Dynamic Top-K Range Reporting in External Memory

Cheng Sheng† Yufei Tao‡
†Department of CSE, CUHK, Hong Kong
‡Division of WebST, KAIST, Republic of Korea

ABSTRACT
In the top-K range reporting problem, the dataset contains \( N \) points in the real domain \( \mathbb{R} \), each of which is associated with a real-valued score. Given an interval \([x_1, x_2]\) in \( \mathbb{R} \) and an integer \( K \leq N \), a query returns the \( K \) points in \([x_1, x_2]\) having the smallest scores. We want to store the dataset in a structure so that queries can be answered efficiently. In the external memory model, the state of the art is a static structure that consumes the external memory model, the state of the art is a static structure so that queries can be answered efficiently. In this paper, we study the problem in a dynamic setting, i.e., points can be inserted into and deleted from \( P \). The top-K range reporting problem can also be defined in a symmetric manner, so that a query returns the \( K \) points with the highest scores.

We consider the external memory (EM) model [2], where a machine has a disk of an unlimited capacity, and memory of \( M \) words. The disk is formatted into disjoint blocks of \( B \) words. The value of \( M \) is at least \( 2B \). We further require that \( B \geq 64 \). Time is measured by the number of I/Os performed, and space is measured by the number of blocks occupied. On an input of size \( N \), a linear complexity is \( O(N/B) \), whereas a poly-logarithmic complexity is \( O(\log^c B N) \) for some positive constant \( c \).

1.1 Applications
The top-K range reporting problem has been studied in a variety of areas, e.g., information retrieval [1, 5], OLAP [9], data streams [8], etc. It finds use in applications where people would like to identify the best few objects, among only a subset of the dataset satisfying a range predicate. For example, a user of a hotel database may be interested in discovering the \( K \) best rated hotels whose prices are in a designated range. Likewise, for promotion purposes, the manager of a company may want to find the \( K \) salesmen with the best performance, among those salesmen whose salaries are in a certain range. In fact, one can easily notice that the top-K range reporting problem generalizes conventional SQL construct of the form

\[
\text{SELECT MIN}(A_1) \text{ FROM } \ldots \text{ WHERE } A_2 \in [x_1, x_2].
\]

That is, instead of retrieving simply the min (a.k.a. top-1) of \( A_1 \), a top-K range query aims at returning the \( K \) objects with the best \( A_1 \) values. Therefore, an efficient structure for this problem augments a relational database with the power to support the top-K version of MIN aggregation (and hence, also MAX aggregation) in the presence of a range predicate.

1.2 Related work
We will focus on only structures with non-trivial worst-case query cost, as they are the subject of this paper. Afshani, Brodal and Zeh [1] developed a static structure that occupies \( O(N/B) \) space, answers a query in

Categories and Subject Descriptors
F.2.2 [Analysis of algorithms and problem complexity]: Nonnumerical Algorithms and Problems—computations on discrete structures; H.3.1 [Information storage and retrieval]: Content analysis and indexing—indexing methods

General Terms
Theory

Keywords
Range top-k, data structure, external memory, logarithmic sketch

1. INTRODUCTION
In the top-K range reporting problem, the dataset is a set \( P \) of \( N \) points in the real domain \( \mathbb{R} \), where each point \( x \) is associated with a distinct score, denoted as \( \text{score}(x) \), in \( \mathbb{R} \). Given an interval \([x_1, x_2]\) in \( \mathbb{R} \) and an integer \( K \leq N \), a query reports, among all the points of \( P \) in \([x_1, x_2]\), the \( K \) points with the lowest scores. As a special case, if \( P \cap [x_1, x_2] \) contains less than \( K \) points, the query returns all of them. The objective is to maintain \( P \) in a structure such that every query can be answered efficiently. In this paper, we study the problem in a dynamic setting, i.e., points can be inserted into and deleted from \( P \). The top-K range reporting problem can also be defined in a symmetric manner, so that a query returns the \( K \) points with the highest scores.

