I/O-Efficient Top-k Range Reporting with Logarithmic Update Cost

Cheng Sheng
Chinese University of Hong Kong
Hong Kong
jeru.sheng@gmail.com

Yufei Tao
Chinese University of Hong Kong
Hong Kong
taoyf@cse.cuhk.edu.hk

June 5, 2015

Abstract

In the top-k range reporting problem, the input set $S$ consists of $n$ points in the real domain, each of which is associated with a distinct real-valued score. Given an interval $q = [x_1, x_2]$ and an integer $k \in [1, n]$, a query returns the $k$ points in $q$ having the largest scores. Specially, if $q$ covers less than $k$ points of $S$, all of them are returned. We want to store $S$ in a dynamic structure so that updates (both insertions and deletions) and queries can be supported efficiently. We present a structure in external memory that uses linear space, answers a query in $O(\lg_B n + k/B)$ I/Os, and supports an update in $O(\lg_B n)$ amortized I/Os, where $B$ is the block size.

Keywords: Top-k Range Reporting, Dynamic Structures, External Memory

*Preliminary versions of this article appeared in PODS’12 and PODS’14.
1 Introduction

In the top-k range reporting problem, the input is a set $S$ of $n$ points in $\mathbb{R}$, where each point $e \in S$ carries a distinct real-valued score, denoted as $\text{score}(e)$. Given an interval $q = [x_1, x_2]$ and an integer $k$, a query returns the $k$ points in $S \cap q$ with the highest scores. If $|S \cap q| < k$, the entire $S \cap q$ should be returned. The goal is to store $S$ in a structure so that queries and updates (i.e., insertions and deletions on $S$) can be supported efficiently.

Motivation. Top-k search in general is widely acknowledged as an important operation in a large variety of information systems (see an excellent survey [11]). It plays a central role in applications where an end user wants only a small number of elements with the best competitive quality, as opposed to all the elements satisfying a query predicate. Top-k range reporting—being an extension of classic range reporting—is one of the most fundamental forms of top-k search. A representative query on a hotel database is “find the 10 best-rated hotels whose prices are between 100 and 200 dollars per night”. Here, each point $e \in S$ represents the price of a hotel, with $\text{score}(e)$ corresponding to the hotel’s user rating. In fact, queries like the above are so popular that database systems nowadays strive to make them first-class citizens with direct algorithm support. This calls for a space-economic structure that can guarantee attractive query and update efficiency.

Computation Model. We study the problem in the external memory (EM) model [2]. A machine is equipped with $M$ words of memory, and a disk of unbounded size that has been formatted into blocks of size $B \geq 64$ words. An I/O either reads a block of data from the disk to memory, or conversely, writes $B$ words in memory to a disk block. The space of a structure is the number of blocks it occupies, whereas the time of an algorithm is the number of I/Os it performs. CPU calculation is for free. A word has $\Omega(lg n)$ bits, where $n \geq B$ is the input size of the problem at hand. Each real value and each integer are assumed to fit in a single word. The values of $M$ and $B$ satisfy the condition $M = \Omega(B)$.\(^1\)

Throughout this paper, a space/time complexity holds in the worst case by default. A logarithm $\log_b x$ is defined as $\max\{1, \log_b x\}$, and $b = 2$ if omitted. Linear cost should be understood as $O(n/B)$ whereas logarithmic cost as $O(lg B n)$.

1.1 Previous Work

Top-k range reporting was first studied by Afshani, Brodal and Zeh [1], who gave a static structure of $O(n/B)$ space that answers a query in $O(lg_B n + k/B)$ I/Os. The query cost is optimal, as can be shown via a reduction from predecessor search [16].

The authors of [1] also considered a variant of the problem called ordered top-k range reporting where the points in the query result must be sorted by score. For this variant, they showed that, for any parameter $\alpha \in [1, \lg M/B(n/B)]$, a data structure achieving $\lg^{O(1)} n + O(\alpha k/B)$ query time must consume

$$\Omega\left(\frac{n}{B} \frac{\alpha^{-1} \lg_M(n/B)}{\lg(\alpha^{-1} \lg_M(n/B))}\right)$$

\(^1\) $M$ can be as small as $2B$ in the model defined in [2]. However, any algorithm that works on $M = cB$ with constant $c \geq 2$ can be adapted to work on $M = 2B$ with only a constant blowup in space and time. Therefore, one might as well consider that $M = \Omega(B)$. 
space. In other words, if only linear space is allowed, $\alpha$ has to be
\[ \Omega \left( \frac{\lg M(n/B)}{\lg \lg M(n/B)} \right). \]
Interestingly, this provides motivation to focus on the unordered version (i.e., the target problem of this paper) because, as long as an unordered query can be solved in $O(\lg_B n + k/B)$ I/Os, we can trivially produce the ordered output by sorting. The total cost $O(\lg_B n + (k/B) \lg M_B(k/B))$ is already optimal up to a small factor, which is $\lg \lg M(n/B)$ for $M = \Omega(B^{1+\epsilon})$ where $\epsilon$ is any positive constant.

In internal memory, by combining a priority search tree [14] and Frederickson’s selection algorithm [9] on heaps, one can obtain a pointer-machine structure that uses $O(n)$ words, answers a query in $O(\lg n + k)$ time, and supports an update in $O(\lg n)$ time. In RAM, Brodal et al. [7] considered a special instance of the problem where the input points of $\mathcal{S}$ are exactly the $n$ integers $1, 2, ..., n$. They gave a linear-size structure with $O(1+k)$ query time (which holds also for the ordered version).

It is worth mentioning that in recent years top-$k$ reporting has also received considerable attention in a variety of other contexts. We refer the interested readers to recent works [12, 15, 17, 18, 19, 20] for entry points into the literature.

1.2 Our Results and Techniques
The main result of this paper is a new structure for solving top-$k$ range reporting with logarithmic query and update cost:

**Theorem 1.** For top-$k$ range reporting, there is a structure of $O(n/B)$ space that answers a query in $O(\lg_B n + k/B)$ I/Os, and supports an insertion and a deletion in $O(\lg_B n)$ I/Os amortized.

We obtain the above result by combining three structures which are designed with different techniques:

**Structure I.** Our first structure, presented in Section 2, adapts the aforementioned pointer machine structure—which combines a priority search tree with Frederickson’s heap selection algorithm—to external memory. This gives a linear-size structure that can be updated in $O(\lg_B n)$ amortized I/Os, but answers a query in $O(\lg n + k/B)$ I/Os (note that the log base is 2). We use the structure to handle $k \geq B \lg n$ in which case its query cost is $O(\lg n + k/B) = O(k/B)$.

**Structure II.** Our second structure guarantees the desired $O(\lg_B n + k/B)$ query cost, but has a somewhat bizarre update complexity of $O(\lg_B n + \frac{\lg n}{B^{1/5}} \lg_B n)$. We use the structure to cover the scenario where $\lg n \leq B^{1/5}$: in such a case the update cost becomes $O(\lg_B n)$.

To design the structure, we start from an obvious connection between top-$k$ range reporting and 3-sided range searching. In the latter problem, we want to index a set of 2d points such that, given a 3-sided rectangle $q = [x_1, x_2] \times [y, +\infty)$, all the points in $q$ can be reported efficiently. To see the connection, let us map each point $e$ in the input set $S$ (of top-$k$ range reporting) to a 2d point $(e, \text{score}(e))$. Denote by $P$ the set of resulting 2d points. Given a top-$k$ range reporting query with search range $[x_1, x_2]$, we can answer it by finding all the points in $P$ that fall in the rectangle $q^* = [x_1, x_2] \times [\tau, \infty)$ for some properly chosen $\tau$.

3-sided range searching has been well solved [4]. The challenge, however, lies in finding a good $\tau$. Ideally, we would like $q^*$ to cover exactly $k$ points of $P$, but as remarked by Afshani, Brodal and
Zeh [1], this “does not seem to be any easier” than the original problem. They circumvented the issue by resorting to shallow cutting. Unfortunately, a shallow cutting is costly to maintain, which explains why the structure of [1] is difficult to update.

Motivated by this, we turned our attention back to the problem of finding \( \tau \). The key to our eventual success is that, we do not need an ideal \( \tau \), but any \( \tau \) that makes \( q^* \) cover \( c k \) (for some constant \( c \geq 1 \)) points in \( P \) is good enough. After retrieving those \( O(k) \) points, we can run an external version of the \( k \)-selection algorithm (e.g., the one by Aggarwal and Vitter [2]) to find the point having the \( k \)-th largest score among them. This algorithm requires only linear time, or in our case, \( O(k/B) \) I/Os.

We therefore introduce the approximate \( k \)-threshold problem:

The input is the same set \( S \) of \( (1d) \) points as in top-\( k \) range reporting. Given an interval \( q = [x_1, x_2] \) in \( \mathbb{R} \) and an integer \( k \in [1, n] \), a query reports a value \( \tau \in \mathbb{R} \) such that at least \( k \) but at most \( O(k) \) points \( e \in S \cap q \) satisfy the condition \( \text{score}(e) \geq \tau \). If \( q \) covers less than \( k \) points in \( S \), the query returns \(-\infty\). The goal is to store \( S \) in a structure so that queries and updates can be supported efficiently.

As a side product of independent interest, we prove:

**Lemma 1.** For the approximate \( k \)-threshold problem, there is a structure that consumes \( O(n/B) \) space, answers a query in \( O(lg_B n) \) time, and supports an insertion and a deletion in \( O(lg_B n + \frac{lg n}{B^{\frac{1}{2}} lg_B n}) \) amortized I/Os.

The above lemma, combined with the external priority search tree\(^2\) of Arge, Samoladas and Vitter [4], leads directly to a top-\( k \) range reporting structure of linear space, \( O(lg_B n + k/B) \) query cost, and \( O(lg_B n + \frac{lg n}{B^{\frac{1}{2}}} lg_B n) \) amortized update cost.

To tackle the approximate \( k \)-threshold problem, it would be natural to index \( S \) with a B-tree. Searching the B-tree with a query range \([x_1, x_2]\) in a standard way yields \( h = O(lg_B n) \) canonical subsets \( S_1, S_2, ..., S_h \) that partition the points of \( S \cap [x_1, x_2] \). One source of difficulty is the lack of a clear decomposability property that allows us to deal with each \( S_i \) individually. We overcome this by precomputing a logarithmic sketch for each \( S_i \), which is a subset of \( S_i \) containing the points with the highest score, the 2nd highest, the 4th highest, and so on. As it probably has become clearer, by adapting the algorithm of Frederickson and Johnson [10] for approximate rank selection from multiple sorted lists, we manage to find a good \( \tau \) by using just the sketches of \( S_1, S_2, ..., S_h \).

Some other technical challenges then arise. First, unlike RAM, an EM structure typically has a large, non-constant, node fanout. In this case, care must be exercised in deciding which sketches to store, so that not many sketches will be needed by a query, and yet, the overall space can still be kept linear. Second, updating a sketch is problematic because, as can be imagined, a single insertion/deletion in a set could destroy its sketch completely. We managed to achieve the result in Lemma 1 by (i) replacing a sketch with a relaxed version where the \( i \)-th point does not have exactly the \( 2^i \)-th highest score, but has instead the \( \Theta(2^i) \)-th highest, and (ii) creating several sets of structures, with each set designed to update the \( i \)-th point of a sketch for a different range of \( i \). The details are presented in Section 3.

**Structure III.** Structure II does not achieve the performance guarantees in Theorem 1 when \( lg n > B^{1/5} \), or equivalently, \( B < lg^5 n \). However, since \( k \geq B lg n \) has already been taken care of by Structure I, it remains to target \( k < B lg n < lg^6 n \). Motivated by this, in Sections 4 and 5, we

\(^2\)An external priority search tree on \( n \) points uses \( O(n/B) \) space, answers a 3-sided range query in \( O(lg_B n + k/B) \) time (where \( k \) is the number of points reported), supports an update in \( O(lg_B n) \) I/Os.
develop a linear-size structure that can be updated in $O(\log_B n)$ I/Os, and answers a query with $k = O(\text{poly} \log n)$ in $O(\log_B n + k/B)$ I/Os. The most crucial idea behind this structure is to use a suite of “RAM-reminiscent” techniques to unleash the power of manipulating individual bits.

1.3 Top-$k$ Range Reporting without the Distinct-Score Condition

Points having distinct scores is a condition in the original formulation of top-$k$ range reporting in [1]. If two points are allowed to have an identical score, the query semantics can be adapted in two natural ways, depending on how ties are treated. The first one is to break ties arbitrarily, namely, if multiple points have the $k$-th highest score $c$ (among the points in the query range $[x_1, x_2]$), return $k - z$ of them arbitrarily, where $z$ is the number of points with scores larger than $c$. The second adaptation is not to break ties at all, that is, all the points with scores at most $c$ are reported.

