I/O-efficient Range Minima Queries

Peyman Afshani[1⋆] and Nodari Sitchinava[2]

[1] MADALGO, Department of Computer Science, University of Aarhus, Denmark
peyman@madalgo.au.dk
[2] Department of Information and Computer Sciences, Univ. of Hawaii – Manoa, USA
nodari.sitchinava@hawaii.edu

Abstract. In this paper we study the *offline (batched) range minima query (RMQ)* problem in the external memory (EM) and cache-oblivious (CO) models. In the *static* RMQ problem, given an array A, a query $\text{RMQ}_A(i, j)$ returns the smallest element in the range $A[i, j]$.

If B is the size of the block and m is the number of blocks that fit in the internal memory in the EM and CO models, we show that Q range minima queries on an array of size N can be answered in $\text{O}\left(\frac{N}{B} + \frac{Q}{B}\log_m \frac{Q}{B}\right) = \text{O}(\text{scan}(N)+\text{sort}(Q))$ I/Os in the CO model and slightly better $\text{O}(\text{scan}(N)+\frac{Q}{B}\log_m \min\{\frac{Q}{B}, \frac{N}{B}\})$ I/Os in the EM model and linear space in both models. Our cache-oblivious result is new and our external memory result is an improvement of the previously known bound. We also show that the EM bound is tight by proving a matching lower bound. Our lower bound holds even if the queries are presorted in any predefined order.

In the batched *dynamic* RMQ problem, the queries must be answered in the presence of the updates (insertions/deletions) to the array. We show that in the EM model we can solve this problem in $\text{O}\left(\text{sort}(N) + \text{sort}(Q)\log_m \frac{N}{B}\right)$ I/Os, again improving the best previously known bound.

1 Introduction

Given an array A on N entries, the *range minimum query (RMQ)* $\text{RMQ}(i, j)$, such that $1 \leq i \leq N$, asks for the item in the range $A[i..j]$ with the smallest value.[3] Range minima queries have many practical applications such as data compression, text indexing and graph algorithms and they have been studied extensively. In internal memory, there are many papers that deal with answering range minima queries in constant time and the main basic idea is to use Cartesian trees [12] and to find least common ancestors [10] (see also [7, 8, 3] for a subset of other results on reducing space and other improvements).

In this paper we are interested in the RMQ problem in the *external memory* model. The external memory model (also known as the *I/O model* or *disk*

⋆ Work supported in part by the Danish National Research Foundation grant DNRF84 through Center for Massive Data Algorithmics (MADALGO)

[3] The query might ask for the index of the item instead, but this variation is an easy adaptation of the known solutions – including the ones in this paper.

Problem	I/Os	Space	Notes
Static RMQ, EM	$\mathrm{O}((n+q)\log_m(n+q))$	$\mathrm{O}(Q+N\log_m N)$	[5]
Static RMQ, EM	$\mathrm{O}((n+q)\log_m(n+q))$	$\mathrm{O}(N+Q)$	[2]
Static RMQ, EM	$\mathrm{O}(n+q\log_m \min\{q,n\})$	$\mathrm{O}(N+Q)$	**new**
Static RMQ, EM, CO	$\Omega(n+q\log_m \min\{q,n\})$	-	**new**
Static RMQ, CO	$\mathrm{O}(n+q\log_m q)$	$\mathrm{O}(N+Q)$	**new**
Dynamic RMQ, EM	$\mathrm{O}((n+q)\log_m^2(n+q))$	$\mathrm{O}(N+Q)$	[2]
Dynamic RMQ, EM	$\mathrm{O}((n+q\log_m q)\log_m n)$	$\mathrm{O}(N+Q)$	**new**

Table 1. Previous and new results on static and dynamic RMQs in the external memory model (EM) and the cache-oblivious model (CO).

access model (DAM)) was introduced by Aggarwal and Vitter [1] and addresses situations where the data is so big that it can only be stored in slow external storage. The external storage is divided into blocks of size B and all the computations must be done in the internal memory of size M. Each data transfer, an *input/output (I/O)* operation, between the external and internal memory can transfer a single block. The complexity metric of the model, *I/O complexity*, measures the number of such transfers. In this paper we use the common notations $n = N/B$, $m = M/B$, $q = Q/B$, and $\mathrm{sort}(N) = \mathrm{O}(n\log_m n)$ – the I/O complexity to sort an array of N elements.

In the external memory model, the online RMQ problem where we require that the answer to each query must be provided immediately, one must spend at least one I/O operation to report the output and, therefore, constant time solutions in the RAM model translate to the optimal solutions in the EM model as well. Instead, Chiang et al. [5] considered the offline version of the problem. In the *offline (batched)* range minima problem we are given a sequence of Q range minima queries $\textsc{rmq}(i,j)$ and we are asked to answer each query eventually and in arbitrary order by presenting the output as pairs of the input queries and the corresponding answers.

