Categorical Range Maxima Queries*

Manish Patil
Louisiana State University, USA
mpatil@csc.lsu.edu

Sharma V. Thankachan
University of Waterloo, Canada
thanks@uwaterloo.ca

Rahul Shah
Louisiana State University, USA
rahul@csc.lsu.edu

Yakov Nekrich
University of Waterloo, Canada
yakov.nekrich@gmail.com

Jeffrey Scott Vitter
The University of Kansas, USA
jsv@ku.edu

ABSTRACT

Given an array $A[1...n]$ of n distinct elements from the set $\{1, 2, ..., n\}$ a range maximum query RMQ(a, b) returns the highest element in $A[a...b]$ along with its position. In this paper, we study a generalization of this classical problem called *Categorical Range Maxima Query* (CRMQ) problem, in which each element $A[i]$ in the array has an associated category (color) given by $C[i] \in [\sigma]$. A query then asks to report each distinct color c appearing in $C[a...b]$ along with the highest element (and its position) in $A[a...b]$ with color c. Let p_c denote the *position* of the highest element in $A[a...b]$ with color c. We investigate two variants of this problem: a threshold version and a top-k version. In threshold version, we only need to output the colors with $A[p_c]$ more than the input threshold τ, whereas top-k variant asks for k colors with the highest $A[p_c]$ values.

In the word RAM model, we achieve linear space structure along with $O(k)$ query time, that can report colors in sorted order of $A[\cdot]$. In external memory, we present a data structure that answers queries in optimal $O(1+\frac{k}{B})$ I/O's using almost-linear $O(n \log^* n)$ space, as well as a linear space data structure with $O(\log^* n + \frac{k}{B})$ query I/Os. Here k represents the output size, $\log^* n$ is the iterated logarithm of n and B is the block size. CRMQ has applications to document retrieval and categorical range reporting – giving a one-shot framework to obtain improved results in both these problems. Our results for CRMQ not only improve the existing best known results for three-sided categorical range reporting but also overcome the hurdle of maintaining color uniqueness in the output set.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Trees; Tables; F.2 [**ANALYSIS OF ALGORITHMS AND PROBLEM COMPLEXITY**]: Tradeoffs among Complexity Measures

Keywords

I/O Efficiency, Categorical Queries

*This work is supported in part by National Science Foundation (NSF) Grants CCF–1017623 (R. Shah and J. S. Vitter) and CCF–1218904 (R. Shah).

PODS'14, June 22–27, 2014, Snowbird, UT, USA.

1. INTRODUCTION

Given an array A of n elements from a totally ordered set, a natural question is to ask for the position of a maximum element between two specified indices a and b. Queries of this form are known as range maximum queries (RMQ). Consider a sample query: "Give me the highest paid employee within age group 18 to 22 years". By arranging all employees in a age-sorted array with his/her salary as the key, this query translates into an RMQ problem. Being an important tool in designing data structures for numerous problems in string processing and computation geometry, RMQ has been extensively studied in the literature [5, 4, 10]. There are several variants of the problem, the most prominent being the one where the array is static and known in advance. The current best known result for such a scenario is by Fischer and Heun [10], where they present a $2n + o(n)$-bit structure capable of answering queries in constant time.

However, in many applications, the standard RMQ problem does not suffice. Consider the generalization of the above query as a motivating example: "Give me the list of highest paid employees for different job positions (one per job position) with age between 18 to 22 years". This problem can obviously be solved by maintaining age-sorted array of employees as before for each designation in the organizational hierarchy and then issuing a RMQ for all of them. However, this solution may be very inefficient as the job positions held by employees within the specified age group can be only a fraction of all listed positions for the organization. We call the above problem to be an instance of *Categorical Range Maxima Query* (CRMQ). For CRMQ, we assume that each element in the input array A is assigned a color. The goal is to preprocess the array and maintain a data structure, such that given a query range $[a, b]$, one can efficiently report each distinct color c in the query range along with the highest element in $A[a...b]$ with color c. Further continuing the example under consideration, lets say we only need to output the job positions where the highest paid employee with that designation earns more than \$80,000 per year. This natural extension of CRMQ called "threshold-CRMQ" problem is formally defined below.

PROBLEM 1. *[Threshold-CRMQ] Let $A[1...n]$ be an array of n distinct integers in $[1, n]$ with each element $A[i]$ associated with a color $C[i] \in [\sigma]$. Then, goal is to build a data structure such that, given a query (a, b, τ), we can report the triplet $(c, p_c, A[p_c])$ for those colors $c \in [\sigma]$ with $A[p_c] \geq \tau$. Here $A[p_c]$ represents the highest element in $A[a...b]$ with color c. If there does not exist an element in $A[a...b]$ with color c, then $A[p_c] = -\infty$.*

Top-k queries are widely popular in database and information retrieval systems as they allow end-users to focus on the most important (top-k) outputs amongst those which satisfy the query. We study top-k version of CRMQ problem (top-CRMQ) as well, where the query input consists of a range $[a, b]$ and an integer $k \leq \sigma$, and we are required to output only k colors with the highest $A[p_c]$ values.

PROBLEM 2. *[Top-CRMQ] Let $A[1...n]$ be an array of n distinct integers in $[1, n]$ with each element $A[i]$ associated with a color $C[i] \in [\sigma]$. Then, goal is to build a data structure such that, given a query (a, b, k), we can report k triplets $(c, p_c, A[p_c])$ for colors $c \in [\sigma]$ with the highest $A[p_c]$ values, where $A[p_c]$ represents the highest element in $A[a...b]$ with color c. If there does not exist an element in $A[a...b]$ with color c, then $A[p_c] = -\infty$.*

In this article, we focus on top-CRMQ as our central problem. We distinguish between the sorted and unsorted version of this problem. In the sorted version, a triplet $(c, p_c, A[p_c])$ is reported before $(c', p_{c'}, A[p_{c'}])$, if $A[p_c] > A[p_{c'}]$, whereas unsorted version do not place any such restrictions. We focus on sorted version in Word-RAM model and unsorted version in external memory. Our main results are summarized in following theorems.

THEOREM 1. *There exists a linear space (in words) and optimal $O(k)$ time solution for the (sorted) top-CRMQ problem in Word-RAM model.*

THEOREM 2. *There exists an external memory structure of $O(n \log^* n)$ space and optimal $O(1+\frac{k}{B})$ query I/Os for the top-CRMQ problem, where $\log^* n$ is the iterated logarithm of n and B is the block size.*

THEOREM 3. *There exists an external memory structure of linear-space and near-optimal $O(\log^* n + \frac{k}{B})$ query I/Os for the top-CRMQ problem, where $\log^* n$ is the iterated logarithm of n and B is the block size.*

We improve the query I/O bound of the linear space solution in the above theorem by trading off space to achieve space-time bounds with Inverse Ackermann function. We summarize the result in following theorem with its proof deferred to Appendix D.

THEOREM 4. *The top-CRMQ problem can answered in near-optimal $O(\alpha^3 + \frac{k}{B})$ I/Os using an $O(n\alpha)$-word space structure, α being the Inverse Ackermann function of n.*

Answering Threshold-CRMQ: Data structures for answering top-CRMQ as summarized in theorems above, can be used for answering the threshold-CRMQ as well. Given a threshold-CRMQ (a, b, τ), we issue multiple top-CRMQ's as follows. Assume, we are using the I/O-optimal structure in Theorem 2, then we choose $K_j = 2^j B$ and issue top-CRMQ (a, b, K_j) for $j = 0, 1, 2, 3, ...$ until we find the smallest K_j (say K') where at least one of the triplet $(x, p_x, A[p_x])$ in the output set violates the condition $A[x] \geq \tau$. Then all those triplets corresponding to the output of top-CRMQ (a, b, K') satisfying the condition $A[\cdot] \geq \tau$ can be reported as the final answers. The number of I/O's required is $O(1+2+4+...+K'/B) = O(1+K'/B) = O(1+k/B)$, where k is the output size. If we are using the linear-space structure, we use the same procedure, with $K_j = 2^j B \log^* n$ and the query I/Os can be bounded by $O(\log^* n + (1+2+4+...+k/B)) = O(\log^* n + k/B)$. In conclusion, results in Theorem 2 and Theorem 3 are applicable for threshold-CRMQ as well.

Outline: Section 2 introduces a few existing data structures for several orthogonal range searching problems and give a brief summary of the external-memory model [2]. While CRMQ is an interesting problem in its own right, it is also closely related to other important problems. In Section 3 we describe the applications of our results to categorical range reporting and document retrieval. Section 4- 7 are dedicated for deriving external memory data structures for the top-CRMQ problem. We begin by reducing the problem under consideration to a geometric problem in Section 4. Using the equivalent geometric formulation, we present a simple external memory solution for top-CRMQ in Section 5. We build upon this solution incrementally in Section 6 and 7 to obtain I/O optimal and linear space structures. Internal memory result for top-CRMQ is discussed in Section 8. Finally we conclude in Section 9.

2. PRELIMINARIES

2.1 External Memory Model

The external memory (EM) model [2, 27] is a popular model for analyzing the performance of algorithms when input data set is too large to be accommodated in internal memory and hence resides on the disk. In EM, the CPU is connected directly to an internal memory, which is then connected to a much slower disk. The disk is of an unbounded size and is formatted into disjoint blocks, each of which contains B consecutive words. An I/O operation reads a block of data from the disk into memory, or conversely, writes a block of memory information into the disk. Main memory can accommodate M words and is assumed to have at least two blocks, i.e., $M \geq 2B$. The cost of answering a query is measured in the number of I/Os performed by the algorithm.

2.2 Three-dimensional Dominance Reporting

Given a set S of n points in three dimensions and query point $q = (q_1, q_2, q_3)$, the three-dimensional dominance reporting asks for all the points $s = (x_1, x_2, x_3) \in S$ such that $x_i < q_i$, $1 \leq i \leq 3$. The best known result for the problem is by Afshani [1] which achieves linear space along with optimal $O(\log_B n + k/B)$ query I/Os.

2.3 Three-sided Orthogonal Range Reporting

Given a set S of n points in two dimensions, three-sided orthogonal range reporting asks for all points inside a query rectangle of the form $[x_1, x_2] \times (-\infty, y]$. The best I/O model solution to this range reporting problem is due to Arge et al. [3] which takes linear space and report all the points the query rectangle in $O(\log_B n + k/B)$ I/Os. When the two-dimensional points are on the $[n] \times [n]$ grid, Larsen et. al [17] achieve improved query bound of $O(1 + k/B)$ I/Os.

3. APPLICATIONS OF CRMQ

3.1 Categorical Range Reporting Without Duplicates

In the categorical (or colored) range reporting problem the set of input points is partitioned into categories and stored in a data structure; a query asks for categories of points that belong to the query range. The problem has been extensively studied in computational geometry and database communities [14, 12, 6, 20, 16, 23, 17, 18].

In three-sided color reporting, the query asks to report the set of colors of the points in an input region $[a, b] \times [\tau, +\infty)$. Without loss of generality, we assume that the points are in rank-space ‖. The first external memory result for this problem was given by Nekrich [23]. His results on this problem were further improved by Larsen and Walderveen [18], where they presented an $O(nh)$-word data structure with $O(\log^{(h)} n + \frac{k}{B})$ query cost, k being the output size, $1 \leq h \leq \log^* n, \log^{(h)} n = \log\log^{(h-1)} n$ and $\log^{(1)} n = \log n$. Thus by choosing $h = \log^* n$, an I/O-optimal structure can be obtained. On the other-hand, a linear space structure can be obtained by choosing $h = O(1)$.