As long as the original top-$k$ range reporting problem (i.e., with distinct scores) has been solved, both of the above semantics can be supported easily. In fact, this is trivial for the first semantics, in which case we can break ties by letting an object with a smaller id have a higher score, and apply our distinct-score structure directly. For the second semantics, we can maintain a separate B-tree where points $e$ are sorted first by $\text{score}(e)$, and then by coordinate $e$. In this way, after we have found score $c$ using our distinct-score structure, all the points (covered by the query range $[x_1, x_2]$) with score $c$ can be retrieved in $O(\log_B n)$ I/Os, plus the linear output time. Therefore, for each semantics, we obtain a linear-size structure that answers a query in $O(\log_B n)$ time plus the linear output time, and can be updated with the same cost as in Theorem 1.

2 Structure I: For $k = \Omega(B \log n)$

In this section, we will prove:

**Lemma 2.** For top-$k$ range reporting, there is a structure of $O(n/B)$ space that answers a query in $O(\log n + k/B)$ I/Os, and supports an insertion and a deletion in $O(\log_B n)$ I/Os amortized.

As mentioned in Section 1.2, Top-$k$ range reporting has a geometric interpretation. We can convert $S$ to a set $P$ of points, by mapping each element $e \in S$ to a 2d point $(e, \text{score}(e))$. Then, a top-$k$ query with $q = [x_1, x_2]$ equivalently reports the $k$ highest points of $P$ in the vertical slab $q \times (-\infty, \infty)$. This is the perspective we will take to prove Lemma 2.

Our structure is essentially an external priority search tree [4] on $P$ with a constant fanout. However, we make two contributions. First, we develop an algorithm using this structure to perform top-$k$ range reporting. Second, while an update by the standard algorithm of [4] requires $O(\log n)$ I/Os, we lower the update cost to $O(\log_B n)$. The technical nuance lies in the analysis. Our update algorithm is a standard application of the buffering technique [3], but the analytic machinery of [3] falls short in our context. Our analysis of the update overhead is based on a new token argument.

**Structure.** Let $T$ be a weight balanced $B$-tree (WBB-tree) [6] on the x-coordinates of the points in $P$. The leaf capacity and branching parameter of $T$ are both set to $B$. We number the levels of $T$ bottom up, with the leaves at level 0 (this will be a convention on trees throughout the paper). For each node $u$ in $T$, we use $P(u)$ to denote the set of points whose x-coordinates are stored in the subtree of $u$. As a property of the WBB-tree, if $u$ is at level $i$, then $|P(u)|$ falls between $B^{i+1}/4$ and $B^{i+1}$; if $|P(u)|$ is outside this range, $u$ becomes unbalanced and needs to be remedied.
Each node \( u \) naturally corresponds to a vertical slab \( \sigma(u) \) with \( P(u) = \sigma(u) \cap P \).\(^3\) Let \( u, u' \) be child nodes of the same parent. We say that \( u' \) is a right sibling of \( u \) if \( \sigma(u') \) is to the right of \( \sigma(u) \). Otherwise, \( u' \) is a left sibling of \( u \). Note that a node can have multiple left/right siblings, or none (if it is already the left/right most child).

Consider now \( u \) as an internal node with child nodes \( u_1, ..., u_f \) where \( f = O(B) \) (we always follow the left-to-right order in listing out child nodes). We associate \( u \) with a binary search tree \( T(u) \) of \( f \) leaves, which correspond to \( \sigma(u_1), ..., \sigma(u_f) \), respectively. Let \( v \) be an internal node in \( T(u) \). We define \( \sigma(v) = \bigcup_{j=1}^{2} \sigma(u_{j_1}) \), where \( \sigma(u_{j_1}), \sigma(u_{j_1}+1), ..., \sigma(u_{j_2}) \) are the leaves of \( T(u) \) below \( v \), and accordingly, define \( \bar{P}(v) = \sigma(v) \cap P \).

Notice that we can view \( T \) instead as one big tree \( \mathcal{T} \) that concatenates the secondary binary trees of all the nodes in \( T \). Specifically, if \( u' \) is a child of \( u \) in \( T \), the concatenation makes the root of \( \mathcal{T}(u') \) the only child of the leaf \( \sigma(u') \) of \( \mathcal{T}(u) \). See Figure 1. \( \mathcal{T} \) is almost a binary tree except that some internal nodes have only one child which is an internal node itself. However, this is only a minor oddity because any path in \( \mathcal{T} \) of 3 nodes must contain at least one node with two children. The height of \( \mathcal{T} \) is \( O(\lg n) \).

Each node \( v \) in \( \mathcal{T} \) is associated with a set—denoted as \( \text{pilot}(v) \)—of pilot points satisfying two conditions:

- The points of \( \text{pilot}(v) \) are the highest among all points \( p \in P(v) \) that are not stored in any \( \text{pilot}(\hat{v}) \), where \( \hat{v} \) is a proper ancestor of \( v \) in \( \mathcal{T} \).

- If less than \( B/2 \) points satisfy the above condition, \( \text{pilot}(v) \) includes all of them. Otherwise, \( B/2 \leq |\text{pilot}(v)| \leq 2B \). In any case, \( \text{pilot}(v) \) is stored in \( O(1) \) blocks.

The lowest point in \( \text{pilot}(v) \) is called the representative of \( \text{pilot}(v) \).

Finally, for each internal node \( u \) in \( T \), we collect the representatives of the pilot sets of all the nodes in \( \mathcal{T}(u) \), and store these \( O(B) \) representatives in \( O(1) \) blocks—referred to as the representative blocks of \( u \).

**Query.** Given a top-\( k \) query with range \( q = [x_1, x_2] \), we descend two root-to-leaf paths \( \pi_1 \) and \( \pi_2 \) in \( \mathcal{T} \) to reach the leaf nodes \( z_1 \) and \( z_2 \) whose slabs’ x-ranges cover \( x_1 \) and \( x_2 \), respectively. In \( O(\lg n) \) I/Os, we retrieve all the \( O(B \lg n) \) pilot points of the nodes on \( \pi_1 \cup \pi_2 \), and eliminate those outside \( q \times (\neg \infty, \infty) \). Let \( Q_1 \) be the set of remaining points.

\(^3\) Precisely, the slab of a leaf node \( u \) is \( [x, x'] \times (\neg \infty, \infty) \) where \( x \) is the smallest x-coordinate stored at \( u \), and \( x' \) is the smallest x-coordinate in the leaf node \( u' \) succeeding \( u \). If \( u' \) does not exist, \( x' = \infty \). The slab of an internal node unions those of all its child nodes.
Figure 2: The gray nodes in Figure (a) constitute set Π. Each number is a node's sorting key in the heap rooted at that node. Figure (b) shows $H$ after heap concatenation.

Let $v^*$ be the least common ancestor of $z_1$ and $z_2$. Define $\pi'_1$ ($\pi'_2$) as the path from $v^*$ to $z_1$ ($z_2$). Let Π be the set of nodes $v$ satisfying two conditions:

(i) $v \notin \pi'_1 \cup \pi'_2$, but the parent of $v$ is in $\pi'_1 \cup \pi'_2$;

(ii) The x-range of $\sigma(v)$ is covered by $q$.

For every such $v$, we can regard its subtree as a max-heap $H(v)$ as follows. First, $H(v)$ includes all the nodes $v'$ in the subtree of $v$ (in $T$) with non-empty pilot sets. Second, the sorting key of $v'$ is the y-coordinate of the representative of $\text{pilot}(v')$. In this way, we have identified at most $|\Pi|$ non-empty max-heaps, each rooted at a distinct node in Π. Concatenate these heaps into one, by organizing their roots into a binary max-heap based on the sorting keys of those roots. This can be done in $O(\lg n)$ I/Os\footnote{Using a linear-time “make-heap” algorithm; see [8].}. Denote by $H$ the resulting max-heap after concatenation. See Figure 2.

Set $\phi$ to a sufficiently large constant. We now invoke Frederickson’s algorithm to extract the set $R$ of $\phi \cdot (\lg n + k/B)$ representatives in $H$ with the largest y-coordinates; this entails $O(\lg n + k/B)$ I/Os. Let $S_R$ be the set of nodes whose representatives are collected in $R$. Gather all the pilot points of the nodes of $S_R$ into a set $Q_1$.

Define a set $S^*_R$ of nodes as follows. For each node $v \in S_R$, we first add to $S^*_R$ all such siblings $v'$ of $v$ (in $T$) that (i) $v' \notin S_R$, and (ii) the x-range of $\sigma(v')$ is contained in $q$. Second, if $v$ is an internal node, add all its child nodes in $T$ to $S^*_R$. Note that $|S^*_R| = O(|S_R|) = O(\lg n + k/B)$. We now collect the pilot points of all the nodes of $S^*_R$ into a set $Q_2$.

At this moment, we have collected three sets $Q_1, Q_2, Q_3$ with a total size of $O(B \lg n + k)$. We can now report the $k$ highest points in $Q_1 \cup Q_2 \cup Q_3$ in $O(\lg n + k/B)$ I/Os. The query algorithm performs $O(\lg n + k/B)$ I/Os in total. Its correctness is ensured by the fact below:

**Lemma 3.** Setting $\phi = 16$ ensures that $Q_1 \cup Q_2 \cup Q_3$ includes the $k$ highest points in $q \times (\infty, \infty)$.

**Proof.** We will focus on the scenario that the heap $H$ has at least $\phi \cdot (\lg n + k/B)$ representatives. Otherwise, $P$ has $O(B \lg n + k)$ points in $q \times (\infty, \infty)$, and all of them are in $Q_1 \cup Q_2$ ($Q_3$ is empty).

We will first show that $|Q_1 \cup Q_2| \geq k$. This is very intuitive because $Q_1 \cup Q_2$ collects the contents of $\Omega(\lg n + k/B)$ pilot sets. However, a formal proof requires some effort because the pilot set of a
node \(v\) can have arbitrarily few points (in this case all the nodes in the proper subtree of \(v\) must have empty pilot sets). We need a careful argument to address this issue.

We say that a representative in \(R\) is \textit{poor} if its pilot set has less than \(B/8\) points; otherwise, it is \textit{rich}. Consider a poor representative \(r\) in \(R\); and suppose that it is a pilot point of node \(v\), and its \(x\)-coordinate is stored in leaf node \(z\). Note that \(z\) stores the \(x\)-coordinates of at least \(B/4\) points, all of which fall in \(q\). By the fact that \(r\) represents less than \(B/8\) points, we know that at least \(B/8\) points (with \(x\)-coordinates) in \(z\) are pilot points of some proper ancestors of \(v\) in \(T\), and therefore, appear in either \(Q_1\) or \(Q_2\). We associate those \(B/8\) points with \(r\). On the other hand, we associate each rich representative with the at least \(B/8\) points in its pilot set.

Thus, the \(\phi \cdot (\lg n + k/B)\) representatives in \(R\) are associated with at least \((\phi/8)(B \lg n + k)\) points in \(Q_1 \cup Q_2\). Each point \(p \in Q_1 \cup Q_2\), on the other hand, can be associated with at most 2 representatives: the representative of the node where \(p\) is a pilot point, and a poor representative whose \(x\)-coordinate is stored in the same leaf as \(p\).\(^5\) This implies \(|Q_1 \cup Q_2| \geq (\phi/16)(B \lg n + k)\). Hence, \(\phi = 16\) ensures \(|Q_1 \cup Q_2| \geq k\).

Finally, the inclusion of \(Q_3\) ensures that no pilot point in \(q \times (-\infty, \infty)\) but outside \(Q_1 \cup Q_2 \cup Q_3\) can be higher than the lowest point in \(Q_1 \cup Q_2\). The lemma then follows.

**Insertion.** To insert a point \(p\), first update the B-tree \(T\) by inserting the \(x\)-coordinate of \(p\). Let us assume for the time being that no rebalancing in \(T\) is required. Then, we identify the node \(v\) in \(T\) whose pilot set should incorporate \(p\). This can be achieved in \(O(\lg_B n)\) I/Os by descending a single root-to-leaf path in \(T\) (note: not \(T\)), and inspect the representative blocks of the nodes on the path. We add \(p\) to \(\text{pilot}(v)\).

We say that a pilot set \textit{overflows} if it has more than \(2B\) points. If \(\text{pilot}(v)\) overflows, we carry out a \textit{push-down} operation at \(v\), which moves the \(|\text{pilot}(v)| - B\) lowest points of \(\text{pilot}(v)\) to the pilot sets of its at most 2 child nodes in \(T\). The resulting \(\text{pilot}(v)\) has size \(B\). If the pilot set of a child \(v'\) now overflows, we treat it in the same manner by performing a push-down at \(v'\). We will analyze the cost of push-downs later.

**Deletion.** To delete a point \(p\), we identify the node \(v\) in \(T\) whose pilot set contains \(p\). This can be done in \(O(\lg_B n)\) I/Os by inspecting the representative blocks. We then remove \(p\) from \(\text{pilot}(v)\).

We say that a pilot set \textit{underflows} if it has less than \(B/2\) points, and yet, one of its child nodes has a non-empty pilot set. To remedy this, we define a \textit{pull-up} operation at node \(v'\) in \(T\) as one that moves the \(\min\{\lfloor B/2 \rfloor, B - |\text{pilot}(v')|\}\) highest points from

\[
\bigcup_{\text{child } v'' \text{ of } v' \text{ in } T} \text{pilot}(v'')
\]

(1)

to \(\text{pilot}(v')\). If (1) has less than the requested number of points, the pull-up moves all the points of (1) into \(\text{pilot}(v)\), after which all proper descendants of \(v'\) have empty subsets; we call such a pull-up a \textit{draining one}.