Previous results in the EM model. Chiang et al. [5] presented an algorithm that answers a batch of Q queries using $\mathrm{O}(\mathrm{sort}(N+Q)) = \mathrm{O}((n+q)\log_m(n+q))$ I/Os and $\mathrm{O}(Q+N\log_m N)$ space. Very recently, Arge et al. [2] improved the space to $\mathrm{O}(N+Q)$ while keeping the same I/O complexity. They also showed a solution for the *dynamic* version of the problem where the sequence of queries is intermixed with insertions and deletions of entries to and from the array. Their solution requires $\mathrm{O}\left((n+q)\log_m^2(n+q)\right)$ I/Os. They left a few open questions and in fact they explicitly conjectured that even the static range minima queries should require $\Omega((n+q)\log_m(n+q))$ I/Os in the worse case. The conjecture is non-trivial and interesting because in internal memory, the constant time per query trivially implies $O(N+Q)$ time to answer Q queries in an array of size N.

Our Results. We offer a number of improvements to both static and dynamic batched RMQs. In Section 2, we prove a lower bound of $\Omega(n+q\log_m \min\{q,n\})$ I/Os for the static batched RMQ problem, partially confirming the suspicion of Arge et al. [2] that it is impossible to achieve linear $O(n+q)$ I/O complexity in the EM model. Our lower bound assumes the standard indivisibility of individual records and holds even if the queries are presorted. In the process of proving the lower bound we present an algebraic notation which simplifies the presentation of permutation lower bound proofs and might be of independent interest.

In Section 3 we present a matching upper bound in the EM model, thus proving that our lower bound is asymptotically optimal. Our upper bound immediately implies an improvement to the dynamic version of the RMQ problem by Arge et al. [2], which can be solved in $O(\mathrm{sort}(N)+\mathrm{sort}(Q)\cdot\log_m n)$ I/Os (Section 5).

In Section 4 we present the first solution for the static RMQs in the cache-oblivious model[4]. The *cache-oblivious (CO)* model [9] is similar to the EM model, except the algorithms are not allowed to make use of the parameters M and B. Instead, the data transfer between the external and internal memory is performed automatically by a separate paging algorithm implemented by the system with a reasonable cache replacement strategy, e.g., least recently used (LRU) strategy. Our cache-oblivious solution requires $O(n+q\log_m q)$ beating all the previous results in the EM model.

Table 1 lists our results in comparison with the previous results in the external memory and the cache-oblivious models.

Finally, in Section 5 we discuss some additional simple improvements if some blocks of the input array are not covered by any queries.

2 Lower Bound In Both Models

In this section, we prove a lower bound showing that under a standard assumption of indivisibility of individual items it is impossible to answer Q RMQ queries on a static array of size N in fewer than $\Omega(n+q\log_m \min\{q,n\})$ I/Os.

Atomic elements. We assume each query is accompanied by a label that is a string obtained by concatenating the representation of its left and right boundaries. So, a query $q_i=[\ell_i,r_i]$ is represented by $(s_{\ell_i r_i},\ell_i,r_i)$, where $s_{\ell_i r_i}$ is its label. Query labels and values in the array A are considered *atomic elements*.

The Model. We conceptually view the external memory as a (horizontal) tape of infinite size consisting of cells arranged from left to right that are also organized into blocks of B cells. Each cell can store one atomic element. Any other information can be stored and accessed for free by the algorithm (i.e., we assume unlimited computational power and full information). The only restriction placed on the algorithm is that it cannot create new atomic elements, but can only make copies of the existing ones. Thus, to manipulate labels or values, the

[4] Previously, only *online* results were known. E.g., see [6, 11].

algorithm can load one block (containing some atomic elements) from the tape into the internal memory or it can select B atomic elements from the internal memory and write copies of them somewhere on the tape as one block. The algorithm starts with a tape that contains the input values of A in n blocks and the Q queries in the q following blocks and it must end with a tape configuration where each query label is followed by its answer (i.e., a pair $(s_{\ell_i r_i}, A[j])$ where $A[j]$ is the answer to the query $[\ell_i, r_i]$ labeled s_{ℓ_i, r_j}).

Sequences. In this model, a subset of K cells naturally defines a sequence of K atomic elements, by considering the atomic elements stored in the cells in the left-to-right order. In the rest of this section, we slightly extend the definition of a sequence: a *sequence representation* (*seq-rep* for short) is a sequence of K atomic elements that is stored in $O(K/B)$ blocks[5] on the tape from left to right. Note that we allow some inefficiency in the storage as there could be blocks that store only a few atomic elements. Observe that one sequence can have two different seq-reps S_1 and S_2 and the atomic elements of each block could occupy different addresses within that block. Nonetheless, one can convert one representation into another in $O(K/B)$ I/Os. This implies that for a given sequence, all the seq-reps are essentially equivalent up to an additive term of $O(K/B)$ I/Os.