The data structures described in [23, 18] have a limitation that can compromise their usefulness in some situations: the list of colors in the output set may contain several (yet constant) occurrences of the same color. Eliminating such duplicates (in the current settings) needs extra I/Os (sorting is inevitable in these solutions, which makes these results less-optimal in terms of query I/Os). In [23], another data structure that uses linear space and reports every color exactly once is described. Unfortunately, this data structure needs $O((\frac{n}{B})^\varepsilon + \frac{k}{B})$ I/Os to answer a query, where ε is an arbitrarily small positive constant. This makes the design of an efficient external data structure that reports every color exactly once an important open problem, and we provide the following solution for it.

THEOREM 5. *A three-sided color reporting query on a set of n points in rank-space can be answered in $O(1 + \frac{k}{B})$ I/Os using an $O(n \log^* n)$-word structure, or in $O(\log^* n + \frac{k}{B})$ I/Os using an $O(n)$-word structure, such that the output set contains* exactly one copy of each answer*, where k is the output size, $\log^* n$ is the iterated logarithm of n and B is the block size.*

PROOF. Let $P = \{(i, y_i) | i = 1, 2, 3, ..., n\}$ be the set of points, then construct the array A, where $A[i] = y_i$ and its color is same as that of (i, y_i). Then the output of any three-sided color reporting query on P with $[a, b] \times [\tau, +\infty)$ as an input is the same as that of a threshold-CRMQ (a, b, τ) on A. Thus, we obtain the results summarized in above theorem using Theorem 2 and Theorem 3. □

Consequently, we achieve a smaller (non-optimal) term of $\log^* n$ in the I/O bound of the linear-space structure compared to the $(\frac{n}{B})^\varepsilon$ or $\log^{(O(1))} n$ terms in the existing solutions. Further, using standard techniques [23, 18] in conjunction with results in Theorem 2, Theorem 3, we obtain following results for (two dimensional) four-sided color reporting problem. Although this improves the known results of the problem [18], the output set may contain multiple (at most two) copies of the same color.

THEOREM 6. *A four-sided color reporting query on a set of n points in an $[n] \times [n]$ grid can be answered in $O(1 + \frac{k}{B})$ I/Os using an $O(n \log n \log^* n)$-word structure, or in $O(\log^* n + \frac{k}{B})$ I/Os using an $O(n \log n)$-word structure. Here k is the output size, $\log^* n$ is the iterated logarithm of n and B is the block size.*

Hardness of color counting: In a color counting problem, our task is to simply report the cardinality of the output set of the corresponding reporting problem. Color counting problems are considered to be much harder than the reporting counterparts. For example, the best known space-time trade-off for two-dimensional four-sided color counting is $O(n^2 \log^2 n)$ words and $O(\log^2 n)$ time [12, 15]. In [18], Larsen and Walderveen show that two-dimensional range counting problem is equivalent to one-dimensional color counting problem. Using a simple extension of their techniques, we can obtain a similar result for three-sided color counting problem as summarized below.

THEOREM 7. *Three-sided color counting problem (in two dimension) is at least as hard as three-dimensional orthogonal range counting problem.*

3.2 Ranked Document Retrieval

Suppose that we want to store a collection $\mathcal{D} = \{d_1, d_2, ..., d_D\}$ of D documents (strings) of total n characters, so that for a given query string P all documents containing P can be reported. This problem can be reduced to one-dimensional color reporting problem and can be solved optimally [20]. A more general and arguably the most important query, known as the *top-k document retrieval query* asks to find those k documents in $\mathcal{D}$ which are most relevant to P, where k is also an input parameter. The relevance of a document d w.r.t a pattern P is captured using a predefined ranking function $w(P, d)$, which is dependent on the set of occurrences of P in d. A popular example is the *term frequency*, where $w(P, d)$ is the number of occurrences of P in d. This problem has been studied extensively in string searching community (See [21] for an excellent survey) and linear-space and optimal query time internal memory results are known [13, 22]. Whereas in external memory, the best known linear space index is given by Shah et al. [25], however the query I/O bound is $O(\frac{|P|}{B} + \log_B n + \log^{(h)} n + \frac{k}{B})$ I/Os for any constant $h \geq 1$. We show that our solution for top-CRMQ can be used to obtain the following new result. Please refer to Appendix B for more details.

THEOREM 8. *If the ranking function is such that, the relevance of a document w.r.t. a pattern is not more that its relevance w.r.t. to any prefix of the same pattern, then we can construct a linear-space structure for answering top-k document retrieval queries in $O(\frac{|P|}{B} + \log_B n + \log^* B + \frac{k}{B})$ I/Os, where P is the input pattern.*

Although, our results require relevance to be a monotonic function (less general than the one considered by Shah et al. [13]), the most popular relevance measures such as term-frequency, term-proximity, Page-Rank etc. are monotonic.

‖By rank-space we assume that the points are in $[n] \times [n]$ grid, and the projections of any two points to any axis is different. If the points are in a $[U] \times [U]$ grid, we can reduce them to $[n] \times [n]$ grid using standard techniques. However the space will increase by an $O(n)$ words and the query cost by $O(\log\log_B U)$ I/Os (or $O(\log\log U)$ time). If the coordinate values are unbounded, the extra term in space is again $O(n)$, but in the query cost is $O(\log_B n)$ I/Os (or $O(\log n)$ time).

3.3 Sorted Reporting

In this problem we want to report all elements of an array A in sorted order. Suppose that we want to store an array A in a data structure such that for any query range $[a, b]$ all elements $A[i]$, $a \leq i \leq b$, can be reported in sorted order. Brodal et al. [7] described a linear space data structure that answers such queries in $O(b-a+1)$ time: moreover their data structure can be also used to report k highest points in the range in sorted order. Karpinski and Nekrich [16] considered the same problem in the color scenario: elements of the array are also assigned colors. We assume that colors are from an ordered set; now the query answer must report the k highest colors that occur in the query range and colors must be reported in the reverse order. We observe that the optimal data structure described in Theorem 1 generalizes the result of [16, 7]. This result is obtained using a new data structure for sorted three-dimensional dominance queries, which may be of independent interest. The result is summarized below (Proof is deferred to Appendix C).

THEOREM 9. *A given set of n three-dimensional points can be stored as an $O(n)$-word data structure that can answer a three-dimensional dominance reporting query in $O(\log n + output)$ time in Word-RAM model, with outputs reported in the sorted order of z coordinate.*

4. THE FRAMEWORK

For color listing problem i.e., to simply enumerate all distinct colors in $C[a...b]$, Muthukrishnan [20] proposed the *chaining* idea, where each occurrence of a particular color points to (or chains to) its predecessor of the same color ¶. Therefore, among all occurrences of a particular color $c \in [\sigma]$ occurring in $C[a...b]$, only the first ones chain will be pointing outside the range $[a, b]$. Based on this observation, he reduced the problem to a (two-dimensional) three-sided range reporting query, which can be solved optimally using known structures. We introduce a generalization of this approach for solving our top-CRMQ problem. Formally, for each position $i \in [1, n]$ in the array A, we define *previous* and *next* pointers as follows:

$$prev(i) = max\{\{j \in [1, i) | A[j] > A[i], C[j] = C[i]\} \cup \{-\infty\}\}$$
$$next(i) = min\{\{j \in (i, n] | A[j] > A[i], C[j] = C[i]\} \cup \{+\infty\}\}$$

Using these pointers, for each position $i \in [1, n]$ in A we obtain a (weighted) interval-pair with $(prev(i), i)$ as a backward interval, $(i, next(i))$ as a forward interval, and $A[i]$, $C[i]$ being the weight and color associated with the interval-pair respectively. We represent such an interval-pair by a pentuple $(i, A[i], C[i], prev(i), next(i))$. The following is a key observation for the *two-sided chaining* just introduced.

LEMMA 1. *For a given range $[a, b]$ and a color c, let $S_{a,b,c} = \{i_1, i_2, ..., i_r\}$ be the (possibly empty) set of all positions within $[a, b]$ such that $C[i_1] = C[i_2] = ... = C[i_r] = c$. If $S_{a,b,c}$ is not an empty set, then exactly one element $p_c \in S_{a,b,c}$ satisfies the following: $prev(p_c) < a, b < next(p_c)$, where $A[p_c] = max\{A[i_1], A[i_2], ..., A[i_r]\}$.*

In order to utilize the above lemma for answering top-CRMQ, we use an $O(n)$-word structure that can compute a threshold $\tau^k_{a,b}$ for a given top-CRMQ (a, b, k) in $O(1)$ time such that size of $Out_\tau = \{(c, p_c, A[p_c]) \mid c \in \sigma, A[p_c] \geq \tau^k_{a,b}\}$ is bounded by $\hat{k} = k + O(k)$, where $A[p_c]$ represents the highest element in $A[a...b]$ with color c (see Appendix A for details). Then, Lemma 1 suggests that if a triplet $(c, p_c, A[p_c])$ is an answer for a top-CRMQ, then the pentuple $(p_c, A[p_c], C[p_c], prev(p_c), next(p_c))$ satisfies the following conditions, and vice versa: $p_c \in [a, b]$, $prev(p_c) < a$, $next(p_c) > b$ and $A[p_c] \geq \tau^k_{a,b}$. Therefore, top-CRMQ can be reduced to a new problem as defined below.

PROBLEM 3. *Store a set $\mathcal{I}$ of n interval-pairs of the form $(i, A[i], C[i], prev(i), next(i))$ in a data structure, such that given a query $(a, b, k, \tau^k_{a,b})$, we can efficiently report all those interval-pairs with weight $\geq \tau^k_{a,b}$ and its backward, forward intervals stabbed by a, b respectively. i.e., output the interval-pairs satisfying the following five constraints:*
(1) $prev(i) < a$ (2) $a \leq i$
(3) $i \leq b$ (4) $b < next(i)$ (5) $A[i] \geq \tau$

Notice that the output set Out_τ for the above problem, is a super set of the output set Out_k of our top-CRMQ, because $\hat{k} \geq k$. Therefore, in order to answer a top-CRMQ, we first find the triplet $(c^*, p_{c^*}, A[p_{c^*}]) \in Out_\tau$ using a selection algorithm such that the number of triplets $(c, p_c, A[p_c]) \in Out_\tau$ with $A[p_{c^*}] \leq A[p_c]$ is k. This takes only $O(\hat{k}/B) = O(k/B)$ I/Os [26]. Then, all those triplets in Out_τ with $A[p_{c^*}] \leq A[p_c]$ can be reported as the final outputs. Both the problems being equivalent, we use the term "top-CRMQ" to refer to either of these problems. In particular, by top-CRMQ (a, b, k) we refer to Problem 2 whereas by top-CRMQ $(a, b, k, \tau^k_{a,b})$ we refer to the Problem 3. Moreover, for notational simplicity, input to the Problem 3 is defined as a quadruple (a, b, k, τ).

5. INTERVAL TREE BASED SOLUTION

In this section, we present a simple interval-tree based external memory data structure and achieve the result summarized in following lemma.

LEMMA 2. *A given set $\mathcal{I}$ of interval-pairs can be maintained as an $O(|\mathcal{I}|)$-space structure such that given a top-CRMQ (a, b, k, τ), we can report all interval-pairs $(i, A[i], C[i], prev(i), next(i)) \in \mathcal{I}$ with $i \in [a, b]$, $prev(i) < a$, $next(i) > b$ and $A[i] \geq \tau$ using $O(\log^3(|\mathcal{I}|/B) + \frac{k}{B})$ I/Os.*

We begin by describing a linear space external memory interval tree (which is not optimal, but is sufficient for our purpose) and then use it to answer top-CRMQ.