In general, if the pilot set of a node \(v'\) underflows, we carry out at most two pull-ups at \(v'\) until either \(|\text{pilot}(v')| = B\), or a draining pull-up has been performed. After the first pull-up, if the pilot set at a child node of \(v'\) underflows, we should remedy that first (in the same manner recursively) before continuing with the second pull-up at \(v'\). We will analyze the cost of pull-ups later.

---

\(^5\) No two poor representatives can have their \(x\)-coordinates stored in the same leaf.
Rebalancing. It remains to clarify how to rebalance $T$. Let $u^*$ be the highest node in $T$ that becomes unbalanced after inserting $p$. Let $\hat{u}$ be the parent of $u^*$. We rebuild the whole subtree of $\hat{u}$ in $T$, and the corresponding portion in $T$. Let $\ell$ be the level of $\hat{u}$ in $T$. Our goal is to complete the reconstruction in $O(B^\ell)$ I/Os. Then, a standard argument with the WBB-tree shows that every update accounts for $O(\log_B n)$ I/Os of all the reconstructions.

Let $\hat{v}$ be the root of $T(\hat{u})$. Essentially, we need to rebuild the subtree of $\hat{v}$ in $T$, which has $O(B^\ell)$ nodes. The first step of our algorithm is to distribute all the pilot points stored in the subtree of $\hat{v}$ down to the leaves where their x-coordinates are stored, respectively. For this purpose, we simply push down all the pilot points of $\hat{v}$ to its child nodes in $T$, and do so recursively at each child. We call this a pilot grounding process.

We now reconstruct the subtrees of $\hat{u}$ and $\hat{v}$. First, it is standard to create all the nodes of $T$ in the subtree of $\hat{u}$, and all the nodes of $T$ in the subtree of $\hat{v}$ in $O(B^\ell)$ I/Os. What remains to do is to fill in the pilot sets. We do so in a bottom up manner. Suppose that we are to fill in the pilot set of $v$, knowing that the pilot sets of all the proper descendants of $v$ (in $T$) have been computed properly. We populate $\text{pilot}(v)$ using the same algorithm as treating a pilot set underflow at $v$.

Next, we will prove that each update accounts for only $O(1)$ I/Os incurred by push-downs and pull-ups. At first glance, this is quite intuitive: inserting a point into a pilot set may “edge out” an existing point there to the next level of $T$, which may then create a cascading effect every level down. Viewed this way, an insertion creates $O(\log n)$ demotion events, and conversely, a deletion creates $O(\log n)$ promotion events. As $O(B)$ such events are handled by a push-down or pull-up using $O(1)$ I/Os, the cost amortized on an update should be $O(\frac{1}{B} \log n)$.
complicates things, however, is the fact that pilot points may bounce up and down across different levels. Below we give an argument to account for this complication.

We imagine some conceptual tokens that can be passed by a node to a child in $T$, but never the opposite direction. Specifically, the rules for creating, passing, and deleting tokens are:

1. When a point $p$ is being inserted into $T$, we give $v$ an insertion token if $p$ is placed in $\text{pilot}(v)$.
2. When a point $p$ is deleted from $T$, we give $v$ a deletion token if $p$ is removed from $\text{pilot}(v)$.
3. In a push-down, when a point $p$ is moved from $\text{pilot}(v)$ to $\text{pilot}(v')$ (where $v'$ is a child of $v$), we take away an insertion token from $v$, and give it to $v'$. We will prove shortly that $v$ always has enough tokens to make this possible.
4. In a pull-up, when a point $p$ is moved from $\text{pilot}(v')$ to $\text{pilot}(v)$ (where $v'$ is a child of $v$), we take away a deletion token from $v$, and give it to $v'$. Again, we will prove shortly that this is always do-able.
5. When an insertion/deletion token reaches a leaf node, it disappears.
6. After a draining pull-up is performed at $v$, all the tokens in the subtree of $v$ disappear.
7. When the subtree of a node $v$ is reconstructed, all the tokens in the subtree disappear.

**Lemma 4.** Our update algorithms enforce two invariants at all times:

- **Invariant 1:** every internal node $v$ in $T$ has at least $|\text{pilot}(v)| - B$ insertion tokens.
- **Invariant 2:** every internal node $v$ in $T$ has at least $B - |\text{pilot}(v)|$ deletion tokens, unless all proper descendants of $v$ in $T$ have empty pilot sets.

Notice that, by Invariant 1, a node $v$ with $|\text{pilot}(v)| \leq B$ is not required to hold any insertion tokens; likewise, by Invariant 2, a node $v$ with $|\text{pilot}(v)| \geq B$ is not required to hold any deletion tokens. Furthermore, the two invariants ensure that the token passing described in Rules 3 and 4 is always do-able.

**Proof of Lemma 4.** Both invariants hold on $v$ right after the subtree of $v$ has been reconstructed because at this moment either (i) $|\text{pilot}(v)| = B$, or (ii) $|\text{pilot}(v)| < B$ and meanwhile all proper descendants of $v$ in $T$ have empty pilot sets.

Inductively, assuming that the invariants are valid currently, next we will prove that they remain valid after applying our update algorithms.

- Putting a newly inserted point $p$ into $\text{pilot}(v)$ gives $v$ a new insertion token, which accounts for the increment of $|\text{pilot}(v)| - B$. Hence, Invariant 1 still holds. Invariant 2 also holds because $B - |\text{pilot}(v)|$ has decreased.
- Physically deleting a point $p \in \text{pilot}(v)$ from $T$ gives $v$ a new deletion token, which accounts for the increment of $B - |\text{pilot}(v)|$. Hence, Invariant 2 still holds. Invariant 1 also holds because $|\text{pilot}(v)| - B$ has decreased.
- Consider a push-down at node $v$. After the push-down, $|\text{pilot}(v)| = B$; thus, Invariants 1 and 2 trivially hold on $v$. Let $v'$ be a child of $v$. Invariant 1 still holds on $v'$ because $v'$ gains as many insertion tokens as the increase of $|\text{pilot}(v')| - B$. Invariant 2 also continues to hold on $v'$ because the value of $B - |\text{pilot}(v')|$ has decreased.
Consider a pull-up at node $v$. After the pull-up, $|\text{pilot}(v)| \leq B$; hence, Invariant 1 trivially holds on $v$. Invariant 2 also holds on $v$ because $v$ loses as many deletion tokens as the decrease of $B - |\text{pilot}(v)|$. Let $v'$ be a child of $v$. Invariant 1 continues to hold on $v'$ because the value of $|\text{pilot}(v')| - B$ has decreased. Invariant 2 also holds on $v'$ because $v'$ gains as many deletion tokens as the increase of $B - |\text{pilot}(v')|$. 

Recall that a push-down is necessitated at a node $v$ only if $\text{pilot}(v) > 2B$. Therefore, by Invariant 1, after the operation $|\text{pilot}(v)| - B = \Omega(|\text{pilot}(v)|)$ insertion tokens must have descended to the next level of $T$. The operation itself takes $O(|\text{pilot}(v)|/B)$ I/Os; after amortization, each of those insertion tokens bears only $O(1/B)$ I/Os of that cost.

Now consider the moment when a pilot set underflow happens at $v$. By Invariant 2, $v$ must be holding at least $B/2$ deletion tokens at this time. Our algorithm performs one or two pull-ups at $v$ using $O(1)$ I/Os. We account for such cost as follows. If neither of the two pull-ups is a draining one, at least $B/2$ deletion tokens must have descended to the next level; we charge the cost on those tokens, each of which bears $O(1/B)$ I/Os. On the other hand, if a draining pull-up occurred, at least $B/2$ deletion tokens must have disappeared; each of them is asked to bear $O(1/B)$ I/Os.

In summary, each token before its disappearance is charged $O(1/B \lg n)$ I/Os in total. Since an update creates only one token, the amortized update cost only needs to increase by $O(1/B \lg n)$ to cover the cost of push-downs and pull-ups.

*Remark.* The above analysis has assumed that the height of $T$ remains $\Theta(\lg n)$. The assumption can be removed by the standard technique of global rebuilding, e.g., rebuilding the whole structure every time $n$ has been doubled or halved from its value when the previous rebuilding happened. With this, we have completed the proof of Lemma 2.

We note that our token argument is reminiscent of the “signal analysis” in [13]. However, unlike the signals in [13] which are explicit items trickling down a tree, the tokens in our analysis are implicit, such that their definitions and behavior need to be carefully formulated to make the argument work. The formulation is, in our opinion, the essence and novelty of our analysis.

### 3 Structure II: For $\lg n \leq B^{1/5}$

In this section, we will prove:

**Lemma 5.** For top-$k$ range reporting, there is a structure of $O(n/B)$ space that answers a query in $O(\lg_B n + k/B)$ I/Os, and supports an insertion and a deletion in $O(\lg_B n + \frac{\lg n}{B^{1/5}} \lg_B n)$ I/Os amortized.

Due to the reasons explained in Section 1.2, we will focus exclusively on the approximate $k$-threshold problem, and more specifically, on proving Lemma 1. We will consider only that the query range $q = [x_1, x_2]$ contains at least $k$ points of $S$, namely, $|S \cap q| \geq k$. Otherwise, the (approximate $k$-threshold) query can be easily answered in $O(\lg_B n)$ time. For this purpose, we only need to create a slightly augmented B-tree on the points of $S$ such that, $|S \cap q|$ can be retrieved in $O(\lg_B n)$ I/Os. If this number is below $k$, we simply return $-\infty$.

In Section 3.1, we will propose a tool called the *logarithmic sketch* for performing approximate rank selection from a union of sorted arrays. This tool will be indispensable in both our structure for Lemma 1 (presented in Sections 3.2-3.3), and our structure III (in Section 4).
3.1 The Logarithmic Sketch

Let $L$ be a set of real values. Given a real value $e$ (which is not necessarily in $L$), we define its rank in $L$ as $|\{e' \in L \mid e' \geq e\}|$. Note that the largest element of $L$ has rank 1.

Frederickson and Johnson [10] considered the following union-rank selection (URS) problem:

Let $L_1, L_2, ..., L_m$ be $m$ disjoint sets of real values, such that each $L_i$ ($1 \leq i \leq m$) is stored in an array, where its elements are sorted in descending order. Given a real value $k$ satisfying $1 \leq k \leq \sum_{i=1}^{m} |L_i|$, a query returns a real value $\tau$ whose rank in $\bigcup_{i=1}^{m} L_i$ falls in $[k, O(k)]$.

They proved the following classic result:

**Lemma 6** ([10]). Regarding the URS problem in the RAM model, there is an algorithm that answers a query with integer parameter $k$ in $O(m)$ CPU time. The $\tau$ returned is an element in $\bigcup_{i=1}^{m} L_i$. Furthermore, the algorithm performs only the following two operations on $L_1, L_2, ..., L_m$: (i) reading an arbitrary element in any $L_i$ ($1 \leq i \leq m$), and (ii) comparing two elements in $\bigcup_{i=1}^{m} L_i$.

In external memory, the above lemma implies that a query can be answered in $O(m)$ I/Os, by trivially simulating the RAM algorithm of [10]. Note that this is true even if $k$ is not an integer: in this case, we only need to replace $k$ with $\lceil k \rceil$ before doing the simulation.

We observe that it is possible to solve the URS problem with the same I/O cost, but significantly less space. For this purpose, we define:

**Definition 1** (Logarithmic Sketch). Let $L$ be a set of real $l$ values. Its logarithmic sketch $\Sigma$ is an array of size $\lceil \lg |L| \rceil + 1$, where the $j$-th ($1 \leq j \leq \lceil \lg |L| \rceil + 1$) entry $\Sigma[j]$ —called a pivot—is a real value whose rank in $L$ falls in $[2^{j-1}, 2^j)$.

Henceforth, we will abbreviate the name “logarithmic sketch” simply as “sketch”. Let us now return to the URS problem. Suppose that, for each input set $L_i$ ($1 \leq i \leq m$), we do not store $L_i$ entirely, but instead store only its sketch $\Sigma_i$. The lemma below shows that we can still answer a query in $O(m)$ I/Os.

**Lemma 7.** Given the sketches $\Sigma_1, \Sigma_2, ..., \Sigma_m$ and a real value $k$ satisfying $1 \leq k \leq |\bigcup_{i=1}^{m} L_i|$, we can find in $O(m)$ I/Os a real value $\tau$ whose rank in $\bigcup_{i=1}^{m} L_i$ is between $k$ and $O(k)$. Furthermore, the $\tau$ returned is either $-\infty$ or an element in $\bigcup_{i=1}^{m} \Sigma_i$.

The rest of the subsection is devoted to proving the lemma. We will first explain our algorithm, then analyze its cost, and finally prove its correctness. It suffices to consider that $k$ is an integer.