The Main Idea and Intuition. We prove our lower bound using known hardness results for the problem of permuting array entries. Intuitively, the hard input instance to the RMQ algorithm is a set of queries where the left end points and the right end points correspond to two very "different" permutations; our lower bound follows from the fact that the permutation corresponding to the left end points needs $\Omega(\min\{Q, q \log_m n\}$ I/Os to be transformed into the permutation corresponding to the right end points.

Although our lower bound approach does not introduce fundamentally new techniques, it does require rather complicated logical steps. To follow the argument with greater ease, we introduce a new algebraic notation, which could be considered an interesting way of presenting permutation lower bounds.

An Algebraic Notation. Let $\mathbf{X} := X_1, \ldots, X_N$ be a sequence of N atomic elements. For a given permutation $\pi : \{1, \cdots, N\} \to \{1, \cdots, N\}$, $\pi(\mathbf{X})$ is defined as the sequence $X_{\pi(1)}, \ldots, X_{\pi(N)}$ and we denote $X_{\pi(i)}$ with $\pi^{(i)}(\mathbf{X})$. Furthermore, if we can permute one seq-rep of $\pi(\mathbf{X})$ into another seq-rep of $\varkappa(\mathbf{X})$ using t I/Os, then we can permute *any* seq-rep of $\pi(\mathbf{X})$ into *any* seq-rep of $\varkappa(\mathbf{X})$ using $t + O(n)$ I/Os ($n = N/B$). Note that we can also permute *any* seq-rep of $\varkappa(\mathbf{X})$ into *any* seq-rep of $\pi(\mathbf{X})$ using the same $t + O(n)$ I/Os. We denote such transformation with $\pi(\mathbf{X}) \overset{t+O(n)}{\leftsquigarrow} \varkappa(\mathbf{X})$. Easy to see but important consequences of the indivisibility assumption are summarized below.

[5] The O-notation here hides a universal constant that does not depend on any machine or input parameter. We need this constant since during some steps of our proof, we will be working with the sequences that do not necessarily pack B elements in each block.

Observation 1 *Consider two sequences of symbols* $\mathbf{X} := X_1, \ldots, X_N$ *and* $\mathbf{Y} := Y_1, \ldots, Y_N$. *Let* $\varkappa, \pi$ *and* φ *be three permutations. The following properties hold in the indivisibility model regarding the seq-reps of these sequences.*

(a) If $\pi(\mathbf{X}) \overset{t+O(n)}{\longleftrightsquigarrow} \varkappa(\mathbf{X})$ *then* $\pi(\mathbf{Y}) \overset{t+O(n)}{\longleftrightsquigarrow} \varkappa(\mathbf{Y})$

(b) If $\pi(\mathbf{X}) \overset{t+O(n)}{\longleftrightsquigarrow} \varkappa(\mathbf{X})$ *then* $\pi(\varphi(\mathbf{X})) \overset{t+O(n)}{\longleftrightsquigarrow} \varkappa(\varphi(\mathbf{X}))$ *and* $\varphi(\pi(\mathbf{X})) \overset{t+O(n)}{\longleftrightsquigarrow} \varphi(\varkappa(\mathbf{X}))$

(c) If $\pi(\mathbf{X}) \overset{t+O(n)}{\longleftrightsquigarrow} \varkappa(\mathbf{X})$ *and* $\varkappa(\mathbf{X}) \overset{r+O(n)}{\longleftrightsquigarrow} \varphi(\mathbf{X})$ *then* $\pi(\mathbf{X}) \overset{t+r+O(n)}{\longleftrightsquigarrow} \varphi(\mathbf{X})$.

Remark. The constants hidden in the O-notations above can grow. This is because we are working with *any* seq-rep of permutations rather than specific ones.

The Query Order. We actually prove a stronger lower bound claim. We show that the problem stays hard even if the queries are given in the order of the left end points, or the right end points, or any other ordering that does *not* depend on the input array A. We model this claim precisely. Let $\mathcal{Q}$ be the list of queries. Before showing the algorithm the input set A, we allow the algorithm to pick whatever order that it desires for the queries, i.e., the algorithm can permute the queries for free. Once that order is picked (for example, the algorithm can sort the list of queries by the left end points), the algorithm is given an array A. We show that even in this relaxed formulation, the algorithm cannot achieve $O(n+q)$ bound on the number of I/Os.

Observe in the case $Q < N$ we simply need to prove a lower bound for Q queries and an input array of size Q since the upper bound in the previous section has linear dependency on n. Thus, the non-trivial case of the problem is when $Q = \Omega(N)$. Due to this, w.l.o.g, we assume the following in the rest of this section: the range of the queries run from 1 to N and $Q = \alpha N$ where $\alpha \geq 1$ is an integer. We also need the following lemma, which is an easy generalization of the permutation lower bound [1].