5.1 Linear Space Interval Tree

Given a set $\mathcal{I}$ of n intervals of the form (s_i, e_i), where s_i and e_i represent the start and end points, the output of an interval stabbing query is the set of intervals stabbed by a input point q; i.e., we need to output all those intervals (s_j, e_j) such that $q \in [s_j, e_j]$. For simplicity we assume all start and end points to be distinct; otherwise ties can be broken arbitrarily.

The proposed interval tree construction begins with building a balanced binary search tree (BST) of n nodes over all end points e_i of set $\mathcal{I}$. Thus each node u in BST is associated with a unique end point which we denote as $stab(u)$**.

¶If there is no such predecessor, then points to $-\infty$.

**For any given nodes u_1 and u_2, $stab(u_1) \leq stab(u_2)$ if u_1 comes before u_2 during the in-order traversal of BST.

Further each node u is associated with a set of intervals $\mathcal{I}(u) = \{(s_i, e_i) | stab(u) \in [s_i, e_i], stab(v) \notin [s_i, e_i]$, where v is any ancestor of $u\}$. Let $size(u)$ represent the number of leaves in the subtree of u. We finish the construction by making each node u with $size(u) \leq B$, $size(parent(u)) > B$, a leaf node by first setting $\mathcal{I}(u) = \cup_{v \in subtree(u)} \mathcal{I}(v)$ and then pruning its subtree. We emphasize that, in this interval tree, for each leaf u, $\mathcal{I}(u)$ is bounded by $O(B)$[††]. The size of interval tree can now be bounded as $O(n)$ words since $\sum_u |\mathcal{I}(u)| = |\mathcal{I}| = n$. To answer a stabbing query, we first identify the node u_q such that value $stab(u_q)$ is the predecessor of q. Then any interval stabbed by a query point q will be associated with one of the $O(\log(\frac{n}{B}))$ nodes on the path from the root to node u_q. We summarize this property in the following lemma.

Lemma 3. *Given a query point q, we can obtain a set of $O(\log(\frac{n}{B}))$ nodes in the proposed linear space interval tree in $O(\log(\frac{n}{B}))$ I/Os such that any interval stabbed by q is associated with one of these nodes.*

For query point q and each interval (s_j, e_j) associated with any of the $O(\log(\frac{n}{B}))$ nodes obtained by the above lemma, either $s_j \leq q$ or $q \leq e_j$ is true. The interval stabbing query can now be answered by issuing $O(\log(\frac{n}{B}))$ single-constraint queries (i.e., check if $q \leq e_j$ in the case one already knows $s_j \leq q$, and vice versa) on these nodes. Therefore, Lemma 3 can be rewritten as follows.

Lemma 4. *A set $\mathcal{I}$ of n intervals can be categorized into subsets using an interval tree structure, such that an interval stabbing query (with two constraints) can be decomposed into $O(\log(\frac{n}{B}))$ queries with a single constraint.*

5.2 Interval Tree within an Interval Tree

Taking a clue from Lemma 4, we aim to decompose top-CRMQ problem into a set of simpler queries. Intuitively, we can maintain an interval tree structure with respect to the backward intervals of all interval-pairs and reduce the original problem (which is a five-constraints query) to $O(\log(\frac{n}{B}))$ four-constraints queries. Each of these four-constraints query can be further reduced to $O(\log(\frac{n}{B}))$ three-constraints queries by employing another interval tree structure with respect to the forward intervals on a smaller set of interval-pairs. We elaborate on such an interval-tree-within-an-interval-tree approach below to achieve the result summarized in Lemma 2.

Data Structure: The proposed data structure consists of three components described as follows:

- Backward interval tree: This is an interval tree based on backward intervals of all the interval-pairs in $\mathcal{I}$ as described earlier in the beginning of this section.
- Forward interval trees: The backward interval tree partitions the set $\mathcal{I}$ of interval-pairs into disjoint sets such that each of the set is associated with some node in the interval tree. Let $\mathcal{I}(u_b)$ be such a set associated with node u_b in backward interval tree. We maintain an interval tree at each node u_b based on the forward intervals of all interval-pairs in $\mathcal{I}(u_b)$.
- Dominance structures: Let $\mathcal{I}(u_b, v_f)$ be the set of the interval-pairs associated with node v_f in forward interval tree that is in turn associated with node u_b in backward interval tree. For each possible set $\mathcal{I}(u_b, v_f)$ we maintain data structures for answering the three-dimensional dominance queries [1] listed below.

Q_1 : (1) $prev(i) < a$, (4) $b < next(i)$ and (5) $A[i] \geq \tau$
Q_2 : (2) $a \leq i$, (3) $i \leq b$ and (5) $A[i] \geq \tau$
Q_3 : (2) $a \leq i$, (4) $b < next(i)$ and (5) $A[i] \geq \tau$
Q_4 : (1) $prev(i) < a$, (3) $i \leq b$ and (5) $A[i] \geq \tau$

With each of the above three components occupying linear space total space required for the proposed data structure can be bounded by $O(|\mathcal{I}|)$ words. Space requirement of the backward interval tree is $O(|\mathcal{I}|)$ words (Lemma 3). By the same argument space requirement of a forward interval tree associated with node u_b of backward interval tree is bounded by $O(|\mathcal{I}(u_b)|)$. Thus the total space required for all forward interval trees is $O(|\mathcal{I}|)$ words. Moreover since each interval-pair belongs to exactly one of the $\mathcal{I}(u_b, v_f)$ set, all dominance structures collectively occupy linear space as well.

Query Algorithm: We begin by employing the standard interval tree algorithm (Lemma 3) to identify $O(\log(|\mathcal{I}|/B))$ nodes in the backward interval tree such that any interval-pair that has its backward interval stabbed by a is associated with one of these $O(\log(|\mathcal{I}|/B))$ nodes. We then apply the same algorithm to each of the forward interval tree associated with these $O(\log(|\mathcal{I}|/B))$ nodes to obtain $O(\log(|\mathcal{I}|/B))$ nodes in a single forward interval tree and $O(\log^2(|\mathcal{I}|/B))$ nodes overall such that any interval-pair that has its backward interval stabbed by a and forward interval stabbed by b is associated with one of these $O(\log^2(|\mathcal{I}|/B))$ nodes. We call these nodes candidate nodes and the set of interval-pairs associated with these nodes candidate sets. We now need to further explore only the retrieved candidate sets to get the desired outputs.

For each candidate node v_f belonging to a forward interval tree that in turn is associated with the node u_b in the backward interval tree, let $stab(v_f)$ and $stab(u_b)$ be the end points maintained at nodes v_f and u_b respectively. Then, each interval-pair in $\mathcal{I}(u_b, v_f)$ is stabbed by $stab(u_b)$ and $stab(v_f)$ on its backward and forward interval respectively. By careful examination of the relative values of $a, b, stab(u_b)$ and $stab(v_f)$, we can eliminate two constraints out of five for top-CRMQ and is one of the crucial observations of our paper. We classify node v_f into one the following categories based on which two constraints are satisfied by the interval-pairs in set $\mathcal{I}(u_b, v_f)$:

T_1 : $a \leq stab(u_b) \leq stab(v_f) \leq b$
T_2 : $stab(u_b) \leq a \leq b \leq stab(v_f)$
T_3 : $stab(u_b) \leq a \leq stab(v_f) \leq b$
T_4 : $a \leq stab(u_b) \leq b \leq stab(v_f)$

It can be easily verified that each of these categories lead to the query types Q_1, Q_2, Q_3, and Q_4 respectively on set $\mathcal{I}(u_b, v_f)$ to obtain the interval-pairs satisfying all five constraints required for top-CRMQ problem.

Thus, by first obtaining the candidate nodes and then applying appropriate three-dimensional dominance query on each of them all desired outputs can be retrieved. Using Lemma 3, number of I/Os spent on querying backward interval tree as well as each of the forward interval trees are bounded by $O(\log(|\mathcal{I}|/B))$ I/Os. Therefore all candidate

[††]For any node u, the total number of intervals assigned to nodes in its subtree is $O(size(u))$. This fact follows because (1) all our start and end points are distinct, and (2) for any interval assigned to node u, both its start and end points should be some value associated with one of its descendants.

nodes can be obtained by spending $O(\log^2(|\mathcal{I}|/B))$ I/Os. Moreover, data structure from [1] used for dominance query also requires additional $O(\log_B |\mathcal{I}|)$ I/Os. Therefore total number of I/Os required is $O(\log^2(|\mathcal{I}|/B) \log_B |\mathcal{I}| + \frac{k}{B}) = O(\log^3(|\mathcal{I}|/B) + \frac{k}{B})$. This completes the proof of Lemma 2.

6. BOOTSTRAPPING

The I/O bound in Lemma 2 is optimal for the case $k \geq B\log^3(n/B)$. In the present section, we bootstrap this result to optimally answer "special" top-CRMQ. We start by introducing a blocking scheme that forms the basis of all subsequent external memory results.

Blocking Scheme: Let blocking factor $\delta_j = B(\log^{(j)}(\frac{n}{B}))^5$ and $k_j = B(\log^{(j)}(\frac{n}{B}))^3$ for $j = 1, 2, 3, ..., \log^*(\frac{n}{B})$. Without loss of generality, we further assume that both δ_j and k_j are always rounded to the next highest power of 2 [§]. We partition the array $A[1...n]$ into $\frac{n}{\delta_j}$ disjoint blocks each of size δ_j such that block $A_{j,t} = A[(t-1)\delta_j+1...t\delta_j]$. Define $f_{j,t}$ to denote the left boundary of the block $A_{j,t}$. We say that a block of size δ_j is δ_j-block and a blocking boundary of partitioning based on δ_j (i.e., $f_{j,t}$) is δ_j-boundary. For consistency, fix $\delta_0 = n$ and $A_{0,1} = A[1...n]$. Given a range $[a, b]$, let $A[a^j...b^j]$ be the longest span of δ_j blocks that is completely within $A[a...b]$. Suppose query range $[a, b]$ intersects blocks $A_{j,l}, A_{j,l+1}, ..., A_{j,t}$ then $a^j = f_{j,l+1}$ and $b^j = f_{j,t} - 1$. We prove the following results in the remainder of this section.

LEMMA 5. *A top-CRMQ (a, b, k, τ) can be answered in $O(\frac{k_{\mu+1}}{B} + \frac{k}{B})$ I/Os using an $O(n\log^* n)$-space structure if the span $A[a...b]$ is completely within a δ_μ-block for $\mu \in [0, \log^*(\frac{n}{B})]$.*

LEMMA 6. *A top-CRMQ (a, b, k, τ) can be answered in $O(\frac{k_{\mu+1}}{B} + \frac{k}{B} + \log^* n)$ I/Os using an $O(n)$-space structure if the span $A[a...b]$ is completely within a δ_μ-block for $\mu \in [0, \log^*(\frac{n}{B})]$.*

6.1 Proof of Lemma 5

For each block $A_{j,t}$, we maintain a data structure $IT_{j,t}$ (of size $|IT_{j,t}|$ words) summarized in Lemma 2 [‡]. The total space occupancy is $O(\sum_j \sum_t |IT_{j,t}|) = O(n\log^* n)$ space. Then the δ_μ-block containing span $A[a...b]$ i.e., $A_{\mu,t}$ with $t = \lceil \frac{a}{\delta_\mu} \rceil$ can be queried using structure $IT_{\mu,t}$ to obtain the desired answers in $O(\log^3(\frac{\delta_\mu}{B}) + \frac{k}{B}) = O(\frac{k_{\mu+1}}{B} + \frac{k}{B})$ I/Os.