**Algorithm for Lemma 7.** For each $i \in [1, m]$, we conceptually construct an array $A_i$ of size $|L_i|$ as follows:

- $A_i[1] = \Sigma_i[1]$.
- For each $j$ satisfying $2 \leq j \leq \lceil \lg |L_i| \rceil$, all of $A_i[2^{j-1}], A_i[2^{j-1} + 1], ..., A_i[2^j - 1]$ are set to $\Sigma_i[j + 1]$.
- All the remaining elements of $A_i$ are set to $-\infty$.

Note that the only pivot of $\Sigma_i$ that does not appear in $A_i$ is $\Sigma_i[2]$.

Another, more intuitive, way of understanding the construction of $A_i$ is as follows. Chop $A_i$ by rank into $\lambda = \lceil \lg |L_i| \rceil + 1$ rank ranges: $[2^0, 2^1 - 1], [2^1, 2^2 - 1], ..., [2^\lambda, |L_i|]$. Recall from Definition 1
that, the $j$-th ($1 \leq j \leq \lambda$) pivot of $\Sigma_i$ is “responsible for” the rank range $[2^{j-1}, 2^j)$. We use this pivot to fill in the preceding rank range of $A_i$, namely, the rank range $[2^{j-2}, 2^{j-1})$. The only exception is in the first rank range, for which we demand $A_i[1] = \Sigma_i[1]$ (which explains why $\Sigma_i[2]$ is never used). Finally, the above steps left the the last rank range $[2^\lambda, |L_i|]$ of $A_i$ unfilled; all those elements are then set to $-\infty$.

We answer a query with integer $k$ by simulating the RAM algorithm of [10] (Lemma 6) on $A_1, A_2, ..., A_m$. Specifically, whenever the algorithm of [10] needs to read $A_i[j]$, we supply the corresponding pivot of $\Sigma_i$ or $-\infty$, according to the rules set forth as above. In this way, we provide accesses to $A_i$ without actually materializing it. The I/O cost of our algorithm is clearly $O(m)$.

A technical detail is worth mentioning. The algorithm of [10] assumes that the elements in $A_1, A_2, ..., A_m$ are all distinct, whereas this is not true in our scenario. We resolve the issue by breaking ties in the following manner. Given two elements $A_{i_1}[j_1]$ and $A_{i_2}[j_2]$, we define $A_{i_1}[j_1] > A_{i_2}[j_2]$ in any of the situations below:

- $A_{i_1}[j_1] > A_{i_2}[j_2]$;
- $A_{i_1}[j_1] = A_{i_2}[j_2]$ and $i_1 < i_2$;
- $A_{i_1}[j_1] = A_{i_2}[j_2]$, $i_1 = i_2$ and $j_1 < j_2$.

Note that $>$ is a total order on the elements of $A_1, A_2, ..., A_m$. We write $e_1 \geq e_2$ to indicate that either $e_1 > e_2$ or $e_1, e_2$ refer to the same element of some array $A_i$. Clearly, $e_1 \geq e_2$ implies $e_1 \geq e_2$.

Correctness. Let $\tau$ be the value returned by our algorithm. For each $i \in [1, m]$, define $\rho_i$ to be the number of elements $e \in A_i$ such that $e \geq \tau$. Lemma 6 tells us that

$$\sum_{i=1}^{m} \rho_i \in [k, O(k)].$$

For each $1 \leq i \leq m$, define $\rho_i'$ to be the rank of $\tau$ in $L_i$. Next, we will prove that

$$\rho_i' \in [\rho_i, 8\rho_i]$$

which suggests

$$\sum_{i=1}^{m} \rho_i' \in \left[ \sum_{i=1}^{m} \rho_i, 8 \sum_{i=1}^{m} \rho_i \right] \subseteq [k, O(k)]$$

as claimed in Lemma 7.

To prove (2), first notice that our construction ensures $L_i[j] \geq A_i[j]$ for all $j \in [1, |L_i|]$. In other words, $A_i[j] \geq \tau$ always implies $L_i[j] \geq A_i[j] \geq \tau$, which in turn indicates that $\rho_i \leq \rho_i'$.

To prove $\rho_i' \leq 8\rho_i$, we distinguish several cases:

- Case 1: $\tau > \Sigma_i[1]$, that is, $\tau$ is greater than even the largest element in $L_i$. Thus, $\rho_i = \rho_i' = 0$.
- Case 2: $\Sigma_i[1] \geq \tau > \Sigma_i[3]$. In this case, our construction ensures $\rho_i = 1$ while the value of $\rho_i'$ can be any integer from 1 to 6.
- Case 3: $\tau \leq \Sigma_i[3]$. Let $j^o \geq 3$ be the integer such that $\rho_i' \in [2^{j^o-1}, 2^{j^o})$. By Definition 1, we know $\Sigma_i[j^o - 1] > \tau$. 

12
We can see that $\rho_i \leq 8\rho_i$ holds in all the above scenarios. This completes the proof of Lemma 7.

3.2 The Structure and Query Algorithm

We are now ready to describe our structure for solving the approximate $k$-threshold problem. We assume that $\lg_B n$ does not change; the assumption can be removed by global rebuilding. For convenience, given a set $L$ of points $e \in \mathbb{R}$ each of which is associated with a distinct score$(e)$, we use the term top-$t$ points $(1 \leq t \leq |L|)$ to refer to the $t$ points in $L$ with the highest scores. Similarly, we use the term bottom point to refer to the point in $L$ with the lowest score.

We create a WBB-tree $T$ on the input set $S$ with leaf capacity $b = B \lg_B n$ and branching parameter $F = B^{3/5}$. Its height is $O(\lg_F (n/b)) = O(\lg_B n)$. For each node $u$ of $T$, denote by $S(u)$ the set of points stored in the subtree rooted at $u$. As a property of the WBB-tree, if $u$ is at level $\ell$, $|S(u)|$ is in the range $[bF^\ell/4, F^\ell]$; otherwise, $u$ is unbalanced. Each node $u$ is naturally associated with an $x$-range in $\mathbb{R}$.

Now consider $u$ to be an internal node with child nodes $u_1, ..., u_f$ where $f \leq 4F$, as guaranteed by the WBB-tree. Define a multi-slab $u[i, i']$ to be the union of the $x$-ranges of $u_i, u_{i+1}, ..., u_{i'}$ for some $i, i'$ satisfying $1 \leq i \leq i' \leq f$. Denote by $S(u[i, i'])$ the set of points in $u[i, i']$, i.e., $S(u[i, i']) = S(u_i) \cup S(u_{i+1}) \cup ... \cup S(u_{i'})$. Let $S(u[i, i'])$ be the set of scores of the points in $S(u[i, i'])$.

For each internal node $u$, we store the following information:

1. The bottom point of $S(u_i)$ and the size of $S(u_i)$ for each child $u_i$ $(1 \leq i \leq f)$. All this information fits in a block. After reading the block, we can obtain for free (i) the bottom point of $S(u[i, i'])$ of any multi-slab $u[i, i']$ $(1 \leq i \leq i' \leq f)$, and (ii) the size of $S(u[i, i'])$.

2. A multi-way list, which collects, for each child node $u_i$ $(1 \leq i \leq f)$ of $u$, the top-$(B^{4/5}/4)$ points in $S(u_i)$. A multi-way list can be stored in a single block because $f \cdot B^{4/5}/4 \leq 4F \cdot B^{4/5}/4 = B$. Because of this, the ordering of the elements in the multi-way list will not matter.

3. A one-way list: It contains the top-$(B^{3/5} \lg_B n)$ points of $S(u)$, sorted in descending order of score.

4. $O(f^2)$ sketches: For each multi-slab $u[i, i']$, we maintain a sketch $\Sigma(u[i, i'])$ of $S(u[i, i'])$. For each pivot $\sigma$ in the sketch, we always store with $\sigma$ its precise rank of in $S(u[i, i'])$.

Now, consider $\sigma'$ as the last pivot of $\Sigma(u[i, i'])$. We enforce an invariant on $\sigma'$ at all times. To explain, let $l$ be the size of $S(u[i, i'])$; in other words, $\sigma'$ is the $\lambda$-th pivot of $\Sigma(u[i, i'])$, where $\lambda = \lfloor \lg l \rfloor + 1$. The invariant says:
If \( l \leq \frac{3}{2} \cdot 2^{\lambda-1} \), \( \sigma' \) is always the lowest score in \( S(u[i, i']) \), namely, the score of the bottom point of \( S(u[i, i']) \). On the other hand, if \( l > \frac{3}{2} \cdot 2^{\lambda-1} \), then \( \sigma' \) is allowed to be any score whose rank in \( S(u[i, i']) \) falls in \([2^{\lambda-1}, l]\).

The information in \( \textbf{I1} \) and \( \textbf{I2} \) fits in \( O(1) \) blocks, which we refer to as the \textit{meta blocks} of \( u \).

For each leaf node \( z \), we store the points of \( S(z) \) in descending order of score. For convenience, we consider that the first \( B^{3/5} \lg_B n \) points of this ordering constitute the \textit{one-way list} of \( z \). Finally, we maintain an external priority search tree \([4]\) on the set \( P \) of 2d points converted from \( S \): each point \( e \in S \) is mapped to a 2d point \((e, \text{score}(e))\) in \( P \), which we call the \textit{image} of \( e \).

**Space.** The WBB-tree and the external priority search tree use \( O(n/B) \) space. To bound the space of sketches, notice that (i) each internal node carries \( O(F^2) \) sketches, (ii) each level-\( \ell \) sketch occupies \( O(\lg(bF^\ell)) \) words, and (iii) the number of level-\( \ell \) nodes is at most \( \frac{n}{bF^{\ell-4}} \), the space consumption of all sketches is

\[
\sum_{\ell=1}^{O(\lg_B n)} \frac{n}{bF^\ell/4} \cdot O\left(1 + \frac{F^2 \lg(bF^\ell)}{B}\right) = O\left(\frac{n}{bF}\right) + O\left(\frac{n}{B}\right) \cdot \sum_{\ell=1}^{O(\lg_B n)} \left(\frac{F^2 \lg b}{b} \cdot \frac{1}{F^\ell} + \frac{F^2 \lg F}{b} \cdot \ell\right) = O\left(\frac{n}{bF}\right) + O\left(\frac{n}{B}\right) \cdot O\left(\frac{F \lg b}{b} + \frac{F \lg F}{b}\right)
\]

\[= o(n/B)
\]

(by \( F = B^{1/5} > 1 \) and \( b = B \lg_B n \)).

Since (i) the multi-way and one-way lists of an internal node together consume \( O(1 + (1/B) \cdot B^{3/5} \lg_B n) \) space and (ii) the number of internal nodes is \( O(n/(bF)) \), all these lists occupy

\[O(1 + (1/B) \cdot B^{3/5} \lg_B n) \cdot O(n/(bF)) = O\left(\frac{n}{bF}\right) + O\left(\frac{nB^{3/5} \lg_B n}{BbF}\right)
\]

(as \( b = B \lg_B n \))

\[= o(n/B)
\]

space. Therefore, our structure consumes linear space overall.

**Query.** Given an approximate \( k \)-threshold query with a search interval \( q = [x_1, x_2] \), first identify, in a standard way, \( h = O(\lg_B n) \) canonical sets \( L_1, \ldots, L_h \) that constitute a partition of \( S \cap q \). For each \( L_i \), its logarithmic sketch is either available as the sketch of some multi-slab, or can be computed in \( O(\lg_B n) \) I/Os from a leaf node (only at most two leaf nodes are involved). Then, we apply Lemma 7 to answer a URS query with parameter \( k \) on \( L_1, \ldots, L_h \) in \( O(h) \) I/Os (as explained at the beginning of Section 3, we are guaranteed that \( k \leq \sum_{i=1}^h |L_i| \)). The output of this URS query is also our final answer for the approximate \( k \)-threshold query. The total query cost is \( O(h) = O(\lg_B n) \) I/Os.

### 3.3 Construction

Let \( \hat{u} \) be a node in \( T \), and \( Z = |S(\hat{u})| \). In this section, we describe an algorithm to reconstruct the subtree of \( \hat{u} \) in \( O((Z/B) \lg_{M/B}(Z/B)) \) I/Os, which will be used as a subroutine in the update
algorithms. Given \( S(\bar{u}) \), the algorithm outputs a new subtree \( \mathcal{T} \) of \( \bar{u} \) in which all the secondary structures have been properly constructed. It is standard to build the nodes of \( \mathcal{T} \), as well as the information \( \mathbf{I}_1 \) of each node, in \( O((Z/B) \log_{M/B}(Z/B)) \) I/Os [5]. In the sequel, we focus on creating the information \( \mathbf{I}_2, \mathbf{I}_3, \) and \( \mathbf{I}_4 \) of the nodes in \( \mathcal{T} \).

For a node \( u \) in \( \mathcal{T} \), define the ranked list of \( u \), denoted as \( rlist(u) \), to be the list of points from \( S(u) \) in descending order of score. We first explain the computation of sketches, which will be used as a building block of the full construction algorithm.