Lemma 1. *Let* N *and* α *be two integral parameters, and let* S_1 *be the non-decreasing sequence of length* αN *composed of* α *repetitions of* i, $i = 1, \cdots, N$. *Assuming* $2 < B < cM < N$ *for a constant* c, *there exists a permutation* S_2 *of* S_1, *s.t., permuting* S_1 *into* S_2 *requires* $\Omega\left(\min\{\alpha N, \frac{\alpha N}{B} \log_m \frac{N}{B}\}\right)$ *I/Os.*

Proof. The proof is almost identical to the one presented by Aggarwal and Vitter [1] for the general permutations. The only difference is that we need to calculate the number of permutations of S_1. Using straightforward combinatorial arguments and Stirling's formula, the number of permutations of S_1 is $\prod_{i=0}^{N-1} \binom{\alpha(N-i)}{\alpha} \geq \left(\frac{1}{\sqrt{\alpha}}\right)^N (N!)^\alpha$. Using this bound instead of $N!$ at the right hand side of the inequality in Section 4 of Aggarwal and Vitter's paper [1] gives the claimed bound. □

Now we are ready to prove our lower bound result.

Theorem 1. *A set of* Q *range minima queries on a static array of size* N *requires* $\Omega(\min\{Q, q \log_m n\}$ *I/Os in the worst case, assuming indivisibility.*

Proof. Consider the sequences S_1 and S_2 defined in Lemma 1; observe both have α repetitions of every value i between 1 and N. We create the sequence of queries $\mathcal{Q}$ based on S_2 in the following way: if the i-th element of S_2 is j, we create the query interval $[\lfloor i/\alpha \rfloor, j]$ with its appropriate label. We present $\mathcal{Q}$ to the algorithm and let $\varkappa(\mathcal{Q})$ be the ordering of the queries picked by the algorithm (remember this is done for free). Note that $\mathcal{Q}$ is sorted by the left end point.

We now define two different input arrays, $A_1[1, \cdots, N]$ and $A_2[1, \cdots, N]$: A_1 is strictly increasing and A_2 is strictly decreasing. This means, the left end points of the queries give the indices of the answers for the queries on A_1, while the right end points do the same on A_2. However, remember that the final answer should contain the labels of the queries. We claim that one of these two inputs should be difficult to solve regardless of choice of $\varkappa$.

Let r_1 be the number of I/Os used by an algorithm to solve the problem when presented with A_1 and σ_1 be the permutation that describes the order of the atomic elements (the query label, value pair) in the output. For simplicity, we assume $\mathcal{Q}$ is the sequence of the query labels. Observe that $\sigma_1(\varkappa(S_1))$ describes the sequence of indices of the answers to the queries in the first input: a query interval $[i, j]$ in the output is followed by $A_1[i]$ and since queries were originally ordered by the left end point, $\sigma_1(\varkappa(S_1))$ gives the ordering of the indices of the answer.

Now consider the input A_2. Let r_2 be the number of I/Os used by an algorithm to solve the problem when presented with A_2 and σ_2 be the permutation that describes the order of the atomic elements (the query label, value pair) in the output. A query interval $[i, j]$ in the output if followed by $A_2[j]$. This means the sequence of indices of the answers to the queries in the second input is described by $\sigma_2(\varkappa(\varphi(S_1)))$ where φ is a permutation such that $\varphi(S_1) = S_2$.

Thus, we have the following (explanations below):

$$\varkappa(\mathcal{Q}) \overset{r_1}{\leftrightsquigarrow} \sigma_1(\varkappa(\mathcal{Q})) \tag{1}$$

$$\varkappa(\mathcal{Q}) \overset{r_2}{\leftrightsquigarrow} \sigma_2(\varkappa(\mathcal{Q})) \tag{2}$$

$$S_1 \overset{r_1}{\leftrightsquigarrow} \sigma_1(\varkappa(S_1)) \tag{3}$$

$$S_1 \overset{r_2}{\leftrightsquigarrow} \sigma_2(\varkappa(\varphi(S_1))) \tag{4}$$

The above equations describe how the order of the atomic elements in the output correspond to the order of the atomic elements given to the algorithm, with the difference that (for simplicity) instead of dealing with the values in the arrays A_1 and A_2, we are dealing with their indices; S_1 in the left hand side of the equations correspond to the indices of the values in arrays A_1 and A_2.