6.2 Proof of Lemma 6

The space blowup in Lemma 5 comes from the fact that, each interval-pair in $\mathcal{I}$ is repeated $\log^*(\frac{n}{B})$ times as a part of $\log^*(\frac{n}{B})$ number of $IT_{\{\cdot,\cdot\}}$'s. We introduce a categorization technique based on the blocking scheme described earlier that avoids this space blowup, though at the cost of (acceptable) slow-down in query performance. We categorize the input interval-pairs in set $\mathcal{I}$ into $\log^*(\frac{n}{B})+1$ types based on the following rule:

> *An interval-pair $(i, \cdot, \cdot, \cdot, \cdot)$ is categorized as type-j if its both intervals (i.e., backward and forward) are stabbed by a δ_j-boundary, but at least one of them is not stabbed by a δ_{j-1}-boundary.*

[§] In order to ensure δ_{j-1} is always divisible by δ_j

[‡] $IT_{j,t}$ is the structure in Lemma 2 over the following set of interval-pairs $\mathcal{I}_{j,t} = \{(i, \cdot, \cdot, \cdot, \cdot) \in \mathcal{I} | i \in [(t-1)\delta_j + 1, t\delta_j]\}$.

Taking into account the boundary conditions, an interval-pair is termed as type-1 if its both intervals are stabbed by a δ_1-boundary, whereas for an interval-pair of type-$(\log^*(\frac{n}{B})+1)$, none of its intervals is stabbed by any boundary i.e., i and $prev(i)/next(i)$ are within the same $\delta_{\log^*(\frac{n}{B})}$-block (which is of size $\Theta(B)$). Let n_j represent the number of type-j interval-pairs, then $n_1 + n_2 + ... + n_{\log^*(\frac{n}{B})+1} = n$.

We now describe the data structure and query algorithm to achieve the result in Lemma 6. Intuitively, our idea is to make separate linear space data structures for interval-pairs in each type thereby restricting the total space to $O(n)$ words. However, this requires multiple structures to be queried incurring an additive $\log^*(\frac{n}{B})$ term in query I/Os.

Data Structure: We maintain the following substructures.

- For each block $A_{j,t}$ maintain a structure $IT_{j,t}$ summarized in Lemma 2 by considering only type-$(j+1)$ and type-$(j+2)$ interval-pairs. This occupies a total of $O(\sum_j (n_{j+1} + n_{j+2})) = O(n)$ space.
- We create a collection of two-dimensional points by mapping each type-j interval-pair $(i, A[i], prev(i), next(i))$ to a point $(i, A[i])$. Then we apply rank space reduction to these two-dimensional points and maintain a three-sided range reporting structure TS_j by Larsen et al. [17] on this collection. All those type-j interval pairs within $i \in [a, b]$ and $A[i] \geq \tau$ for any given a, b, and τ can be answered in optimal I/Os using TS_j. Further, we associate each two-dimensional point with its corresponding interval-pair, so that the interval-pairs corresponding to the points reported by structure from [17] can be obtained without spending any additional I/Os. Moreover, to be able to query data structure in [17] we need to map the boundary points (a and b) and the threshold τ to rank space. This can be achieved in constant time by maintaining two bit vectors (along with rank-select structure [24]) of length n. Total space required for this component is bounded by $O(n_j)$ words + $O(n)$ bits $= O(n_j + \frac{n}{\log n})$ words. Thus over all space corresponding to $j = 0, 1, 2, ..., \log^*(\frac{n}{B}) + 1$ is $O(n)$ words.
- We also maintain a list A' of all interval pairs $(i, \cdot, \cdot, \cdot, \cdot)$ in the ascending order of i. Space occupancy is $O(n)$ words.

As each of the components described above occupies $O(n)$ words the overall space requirement is linear.

Query Algorithm: As before, let $A_{\mu,t}$ with $t = \lceil \frac{a}{\delta_\mu} \rceil$ be the δ_μ-block containing $A[a...b]$. Then we query $IT_{\mu,t}$ by spending $O(\frac{k_{\mu+1}}{B} + \frac{k}{B})$ I/Os. However, this will give only the outputs of type $(\mu + 1)$ and $(\mu + 2)$. It remains to show how to retrieve the outputs of type-h, for $h \leq \mu$ or $h \geq \mu+3$.

We first demonstrate how type-h outputs with $h \leq \mu$ are retrieved when span $A[a...b]$ is known to be completely within a δ_μ block i.e., $A_{\mu,t}$. We note that any type-h link $(i, \cdot, \cdot, \cdot, \cdot)$ with $h \leq \mu$ and i falling within the block $A_{\mu,t}$ (i.e., $i \in [f_{\mu,t}, f_{\mu,t+1} - 1]$), both its forward as well as backward intervals are stabbed by δ_μ-boundaries ($f_{\mu,t}$ and $f_{\mu,t+1}$ respectively). Therefore, such an interval-pair implicitly satisfies constraints $prev(i) < a$, $b < next(i)$. Hence, for $h \leq \mu$ we only need to take into account the position and weight constraint of the interval-pair (i.e., $i \in [a, b]$ and $A[i] \geq \tau$) and all such type-h outputs can be obtained in optimal I/Os by querying structure TS_h. Therefore, overall I/Os required for retrieving all type-h outputs for $h \leq \mu$ are bounded by $O(\mu + \frac{k}{B}) = O(\log^*(\frac{n}{B}) + \frac{k}{B})$.

Finally all type-h outputs for $h \geq \mu + 3$ can be efficiently retrieved using the following key observation. Any type-h interval-pair $(i, \cdot, \cdot, \cdot, \cdot)$, with $h \geq \mu + 3$ is an output, only if i falls within a $\delta_{\mu+1}$-block that contains either a or b. Otherwise at-least one of two conditions $prev(i) < a$, $b < next(i)$ will be violated. Therefore, the number of candidate interval-pairs in this case is only $2\delta_{\mu+2}$, and the output interval-pairs can be obtained by scanning the two $\delta_{\mu+2}$-blocks in A' to evaluate the five conditions listed in Observation 1 for each of the candidate. The I/Os required in this step are bounded by $O(\frac{\delta_{\mu+2}}{B}) = o(\frac{k_{\mu+1}}{B})$.

Putting together all pieces, the number of I/Os required to answer a top-CRMQ (a, b, k, τ) with $A[a...b]$ completely within a δ_μ-block, can be bounded by $O(\frac{k_{\mu+1}}{B} + \frac{k}{B} + \log^* n)$.

7. THE FINAL DATA STRUCTURES

This section is dedicated to proving Theorem 2 and Theorem 3. Given a top-CRMQ (a, b, k), the structure presented in Lemma 2 can be maintained in $O(n)$-space to optimally handle queries with $k = \Omega(B \log^3(n/B))$. Otherwise, we find the parameter $\pi \in [1, \log^*(n/B)]$, where $k_{\pi+1} < k \leq k_\pi$ (for consistency, assume $k_{\log^*(n/B)+1} = 0$). Then we decompose the original query into following subqueries:

1. top-CRMQ $(a, a^\pi - 1, k, \tau)$
2. top-CRMQ (a^π, b^π, k)
3. top-CRMQ $(b^\pi + 1, b, k, \tau)$

Here $A[a^\pi...b^\pi]$ represents the longest span of δ_π blocks that is completely within $A[a...b]$. Let Out_i represent the set of answers corresponding to the above queries for $i = 1, 2, 3$ (a procedure to obtain them will be described later). Notice that these are disjoint sets and cardinality of each of them is $O(k)$. Moreover, $\cup_{i=1}^3 Out_i$ is a superset of final answers for the original query (a, b, τ). Therefore, those interval-pairs $(i, A[i], C[i], prev(i), next(i)) \in \cup_{i=1}^3 Out_i$ with $prev(i) < a$, $next(i) > b$ and $A[i] \geq \tau$ can be *uniquely* reported as the final answers (the condition $i \in [a, b]$ is satisfied implicitly).

It remains to show, how to retrieve the output set for each of the subqueries efficiently. Both Out_1 and Out_3 can be obtained in $O(k_{\pi+1}/B + k/B) = O(1 + k/B)$ I/Os by maintaining an $O(n \log^* n)$-space structure (refer to Lemma 5). By querying on the structure described in the following lemma, Out_2 also can be obtained in optimal I/Os. This completes the proof of Theorem 2.

LEMMA 7. *There exists an $O(n \log^* n)$-space structure that supports a top-CRMQ (α, β, K) in optimal $O(1+K/B)$ I/Os if $A[\alpha...\beta]$ is a span of several δ_π-blocks and $K \leq k_\pi$ for $\pi \in [0, \log^*(\frac{n}{B})]$.*

Similarly, using the linear space structure in Lemma 6, both Out_1 and Out_3 can be obtained in $O(k_{\pi+1}/B + k/B + \log^* n) = O(\log^* n + k/B)$ I/Os. Combining this with the following lemma for retrieving Out_2, we achieve the result summarized in Theorem 3.

LEMMA 8. *There exists an $O(n)$-space structure that supports a top-CRMQ (α, β, K) in $O(\log^* n + K/B)$ I/Os if $A[\alpha...\beta]$ is a span of several δ_π-blocks and $K \leq k_\pi$ for $\pi \in [0, \log^*(\frac{n}{B})]$.*

The remaining part of this section is dedicated to prove these two lemmas i.e., Lemma 7 and 8.

7.1 Proofs of Lemma 7 and Lemma 8

We identify the parameter θ as the smallest i such that, there exists a δ_i-boundary in $[\alpha, \beta]$. Using θ we decompose top-CRMQ (α, β, K) further into the following subqueries, and obtain the desired answers by merging the outputs of individual subqueries.

- Q_{left}: top-CRMQ $(\alpha, \alpha^\theta - 1, K)$
- O_{middle}: top-CRMQ $(\alpha^\theta, \beta^\theta, K)$
- Q_{right}: top-CRMQ $(\beta^\theta + 1, \beta, K)$

Here $A[\alpha^\theta...\beta^\theta]$ represents the longest span of δ_θ blocks that is completely within $A[\alpha...\beta]$. We now describe the necessary structure for handling each of these queries, followed by the query algorithm.

7.1.1 *Answering Q_{middle}*

Data Structure: Starting from left boundary of each block $A_{j,t}$ i.e., $f_{j,t}$, consider the spans covering $1, 2, 4, 8, ...$ blocks of size δ_j such that it does not cross the first δ_{j-1}-boundary that follows $f_{j,t}$. We maintain the top-k_j answers (i.e., the corresponding pentuples) for each of these spans explicitly (in descending order of weight) i.e., we maintain the list $ML(j, t, i)$ that contains the answers for top-CRMQ with k_j as an input on the span $A[f_{j,t}...f_{j,t+2^i} - 1]$ for any $1 \leq j \leq \log^*(\frac{n}{B})$, $1 \leq t \leq \frac{n}{\delta_j}$ and $i = 0, 1, 2, ..., \log(\frac{\delta_{j-1}}{\delta_j})$. Overall space requirement for such a storage is $O(\sum_j (\frac{n}{\delta_j}) k_j \log(\frac{\delta_{j-1}}{\delta_j}))$ $= O(\sum_j \frac{n}{\log^{(j)}(\frac{n}{B})}) = O(n)$ words.