**Computing Sketches.** Consider \( u \) to be an internal node in \( \mathcal{T} \) with \( f \) child nodes. Given \( rlist(u) \), we next show how to compute the \( O(f^2) \) sketches of \( u \) in three steps. The first one scans \( rlist(u) \) to generate all the sketch pivots of \( u \). At this moment, those pivots are not necessarily grouped by the sketches they belong to. Then, the second step achieves the grouping by sorting. The final step generates the last pivot of each sketch. Below are the details:

- For the first step, allocate one block of memory as the output buffer, and another block to keep track of \( O(f^2) \) counters, one for each multi-slab \( u[i, i'] \) \((1 \leq i \leq i' \leq f)\). The counter of \( u[i, i'] \) counts how many points in the multi-slab have already been scanned in \( rlist(u) \). Every time the counter reaches \( \frac{3}{2} 
\end{align*}

\[
O(1 + (f^2 \log Z)/B) = O(1 + (1 + B^{-2/5}) \log_B Z) \tag{3}
\]

- Given a tuple \((i, i', j, \sigma)\), we refer to \((i, i')\) as its key. The second step groups the \( O(f^2 \log Z) \) tuples by key. Since the number of the distinct keys is \( O(f^2) \), this can be done using the distribution sort algorithm of [2] with I/O cost

\[
O \left( 1 + \frac{f^2 \log Z}{B} \log_{M/B}(f^2) \right) = O(1 + (1/B^{2/5}) \log_B Z). \tag{4}
\]

The algorithm is stable, i.e., at its termination, the tuples with the same key are still in descending order of their values of \( j \). The sketches can then be created by reading the sorted list once more.

- At this moment, a sketch may have one less pivot than required. If so, we simply create the last pivot of the sketch from the information \( \mathbf{I}_1 \) of \( u \), enforcing the invariant explained in Section 3.2 (for \( \mathbf{I}_4 \)). This can be done easily in

\[
O \left( 1 + \frac{f^2 \log Z}{B} \right) = O(1 + (1/B^{2/5}) \log_B Z) \tag{5}
\]

I/Os.

Therefore, the sketches of \( u \) can be computed in

\[
O(|rlist(u)|/B) + O((1/B^{2/5}) \log_B Z) = O(|rlist(u)|/B + \log_B n) = O(|rlist(u)|/B)
\]

I/Os, where the last equality used the fact that \(|rlist(u)| \geq bF > \log_B n\) because \( u \) is an internal node. In the sequel, we will refer to the above algorithm as sketch build.
Full Construction Algorithm. We are now ready to explain the details of building the secondary structures for a subtree containing \( Z \) points. Let us first consider \( M = O(B^2) \). In this case, we generate the ranked lists of all the internal nodes of \( \cal{T} \) in a bottom-up manner, where the ranked list of a node is obtained by an \( O(F) \)-way merge which combines the ranked lists of its children. All the \( O(F) \)-way merges that take place at the same level perform \( O((Z/B)\lg_{M/B} F) \) I/Os. As there are \( O(\lg_F (Z/b)) \) levels, all the ranked lists can be produced in \( O((Z/B)\lg_{M/B} (Z/B)) \) time. Then, for each internal node \( u \), compute its sketches as explained earlier in \( O(|rlist(u)|/B) \) I/Os. The multi-way list of \( u \) can be obtained by reading \( rlist(u) \) once—recall that a multi-way list occupies only one block. The one-way list of \( u \) can be generated by another scan of \( rlist(u) \), as it is just a prefix of \( rlist(u) \). Therefore, all the sketches, multi-way lists, and one-way lists can be produced in \( O((Z/B)\lg_B (Z/B)) = O((Z/B)\lg_{M/B} (Z/B)) \) I/Os.

Now, consider \( M = \Omega(B^2) \). In this case, we cannot afford to generate the ranked lists of all the internal nodes. This is because, the height of the subtree, which is \( \Omega(\lg_F (Z/b)) \), can be much larger than \( \lg_{M/B} (Z/B) \), such that we can no longer spend \( O(Z/B) \) I/Os at each level. We circumvent this obstacle by computing the ranked lists for the nodes at only \( O(\lg_{M/B} (Z/B)) \) levels, and deploying each ranked list to build secondary structures for nodes of different levels simultaneously.

In general, suppose that we have obtained all the ranked lists at some level \( \ell \). We will proceed to compute the ranked lists for the nodes at level \( \ell + \eta \), where \( \eta \) is the maximum integer satisfying

\[
4F^\eta < \frac{M}{2B} \cdot \min\{1, F - 1\}. \tag{6}
\]

As \( B \geq 64 \), \( F - 1 = \Omega(1) \), meaning that the right hand side of the above inequality is \( \Omega(M/B) = \Omega(B) \), which guarantees the existence of a valid \( \eta \). For each level-\( (\ell + \eta) \) node \( u \), its ranked list \( rlist(u) \) can be obtained by merging the ranked lists of the descendants of \( u \) at level \( \ell \). Node \( u \) has at most \( 4F^\eta \) descendants at level \( \ell \) because (as a property of the WBB-tree), a node at level \( \ell \) has at least \( bF^\ell /4 \) points in its subtree, whereas \( u \) has at most \( bF^{\ell+\eta} \) points in its subtree. As a result, the merge can be completed in \( O(|rlist(u)|/B) \) I/Os by assigning a memory block to each of the \( 4F^\eta < M/(2B) \) descendants. After that, the secondary structures of \( u \) can be created as discussed before with \( O(|rlist(u)|/B) \) I/Os.

Next, we will explain how to build, together, the secondary structures for all the descendants of \( u \) at levels from \( \ell + 1 \) to \( \ell + \eta - 1 \). Let \( t \) be the number of such descendants, denoted as \( v_1, \ldots, v_t \). Since each level-\( (\ell + i) \) node has at least \( bF^{\ell+i}/4 \) points in its subtree, the number of level-\( (\ell + i) \) descendants of \( u \) is at most \( 4F^{\eta-i} \), meaning that

\[
t \leq \sum_{i=1}^{\eta-1} 4F^{\eta-i} < 4F^\eta/(F - 1) < M/(2B).
\]

Let us first elaborate how to compute the sketches of \( v_1, \ldots, v_t \) at the same time. For any \( v_i \), if we had the ranked list of \( v_i \), then we could simply invoke the sketch build algorithm on \( v_i \). Recall that sketch-build involves three steps. A crucial observation is that, the first step can still be performed with \( rlist(u) \), in replacement of the ranked list of \( v_i \). This is because, as \( rlist(u) \) is scanned, we can ignore those points that do not belong to the subtree of \( v_i \), whereas those that do belong are encountered in descending order of score. Hence, using only 2 blocks of memory (excluding the input buffer for \( rlist(u) \)), we can carry out the first step on \( v_i \) by reading \( rlist(u) \) only once. Recall that \( 2t \) is smaller than \( M/B \). Thus, we can dedicate 2 blocks of memory for each of \( v_1, \ldots, v_t \), so that we can perform the first step for all of them simultaneously with a single scan of \( rlist(u) \). Summing up the I/O cost of reading \( rlist(u) \) and that of outputting the \( O(F^2 \lg Z) \)
tuples for each $v_i$, we know from (3) that the total cost is
\[
O(|\text{rlist}(u)|/B) + t \cdot O \left( 1 + (1/B^{2/5}) \lg_B Z \right).
\] (7)

Now, we can carry out the second and third steps of sketch build for each $v_i$ individually. By (4) and (5), doing so for all $v_i$ requires
\[
t \cdot O \left( 1 + (1/B^{2/5}) \lg_B Z \right)
\] I/Os in total. Therefore, the overall cost of computing the sketches of $v_1, \ldots, v_t$ is dominated by (7). Since $v_1, \ldots, v_t$ are internal nodes in the subtree of $u$, $t$ is bounded by $O(|\text{rlist}(u)|/(bF))$, meaning that
\[
t \cdot O \left( 1 + (1/B^{2/5}) \lg_B Z \right) = t \cdot O(\lg_B N)
\] 
\[
= O(|\text{rlist}(u)|/(bF)) \cdot O(\lg_B N)
\] (as $b = B \lg_B N$) = $O(|\text{rlist}(u)|/(bF))$.

Therefore, (7) = $O(|\text{rlist}(u)|/B)$.

Finally, by allocating one block of memory for each node $v_i$, the one-way lists of all $v_1, \ldots, v_t$ can be computed with one scan of $\text{rlist}(u)$. Similarly, the multi-way lists of all $v_1, \ldots, v_t$ can be computed with another scan. At this point, we have finished constructing the secondary structures of $v_1, \ldots, v_t$ with $O(|\text{rlist}(u)|/B)$ I/Os in total.

The above analysis shows that, in $O(|\text{rlist}(u)|/B)$ I/Os, we can build the secondary structures for $u$ and all of its descendants at level $\ell + 1$ or above. In other words, the secondary structures of all the nodes from levels $\ell + 1$ to $\ell + \eta$ can be computed in $O(Z/B)$ time. As the height of the subtree $T$ we are reconstructing is $O(\lg_B Z)$, we need to pay the $O(Z/B)$ cost $O(\lg_B Z)/\eta$ times. Since $\eta$ is the maximum integer satisfying (6), $4F^{\eta+1} = \Omega(M/B)$. This means that $\eta = \Omega(\lg_F(M/B)) = \Omega(\lg_B(M/B))$, indicating $O(\lg_B Z)/\eta = O(\lg_{M/B}(Z/B))$. Therefore, the total reconstruction cost of $T$ is $O((Z/B)\lg_{M/B}(Z/B))$ for the case $M = \Omega(B^2)$.

### 3.4 Update

We now describe how to update the structure in Section 3.2. The insertion/deletion of a point $e$ into/from $S$ is carried out as follows:

1. Insert/delete $e$ in the WBB-tree $T$. We refer to the path from the root to the leaf node containing $e$ as the update path, denoted as $\pi$.

2. Only nodes on $\pi$ may have become unbalanced. Find the highest unbalanced node $u^*$. If $u^*$ is the root, reconstruct the whole tree; otherwise, reconstruct the subtree rooted at the parent $\hat{u}$ of $u^*$. In both cases, reconstruction is done with the algorithm in the previous subsection. If $u^*$ or $\hat{u}$ is the root, the update is now complete; otherwise, we proceed to the next step.

3. Let $U$ be the set of proper ancestors of $\hat{u}$ if $u^*$ exists (namely, reconstruction was done in Step 2); otherwise, define $U$ to be the set of nodes on $\pi$. Perform the following steps on the nodes $u \in U$ in the bottom-up order.
(a) Let \( v \) be the child of node \( u \) on \( \pi \). In the one-way list of \( u \), only the points coming from the subtree of \( v \) can be affected by the update. Hence, we update the one-way list of \( u \) by (i) first scanning the list to remove all the points from the subtree of \( v \), and (ii) merging the (resulting) one-way list of \( u \) and that of \( v \) by scanning both lists once. The merged list is then the new one-way list of \( u \).

(b) Let \( v \) again be the child of \( u \) on the update path. In the multi-way list of \( u \), only the \( B^{4/5} \) points from the subtree of \( v \) can be affected by the update. Replace them with the top-(\( B^{4/5} \)) points in the multi-way list of \( v \).

(c) Load the meta blocks of \( u \) into memory. Then, update all the sketches of \( u \). We defer the details to Section 3.5.

**Analysis.** Step 1 obviously takes \( O(\lg B n) \) I/Os. If the reconstruction of Step 2 involves \( Z \) points, (as a property of the WBB-tree) we can amortize the cost over \( \Omega(Z/F) \) updates, so that each update bears cost

\[
O((Z/B) \lg M/B (Z/B)) = O\left( \frac{\lg M/B (Z/B)}{B^{4/5}} \right) = O\left( \frac{\lg (n/B)}{B^{4/5}} \right).
\]

As each update needs to bear such cost at most once for each level, the amortized update cost is increased by \( O(\frac{\lg n}{B^{4/5}} \lg B n) \).

Since a one-way list contains \( B^{3/5} \lg B n \) points, Step 3a (which essentially scans two such lists \( O(1) \) times) performs

\[
O \left( 1 + \frac{B^{3/5} \lg B n}{B} \right) = O \left( 1 + \frac{\lg B n}{B^{2/5}} \right)
\]

I/Os. Step 3b require only \( O(1) \) I/Os. As \(|U| = O(\lg B n)\), Steps 3a and 3b entail in total \( O(\lg B n + \frac{\lg n}{B^{2/5}} \lg B n) \) I/Os.

It remains to discuss Step 3c, for which we prove the lemma below in the next subsection:

**Lemma 8.** There is an algorithm to perform Step 3c such that the amortized update cost is increased by \( O(\lg g n + \frac{\lg n}{B^{2/5}} \lg B n) \).

With this, we complete the proof of Lemma 1, and hence, also the proof of Lemma 5.

### 3.5 Proof of Lemma 8

Let \( u \) be an internal node with child nodes \( u_1, ..., u_f \). Next, we describe how to update a sketch \( \Sigma(u[i, i']) \) when a point \( e \) is inserted in or deleted from a multi-slab \( u[i, i'] \). We assume that the meta blocks of \( u \) are already in memory. Denote by \( \lambda \) the number of pivots in \( \Sigma(u[i, i']) \), and by \( l \) the size of \( S(u[i, i']) \). Remember that \( \lambda = \lceil \lg l \rceil + 1 \).