Applying Observation 1(c) to (1) through (4), we get

$$\sigma_1(\varkappa(\mathcal{Q})) \overset{r_1+r_2+O(q)}{\leftrightsquigarrow} \sigma_2(\varkappa(\mathcal{Q})) \tag{5}$$

$$\sigma_1(\varkappa(S_1)) \overset{r_1+r_2+O(q)}{\leftrightsquigarrow} \sigma_2(\varkappa(\varphi(S_1))) \tag{6}$$

Applying Observation 1(a) to (5) we get $\sigma_1(\varkappa(S_1)) \overset{r_1+r_2+O(q)}{\longleftrightsquigarrow} \sigma_2(\varkappa(S_1))$. Finally, with (6) and Observation 1(c) we obtain $\sigma_1(\varkappa(S_1)) \overset{O(r_1+r_2+q)}{\longleftrightsquigarrow} \sigma_1(\varkappa(\varphi(S_1)))$. Set φ as an inverse of σ_1 in Observation 1(b) and this gives $\varkappa(S_1) \overset{O(r_1+r_2+q)}{\longleftrightsquigarrow} \varkappa(\varphi(S_1))$ and similarly $S_1 \overset{O(r_1+r_2+q)}{\longleftrightsquigarrow} \varphi(S_1) = S_2$. Thus, by Lemma 1, we must have $r_1 + r_2 = \Omega(\min\{Q, q \log_m n\})$, so the problem is hard on either A_1 or A_2. □

3 Solution In The External Memory Model

In this section we prove a matching upper bound for the static RMQ problem in the EM model.

Theorem 2. *A set of Q range minima queries on a static array of N elements can be answered in $\mathrm{O}(n + q \cdot \min\{\log_m n, \log_m q\})$ I/Os and $\mathrm{O}(N + Q)$ space.*

Note that when $Q = \Theta(N)$ the I/O complexity in the above theorem matches the I/O complexity $\mathrm{O}(\mathrm{sort}(N + Q)) = \mathrm{O}((n + q) \log_m(n + q))$ of Arge et al. [2]. Thus, we concentrate on two cases: (i) when $N = \omega(Q)$ and (ii) when $Q = \omega(N)$.

Without loss of generality we assume that each query $\textsc{rmq}(i, j)$ has a unique identifier – it can be, for example, the initial index in the list of the input queries.

Lemma 2. *The problem of answering a set of Q range minima queries on a static array A of $N = \omega(Q)$ elements can be reduced to the problem of answering Q range minima queries on a static array A' of size $\mathrm{O}(Q)$ in $\mathrm{O}(n + q \log_m q)$ I/Os and $\mathrm{O}(N + Q) = \mathrm{O}(N)$ space.*

Proof. Consider any two adjacent array entries $A[i]$ and $A[i+1]$. Observe that if no query starts or ends with an index i and $i+1$, then the larger of the two entries $A[i]$ and $A[i + 1]$ will not be the answer to any of the queries. More generally, for any contiguous region of the array $A[i..j]$, $i < j$, if there are no queries with endpoint indices in the range $[i, j]$, then we can compact the subarray $A[i..j]$ to a single element that is the minimum in the range $A[i..j]$ without affecting the answers to the queries. Since there are $2Q$ query endpoints, the size of the compacted array is $\mathrm{O}(Q)$. Obviously, if we compact the input array to a smaller array, we have to adjust the query endpoints appropriately, which we show how to do next.

For each query $\textsc{rmq}(i, j)$ we create two items e_i and e_j associated with the two endpoints of the query. Each endpoint e_i (resp. e_j) contains full information about the query $\textsc{rmq}(i, j)$, i.e., the unique identifier of the query and the index j (resp. i) of the other endpoint.

We sort the set of endpoints e_x by their indices x. By simultaneously scanning the input array and the sorted set of endpoints we can identify the ranges of array indices that contain no query endpoints. During the scan we can also identify the minimum within each range and copy them into a new array A'. Let $s[i]$ be the number of items among $A[1..i]$ that were not copied to A'. We can compute

the values $s[i]$ for all $1 \leq i \leq N$ during the scan. To adjust the queries, we need to update the index of each query endpoint e_i from i to $i - s[i]$. This can be accomplished with a simultaneous scan of the sorted endpoints and the values $s[i]$. Finally, a sort of the endpoints by the query identifiers will place the two endpoints of each query in adjacent memory locations and with a final scan of this sorted sequence we can create the updated queries $\textsc{rmq}(i - s[i], j - s[j])$ for each original query $\textsc{rmq}(i, j)$. The I/O complexity of the whole process is $\mathrm{O}(n + \mathrm{sort}(Q)) = \mathrm{O}(n + q \log_m q)$ I/Os because it is just $\mathrm{O}(1)$ scans of arrays of size $\mathrm{O}(N)$ and $\mathrm{O}(1)$ sorts of sets of size $\mathrm{O}(Q)$. □

Lemma 3. *A set of Q range minima queries on a static array A of $N = o(Q)$ elements can be answered in $\mathrm{O}(q \log_m n)$ I/Os and $\mathrm{O}(N + Q) = \mathrm{O}(Q)$ space.*