We consider the external memory (EM) model [2], where a machine has a disk of an unlimited capacity, and memory of \( M \) words. The disk is formatted into disjoint blocks of \( B \) words. The value of \( M \) is at least \( 2B \). We further require that \( B \geq 64 \). Time is measured by the number of I/Os performed, and space is measured by the number of blocks occupied. On an input of size \( N \), a linear complexity is \( O(N/B) \), whereas a poly-logarithmic complexity is \( O(\log^c B N) \) for some positive constant \( c \).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
PODS’12, May 21–23, 2012, Scottsdale, Arizona, USA.
Copyright 2012 ACM 978-1-4503-1248-6/12/05 ...$10.00.
We consider a variant of the problem called ordered top-K range reporting where the points in the query result must be sorted by their scores. For this variant, they show that, for any parameter $\alpha \in [1, \log_2(B/(N/B))]$, a data structure that achieves $O((\log B N + K/B))$ query time must consume
\[
\Omega \left( \frac{\log M(N/B)}{\log \log M(N/B)} \right)
\]
space. In other words, if only linear space is allowed, $\alpha$ has to be
\[
\Omega \left( \frac{\log M(N/B)}{\log \log M(N/B)} \right).
\]

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 $O((\log B N + K/B))$ time, we can trivially produce the ordered output by sorting. The total cost $O((\log B N + (K/B) \log M/(B/(K/B)))$ is already optimal up to a small factor, which is $\log \log M(N/B)$ for $M = \Omega(B^{1+\epsilon})$ where $\epsilon$ is any positive constant.

In RAM, by combining the priority search tree of McCreight [10] with the algorithm of Frederickson [6] for selecting the $K$ smallest items from a priority queue, one can obtain a structure for solving the unordered top-$K$ range reporting problem that consumes $O(N)$ space, answers a query in $O(\log N + K)$ time, and supports an update in $O(\log(N))$ time. Brodal, Fagerberg, Greve and Lopez-Ortiz [5] considered a special version of the problem where the points of $T$ take distinct values from the integer set $\{1, \ldots, N\}$. In this case, they gave a static structure that uses $O(N)$ space and answers a query in $O(K)$ time. Their solution also works for the ordered variant.

### 1.3 Our results

We give the first dynamic structure for the top-$K$ range reporting problem in external memory:

**Theorem 1.** For the top-$K$ range reporting problem, there is a data structure that consumes linear space, answers a query in $O((\log_B N + K/B))$ time, and can be updated in $O(\log_B^2 N)$ amortized time per insertion and deletion. The structure can be constructed in $O((N/B) \log M/(B/(N/B)))$ I/Os.

Our construction time improves the result of [1] by a factor of nearly $B$ (also recall that the structure of [1] is static). The starting point of our techniques is an obvious connection between the top-$K$ range reporting problem 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 $q = [x_1, x_2] \times (-\infty, y]$, all the points in $q$ can be reported efficiently. To see the connection, let us map each point $x$ in the input dataset $P$ (of the top-$K$ range reporting problem) to a 2d point $(x, \text{score}(x))$. Denote by $S$ the set of resulting $2d$ points. Given a top-$K$ range query with search range $[x_1, x_2]$, we can answer it by finding all the points in $S$ that fall in the rectangle $q = [x_1, x_2] \times (-\infty, \tau]$ for some properly chosen $\tau$. 3-sided range searching has been well solved [3]. The challenge, however, lies in finding a good $\tau$. Ideally, we would like $q$ to cover exactly $K$ points of $S$, 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 an interesting method called shallow cutting. Unfortunately, a shallow cutting is costly to compute, which explains why the structure of [1] is expensive to construct and 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 $cK$ (for some $c \geq 1$) points in $S$ 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 smallest score among them. This algorithm requires only linear time, or in our case, $O(K/B)$ I/Os.