The first update step maintains the length \( \lambda \) of \( \Sigma(u[i, i']) \). An insertion of \( e \) may necessitate one more pivot. If so, we add the score of the bottom point of \( S(u[i, i']) \) as the last pivot of \( \Sigma(u[i, i']) \) with rank \( l \) (the bottom point can be obtained in memory; see the description of \( I_1 \) in Section 3.2). Conversely, a deletion of \( e \) may render \( \lambda \) to decrease by 1. If so, we simply discard the last pivot of \( \Sigma(u[i, i']) \).

Recall that, for each pivot \( \sigma \) of \( \Sigma(u[i, i']) \), our structure keeps track of its rank in \( S(u[i, i']) \). The second update step is to refresh all these ranks. Let us start by considering that \( \sigma \) is not the last
We are now ready to prove Lemma 8. Inserting/deleting a point can affect the sketches of 
$O(\lg_B n)$ nodes, such that $O(F^2)$ sketches at each node can be modified. Excluding the cost of fixing invalidated pivots, at each node, all the sketches can be updated in 
$O(1 + (F^2 \lg n)/B) = O(1 + \frac{\lg n}{B^{3/5}})$ I/Os by scanning the $O(F^2)$ sketches once (each sketch has $O(\lg n)$ pivots). Such cost sums up to 
$O(\lg_B n + \frac{\lg n}{B^{3/5}} \lg_B n)$ at all levels.
It remains to bound the cost of fixing invalidated pivots. Consider an arbitrary pivot \( \sigma \); and suppose that it is the \( j \)-th pivot of \( \Sigma(u[i, i']) \). The pivot must have been created in one of the following ways:

- as the new pivot \( \sigma_{new} \) after fixing the invalidation of a previous \( j \)-th pivot of \( \Sigma(u[i, i']) \);
- directly set, as the last pivot of \( \Sigma(u[i, i']) \), to the lowest score of \( S(u[i, i']) \) at the moment when the size of \( S(u[i, i']) \) was precisely \( \lfloor \frac{3}{2} \cdot 2^{\lambda - 1} \rfloor \), where \( \lambda \) is the number of pivots in \( \Sigma(u[i, i']) \) at that moment;
- generated when a subtree of \( T \) was re-constructed.

In any case, \( \Omega(0.5 \times 2^j) \) updates must have happened to \( S(u[i, i']) \) since the creation of \( \sigma \). We amortize the I/O cost of invalidation fixing—which is \( O(2^j/B^{3/5}) \) by Lemma 9—on those updates such that each update accounts for only \( O(1/B^{3/5}) \) I/Os. As a sketch has \( O(\lg n) \) pivots, each update can be charged \( O(F^2 \lg n) \) times with respect to one sketch. Hence, the update can be charged \( O(F^2 \lg n) \) times with respect to a node. Finally, the update needs to be charged cost for \( O(\lg B \cdot n) \) nodes. Therefore, overall, the cost amortized on an update is

\[
O(F^2 \lg n) \cdot O(1/B^{3/5}) \cdot O(\lg B n) = O \left( \frac{\lg n}{B^{1/5}} \cdot \lg B n \right).
\]

This concludes the proof of Lemma 8.

4 Structure III: For \( k = O(\text{polylog } n) \)

In this section, we will prove:

**Lemma 10.** For top-\( k \) range reporting with \( k = O(\text{polylog } n) \), there is a structure of \( O(n/B) \) space that answers a query in \( O(\lg B n + k/B) \) I/Os, and supports an insertion and a deletion in \( O(\lg B n) \) I/Os amortized.

We will first introduce two relevant problems in Sections 4.1 and 4.2. Our structure III—presented in Section 4.3—is built upon solutions to those problems. Finally, in Section 4.4, we will complete the proof of Theorem 1.

4.1 Union-Rank Selection with Approximate Rank Accesses

We first define a problem called union-rank selection with approximate rank accesses (URS-ARA). We are given \( m \) disjoint sets \( L_1, ..., L_m \) of real values, such that each \( L_i \) (\( 1 \leq i \leq m \)) can be accessed only by the following operators:

- **MAX:** Returns the largest element of \( L_i \) in \( \text{cost}_{\max} \) I/Os.
- **RANK:** Given a real-valued parameter \( \rho \in [1, |L_i|] \), this operator returns in \( \text{cost}_{\text{rank}} \) I/Os an element \( e \in L_i \) whose rank in \( L_i \) falls in \([\rho, c_1 \rho]\) where \( c_1 \geq 2 \) is a constant integer.

Given a real value \( k \) satisfying

\[
1 \leq k \leq (1/c_1) \cdot \min\{|L_1|, ..., |L_m|\},
\]

a query returns an element \( e \in \bigcup_{i=1}^m L_i \) whose rank in \( \bigcup_{i=1}^m L_i \) falls in \([k, c\prime k]\), where \( c' > 1 \) is a constant dependent only on \( c_1 \).
URS-ARA is similar to the URS problem in Section 3.1. However, URS (by allowing direct accesses to the elements of each \( L_i \)) essentially permits a more powerful \textsc{Rank} operator that returns an element in \( L_i \) with any \textit{precise} rank. Thus, the algorithm of Frederickson and Johnson (Lemma 6) no longer works. In the appendix, we show how to adapt their algorithm to obtain:

**Lemma 11.** Regarding the URS-ARA problem, there is an algorithm that answers a query in \( O(\max(cost_{\text{max}} + cost_{\text{rank}})) \) I/Os.

### 4.2 Approximate \((f,l)\)-Group \( k \)-Selection

Given integers \( f \) and \( l \), we define an \((f,l)\)-\textit{group} \( G \) as a sequence of \( f \) disjoint sets \( G_1, ..., G_f \), where each \( G_i \) (\( 1 \leq i \leq f \)) is a set of at most \( l \) real values. Let \( N \geq fl \) be an integer such that a word has \( \Omega(lg N) \) bits.

In the \textit{approximate \((f,l)\)-group \( k \)-selection problem}—henceforth, the \((f,l)\)-problem for short—the input is an \((f,l)\)-\textit{group} \( G \), where the values of \( f, l, N, \) and \( B \) (block size) satisfy all of the following:

- \( l = O(poly \lg N) \)
- \( f \leq \sqrt{B}lg^\epsilon N \) where \( \epsilon \) is a constant satisfying \( 0 < \epsilon < 1 \).

A query is given:

- an interval \( q = [\alpha_1, \alpha_2] \) with \( 1 \leq \alpha_1 \leq \alpha_2 \leq f \),
- and a real value \( k \) satisfying \( 1 \leq k \leq |\bigcup_{i \in q} G_i| \);

it returns a real value \( x \) whose rank in \( \bigcup_{i \in q} G_i \) falls in \( [k, c_2k] \), where \( c_2 \geq 2 \) is a constant. It is required that \( x \) should be either \(-\infty\) or an element in \( \bigcup_{i \in q} G_i \).

The following lemma is a crucial result that stands at the core of our final structure. Its proof is non-trivial and delegated to Section 5.

**Lemma 12.** For the \((f,l)\)-problem, we can store \( G \) in a structure of \( O(fl/B) \) space that answers a query in \( O(\lg_B(fl)) \) I/Os, and supports an insertion and deletion in \( O(\lg_B(fl)) \) I/Os amortized.

### 4.3 Proof of Lemma 10

We are now ready to elaborate on the structure in Lemma 10. Just like proving Lemma 5, it suffices to consider \textit{approximate \( k \)-threshold problem} defined in Section 1.2. For that problem, fixing an integer \( l = O(poly \lg n) \) and assuming \( k \leq l \), next we describe a linear-size structure that answers a query in \( O(\lg_B n) \) I/Os, and can be updated in \( O(\lg_B n) \) I/Os. This implies a structure claimed in Lemma 10.

**Structure.** We build a WBB-tree \( T \) on the input set \( S \) with branching parameter \( F = \sqrt{B}lg n \), and leaf capacity \( b = FLgB \). Each node \( u \) naturally corresponds to an x-range in \( \mathbb{R} \). If \( u \) is an internal node with child nodes \( u_1, ..., u_f \) (\( f \leq 4F \)), as before, define a \textit{multi-slab} \( u[i, i'] \) to be the union of the x-ranges of \( u_i, u_{i+1}, ..., u_{i'} \) for any \( i, i' \) satisfying \( 1 \leq i \leq i' \leq f \).

Given an (internal/leaf) node \( u \), let \( S(u) \) be the subset of elements in \( S \) that are stored in the subtree of \( u \). Define \( G_u \) as the set of \( c_2l \) highest scores of the elements in \( S(u) \), where \( c_2 \) is the constant in the definition of the \((f,l)\)-problem in Section 4.2.

For each leaf node \( z \), maintain a structure of Lemma 1 to support approximate \( k \)-threshold queries on \( S(z) \). Consider now an internal node \( u \) with child nodes \( u_1, ..., u_f \). We
• maintain an \((f, c_2l)\)-structure of Lemma 12 on the \((f, c_2l)\)-group \(G_u = (G_{u_1}, ..., G_{u_i})\), with \(N\) fixed to some integer in \([n, 4n]\) (this will be guaranteed by our update algorithms).

• store \(G_{u_1} \cup ... \cup G_{u_i}\) in a (slightly augmented) B-tree so that, for any \(1 \leq \alpha_1 \leq \alpha_2 \leq f\), the maximum score in \(\bigcup_{i \in \{\alpha_1, \alpha_2\}} G_u\) can be found in \(O(\log_B(fl))\) I/Os.

There are \(O(n/(Fl))\) internal nodes, each of which occupies \(O(Fl/B)\) blocks. Hence, all the internal nodes use altogether \(O((n/B)^2) = o(n/B)\) space. The overall space cost is therefore \(O(n/B)\).

**Query.** Given an approximate \(k\)-threshold query with parameters \(q = [x_1, x_2]\) and \(k \leq l\), search \(T\) in a standard way to identify a minimum set \(C\) of \(O(\log_B n)\) disjoint canonical ranges whose union covers \(q\), such that each canonical range is either the \(x\)-range of a leaf node or a multi-slab.

Define \(S(u[i, i']) = u[i, i'] \cap S\) for each multi-slab \(u[i, i'] \subseteq C\). Perform URS-ARA with parameter \(k\) on \(\{S(u[i, i']) : u[i, i'] \subseteq C\}\). The \((\Theta(F), c_2l)\)-structure of \(u\) and the B-tree on \(G_u\) allow us to implement the RANK and MAX operators on \(S(u[i, i'])\) in \(O(\log_B(Fl))\) I/Os, respectively. Therefore, by Lemma 11, the URS-ARA query finishes in \(O(\log_F n \cdot \log_B(Fl)) = O(\log_B n)\) I/Os. Denote by \(s\) the value returned by this query.

For each leaf node \(z\) whose \(x\)-range is in \(C\), perform an approximate \(k\)-threshold query on \(S(z)\) using \(q \in \{\log_B b = \log_B n\}\) I/Os. There are at most two such leaf nodes; let \(s_1, s_2\) be the results of the two queries. We return \(\max\{s, s_1, s_2\}\) as the final answer.

**Update.** To support updates, for each internal node \(u\), build a B-tree on the scores in \(G_u\). For each leaf node \(z\), build a B-tree on the scores of the elements in \(S(z)\). Refer to these B-trees as score B-trees, whose space consumption is \(O(n/B)\) in total. Denote by \(parent(u)\) the parent of \(u\).

To insert a point \(e\) in \(S\), first descend a root-to-leaf path \(\pi\) to the leaf node \(z\) whose \(x\)-range covers \(e\). At \(z\), update all its secondary structures in \(O(\log^2_B b) = O((\log_B \log n)^2) = O(\log_B n)\) amortized I/Os. Next, we fix the secondary structures of the nodes along \(\pi\) in a bottom up manner. If \(\text{score}(e)\) enters \(G_z\), at \(parent(z)\), delete the lowest score in \(G_z\), and then insert \(\text{score}(e)\) in \(G_z\). The secondary structures of \(parent(z)\) are then updated accordingly. In general, after updating an internal node \(u\), we check using the score B-tree of \(u\) whether \(\text{score}(e)\) should enter \(G_u\). If so, at \(parent(u)\), delete the lowest score in \(G_u\), insert \(\text{score}(e)\) in \(G_u\), and update the secondary structures of \(parent(u)\).

By Lemma 12, we spend \(O(\log_B(Fl))\) amortized I/Os at each node, and hence, \(O(\log_B n)\) amortized I/Os in total along the whole \(\pi\).

We now explain how to handle node splits. Suppose that a leaf node \(z\) splits into \(z_1, z_2\). First, build the secondary structures of \(z_1\) and \(z_2\) in \(O(b \log^2_B b)\) I/Os. At \(v = parent(z)\), destroy \(G_z\), and include \(G_{z_1}\) and \(G_{z_2}\) into \(G_u\). Rebuild all the secondary structures at \(v\) in \(O(Fl \cdot \log_B(Fl)) = O(b \log_B b)\) I/Os (Lemma 12). This cost can be amortized over the \(\Omega(b)\) updates that must have taken place in \(z\), such that each update is charged only \(O(\log^2_B b) = O(\log_B n)\) I/Os.