Proof. In the algorithm of Arge et al. [2], it is difficult to avoid the $\mathrm{O}(\mathrm{sort}(N+Q))$ cost; to summarize, they do the following: first they build a full k-ary tree T for $k \in \Theta(m)$ on the array A, with each of $\Theta(N/M)$ leaves associated with a contiguous range of $\Theta(M)$ entries. The algorithm processes the queries down this tree level by level, by computing a running answer for each endpoint of a query and distributing the endpoints to the appropriate children of a node. At the leaves of the tree, the answer to each query $\textsc{rmq}(i, j)$ is the minimum of the running answers at the two endpoints e_i and e_j. The two endpoints might be in two different leaves of the tree, i.e., in arbitrary locations in external memory. To compute the minimum of each pair I/O-efficiently, the algorithm sorts the endpoints by the query identifier, which results in the two endpoints being in adjacent memory locations and the minimum can be computed with a simple scan. The I/O complexity of this solution consists of $\mathrm{O}(\mathrm{sort}(N))$ I/Os to build the tree, $\mathrm{O}(q \log_m(N/M))$ to propagate all queries down to the leaves of the tree (the distribution involves scanning Q queries at each of $\mathrm{O}(\log_m(N/M))$ levels of the tree), and $\mathrm{O}(\mathrm{sort}(Q))$ I/Os to compute the minima of pairs of endpoints at the leaves of the tree. Note, when $N = o(Q)$ the I/O complexity of this solution reduces to $\mathrm{O}(\mathrm{sort}(Q)) = \mathrm{O}(q \log_m q)$ I/Os.

To improve the I/O complexity to $\mathrm{O}(q \log_m n)$ we show how to compute the minima of the pairs of endpoints at the leaves of the tree more efficiently. In particular, we observe that the distribution of the query endpoints to the children nodes of the tree is performed stably – that is, the relative order of the queries distributed to each child node is the same as in the (parent) node itself. Thus, we maintain the invariant that at each node the query endpoints are sorted by the initial order of the input queries. Initially, at the root of the node, the invariant is trivially true and the stability of the distribution ensures that the invariant is maintained at each consequent level.

Once the query endpoints reach the leaf level, we do the following. We load $\mathrm{O}(M)$ array entries associated with a leaf into internal memory and scan the endpoints within that leaf, finding and reporting the answers to queries that contain both endpoints within that leaf. Once a query answer is determined unambiguously, we stop considering it any further. At this point, instead of sorting the remaining endpoints, we propagate them up the tree, merging them

by comparing the original indices of the query in the input set. Since the query endpoints at each node are sorted in this order, we can perform this merge I/O efficiently and if the two endpoints e_i and e_j of a query $\textsc{rmq}(i, j)$ are present in the subtrees rooted at two children w_k and $w_{k'}$ of some tree node v, the merging process at node v will place them next to each other and we can compute the minima among both endpoints, report it as the answer to query $\textsc{rmq}(i, j)$ and stop considering the two endpoints any further.

The I/O complexity of the merging process is $O(n + q)$ to process the leaves and $O(q \log_m(N/M))$ I/Os to perform the merge up the tree. Thus the total I/O complexity of the whole algorithm adds up to $O(q \log_m n)$ I/Os. □

The proof of the Theorem 2 follows from Lemma 2 and Lemma 3.

4 Solution In The Cache-Oblivious Model

In this section we will prove the following result:

Theorem 3. *In the cache-oblivious model a set of Q range minima queries on an array of size N can be answered in $O(n + q \log_m q)$ I/Os, assuming $M = \Omega(B^{1+\epsilon})$.*

First, note that Lemma 2 holds in the cache-oblivious model because the reduction consists of a constant number of scans and sorts, which can be accomplished cache-obliviously [9]. Thus, it only remains to show how to answer Q range minima queries on an array of size $N = O(Q)$ in $O(q \log_m q)$ I/Os.

The static solution of Arge et al. [2] can be viewed as using the top-down distribution sweeping approach, where at each node of the recursive tree the queries are considered in some predetermined order (a sweep of queries) and distributed to the $\Theta(M/B)$ children of the node. Brodal and Fagerberg [4] presented a framework to implement distribution sweeping paradigm cache-obliviously by a bottom-up recursive process, where at each recursive level the objects of the children nodes are merged. We will show how to answer the range minima queries by merging the queries bottom up instead, thus allowing us to use the cache-oblivious distribution sweeping framework of Brodal and Fagerberg.

Again, without loss of generality, we assume that each query $\textsc{rmq}(i, j)$ has a unique identifier.