Query Algorithm: We represent $A[\alpha^\theta...\beta^\theta]$ as union of two overlapping spans each of which covers 2^i δ_θ-blocks for some integer i. Let $[f_{\theta,l'}, f_{\theta,l'+2^i} - 1]$ and $[f_{\theta,t'-2^i}, f_{\theta,t'} - 1]$ be the ranges for these overlapping spans such that $f_{\theta,l'} = \alpha^\theta$ and $f_{\theta,t'} - 1 = \beta^\theta$. It is evident that any top-K answer for $A[\alpha^\theta...\beta^\theta]$ should also be in top-K answers of either of the overlapping spans i.e., it should be present in either $ML(j, l', i)$ or $ML(j, t' - 2^i, i)$. Top-K answers (in sorted order) for these two overlapping spans can be directly retrieved from the maintained precomputed answers in $O(\frac{k}{B})$ I/Os. Further, the two lists can be merged to obtain the outputs for Q_{middle} by a simple scan. However, before merging we discard any answer belonging to the region of overlap between two ranges (i.e., span $A[f_{\theta,t'-2^i}...f_{\theta,l'+2^i} - 1]$) from either of the answer lists to ensure uniqueness of reported answers. In conclusion, Q_{middle} can be answered optimally using an $O(n)$-space structure.

7.1.2 *Answering Q_{left} and Q_{right}*

I/O-Optimal Structure: For each $A_{j,t}$ and $h < j$ we maintain top-k_j answers (in descending order of weight) for the span bounded by $f_{j,t}$ and the first δ_h-boundary that follows $f_{j,t}$. Similarly, top-k_j answers for the span bounded by $f_{j,t+1} - 1$ and the first δ_h-boundary that precedes it are maintained. These answers are maintained in two lists SL_r and SL_l. The list $SL_r(j, t, h)$ and $SL_l(j, t, h)$ contains the answer to top-CRMQ with k_j as an input on the span $[f_{j,t}, f_{h,t'+1} - 1]$ and $[f_{h,t'}, f_{j,t+1} - 1]$ respectively for any $1 \leq j \leq \log^*(\frac{n}{B})$, $1 \leq t \leq \frac{n}{\delta_j}$ and $h < j$ with $t' = \lceil \frac{t}{(\delta_h/\delta_j)} \rceil$. Here t' is the δ_h-block that contains the δ_j-block t. Overall space usage for maintaining these inter-level answers can be bounded by $O(\sum_j \frac{n}{\delta_j} k_j (j-1)) = O(\sum_j \frac{nj}{(\log^{(j)}(\frac{n}{B}))^2}) =$ $O(n \log^* n)$ words.

Desired answers for the top-CRMQ query on desired spans $A[\alpha...\alpha^\theta - 1]$ and $A[\beta^\theta + 1...\beta]$ are simply the first K entries in the appropriate lists $SL_r(\pi, \cdot, \theta)$, $SL_l(\pi, \cdot, \theta)$ respectively and the I/Os needed for retrieving are $O(\frac{K}{B})$. Combing this result along with $O(n)$-space structure capable of answering Q_{middle}, we prove Lemma 7.

Linear Space Structure: To achieve linear space, we do the following modification to the data structure just described: maintain $SL_r(j, \cdot, \cdot)$ and $SL_l(j, \cdot, \cdot)$ only for those $j \leq \phi \leq \log^*(\frac{n}{B})$, where $\log^{(\phi)}(\frac{n}{B}) \geq \log^*(\frac{n}{B}) > \log^{(\phi+1)}(\frac{n}{B})$. Then space can be bounded by $O(\frac{n}{(\log^{(2)}(\frac{n}{B}))^2} + \frac{2n}{(\log^{(3)}(\frac{n}{B}))^2} + \frac{3n}{(\log^{(4)}(\frac{n}{B}))^2} + ... + \frac{(\phi-1)n}{(\log^{(\phi)}(\frac{n}{B}))^2}) = O(\frac{n}{\log^*(\frac{n}{B})})$ words. In addition, we maintain all $SL_r(\phi+1, \cdot, \phi)$ and $SL_l(\phi+1, \cdot, \phi)$ as well occupying $O(\frac{n}{(\log^{(\phi+1)}(\frac{n}{B}))^2}) = o(n)$ words. Further, we also assume the availability of the linear space data structure described in Lemma 6. Thus overall space is bounded by $O(n)$-words. In order to answer a query, we consider the following cases:

1. *If* $\pi \leq \phi$: Obtain answers from the appropriate $SL_r(\pi, \cdot, \theta)$ and $SL_l(\pi, \cdot, \theta)$ in $O(\frac{K}{B})$ I/Os.
2. *If* $\pi = \phi + 1$: Obtain answers from appropriately chosen lists $SL_r(\phi+1, \cdot, \phi)$, $SL_r(\phi, \cdot, \theta)$ and then merge them by spending $O(\frac{K}{B})$ I/Os. Similarly appropriate lists $SL_l(\phi+1, \cdot, \phi)$, $SL_l(\phi, \cdot, \theta)$ can be accessed to obtain the desired results.
3. *If* $\pi > \phi + 1$: We first obtain answers for the span $A[\alpha^{\phi+1}...\alpha^\theta - 1]$ and $A[\beta^\theta + 1...\beta^{\phi+1}]$ from appropriate SL_r and SL_l structures in $O(\frac{K}{B})$ I/Os. Whereas answers for $A[\alpha...\alpha^{\phi+1} - 1]$ (resp., $A[\beta^{\phi+1} + 1...\beta]$) can be obtained in $O(\log^3(\frac{\delta_{\phi+1}}{B}) + \frac{K}{B} + \log^*(\frac{n}{B})) = O(\log^*(\frac{n}{B}) + \frac{K}{B})$ I/Os as it is completely within a block of size $\delta_{\phi+1}$ (from Lemma 6).

Therefore, total number of I/Os required to answer Q_{left} and Q_{right} is bounded by $O(\log^*(\frac{n}{B}) + \frac{K}{B})$, when linear space data structure is used. Result summarized in Lemma 8 can now be obtained by using this structure in addition to $O(n)$-space structure for answering Q_{middle}.

8. CRMQ IN INTERNAL MEMORY

In this section, we show how to modify our external memory data structures to achieve the result in Theorem 1. We again begin with an interval tree based solution and obtain internal memory version of Lemma 2 by simply substituting B by 2. i.e., $O(n)$-word space and $O(\log^3 n + k)$ query time. However, outputs are not sorted. Recall that this result is obtained by querying $O(\log^2 n)$ three-dimensional dominance structures. By using our new three-dimensional dominance structure (Theorem 9) instead of the one by Afshani [1], the outputs from each of those three-dimensional dominance queries can be obtained in sorted order. Further, these outputs can be merged to get a complete list of all answers in sorted order using a heap structure. For our purpose, we use an atomic heap [11] that can perform all heap operations in $O(1)$ in Word-RAM model provided the heap size is $\log^{O(1)} n$. By putting everything together, we obtain an $O(n)$-word space and $O(\log^3 n + k)$ query time data structure for the sorted version of Problem 2.

We now apply blocking scheme with a single blocking factor $\delta_1 = \log^4 n$, and maintain the above described interval-tree based structure over each block $A_{1,t} = A[(t-1)\delta_1 + 1...t\delta_1]$ as $IT_{1,t}$, taking overall $O(n)$ space. Recall that $\delta_0 = n$ and we also maintain $IT_{0,1}$. Further we maintain, the structures $ML(\cdot, \cdot, \cdot)$ as described in Section 7.1.1 occupying $O(n)$ word space i.e., from each δ_1-boundary $f_{1,t}$ consider the spans covering $1, 2, 4, 8, ...$ δ_1-blocks and maintain top-k_1 answers ($k_1 = \log^3 n$) for each of these spans explicitly. Whenever query input $k \geq \log^3 n$, it can be answered optimally using $IT_{0,1}$. For $k < \log^3 n$ and the input range $[a, b]$ completely within a δ_1-block, query can be answered in $O(\log^3 \log n + k)$ time only using appropriate $IT_{1,t}$ structure. Otherwise, we can retrieve top-k answers from fringe spans $A[a...a^1 - 1]$, $A[b^1...b]$ and a middle span $A[a^1...b^1 - 1]$ (refer Section 7.1.1, 7.1.2) and merge them to report final top-k answers with identical query time of $O(\log^3 \log n + k)$. The non-optimal $O(\log^3 \log n)$-additive factor is due to the time for querying the interval tree based structure maintained over each δ_1 block. Therefore, for improving the case where $k < \log^3 \log n$ and the query span $A[a...b]$ is completely within a δ_1 blocks, we maintain the following additional structure. Given a δ_1-block $A_{1,t}$, for every span $A[f_{1,t} + i, f_{1,t} + i + 2^j - 1]$ for $i \in 0, 1, 2, 3, ..., (\delta_1 - 1)$ and $j = 0, 1, 2, ..., \log \delta_1$, maintain top-($\log^3 \log n$) answers (in sorted order). Instead of explicitly maintaining, an output element $A[r]$ (or its location r) for a particular span, we simply encode it as an offset from the left boundary of the span i.e., $r - f_{1,t} + i$ in $O(\log \delta_1) = O(\log \log n)$ bits. Thus overall space requirement can be bounded by $o(n \log n)$ bits. Now any span $A[a...b]$ with both a as well as b in the same δ_1-block can be partitioned into two overlapping spans $A[a...y]$ and $A[x...b]$ where $a < x \leq y < b$, such that the top-k answers of these overlapping spans are precomputed and can be retrieved in optimal time. Finally, by merging these answers, we obtain the final output.

9. CONCLUSIONS

In this paper we introduced the problem of colored (categorical) range maxima that generalizes the fundamental problem of computing maxima in a query range to the colored scenario. We show that this problem is related to or generalizes other important problems, such as reporting most relevant documents containing a given string and three-sided categorical range reporting. We provide an optimal solution of the colored range maxima problem in internal memory. Our external memory data structure uses $O(n)$ space and answers queries in $O(\log^* n + k/B)$ I/Os. Design of a linear space data structure with constant query cost or proving a lower bound for this problem remains an interesting open question.

10. REFERENCES

[1] P. Afshani. On dominance reporting in 3d. In *ESA*, pages 41–51, 2008.

[2] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1998.

[3] L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *PODS*, pages 346–357, 1999.

[4] M. A. Bender, M. Farach-Colton, G. Pemmasani, S. Skiena, and P. Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *J. Algorithms*, 57(2):75–94, 2005.