We therefore introduce a new problem:

**Problem 1 (Approximate $K$-threshold problem).** Let $P$ be as defined in the top-$K$ range reporting problem. Given an interval $[x_1, x_2]$ in $\mathbb{R}$ and an integer $K \leq N$, a query $q$ reports a value $\tau \in \mathbb{R}$ such that at least $K$ but at most $O(K)$ points $x \in P$ satisfy the condition that $x \in [x_1, x_2]$ and $\text{score}(x) \leq \tau$. If $[x_1, x_2]$ covers less than $K$ points in $P$, the query returns $\infty$.

As a side product that is of independent interest, we prove:

**Lemma 1.** For the approximate $K$-threshold problem, there is a structure that consumes $O((N/B) \log M/(B/(N/B)))$ I/Os, answers a query in $O((\log_B N))$ time, and can be updated in $O((\log_B N))$ amortized time per insertion and deletion. The structure can be constructed in $O((N/B) \log M/(B/(N/B)))$ I/Os.

The above lemma, combined with the external priority search tree of Arge, Samoladas and Vitter [3], leads directly to Theorem 1. An external priority search tree on $N$ points uses $O(N/B)$ space, answers a 3-sided range query in $O(\log_B N + K/B)$ time (where $K$ is the number of points reported), supports an update in $O(\log_B N)$ time, and can be built in $O((N/B) \log M/(B/(N/B)))$ time. The rest of the paper will therefore focus on the approximate $K$-threshold problem.

From now on, we will consider only that the query range $[x_1, x_2]$ contains at least $K$ points in $P$. Otherwise (i.e., $P \cap [x_1, x_2]$ has less than $K$ points), there is a simple solution to answer the query in $O((\log_B N))$ time. For this purpose, we only need to create a slightly augmented B-tree (see, for example, [11]) on the points of $P$ such that, the number of data points covered by any interval can be retrieved in $O((\log_B N))$ I/Os. If this number for $[x_1, x_2]$ is below $K$, we simply return $\infty$.

### 1.4 Techniques

It would be natural to index $P$ with a B-tree. Searching the B-tree with query range $[x_1, x_2]$ in a standard way yields $h = O((\log_B N))$ canonical subsets $P_1, \ldots, P_h$ that partition the points of $P \cap [x_1, x_2]$. In the approximate $K$-threshold problem, one source of difficulty is the lack of a clear decomposability property that allows us to deal with each $P_i$. The challenge, however, lies in finding a good $\tau$. Ideally, we would like $q$ to cover exactly $K$ points of $S$, 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 an interesting method called shallow cutting. Unfortunately, a shallow cutting is costly to compute, which explains why the structure of [1] is expensive to construct and update.
individually. We overcome this by precomputing a logarithmic sketch for each \( P_i \), which is a subset of \( P \) containing the points with the lowest score, the 2nd lowest, the 4th lowest, and so on. As it probably has become clearer, by adapting the algorithm of Frederickson and Johnson [7] for finding the \( K \) smallest elements from multiple sorted lists, we manage to find a good \( \tau \) by using just the sketches of \( P_1, \ldots, P_h \).

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 an approximate version where the \( i \)-th point does not have exactly the \( 2^i \)-th lowest score, but has instead the \( \Theta(2^i) \)-th lowest, 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 \( i \).

1.5 Top-\( K \) range reporting without the distinct-score condition

Points have distinct scores in the standard top-\( K \) range reporting problem as defined 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 lowest score \( c \) (among the points satisfying the range condition), return \( K - z \) of them arbitrarily, where \( z \) is the number of points with scores less 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 lower score, and apply our distinct-score structure directly. For the second semantics, we can maintain a separate B-tree where the data points are sorted first by their scores, and then by their values. In this way, after we have found score \( c \) using our distinct-score structure, all the points (covered by the query range) 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. A STATIC STRUCTURE

This section will present a static structure, which explains some ingredients of our dynamic structure in the next section. Henceforth, we use the term point to refer to a real value \( x \) associated with a score (which is denoted with \( \text{score}(x) \), as before). From now on, \( B \) would be interpreted as the number of points that can be stored in a block. Furthermore, we say that two point sets \( D_1, D_2 \) are score-disjoint if no point in \( D_1 \) has the same score as a point in \( D_2 \). All logarithms have base 2 by default.