We proceed as follows. For each query $\textsc{rmq}(i, j)$ we create two items e_i and e_j associated with the two endpoints of the query. Each endpoint e_i (resp. e_j) contains full information about the query $\textsc{rmq}(i, j)$, i.e., the unique identifier of the query and the index j (resp. i) of the other endpoint. Each endpoint e_x we will maintain a running answer rmq_{e_x}. At the end of the computation, both rmq_{e_i} and rmq_{e_j} will hold the answer to the query $\textsc{rmq}(i, j)$.

Initially, we sort the endpoints e_x by its index x and initialize $rmq_{e_i} = A[i]$ and $rmq_{e_j} = A[j]$. Next we perform the following merging algorithm. Conceptually, we can visualize a merge tree built on top of the sorted list of endpoints with a single endpoint at each leaf of the merge tree. A node v of the

tree represents a contiguous range $\mathcal{R}(v)$ of the indices in the array, such that $\mathcal{R}(v) = \mathcal{R}(w_L) \cup \mathcal{R}(w_R)$, where w_L and w_R are the two children of v. Each node v of the tree maintains $minS(v)$ – the smallest array entry among the indices in its range $\mathcal{R}(v)$. This value is defined as rmq_e at the leaf node e and can be computed at each internal node v as $minS(v) = \min\{minS(w_L), minS(w_R)\}$ and is updated as the first step before the merging at that node begins.

During the merge up the tree the endpoints are compared by the unique identifiers of the queries associated with that endpoint. For the merge step at each internal node v with the two children w_L and w_R we do the following. If the next two smallest endpoints e_i and e_j are for the same query $\text{RMQ}(i, j)$, we set $rmq_i = rmq_j = \min\{rmq_i, rmq_j\}$ (the final answer to query $\text{RMQ}(i, j)$) and the two endpoints are discarded and never considered again in the merging process. If the next two smallest endpoints are not of the same query, assume the smallest endpoint e is the left endpoint of a query (the right endpoints are treated symmetrically). Then if e is coming from the right child w_R, we propagate e to the output of node v without altering it. If e is coming from the left child w_L, we set $rmq_e = \min\{rmq_e, minS(w_R)\}$ and then propagate it to the output of v.

Lemma 4. *At the end of the merging process, all pairs of items e_i and e_j associated with each query $\text{RMQ}(i, j)$ store the answer to the query in rmq_{e_i} and rmq_{e_j}.*

Proof. The proof is by induction on the level of recursion. First observe that at node v, if $e_i \in \mathcal{R}(w_L)$ and $e_j \in \mathcal{R}(w_R)$ and they represent the same query $\text{RMQ}(i, j)$, they will be considered together at some point during merging at node v, because the comparisons are performed by the query identifiers, which are unique and equal for e_i and e_j. Thus an item $e \in R(v)$ is propagated to the output of a node v iff e's other endpoint is not in $\mathcal{R}(v)$'s subtree. Let A_v represent the subarray of A which is defined by the indices in the range $\mathcal{R}(v)$. Then the correctness of the algorithm follows from the fact that for $i < j$, if $e_i \in w_L$ and $e_j \notin w_R$, $\text{RMQ}_{A_{w_R} \cup A_{w_L}}(i, +\infty) = \min\{\text{RMQ}_{A_{w_L}}(i, +\infty), \text{RMQ}_{A_{w_R}}(-\infty, +\infty)\}$, and if $e_i \in w_R$ and $e_j \notin w_R$, $\text{RMQ}_{A_{w_R} \cup A_{w_L}}(i, +\infty) = \text{RMQ}_{A_{w_R}}(i, +\infty)$. The case of e_j is symmetrical. □

Now we are ready to prove Theorem 3 stated at the beginning of this section.

Proof (of Theorem 3). Creation of the items can be performed with a single scan of the queries, which is trivially cache-oblivious. The initial sorting of the items is implemented using one of the cache-oblivious sorting algorithms [9]. The initialization of rmq_e is implemented using a simultaneous scan of the input array and the sorted items. Finally, the merging is implemented using the lazy funnels [4]. Note, that we compute the value $minS(v)$ only once – the first time a merger at node v is invoked. Both cache-oblivious sorting and lazy funnels require the tall cache assumption $M = \Omega(B^{1+\epsilon})$. The I/O complexity follows from [4]. □

Note, we can extend the above merging algorithm to solve the problem in the external memory model using merging, rather than distribution, which might

be of independent interest. This is accomplished by performing $\Theta(M/B)$-way merging at each node and maintaining at each node $\Theta(M/B)$ minima of all its children.

5 Additional Improvements

The techniques described in the previous sections are quite simple and can be applied to other contexts. We briefly discuss some of these in this section.