[5] O. Berkman and U. Vishkin. Recursive star-tree parallel data structure. *SICOMP*, 22(2):221–242, 1993.
[6] P. Bozanis, N. Kitsios, C. Makris, and A. K. Tsakalidis. New upper bounds for generalized intersection searching problems. In *ICALP*, pages 464–474, 1995.
[7] G. S. Brodal, R. Fagerberg, M. Greve, and A. López-Ortiz. Online sorted range reporting. In *ISAAC*, pages 173–182, 2009.
[8] B. Chazelle and H. Edelsbrunner. Linear space data structures for two types of range search. *DCG*, 2:113–126, 1987.
[9] P. Ferragina and R. Grossi. The String B-tree: A new data structure for string searching in external memory and its application. *JACM*, 46(2):236–280, 1999.
[10] J. Fischer and V. Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *ESCAPE*, pages 459–470, 2007.
[11] M. L. Fredman and D. E. Willard. Trans-dichotomous algorithms for minimum spanning trees and shortest paths. *J. Comput. Syst. Sci.*, 48(3):533–551, 1994.
[12] P. Gupta, R. Janardan, and M. H. M. Smid. Further results on generalized intersection searching problems: counting, reporting, and dynamization. *J. Algorithms*, 19(2):282–317, 1995.
[13] W.-K. Hon, R. Shah, and J. S. Vitter. Space-efficient framework for top-k string retrieval problems. In *FOCS*, pages 713–722, 2009.
[14] R. Janardan and M. A. Lopez. Generalized intersection searching problems. *IJCGA*, 3(1):39–69, 1993.
[15] H. Kaplan, N. Rubin, M. Sharir, and E. Verbin. Efficient colored orthogonal range counting. *SICOMP*, 38(3):982–1011, 2008.
[16] M. Karpinski and Y. Nekrich. Top-k color queries for document retrieval. In *SODA*, pages 401–411, 2011.
[17] K. G. Larsen and R. Pagh. I/O-efficient data structures for colored range and prefix reporting. In *SODA*, pages 583–592, 2012.
[18] K. G. Larsen and F. van Walderveen. Near-optimal range reporting structures for categorical data. In *SODA*, pages 256–276, 2013.
[19] C. Makris and A. K. Tsakalidis. Algorithms for three-dimensional dominance searching in linear space. *IPL*, 66(6):277–283, 1998.
[20] S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *SODA*, pages 657–666, 2002.
[21] G. Navarro. Spaces, trees and colors: The algorithmic landscape of document retrieval on sequences. In *CoRR abs/304.6023*, 2013.
[22] G. Navarro and Y. Nekrich. Top-k document retrieval in optimal time and linear space. In *SODA*, 2012.
[23] Y. Nekrich. Space-efficient range reporting for categorical data. In *PODS*, pages 113–120, 2012.
[24] R. Raman, V. Raman, and S. S. Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *TALG*, 2007.
[25] R. Shah, C. Sheng, S. V. Thankachan, and J. S. Vitter. Top-k document retrieval in external memory. In *ESA*, 2013.
[26] J. F. Sibeyn. External selection. In *STACS*, pages 291–301, 1999.
[27] J. S. Vitter. Algorithms and data structures for external memory. *Foundations and Trends in Theoretical Computer Science*, 2(4):305–474, 2008.
[28] P. Weiner. Linear pattern matching algorithms. In *SWAT*, pages 1–11, 1973.

APPENDIX

A. TOP TO THRESHOLD MAPPING

Data Structure: We partition the array $A[1...n]$ into $\lceil \frac{n}{\log^2 n} \rceil$ disjoint blocks each of size $\log^2 n$ (possibly except for the rightmost block). Starting from each blocking boundary, we consider spans (of length at most n) covering $1, 2, 4, 8, ...$ blocks, and for each such span $S = A[x, y]$, we maintain $\tau^k_{x,y}$ for $k = 1, 2, 4, 8, ..., n$. Here $\tau^k_{x,y} \in \{A[j] | j \in [x, y]\}$ with k as the output size of the threshold-CRMQ $(x, y, \tau^k_{a,b})$. This takes $O(n)$ space. Further, we divide each block into sub-blocks of size $\log^2 \log n$, and starting from each sub-block boundary, we consider spans (of length at most $\log^2 n$) covering $1, 2, 4, 8, ...$ sub-blocks. Again, for each such span $S' = [x', y']$, we maintain $\tau^k_{x',y'}$ for $k = 1, 2, 4, 8, ..., \Theta(\frac{\log^2 n}{\log^2 \log n})$. Notice that the explicit storage of $\tau^k_{x',y'}$'s (in $\log n$ bits per element) is costly. Therefore, we simply encode its relative position within that span in lesser number of $O(\log(\log^2 n)) = O(\log \log n)$ bits. i.e., total $O(n)$-space. Finally answers for the query (a, b, k) where both a, b are completely within a sub-block can be maintained in $o(n)$ bits using tables.

Query Answering: In order to compute the threshold $\tau^k_{a,b}$ corresponding to the input (a, b, k), we get k' by approximating k to the next highest power of 2 i.e., $k' = 2^{\lceil \log k \rceil}$. Then the input range $[a, b]$ can be partitioned into (at most) 6 spans $[a, a' - 1], [a', a'' - 1], [a'', b''], [b'' + 1, b'], [b' + 1, b]$ such that (1) both $[a, a' - 1]$, $[b' + 1, b]$ are within a sub-block, (2) $[a', a'' - 1]$, $[b'' + 1, b']$ are covered by spans of sub-blocks and (3) $[a'', b'']$ is covered by two possibly overlapping spans of blocks. The $\tau^{k'}_{\{\cdot,\cdot\}}$ for each of these spans can be retrieved in constant time and we choose the maximum among them as our threshold $\tau^k_{a,b}$. It can be easily verified that $\hat{k} \leq 6k' < 12k$ and $\hat{k} \geq min(k, dcol)$, where $dcol$ denotes the number of distinct colors in $C[a...b]$.

B. TOP-*k* DOCUMENT RETRIEVAL

In this problem, we are given a set of D string documents $\{d_1, d_2, ..., d_D\}$ of total length n. We need to index these documents so as to answer the query (P, k) that requires us to output k documents with the highest $w(P, d_j)$. The relevance $w(P, d_j)$ depends only on the set of occurrences of P in d_j i.e., $Occ(P, d_j)$ and the document itself. Whenever a relevance measure satisfies the monotonicity property (either $w(P, d_j)$ is always $\leq w(P', d_j)$ or it is always $\geq w(P', d_j)$, where P' is a prefix of P), top-k string retrieval problem can be reduced to top-CRMQ

First, construct a generalized suffix tree [28] of the document collection. Then we mark nodes with document-ids as follows: a leaf node ℓ is marked with document d_j if the suffix represented by ℓ belongs to d_j. An internal node u is marked with d_j if it is the lowest common ancestor of two leaves marked with d_j. Notice that a node can be marked with multiple documents. For each node u (with pre-order rank $rank(u)$) and each of its marked documents d_j, we define a triplet $(rank(u), w(path(u), d_j), d_j)$, where $path(u)$ represents the concatenation of edge labels on the path from root to u. Let (x_i, y_i, d_{c_i}) represents the i-th triplet, where $x_i \leq x_{i+1}$, then we construct A and C as follows: $A[i] = y_i$ and $C[i] = c_i \in [1, D]$. The top-k documents corresponding to the query (P, k) are same as the output colors for top-CRMQ (a, b, k), where $[a, b]$ represents the maximal range such that for all triplets $(x_i, \cdot, \cdot)$ with $i \in [a, b]$, the node with pre-order rank x_i is in the subtree of u_P. Here u_P represents the locus of P, the node closest to root with P as a prefix of $path(u_P)$. Using a String B-tree [9] and some auxiliary structures occupying $O(n)$-word space over all, we can compute u_P in $O(\log_B n + \frac{|P|}{B})$ I/Os.

C. PROOF OF THEOREM 9

Our data structure is based on the same approach as in [8, 19]. But we will also need additional ideas to output points in sorted order.

We associate sets of points $P(v)$ with nodes v of a binary tree T. Let $\max_{xy}(S)$ denote those points of a set S whose projections on the xy-plane are maximal. We set $S(w_r) = S$ for the root w_r of T. In every node v starting with the root, we store set $P(v) = \max_{xy} S(v)$. Then, we divide all points from $S(v) \setminus P(v)$ into two equal parts according to their z-coordinates and associate them with children v_l, v_r of v. In other words, points from $S(v) \setminus P(v)$ are distributed among $S(v_l)$ and $S(v_r)$ so that (1) $p_l.z < p_r.z$ for any $p_l \in S(v_l)$ and $p_r \in S(v_r)$, (2) $|S(v_r)| \leq |S(v_l)| \leq |S(v_r)| + 1$. Finally, we recursively apply the same procedure to $S(v_l)$ and $S(v_r)$.

For every node v, we keep all points of $P(v)$ sorted by their x-coordinates in an array $A(v)$. We also maintain a data structure from [7] that supports sorted reporting queries on $A(v)$: for any query interval $[a, b]$, $D(v)$ reports all points $p \in A[i]$, such that $a \leq i \leq b$ and $p.z \geq c$, sorted in decreasing order of their z-coordinates. As described in [7], $D(v)$ uses $O(|P(v)|)$ space and answers queries in $O(k+1)$ time, where k is the number of reported points. We also store structures $D_x(v)$ and $D_y(v)$ that enable us to answer predecessor and successor queries on x- and y-coordinates of points in $P(v)$.

Using $D(v)$, $D_x(v)$, and $D_y(v)$, we can answer a sorted dominance query $Q = [a, +\infty] \times [b, +\infty] \times [c, +\infty]$ on $P(v)$. Since $P(v)$ contains maximal points with respect to their x- and y-coordinates, all p_1, $p_2 \in P(v)$ have the following property: if $p_1.x > p_2.x$, then $p_1.y < p_2.y$. That is, y-coordinates of points in $P(v)$ decrease monotonously with increasing x-coordinates. Let p_l be the point in $P(v)$ with the smallest x-coordinate, such that $p_l.x \geq a$; let p_r be the point in $P((v)$ with the smallest y-coordinate, such that $p_r.y \geq b$. Let i_l and i_r denote the x-ranks[‡‡] of p_l and p_r respectively. All points p stored in $A[i_l ... i_r]$ and only those points satisfy $p.x \geq a$ and $p.y \geq b$. Hence, we can answer a query Q on $P(v)$ by reporting all points in $A[i_l ... i_r]$ in decreasing order of their z-coordinates until all points p, $p.z \geq c$, are output.

The same sorted dominance query on S is answered as follows. Let Π_q denote the search path for c in T. We report all points $p \in P(v)$ for all nodes $v \in \Pi_q$. For every node u that is a right sibling of $v \in \Pi_q$, we must report relevant points stored in u and its descendants. First, we answer the dominance queries on $P(u)$; if at least one point was reported, we visit both children of u and recursively process both children of u. Let $L(u)$ denote the list of points in $P(u) \cap Q$ sorted by their z-coordinates. The union of $L(u)$ for all visited nodes u contains all points in $S \cap Q$: all points p, $p.z \geq c$, are stored in nodes $v \in \Pi_q$ or in right siblings of nodes $v \in \Pi_q$ and their descendants. Our procedure visits all nodes $v \in \Pi_q$ and their right siblings; our procedure also visits all descendants of the right siblings that contain at least one point $p \in Q$, as can be concluded from the following observation.

Observation 1. *Suppose that u is the right sibling of some node $v \in \Pi_q$ or a descendant of the right sibling of some $v \in \Pi_q$. If $P(u) \cap Q = \emptyset$, then $P(w) \cap Q = \emptyset$ for all descendants w of u.*

Every list $L(u)$ is generated in $O(|L(u)| + 1)$ time: using fractional cascading, we can find indices i_l and i_r in any visited node u in constant time. When i_l and i_r are known, data structure $D(u)$ reports all points $p \in A(u)$, $p.z \geq c$ in $O(|L(u)| + 1)$ time. The total number of nodes u for which lists $L(u)$ were generated is bounded by $O(\log n + k)$. Hence, the total time needed to generate all lists $L(u)$ is $O(\log n + k)$. It remains to show how to merge all $L(u)$ so that the output is sorted by z-coordinates. We will say that a node u is situated to the right of a node v if u and v are stored in respectively the right and the left subtrees of their lowest common ancestor.