A split at an internal level can be handled in a similar way. Suppose that an internal node \(u\) splits into \(u_1, u_2\). Divide \(G_u\) into \(G_{u_1}\) and \(G_{u_2}\) in \(O(Fl/B)\) I/Os, and then rebuild the secondary structures of \(u_1, u_2\) in \(O(Fl \cdot \log_B(Fl))\) I/Os. After discarding \(G_u\) but including \(G_{u_1}, G_{u_2}\), we rebuild the secondary structures of \(parent(u)\) in \(O(Fl \cdot \log_B(Fl))\) I/Os. On the other hand, \(\Omega(Fl)\) updates must have taken place in the subtree of \(u\) (recall that the base tree is a WBB-tree). Hence, each of those updates bears \(O(\log_B(Fl))\) I/Os for the split cost. As an update bears such cost for at most one node per level, the amortized update cost increases by only \(O(\log_B n)\).

\(^6\)The constant \(c_1\) in the RANK operator’s definition (see Section 4.1) equals \(c_2\) here, as is guaranteed by the \((f, c_2l)\)-structures. Given that we focus on \(k \leq l\) while each \(G_u\) has size \(c_2l\), we know that the condition stated in (8) always holds.
An analogous algorithm can be used to handle a deletion in $O(\lg B n)$ amortized I/Os. After $n$ has doubled or halved, we destroy the entire structure, reset $N$ to $2n$, and rebuild everything in $O(n \lg B n)$ I/Os. The amortized update cost is therefore $O(\lg B n)$.

A final remark concerns the maintenance of $l$. Suppose that we want to support all $k \leq \lg c n$ for some constant $c \geq 1$ at all times. We may set $l = 2^c \lg n$ at the moment the structure is globally rebuilt. This ensures that, until $n$ doubles (which would trigger another global rebuilding), the value of $l$ will still remain at least $\lg c n$.

4.4 Completing the Proof of Theorem 1

We now combine Structures I, II, and III—summarized in Lemmas 2, 5, and 10, respectively—to obtain a top-$k$ range reporting structure as is claimed in Theorem 1.

At any moment, we maintain either Structure II only or Structures I and III together. We say that we are in mode II for the former situation, or in mode I-III for the latter. The present mode depends on the values of $B$ and $n$:

- If $\lg n < B^{1/5}$, we are always in mode II.
- If $\lg n > 1 + B^{1/5}$, we are always in mode I-III.
- If $B^{1/5} \leq \lg n \leq 1 + B^{1/5}$, we can be in either mode.

Furthermore, Structure III, if exists, is parameterized to support all $k \leq \lg^6 n$.

**Update.** Suppose that we are in mode II. Given an update, we perform it on Structure II in $O(\lg B n)$ amortized I/Os. Then, check whether $\lg n$ has reached $1 + B^{1/5}$. If not, the update is complete. Otherwise, we destroy Structure II, and build Structures I and III from scratch by inserting each element in the current input set $S$ incrementally. After the two structures are ready, we are in mode I-III. The total cost is $O(n \lg B n)$.

On the other hand, suppose that we are in the I-III mode. Given an update, we perform it on Structures I and III in $O(\lg B n)$ I/Os amortized. Then, check whether $\lg n$ has decreased to $B^{1/5}$. If not, the update is complete. Otherwise, we destroy Structures I and III, and build structure II from scratch by inserting each element in the current $S$ incrementally. After the structure is ready, we are in mode II. The total cost is $O(n \lg B n)$.

Clearly, between two consecutive mode switches, there must have been $\Omega(2^{B^{1/5}}) = \Omega(n)$ updates, each of which is amortized $O(\lg B n)$ I/Os incurred by a mode switch.

**Query.** In mode II, we (obviously) answer a top-$k$ range reporting query using Structure II in $O(\lg B n + k/B)$ I/Os. In mode I-III, we check whether the parameter $k$ is at least $B \lg n$. If so, the query is answered by Structure I in $O(k/B)$ I/Os. Otherwise, it must hold that $k < B \lg n \leq \lg^6 n$. The query thus can be answered by Structure III in $O(\lg B n + k/B)$ I/Os. This completes the proof of Theorem 1.

5 Solving the $(f, l)$-Problem

We devote this section to proving Lemma 12. Henceforth, by “query”, we refer to a query in the $(f, l)$-problem. When no ambiguity can arise, we use $G$ to denote also the union of $G_1, \ldots, G_f$. 

5.1 A Static Structure

We will need again the logarithmic sketch (abbreviated as the “sketch” henceforth) in Section 3.1. Create a sketch $\Sigma_i$ for each $G_i$ ($1 \leq i \leq f$). Call the set $\{\Sigma_1, \ldots, \Sigma_f\}$ a sketch set. We store a compressed form of the sketch set as follows. Describe each pivot $\sigma \in \Sigma_i$ by its global rank in $G$ using $\lg(fl)$ bits, and by its local rank in $G_i$ using $\lg l$ bits. Hence, each $\Sigma_i$ requires $O(\lg l \cdot \lg(fl))$ bits. Since

$$\lg(fl) = \lg B + \lg \lg N$$

a compressed sketch set occupies

$$O(fl \cdot \lg(fl)) = O\left(\sqrt{B} \cdot \lg^2 N \cdot (\lg B + \lg \lg N)\right)$$

bits, and thus fits in a block (which has $B \cdot \Omega(\lg N)$ bits).

Given a query, we first spend an I/O reading the compressed sketched set, and then run the algorithm of Lemma 7 on it in memory. Suppose that this algorithm outputs $s$. If $s = -\infty$, we simply return $-\infty$ as our final answer. Otherwise, $s$ is equal to the global rank of an element in $G$. To convert the global rank to an actual element, we index all the elements of $G$ with a B-tree, which supports such a conversion in $O(\lg_B(fl))$ I/Os. The overall space is $O(fl/B)$ (due to the B-tree); and the query cost is $O(\lg_B(fl))$.

5.2 Supporting Insertions

To facilitate updates, we store the elements of each $G_i$ ($1 \leq i \leq f$) in a B-tree that allows us to obtain the element of any specific local rank in $O(\lg_B l)$ I/Os. In addition, we also maintain a structure of the following lemma, whose proof is deferred to Section 5.4:

**Lemma 13.** We can store an $(f, l)$-group $G = (G_1, \ldots, G_f)$ in a structure of $O(fl/B)$ space such that, in one I/O, we can read into memory a single block, from which we can obtain for free the global rank of the element with local rank $r$ in $G_i$, for every $r \in [1, B^{1/3}\lg_B(fl)]$ and every $i \in [1, f]$. The structure supports an insertion and a deletion in $O(\lg_B(fl))$ I/Os.

Suppose that an element $e_{\text{new}}$ is to be inserted in $G_i$ for some $i \in [1, f]$. Let $r_{\text{new}}$ be the rank of $e_{\text{new}}$ in $G$. We observe that, except perhaps a single pivot, the new compressed sketch set (after the update) can be deduced from: the current compressed sketch set, $r_{\text{new}}$ and $i$. To understand this, consider first a compressed sketch $\Sigma_{i'}$ where $i' \neq i$. Each pivot whose global rank is at least $r_{\text{new}}$ now has its global rank increased by 1 (its local rank is unaffected). Regarding the compressed $\Sigma_i$, the same is true, but additionally every such pivot should also have its local rank increased by 1. Furthermore, a new pivot is needed in $\Sigma_i$ if $|G_i|$ reaches a power of 2 after the insertion; the new pivot is the only one in the compressed sketch set that cannot be deduced (because its global rank is unknown).

Motivated by this observation, to insert $e_{\text{new}}$ in $G_i$, we first obtain $r_{\text{new}}$ from the B-tree of $G$ in $O(\lg_B(fl))$ I/Os, and then update the new compressed sketch set as described earlier; this takes $O(1)$ I/Os because the compressed sketch set fits in one block. Next, $e_{\text{new}}$ is inserted in the B-trees of $G$ and $G_i$ using $O(\lg_B(fl))$ I/Os. If now $|G_i|$ is a power of 2, we retrieve the global rank of the smallest element in $G_i$ in $O(\lg_B(fl))$ I/Os, and add the element to $\Sigma_i$ in memory.

Recall that, the $j$-th ($1 \leq j \leq |\lg l| + 1$) pivot of $\Sigma_i$ should have its local rank confined to $[2^{j-1}, 2^j]$. If this is not true, we say that it is invalidated. The insertion may have invalidated
one or more pivots, (all of which can be found with no I/O because $\Sigma_i$ in memory). Upon the invalidation of $\Sigma_i[j]$, we replace it as the element $e \in G_i$ with local rank $\lfloor \frac{3}{2} \cdot 2^{i-1} \rfloor$ so that $O(2^j)$ updates in $G_i$ are needed to invalidate $\Sigma_i[j]$ again. For the replacement to proceed, it remains to obtain the global rank of $e$. We do so by distinguishing two cases:

- **Case 2$^j \geq B^{1/3} \lg_B (f l)$**: We simply fetch $e$ from the B-tree on $G_i$, and obtain its global rank from the B-tree on $G$. We can now update $\Sigma_i[j]$ in memory.

  In total, the invalidated pivot is fixed with $O(\lg_B (fl)) = O(2^j / B^{1/3})$ I/Os. Since $\Omega(2^j)$ updates must have occurred in $G_i$ to trigger the invalidation of $\Sigma_i[j]$, each of those updates accounts for $O(1/B^{1/3})$ I/Os of the pivot recomputation. As an update can be charged at most $O(\lg l)$ times this way (because a sketch has $O(\lg l)$ pivots), its amortized cost is increased by only $O(\lg B \lg l)$. 

- **Case 2$^j < B^{1/3} \lg_B (f l)$**: There are $O(\lg (B^{1/3} \lg_B (fl)))$ such invalidated pivots in $\Sigma_i$. We can recompute all of them together in $O(1)$ I/Os using Lemma 13.

Overall, an insertion requires $O(\lg_B (fl))$ I/Os amortized.

### 5.3 Supporting Deletions

Suppose that an element $e_{old}$ is to be deleted from $G_i$ for some $i \in [1, f]$. Let $r_{old}$ be the rank of $e_{old}$ in $G$. Except possibly for only one pivot, the new compressed sketch set can be deduced based only on the current compressed sketch set, $r$, and $i$. To see this, consider first $\Sigma_{i'}$ where $i' \neq i$. Each pivot whose global rank is larger than $r_{old}$ now needs to have its global rank decreased by 1. Regarding $\Sigma_i$, the same is true, and every such pivot should also have its local rank decreased by 1. Furthermore, the last pivot of $\Sigma_i$ should be discarded if $|G_i|$ was a power of 2 before the deletion: in such a case, we say that $\Sigma_i$ shrinks. Finally, if $e_{old}$ happens to be a pivot of $\Sigma_i$, a new pivot needs to be computed to replace it—this is the only pivot that cannot be deduced; we call it a dangling pivot.

The concrete steps of deleting $e_{old}$ are as follows. After fetching its global rank $r_{old}$ in $O(\lg_B (fl))$ I/Os, we update the compressed sketch set in memory according to the above discussion. If $\Sigma_i$ shrinks, we delete the last pivot $\Sigma_i$ in memory. If $e_{old}$ was a pivot (say, the $j$-th one), we retrieve the element $e$ with local rank $\lfloor \frac{3}{2} \cdot 2^{i-1} \rfloor$ in $G_i$, and obtain its global rank using $O(\lg_B (fl))$ I/Os. We then replace the dangling pivot $\Sigma_i[j]$ with $e$ in memory.

Finally, recompute the invalidated pivots (if any) in the same way as in an insertion. As analyzed in Section 5.2, such recomputation increases the amortized update cost by only $O(\lg_B l)$.

### 5.4 Proof of Lemma 13

Let us define the list of the $B^{1/3} \lg_B (fl)$ largest elements of $G_i$ (1 ≤ $i$ ≤ $f$) as the prefix of $G_i$, and denote it as $P_i$. Let $P$ be the union of $P_1, ..., P_f$; we refer to $P$ as a prefix set. $P$ contains at most $f B^{1/3} \lg_B (fl)$ points.

We compress $P$ by describing each element $e$ (say, $e \in P_i$ for some $i$) in $P$ using its global rank in $G$ and its local rank in $G_i$, for which purpose $O(\lg (fl))$ bits suffice. Hence, $P$ can be described by

$$f \cdot B^{1/3} \lg_B (fl) \cdot O(\lg (fl)) = \sqrt{B} \lg^e N \cdot B^{1/3} \cdot O((\lg B + \lg \lg N)^2)$$

$$= B^{5/6} \cdot \lg^e N \cdot O((\lg B + \lg \lg N)^2)$$

25
bits, which fit in a block. After loading this block into memory, we can obtain the global rank of the \( r \)-th largest element of \( G_i \) for free, regardless of \( r \in [1, B^{1/3} \lg_B(fl)] \) and \( i \).