Towards an adaptive analysis: In special cases when large portions of the input array do not overlap with the query ranges, one can achieve better I/O complexity than of the algorithms presented here. Let $n' \leq n$ denote the number of blocks of the input which overlap with the union of ranges defined by the queries. Then both our upper bounds and lower bounds can easily be extended to show that the support of batched RMQ queries is within $\Theta(n' + q \log_m \min\{q, n'\})$ accesses and linear space, in both the external memory model and the cache oblivious model.

Dynamic batched RMQ problem: In the dynamic RMQ problem we are given a sequence that contains Q queries and N update operations (insertions and deletions). There are many different ways to model the behavior of the insertions and deletions with respect to the array indices. As discussed by Arge et al. [2], one can consider an *array version* in which the updates shift the indices (an insertion at position i increases all the succeeding indices by one; a deletion reduces them by one), or a *geometric version* in which such shifting does not occur and the indices are in fact x-coordinates, or a *linked list* version where indices are in fact pointers. All these formulations are equivalent up to an additive $\mathrm{O}(\mathrm{sort}(N+Q))$ term. Previously, Arge et al. had shown how to solve such dynamic problems in $\mathrm{O}(\mathrm{sort}(N+Q)\log_m(n+q))$ I/Os. Using our static $\mathrm{O}(n + q\log_m \min\{q, n\})$ solution as the base case in their solution, we can easily improve the I/O complexity of the dynamic batched RMQ solution to $\mathrm{O}(\mathrm{sort}(N) + \mathrm{sort}(Q)\log_m n)$ I/Os.

6 Conclusions

In this paper, we investigate batched range minimum query (RMQ) problem in the external memory (EM) and the cache-oblivious (CO) models. Improving on the previous papers, we obtain matching upper and lower bounds for the static version of the problem in the EM model. Interestingly, our lower bound shows that the problem cannot be solved in linear I/O complexity (in the number of queries) even if we allow the algorithm to reorder the queries in any arbitrary order for free before it is presented with the input array. We also present the first cache-oblivious solution to the problem and although we do not know if it is optimal, it is faster than the previous external memory solutions.

Open problems. Although our work closes the case of the static version of the problem in the EM model, there are still several interesting open problems remaining. There is no better lower bound known for the dynamic version of the problem than the lower bound that we presented here for the static version. And although we improved the upper bound of the dynamic version of the problem, there is still a gap of $O(\log_m n)$ I/Os remaining between the upper and lower bounds. Closing this gap remains an open problem.

In the cache-oblivious model, the merge-based solution presented here seems to require sorting all the queries. Our EM model solutions on the other hand show that when $Q \gg N$ we can avoid the complexity of sorting the queries. It would be interesting to see if similar bound can be shown in the cache-oblivious model or the sorting of the queries is inherently required in the cache-oblivious model.

Acknowledgements. The authors would like to thank Jérémy Barbay for many useful discussions that inspired and motivated us in this work.

References

1. Aggarwal, A., Vitter, J.S.: The input/output complexity of sorting and related problems. Communications of the ACM 31, 1116–1127 (1988)
2. Arge, L., Fischer, J., Sanders, P., Sitchinava, N.: On (dynamic) range minimum queries in external memory. In: Proc. 13th Algorithms and Data Structures Symposium (WADS). pp. 37–48 (2013)
3. Bender, M.A., Farach-Colton, M.: The LCA problem revisited. In: Proc. 4th Latin American Theoretical Informatics Symposium. pp. 88–94 (2000)
4. Brodal, G.S., Fagerberg, R.: Cache oblivious distribution sweeping. In: Proc. 29th International Colloquium on Automata, Languages, and Programming. pp. 426–438 (2002)
5. Chiang, Y.J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E., Vitter, J.S.: External-memory graph algorithms. In: Proc. 6th ACM/SIAM Symposium on Discrete Algorithms. pp. 139–149 (1995)
6. Demaine, E.D., Landau, G.M., Weimann, O.: On cartesian trees and range minimum queries. Algorithmica 68(3), 610–625 (2014)
7. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM Journal on Computing 40(2), 465–492 (2011)
8. Fischer, J.: Optimal succinctness for range minimum queries. In: Proc. 9th Latin American Theoretical Informatics Symposium. pp. 158–169 (2010)
9. Frigo, M., Leiserson, C.E., Prokop, H., Ramachandran, S.: Cache-oblivious algorithms. In: Proc. 40th IEEE Symposium on Foundations of Computer Science. pp. 285–297 (1999)
10. Gabow, H.N., Bentley, J.L., Tarjan, R.E.: Scaling and related techniques for geometry problems. In: Proc. 16th ACM Symposium on Theory of Computation. pp. 135–143 (1984)
11. Hasan, M., Moosa, T.M., Rahman, M.S.: Cache oblivious algorithms for the RMQ and the RMSQ problems. Mathematics in Computer Science 3(4), 433–442 (2010)
12. Vuillemin, J.: A unifying look at data structures. Comm. ACM 23(4), 229–239 (1980)