\[ \begin{array}{cccccccc}
\text{scores of } D_1 & 10 & 17 & 21 & 25 & 28 & 32 & 44 & 57 \\
\text{scores of } A_i & 10 & 17 & 25 & 57 & 57 & 57 & 57 & 57 \\
\end{array} \]

Figure 1: Original array and conceptual array in the proof of Lemma 3 (gray cells represent points in \( \Sigma(D_i) \))

Let us start by reviewing a classic result:

**Lemma 2.** ([7]). Let \( A_1, \ldots, A_h \) be arrays of values from a totally ordered set such that (i) each array is sorted, (ii) the values in each array are distinct, and (iii) the arrays are mutually disjoint. All arrays are in internal memory. Given an integer \( K \leq \sum_{i=1}^{h} |A_i| \), there is a comparison-based algorithm that finds in \( O(h) \) CPU time a value \( \tau \) that is greater than at least \( K \) but at most \( O(K) \) values in \( A_1 \cup \cdots \cup A_h \).

As an immediate corollary, when \( A_1, \ldots, A_h \) are stored in the disk, there is an algorithm that solves the same problem in \( O(h) \) I/Os – just a trivial simulation of the in-memory algorithm suffices. Furthermore, this algorithm accesses only \( O(h) \) elements of the arrays (otherwise, the CPU time of the algorithm in internal memory would not be \( O(h) \)).

Given a value \( \tau \in \mathbb{R} \), define its rank in a point set \( D \), denoted as \( \text{rank}_D(\tau) \), to be the number of points in \( D \) whose scores are at most \( \tau \).

**Definition 1.** The logarithmic sketch \( \Sigma(D) \) of a point set \( D \) is a sequence \( (\Sigma(D_1), \ldots, \Sigma(D_{|D|})) \), where \( \Sigma(D_k) = (0 \leq k \leq |\Sigma(D_k)|) \) is the point in \( D_k \) whose score has rank \( 2^k \).

For a point set \( D_k \), we use \( D(k) \) \((1 \leq k \leq |D|)\) to denote the point with the \( k \)-th lowest score in \( D \). The next lemma shows that in order to compute a suitable \( \tau \), much less information is needed than is required by Lemma 2.

**Lemma 3.** Let \( D_1, \ldots, D_h \) be point sets that are mutually score-disjoint. Given their logarithmic sketches and an integer \( K \leq \sum_{i=1}^{h} |D_i| \), we can find in \( O(h) \) I/Os a value \( \tau \) whose rank in \( D_1 \cup \cdots \cup D_h \) is at least \( K \) and at most \( O(K) \).

**Proof.** For each \( i \in [1, h] \), we construct a conceptual array \( A_i \) of size \( |D_i| \), based on the logarithmic sketch \( \Sigma(D_i) \). Suppose \( \Sigma(D_i) = (x_0, \ldots, x_{|\Sigma(D_i)|-1}) \). The \( k \)-th entry of \( A_i \), equals\(^1\) \( x_{\lfloor \log_2 k \rfloor} \). Figure 1 illustrates an example. It holds that

\[
\text{score}(A_i([k/2])) \leq \text{score}(D_i[k]) \leq \text{score}(A_i[k]) \quad (1)
\]

for all \( k = 1, \ldots, |D_i| \). Specifically, \( \text{score}(D_i[k]) \leq \text{score}(A_i[k]) \) is because the score of \( x_{\lfloor \log_2 k \rfloor} \) has rank \( 2^\lfloor \log_2 k \rfloor \geq k \) in \( D_i \), whereas \( \text{score}(A_i([k/2])) \leq \text{score}(D_i[k]) \) is because the score of \( x_{\lfloor \log_2 [k/2] \rfloor} \) has rank \( 2^\lfloor \log_2 ([k/2]) \rfloor \leq k \) in \( D_i \).