Observation 2. *If $p_u.z > p_w.z$ for some $p_u \in P(u)$ and $p_w \in P(w)$, then u is an ancestor of w or u is situated to the right of w in T.*

Let V denote the set of all visited nodes. Since the height of T is $O(\log n)$, we can use sweepline approach for sorting points in the query range: we maintain the *current path* Π_c, and report points stored in $P(u)$, $u \in \Pi_c$, in sorted order. Suppose that we work with the current path Π_c at some time. Then this means that all nodes $u \in V$ to the right of Π_c were already processed and points from lists $L(u)$ are already in sorted order. To initialize the path Π_c, we start at the root and move down the tree until a leaf is reached or the currently visited node u has no child $u_i \in V$. In every visited node u, we move to its right child u_r if $u_r \in V$; otherwise, we move to its left child u_l. Thus Π_c is initialized to the rightmost path that consists of nodes $u \in V$.

We extract the first point (i.e., the point with the highest z-coordinate) from every $L(u)$, $u \in \Pi_c$, and insert them into a priority queue Q. The following steps are repeated until all points in all $L(u)$, $u \in V$, are sorted. We extract the highest point p from Q and add it to the sorted list of points. If the list $L(u)$, such that $p \in L(u)$, is not empty, we extract the next point p' from $L(u)$ and add it to Q. When some list $L(w)$, $w \in \Pi_c$, becomes empty, we might need to update the path Π_c. If $L(w)$ is empty and w is the lowest node in Π_c, we remove w from Π_c. If w is the right child of its parent and its left sibling v is in V, we also append new nodes to Π_c. This is done by traversing a downward path that starts in v. In every visited node u, starting with v, we add u to Π_c and move down the tree if at least one child of u is in V; if both children of u are in V, we always select the right child. For every new node u in Π_c, we extract the highest point $p \in L(u)$ and add it to Q. Otherwise, if w has

[‡‡]The x-rank of a point p in a set P is the number of points $p' \in P$ such that $p'.x \leq p.x$.

no left sibling or the left sibling of w is not in Π_c, then we move up in the tree and consecutively examine all ancestors w' of w starting with the parent. If $L(w')$ for an ancestor w' of w is empty, we remove w' from Π_c. If w' has a left sibling $w'' \in V$, we append the rightmost path starting at w'' to Π_c as described above. Otherwise, we examine the ancestors of w' until a node u, $L(u) \neq \emptyset$, is reached. When Π_c and Q are empty, we have generated the sorted list of all points in $S \cap Q$. Correctness of our procedure follows from Observation 2. Suppose that a point $p_1 \in L(u_1)$ was reported before $p_2 \in L(u_2)$, then either (1) u_1 is to the right of u_2, or (2) u_1 is an ancestor of u_2, or (3) u_2 is ancestor of u_1. In the case (1) $p_1.z \leq p_2.z$ by Observation 2. In the case (2) u_1 is an ancestor of u_2. If p_1 was reported before u_2 was inserted into Π_c, then $p_1.z \geq p_3.z$ for some $p_3 \in L(u_3)$, where u_3 is to the right of u_2. Hence, $p_1.z \geq p_3.z \geq p_2.z$. If p_1 was reported after u_2 had been included into Π_c, then it follows from the description that $p_1.z \geq p_2.z$. Case (3) is identical with the second part of case (2).

We implement Q using the atomic heap data structure [11]; Since Q contains $O(\log n)$ elements, all operations on Q can be supported in $O(1)$ time. By keeping the depths of all non-empty nodes $u \in \Pi_c$ in another atomic heap, we can determine whether there are non-empty nodes $u' \in \Pi_c$ below a given node u in $O(1)$ time. Thus we can sort all points $p \in L(v)$, $v \in V$, by their z-coordinates in $O(|V| + \sum_{v \in V} |L(v)|)$ time. This completes the proof of Theorem 9.

D. PROOF OF THEOREM 4

To achieve the result in Theorem 4 we once again rely on the blocking scheme and interval-pair categorization introduced in the Section 6. We begin by partitioning the input range $[a, b]$ into disjoint spans as described in Section 7 and investigate each of them independently: $A[a^\theta ... b^\theta]$ as middle span, $A[a^\pi ... a^\theta - 1]$ and $A[b^\theta + 1 ... b^\pi]$ as side spans and $A[a ... a^\pi - 1]$, $A[b^\pi + 1 ... b]$ as fringe spans. Recall that $k_{\pi+1} < k \leq k_\pi$, both a and b are within a single $\delta_{\theta-1}$-block but belong to two distinct δ_θ-blocks and $\theta \leq \pi$. With structure described in Section 7.1.1 capable of querying the middle span within the desired space-time complexity, we focus on fringe and side spans below. We the following notation in this section: ${\log^*}^0(\cdot) = \log(\cdot)$, ${\log^*}^h(\cdot)$ is the number of times function ${\log^*}^{h-1}(\cdot)$ must be iteratively applied before the result is less than or equal to 2 for $h \geq 1$ and $\alpha(\cdot)$ is the minimum h, where ${\log^*}^h(\cdot) \leq 2$. We use α to denote $\alpha(n)$ for simplicity i.e., α is the Inverse Ackermann function of n.

D.1 Handling Fringe Spans

A close look at the Lemma 6 reveals that the additive $\log^*(\frac{n}{B})$ factor in query I/Os is due to the necessity to access as many three-sided range reporting structures. Recall that each such structure TS_j was built by only considering the type-j interval-pairs. Intuitively, additional three-sided range reporting structures $TS_{i,j}$ can be maintained over a collection of type-m interval-pairs for $m = i, i+1, ..., j$ trading off space for better query performance.

To formalize this intuition we begin by proving following lemma that summarizes the way to group the interval-pairs of different types so as to build a collective three-sided range reporting structure over them. We extend the notation used for blocking factor δ_j as below for the purpose of this subsection: $\delta_j(n) = B(\log^{(j)}(\frac{n}{B}))^5$ and we use δ_j for $\delta_j(n)$ simplicity. By choosing $h = \alpha$ in the lemma, we can obtain a set $U(n, \alpha)$ such that each element of $S(n, \alpha) = \{\delta_1, \delta_2, ..., \delta_{\log^*(\frac{n}{B})}\}$ belongs to at most α sets in $U(n, \alpha)$. We now simply maintain a collective three-sided structure for each element U_e in $U(n, \alpha)$ considering all type-j interval-pairs such that $\delta_j \in U_e$. The overall space requirement of such storage can be bounded by $O(n\alpha)$. Moreover the total number of three-sided structures we need to access in query algorithm of Lemma 6 is now bounded by $2\alpha^2 \log(\frac{\delta_\pi}{B}) = \Theta(\alpha^2 (\frac{k_{\pi+1}}{B})^{1/3}) = O(\alpha^2 (\frac{k}{B})^{1/3})$. Thus the query I/Os can be bounded by $O(\alpha^2 (\frac{k}{B})^{1/3} + \frac{k}{B})$.

LEMMA 9. *Given a set $S(n, h) = \{\delta_1, \delta_2, ..., \delta_x\}$ such that $\log^{(x)}(\frac{n}{B}) \geq {\log^*}^h(\frac{n}{B}) > \log^{(x+1)}(\frac{n}{B})$, we can obtain a set $U(n, h)$ with each of its element being a subset of $S(n, h)$ satisfying following conditions:*

- *any element of $S(n, h)$ belongs to at the most $h+1$ sets in $U(n, h)$*
- *set $Q_x = \{\delta_1, \delta_2, ..., \delta_x\}$ can be expressed as a union of h sets in $U(n, h)$*
- *any set $Q_\mu = \{\delta_1, \delta_2, ..., \delta_\mu\}$ with $\mu < x$ can be expressed as a union of $P(n, h, \mu) \leq 2h^2 \log(\frac{\delta_\mu(n)}{B})$ sets in $U(n, h)$*

PROOF. For the base case with $h = 1$ we have $\delta_x \geq B(\log^* \frac{n}{B}) > \delta_{x+1}$. Then construct $U(n, 1) = \{\{\delta_1\}, \{\delta_2\}, ..., \{\delta_{x-1}\}, \{\delta_1, \delta_2, ..., \delta_x\}\}$. Clearly the first and second statements in the Lemma are true. For any $\mu \leq x - 1$, the set Q_x can be expressed as a union of μ sets in $U(n, h)$, where μ can be upper bounded by $\log(\frac{\delta_\mu(n)}{B}) \leq P(1)$ in this case. This is because $\mu \leq \log^*(\frac{n}{B}) < \frac{\delta_x}{B} \leq \log(\frac{\delta_\mu(n)}{B})$ for any $\mu \leq x - 1$. We now assume that the desired $U(n, h)$ can be obtained for $h = 1, 2, ..., m$ and show how $U(n, m+1)$ can be obtained for $S(n, m+1)$.

We use an important property expressed by equality $\delta_{j+1} = \delta_1(\delta_j)$ to obtain $U(n, m+1)$. Let $\log^{(\phi_1)}(\frac{n}{B}) \geq {\log^*}^m(\frac{n}{B}) > \log^{(\phi_1+1)}(\frac{n}{B})$ then $S(n, m) = \{\delta_1, \delta_2, ..., \delta_{\phi_1}\}$. Further we define $\phi_2, \phi_3, ..., \phi_r$ with $\log^{(\phi_j)}(\frac{n}{B}) \geq {\log^*}^m {\log^*}^m$ *...j times* $...(\frac{n}{B}) > \log^{(\phi_j+1)}(\frac{n}{B})$ for $j = 1, 2, 3, ..., r$, and $\log^{(\phi_r)}(\frac{n}{B}) \geq \log^{(x')}(\frac{n}{B}) \geq {\log^*}^{m+1}(\frac{n}{B}) > \log^{(x'+1)}(\frac{n}{B}) > \log^{(\phi_r+1)}(\frac{n}{B})$. Then $S(n, m+1)$ can be written as follows:

$$
\begin{aligned}
& S(n, m+1) = \{\delta_1, \delta_2, ..., \delta_{x'}\} \\
& = \{\delta_1, ..., \delta_{\phi_1}\} \cup \{\delta_{\phi_1+1}, ..., \delta_{\phi_2}\} \cup \{\delta_{\phi_2+1}, ..., \delta_{\phi_3}\} \\
& \cup ... \cup \{\delta_{\phi_r+1}, ..., \delta_{x'}\} \\
& = \{\delta_1(n), \delta_2(n), ...\} \cup \{\delta_1(\delta_{\phi_1}), \delta_2(\delta_{\phi_1}), ...\} \\
& \cup \{\delta_1(\delta_{\phi_2}), \delta_2(\delta_{\phi_2}), ...\} \cup ... \cup \{\delta_1(\delta_{\phi_r}), \delta_2(\delta_{\phi_r}), ...\} \\
& = S(n, m) \cup S(\delta_{\phi_1}, m) \cup S(\delta_{\phi_2}, m) \cup ... \cup S'(\delta_{\phi_r}, m)
\end{aligned}
$$

Note that the last set $S'(\delta_{\phi_r}, m) = \{\delta \geq B {\log^*}^{m+1}(n/B) | \delta \in S(\delta_{\phi_r}, m)\}$. After constructing $U(\cdot, m)$ for each of the $S(\cdot, m)$ in the above equation (using our recursive method for $h = m$ case), we obtain $U(n, m+1) = U(n, m) \cup U(\delta_{\phi_1}, m) \cup U(\delta_{\phi_2}, m) \cup ... \cup U'(\delta_{\phi_r}, m) \cup \{\delta_{\phi_1}, ..., \delta_{x'}\}$.