Besides the aforementioned block, we also maintain a B-tree on each \( G_i \) \( (1 \leq i \leq f) \) and a B-tree on \( G \). The space consumed is \( O(fl/B) \).

**Insertion.** Suppose that we need to insert an element \( e_{\text{new}} \) into \( G_i \). First, we update the B-trees of \( G_i \) and \( G \) in \( O(\lg_B(fl)) \) I/Os. With the same cost, we can also decide whether \( e_{\text{new}} \) should enter \( P_i \). If not, the insertion is complete.

Otherwise, we find the global rank \( r_{\text{new}} \) and its local rank \( r'_{\text{new}} \) in \( G_i \) with \( O(\lg_B(fl)) \) I/Os. Load the compressed prefix set \( P \) into memory with 1 I/O. Then, the new compressed prefix set can be determined for free based on \( P, i, r_{\text{new}}, \) and \( r'_{\text{new}} \). To see this, first consider a compressed prefix \( P' \) with \( i' \neq i \): if an element has global rank at least \( r_{\text{new}} \), it should have its global rank increased by 1. Regarding the compressed prefix \( P_i \), the same is true; furthermore, all such elements in \( P_i \) should also have their local ranks increased by 1. Finally, we add \( e_{\text{new}} \) into \( P_i \); if \( P_i \) has a size over \( B^{1/3} \lg_B(fl) \), we discard its smallest element.

**Deletion.** Suppose that we need to delete an element \( e_{\text{old}} \) from \( G_i \). Using the B-tree on \( G \), we find its global rank \( r_{\text{old}} \) in \( O(\lg_B(fl)) \) I/Os. Then, \( e_{\text{old}} \) is removed from the B-trees of \( G_i \) and \( G \) in \( O(\lg_B(fl)) \) I/Os.

If \( e_{\text{old}} \notin P_i \), the deletion is done. Otherwise, we load the compressed prefix set in 1 I/O, and then update it, except for a single element, in memory. Specifically, in a compressed prefix \( P' \) with \( i' \neq i \), if an element has global rank at least \( r_{\text{old}} \), it should have its global rank decreased by 1. Regarding the compressed prefix \( P_i \), the same is true; furthermore, all such elements in \( P_i \) should also have their local ranks decreased by 1.

The last element of \( P_i \) is the only one that cannot be inferred directly at this point. But it can be filled in simply by retrieving the element with local rank \( B^{1/3} \lg_B(fl) \) in \( G_i \), and then its global rank in \( G \), all in \( O(\lg_B(fl)) \) I/Os.

6 Conclusions

In this paper, we have presented a dynamic structure for solving the top-\( k \) range reporting problem in external memory. If \( n \) is the size of the input set, our structure uses \( O(n/B) \) space, answers a query in \( O(\lg_B(n + k/B)) \) I/Os, and can be updated in \( O(\lg_B n) \) amortized I/Os per insertion and per deletion.

A somewhat subtle note concerns whether our structure is “indivisible”. It is, as far as the elements in the input set are concerned, because we indeed treat them as atoms. This is obvious for our Structures I and II. Structure III, on the other hand, carries out bit-twiddling, but only on ranks that are derived from the input elements (this is done in the structure of Section 5). The elements themselves, as well as their “satellite data” (if exist), are stored only in indivisible structures, or specifically, WBB-trees and external priority search trees. Nevertheless, it would be interesting study if it would be possible to eliminate bit-twiddling altogether, namely, treating every real/integer value as an atom, too.

Acknowledgements

This work was supported in part by projects GRF 4168/13 and GRF 142072/14 from HKRGC.
References


Appendix

Proof of Lemma 11

Given an element \( e \in L_i \) (\( 1 \leq i \leq m \)), we refer to its rank in \( L_i \) as its local rank, and its rank in \( L \) as its global rank. We will assume that \( m \) is a power of \( c_1 \). The assumption will be removed at the end of the proof.

Case \( k \geq m \). Our algorithm executes in \( \lg c_1 m \) rounds. In the \( j \)-th round (\( 1 \leq j \leq \lg c_1 m \)), \( m/c_1^{j-1} \) sets among \( L_1, ..., L_m \) are active, while the others are inactive. At the beginning, \( L_1, ..., L_m \) are all active, whereas at the end, only one of them remains active.

In round \( j \), we execute RANK on each active set \( L_i \) with parameter \( \rho = c_1^j k/m \) (note that since \( k \leq |L_i|/c_1 \), it must hold that \( \rho \leq |L_1| \)). Remember that the operator can return any element whose local rank falls in \( [c_1^j k/m, c_1^{j+1} k/m] \). Let \( X \) be the set of elements fetched. We call each element in \( X \) a marker, and assign it a weight equal to

- \( c_1 k/m \) if \( j = 1 \);
- \( c_1^j k/m - c_1^{j-1} k/m \) if \( j > 1 \).

The \( m/c_1^{j} \) largest markers in \( P' \) are taken as pivots, among which the smallest is the cutoff pivot of this round. An active set remains active in the next round if its marker is a pivot, whereas the other active sets become inactive.

Denote by \( P'_j \) the set of pivots taken in the \( j \)-th round, and by \( P \) the union of \( P_1, P_2, ..., P_{\lg c_1 m} \). Consider a pivot \( p \in P'_j \), and suppose that it comes from \( L_i \) for some \( i \in [1, m] \). Define the local prefix weight of \( p \) as \( c_1^j k/m \). There are two important facts:

- The local prefix weight of \( p \) is exactly the sum of the weights of the first \( j \) pivots fetched from \( L_i \);
- The local rank of \( p \) is at least its local prefix weight.

For each pivot \( p \in P \), define its prefix weight as the total weight of all the pivots that are larger than or equal to \( p \). We have:

Observation 1. The prefix weight of \( p \) is at most the global rank of \( p \).
Proof. Define \( p_i \) as the smallest pivot in \( L_i \) that is at least \( p \); \( p_i' = \text{nil} \) if no such pivot exists. Defining the local prefix weight of a \( \text{nil} \) point to be 0, we have:

\[
\text{prefix weight of } p = \sum_{i=1}^{m} \text{local prefix weight of } p_i \\
\leq \sum_{i=1}^{m} \text{local rank of } p_i \\
\leq \sum_{i=1}^{m} \text{number of elements in } L_i \text{ at most } p \\
= \text{global rank of } p.
\]

Observation 2. Every cutoff pivot has a prefix weight at least \( k \).

Proof. Consider the cutoff pivot \( p^*_j \) of round \( j \in [1, \lg_{c_1} m] \). Let \( P'_j \) be the set of \( m/c_1 j \) pivots of \( P_j \) greater than or equal to \( p^*_j \). The prefix weight of \( p^*_j \) is at least the sum of the local prefix weights of all the pivots in \( P'_j \), which is at least \((m/c_1 j)(c_1 j/k/m) = k\). \( \square \)

We perform a weighted selection to find the largest pivot \( v \in P \) whose prefix weight is at least \( k \) (\( v \) definitely exists by the above observation). The algorithm terminates by returning \( v \).

The algorithm performs in \( O(m \cdot \text{cost}_{\text{rank}}) \) I/Os. To see this, notice that the \( j \)-th round takes \( O((m/c_1 j^{-1}) \cdot \text{cost}_{\text{rank}}) \) I/Os, i.e., geometrically decreasing with \( j \). Furthermore, \(|P| = \sum_{j=1}^{\log_{c_1} m} (m/c_1 j) = O(m)\); and hence, the weighted selection needs only \( O(m/B) \) I/Os. Next, we prove that the algorithm is correct, namely, the global rank of \( v \) is in \([k, c'k]\) for some constant \( c' \) dependent only on \( c_1 \).

Observation 3. The prefix weight of \( v \) is at most \((1 + c_1)k\).

Proof. We define \( v' \) as the smallest pivot in \( P \) that is larger than \( v \). By definition, we know that the prefix weight of \( v' \) is smaller than \( k \). Define \( p'_i \) as the smallest pivot in \( L_i \) that is at least \( v' \); \( p_i' = \text{nil} \) if no such pivot exists. Defining the local prefix weight of a \( \text{nil} \) point to be 0, we have:

\[
k > \text{prefix weight of } v' = \sum_{i=1}^{m} \text{local prefix weight of } p'_i.
\]

It also holds that

\[
\text{prefix weight of } v = \text{prefix weight of } v' + \text{weight of } v < k + \text{weight of } v
\]

Suppose that \( v \) comes from \( L_{i^*} \) for some \( i^* \in [1, m] \). We distinguish two cases:

Case 1: \( p_{i^*}' = \text{nil} \). This means that \( v \) was taken in the first round of our algorithm; hence, its weight is \( c_1 k/m \). Therefore, by (10) the prefix weight of \( v \) is less than \((1 + c_1)k\).

Case 2: \( p_{i^*}' \neq \text{nil} \). Then, the weight of \( v \) is less than \( c_1 \) times the local prefix weight of \( p_{i^*}' \). Together with (9), this implies that the weight of \( v \) is less than \( c_1 k \). Therefore, (10) tells us that the prefix weight of \( v \) is less than \((1 + c_1)k\). \( \square \)

We are now ready to establish the correctness of our algorithm:
Observation 4. The global rank of $v$ is between $k$ and $c_1^2(2 + c_1)k$.

Proof. The fact that the global rank of $v$ is at least $k$ follows from Observation 1 and the definition of $v$. Next, we prove the second part of the claim. For this purpose, we need some definitions:

- Let $p_i$ be the smallest pivot of $L_i$ that is at least $v$. If $p_i$ exists, we say that $L_i$ is pivotal; otherwise, $L_i$ is non-pivotal.
- By Observation 2 and the definition of $v$, all cutoff pivots are at most $v$. This implies that every $L_i$ has at least one marker that is at most $v$. We will denote the largest such marker as $e_i$. Note that $e_i$ is not necessarily a pivot.
- Define $S_i$ to be the set of elements in $L_i$ that are at least $v$.

For a pivotal set $L_i$, we claim:

$$|S_i| < c_1^2 \cdot \text{(local prefix weight of } p_i).$$

To prove the inequality, suppose that $p_i$ was taken in the $j$-th round of our algorithm. Then, $e_i$ was taken in the $(j + 1)$-th round, meaning that the local rank of $e_i$ less than $c_1(j + 2)k/m$. Since the local rank of $e_i$ is an upper bound of $|S_i|$, it follows that $|S_i| < c_1(j + 2)k/m = c_1^2 \cdot \text{(local prefix weight of } p_i)$.

In the pivotal sets, the total number of elements at least $v$ equals:

$$\sum_{i \text{ s.t. } L_i \text{ is pivotal}} |S_i| \leq c_1^2 \sum_{i \text{ s.t. } L_i \text{ is pivotal}} \text{local prefix weight of } p_i$$

(by Observation 3) \leq c_1^2(1 + c_1)k.$$

Every non-pivotal set has less than $c_1^2k/m$ elements at least $v$. Hence, the total number of such elements in all non-pivot sets is less than $c_1^2k$.

Now we conclude that the global rank of $v$ is less than

$$c_1^2(1 + c_1)k + c_1^2k = c_1^2(2 + c_1)k.$$

$\Box$

Case $k < m$. From each $L_i$, we use MAX to request the largest element in $L_i$. Let $X$ be the set of elements fetched (i.e., one from each $L_i$). Obtain the $k$-th largest element $v$ in $X$. Make set $L_i$ inactive if its largest element is smaller than $v$; otherwise, $L_i$ is active. Run the above algorithm on the $k$ active sets. Suppose that the algorithm outputs $v'$. We then return $\max\{v, v'\}$ as the final answer. It is easy to prove that the algorithm is correct, and runs in $O(m(cost_{rank} + cost_{max}))$ I/Os.

When $m$ Is Not a Power of $c_1$. We add dummy sets $L_1^\Delta, L_2^\Delta, ..., L_d^\Delta$ for some $d < c_1m = O(m)$ so that $m + d$ is a power of $c_1$. These dummy sets satisfy:

- For each $j \in [1, d]$, the size of each $L_i^\Delta$ is $c_1k$.
- All elements of $L_j^\Delta$ are larger than the elements in $\bigcup_{i=1}^m L_i$. 

30
• The MAX and RANK operators on a dummy set are performed entirely in memory as follows. 
MAX on \( L_j^\Delta \) returns a dummy element represented as a pair \((i, c_1)\). Given a parameter \( \rho \in [1, c_1] \), RANK on \( L_j^\Delta \) returns a dummy element represented as a pair \((j, |\rho|)\). Given two distinct dummy elements \((j_1, x_1^\Delta)\) and \((j_2, x_2^\Delta)\), we resolve their comparison by declaring the former to be larger if (i) \( j_1 < j_2 \), or (ii) \( j_1 = j_2 \) and \( x_1^\Delta < x_2^\Delta \).

We run our algorithm—as presented above—on the \( m \) original sets and the \( d \) dummy sets. The second bullet guarantees that the output of an algorithm is a correct answer on the \( m \) original sets. Our algorithm performs \( O((m + d)(\text{cost}_{\text{max}} + \text{cost}_{\text{rank}})) = O(m(\text{cost}_{\text{max}} + \text{cost}_{\text{rank}})) \) I/Os.