It can be easily verified that each element in $S(n, m+1)$ belongs to at the most $m+2$ sets in $U(n, m+1)$ thus proving the first statement in Lemma. The second statement also verifiable since $Q_{x'}$ can be expressed as a union of $S(n, m)$ and $\{\delta_{\phi_1}, ..., \delta_{x'}\} \in U(n, m+1)$, where $S(n, m)$ can in-turn be expressed as a union of $m+1$ sets in $U(n, m) \in U(n, m+1)$. Finally the remaining case, where $\mu < x'$ can be proved as follows: let $\phi_j + 1 \leq \mu \leq \phi_{j+1}$, the set Q_μ

can be expressed as $S(n,m) \cup S(\delta_{\phi_1},m) \cup S(\delta_{\phi_2},m) \cup ... \cup S(\delta_{\phi_j},m) \cup \{\delta_{\phi_j+1},...,\delta_\mu\}$. As each $S(.,m)$ can be expressed as the union of m sets in $U(n,m+1)$, $S(n,m) \cup S(\delta_{\phi_1},m) \cup S(\delta_{\phi_2},m) \cup ... \cup S(\delta_{\phi_j},m)$ can be expressed as a union of $(j+1)m$ sets in $U(n,m+1)$. Moreover $j \le \log^{*^{m+1}}(\frac{n}{B}) \le \log(\frac{\delta_\mu(n)}{B})$ for all $\mu < x'$, therefore $(j+1)m \le (\log(\frac{\delta_\mu(n)}{B})+1)m$. The remaining elements in Q_μ, i.e., $\{\delta_{\phi_j+1},...,\delta_\mu\}$ can be represented as the union of $P(\delta_{\phi_j},m,\mu-\phi_j) \le P(n,m,\mu)$ sets in $U(\delta_{\phi_j},m)$. By putting every thing together, $P(n,m+1,\mu) \le P(n,m,\mu) + m\log(\frac{\delta_\mu(n)}{B}) + m \le 2(m+1)^2\log(\frac{\delta_\mu}{B})$. This completes the proof. $\square$

D.2 Handling Side Spans

We demonstrate the proposed data structures for the span $A[a^\pi...a^\theta-1]$ below and note that the span $A[b^\theta+1...b^\pi]$ can handled in a symmetric way. We take a different approach for handling the side spans and instead of obtaining the top-k answers for the query $(a^\pi, a^\theta-1, k)$ as before, we instead choose to retrieve only those outputs from set $Output(a,b,k)$ of size k which belong to the span $A[a^\pi...a^\theta-1]$. This can be achieved by using the index summarized in following lemma. By choosing $h=\alpha$, we obtain linear space index with $O(\alpha\log_B k + \frac{k}{B})$ query I/Os.

LEMMA 10. *There exists an $S(n,h) = O(n/\log^{*^h}(n/B))$ space data structure for answering the following query in $T(n,h)+O(\frac{k'}{B})$ I/Os where $T(n,h) = T(n,h-1)+O(\log_B k)$: Given a top-$k$ categorical maxima query (a,b,k) retrieve the $k' \le k$ outputs in the set $Output(a,b,k)$ which belong to the span $A[a^{dn}...a^{up}-1]$ such that $a \le a^{dn} \le a^{up}-1 \le b$, $\log^{*^h}(\frac{n}{B}) \le \log^{(dn)}(\frac{n}{B}) < \log^{(up)}(\frac{n}{B})$ and $k \le k_{dn} < k_{up}$.*

PROOF. Before moving to the proof recall that, if query range $[a,b]$ intersects blocks $A_{j,l}, A_{j,l+1}, ..., A_{j,t}$ then $a^j = f_{j,l+1}$. We now prove the above result using induction. The base case for $h=1$ can be proved as follows: for each $A_{j,t}$ with $j \le \phi \le \log^*(\frac{n}{B})$, $\log^{(\phi)}(\frac{n}{B}) \ge \log^*(\frac{n}{B}) > \log^{(\phi+1)}(\frac{n}{B})$ and $i < j$ we maintain top-k_j answers for the span bounded by $f_{j,t}$ and the first δ_i-boundary that follows $f_{j,t}$. Instead of maintaining these k_j answers as a single list $SL(j,t,i)$ as before, we view it as a collection of multiple lists $SL(j,t,i,\overline{k_j})$ storing top-$\overline{k_j}$ answers for $\overline{k_j} = 1,2,4,...k_j$ and maintain a three-dimensional dominance structure for each of these lists over the $prev(.)$, $next(.)$ and $weight(.)$ fields. Thus $S(n,1)$ can be bounded by $O(\frac{n}{(\log^{(2)}(\frac{n}{B}))^2} + \frac{2n}{(\log^{(3)}(\frac{n}{B}))^2} + \frac{3n}{(\log^{(4)}(\frac{n}{B}))^2} + ... + \frac{(\phi-1)n}{(\log^{(\phi)}(\frac{n}{B}))^2}) = O(\frac{n}{\log^*(\frac{n}{B})})$ words. In order to answer a query, we use the dominance structure corresponding to the list $SL(dn,t,up,\overline{k})$ with $t = \lceil\frac{a^{dn}}{\delta_{dn}}\rceil$ and $k \le \overline{k} < 2k$. Note that an answer from any list $SL(dn,t,up,.)$ belongs to the span $A[a^{dn}...a^{up}-1]$. Then such an answer only needs to satisfy the conditions $prev(.) < a$, $b < next(.)$ and $A[.] \ge \tau^k_{a,b}$ to be reported as an output for the query (a,b,k). Total query I/Os needed can bounded by $O(\log_B \overline{k} + \frac{k'}{B})$ and $T(n,1) = O(\log_B k)$.

Next we prove the result for $h = m+1$ assuming the claim to be true for all previous cases (i.e., $h = 1,2,...,m$). Let r be such that $\log^{*^m}\log^{*^m}$ ... r *times* ...$(\frac{n}{B}) \ge \log^{*^{m+1}}(\frac{n}{B}) > \log^{*^m}\log^{*^m}$ ... $(r+1)$ *times* ...$(\frac{n}{B})$. Then for $j = 1,2,3,...,r-1$, define ϕ_j as follows: $\log^{(\phi_j)}(\frac{n}{B}) \ge \log^{*^m}\log^{*^m}$ ... j *times* ...$(\frac{n}{B}) > \log^{(\phi_j+1)}(\frac{n}{B})$. Where as ϕ_r is defined as the largest integer (say g) with $\log^{*^m}(\log^{(\phi_r)}(\frac{n}{B})) \ge \log^{*^{m+1}}(\frac{n}{B})$. Note that for $j = 1,2,3,...,r-2$, $\log^{*^m}(\log^{(\phi_j)}(\frac{n}{B})) = \log^{(\phi_{j+1})}(\frac{n}{B})$. We consider the following two cases:

Case 1. $\delta_{\phi_x+1} < \delta_{dn} < \delta_{up} \le \delta_{\phi_x}$: To answer a query in this scenario, we simply maintain the structure $S(.,m)$ for each $A_{\phi_j,t}$ occupying overall space of $O(\frac{n}{\log^{(\phi_1)}(\frac{n}{B})} + \frac{n}{\log^{(\phi_2)}(\frac{n}{B})} + ... + \frac{n}{\log^{*^m}\log^{(\phi_r)}(\frac{n}{B})}) = O(\frac{n}{\log^{*^{m+1}}(\frac{n}{B})})$ words. Since both a^{dn} and a^{up} belong to the same δ_x-block in this case, query can be answered using the structure maintained over the points in that δ_{ϕ_x}-block in $T(\delta_x,m) \le T(n,m)$ time.

Case 2. $\delta_{\phi_x+1} < \delta_{dn} \le \delta_{\phi_x} \le \delta_{y+1} < \delta_{up} \le \delta_y$: We maintain two components described below.

- For each $A_{\phi_j,t}$ and $i < j$ for $j = 1,2,3,..,r$, we maintain top-k_{ϕ_j} answers for the span bounded by $f_{\phi_j,t}$ and the first δ_{ϕ_i}-boundary that follows it. Again these k_{ϕ_j} answers are maintained as a collection of three-dimensional dominance structures over the multiple lists $SL(\phi_j,t,\phi_i,\overline{k_{\phi_j}})$ for $\overline{k_{\phi_j}} = 1,2,4,...,k_{\phi_j}$. The overall space (in words) of this component can be bounded as follows: $O(\frac{n}{\log^{(\phi_2)}(\frac{n}{B})} + \frac{2n}{\log^{(\phi_3)}(\frac{n}{B})} + ... + \frac{(r-1)n}{\log^{(\phi_r)}(\frac{n}{B})}) = O(\frac{n}{\log^{*^{m+1}}(\frac{n}{B})})$. Here note that $r \le \log^{*^{m+1}}(\frac{n}{B})$ and $\log^{(\phi_r)}(\frac{n}{B}) = \Omega((\log^{*^{m+1}}(\frac{n}{B}))^2)$.
- Consider a blocking factor δ_z in our blocking scheme such that $\delta_{\phi_{j+1}} < \delta_z < \delta_{\phi_j}$. Then for each δ_z-boundary i.e., $f_{z,t}$ we maintain top-k_z answers for the span bounded by $f_{z,t}$ and the first δ_{ϕ_j}-boundary that follows it. Once again it is in the form of three-dimensional dominance structures over the lists $sl(z,t,\phi_j,\overline{k_z})$ for $\overline{k_z} = 1,2,4,...,k_z$ occupying $O(\frac{n}{(\log^{(z)}(\frac{n}{B}))^2})$ space. Such pre-computed answers are stored for each δ_z, where $\log^{(z)}(\frac{n}{B}) \ge \log^{*^{m+1}}(\frac{n}{B})$. Therefore total space can be bounded by $o(\frac{n}{\log^{*^{m+1}}(\frac{n}{B})})$.

In order to answer the query, we partition that span $A[a^{dn}...a^{up}-1]$ into three disjoint spans and answer each of them separately as discussed below.

- The first span is bounded by a^{dn} and the first δ_{ϕ_x}-boundary that follows it. By slight abuse of notation let such a δ_{ϕ_x}-boundary be denoted by a^{ϕ_x}. Recall that $\delta_{\phi_x+1} < \delta_{dn} \le \delta_{\phi_x}$. Hence the desired answers can be obtained by querying dominance structure on the list $sl(dn,t,\phi_x,\overline{k})$ with $t = \lceil\frac{a^{dn}}{\delta_{dn}}\rceil$ and $k \le \overline{k} < 2k$ by spending $O(\log_B k + \frac{k''}{B})$.
- To get the outputs for the query (a,b,k) in the span $A[a^{\phi_x}...a^{\phi_y+1}-1]$ we need to query appropriate SL list (three-dimensional dominance structure associated with it). Here a^{ϕ_y+1} represents the first δ_{ϕ_y+1}-boundary that follows a^{ϕ_x} and number of I/Os required for querying SL are bounded by $O(\log_B k + \frac{k'''}{B})$.
- The remaining span for the range a^{ϕ_y+1}, a^{up} falls into the case 1 studied earlier as the range will be completely contained in a δ_{ϕ_y}-block. Therefore I/Os needed in this case are given by $T(n,m) + \frac{k''''}{B}$.

We note that $k' = k'' + k''' + k''''$ and total query I/Os are bounded by $T(n,m) + O(\log_B k + \frac{k'}{B})$. Putting all the pieces together, $S(n,m+1) = O(\frac{n}{\log^{*^{m+1}}(\frac{n}{B})})$ and $T(n,m+1) = T(n,m) + O(\log_B k) = O(m\log_B k)$. $\square$

By putting every thing together, we obtain an $O(n\alpha)$-word data structure with $O(\alpha^2(\frac{k}{B})^{1/3} + \alpha\log_B\alpha + \frac{k}{B}) = O(\alpha^3 + \frac{k}{B})$ query I/Os. This completes the proof of Theorem 4.