Next, given \( K \), we apply Lemma 2 to \( A_1, \ldots, A_h \). Recall that, in Lemma 2, each input array should be stored in the

\(^1\)In case \( |\log_2 k| \geq |\Sigma(D_i)| \), define \( x_{\lfloor \log_2 k \rfloor} \) to be a dummy point with score \( \infty \).
disk. Apparently, we cannot afford to materialize $A_1, \ldots, A_h$ into the disk because their sizes are large. Interestingly, we can apply Lemma 2 without disk materialization as follows. Every time the algorithm asks for a cell in the array, say $A_i[k]$ for some $i, k$ satisfying $1 \leq i \leq h, 1 \leq k \leq |A_i|$, we probe $\Sigma(D_i)$ to return the score of its $\lfloor \log k \rfloor$-th point. As mentioned earlier, the algorithm finishes in $O(h)$ time in internal memory, implying that we only need to probe $\Sigma(D_1)$, \ldots, $\Sigma(D_h)$ for $O(h)$ times in total. Furthermore, Lemma 2 requires that no two numbers, from either the same array or different arrays, should be identical. To fulfill this requirement, when the scores of two entries $A_i[k]$ and $A_i[k']$ are identical, we break the tie by comparing $i$ and $i'$; if there is still a tie, we compare $k$ and $k'$. Therefore, when the algorithm of Lemma 2 finishes, we have obtained a $\tau$ whose rank in $A_1 \cup \cdots \cup A_h$ falls in $[K, cK]$ for some constant $c$.

In the sequel, we will show that

$$\text{rank}_{A_i}(\tau) \leq \text{rank}_{D_i}(\tau) \leq 2 \text{rank}_{A_i}(\tau)$$

for every $i \in [1, h]$. This will conclude the proof since it implies that the rank of $\tau$ in $D_1 \cup \cdots \cup D_h$ is covered by $[K, 2cK]$.

Set $\alpha = \text{rank}_{A_i}(\tau)$. To prove $\text{rank}_{A_i}(\tau) \leq \text{rank}_{D_i}(\tau)$, we point out: (i) $\text{score}(A_i[\alpha]) \leq \text{score}(A_i[\alpha])$, and (ii) $\text{score}(D_i[k]) \leq \text{score}(A_i[\alpha])$, by (1). Therefore, for all $k \leq \alpha$, $\text{score}(D_i[k]) \leq \text{score}(D_i[\alpha])$. Hence, $\text{rank}_{D_i}(\tau) \geq \alpha = \text{rank}_{A_i}(\tau)$.

To prove $\text{rank}_{D_i}(\tau) \leq 2 \text{rank}_{A_i}(\tau)$, suppose that $2\alpha + 1 \leq |D_i|$; otherwise, the statement is trivially true. Observe: (i) $\text{score}(A_i[\alpha + 1]) > \alpha$, by the definition of $\alpha$, and (ii) $\text{score}(D_i[2\alpha + 1]) \geq \text{score}(A_i[\alpha + 1])$, by (1). Therefore, for all $k > 2\alpha$, $\text{score}(D_i[k]) \geq \text{score}(D_i[2\alpha + 1]) > \alpha$. This implies that $\text{rank}_{D_i}(\tau) \leq 2\alpha = 2 \text{rank}_{A_i}(\tau)$. \hfill $\square$

Structure. We are ready to describe our structure for the approximate $K$-threshold problem. The base tree is a B-tree on the points of $P$, where each leaf node contains at least (most) $B/2$ ($B$) points, and each internal node has at least (most) $f/2$ ($f$) children. All the points are stored at the leaf level. Set $f = B^{1/5}$. For every node $u$ of the B-tree, denote by $S(u)$ the set of points in the subtree rooted at $u$. If $u$ is an internal node, define its multi-slabs as follows. Assuming that the children of $u$ are $u_1, \ldots, u_f$ ordered from left to right, define a multi-slab $u[i, j]$ ($1 \leq i \leq j \leq f$) as the union of the subtrees of $u_i, \ldots, u_j$. Use $S(u[i, j])$ to denote the set of points in $u[i, j]$, i.e., $S(u[i, j]) = S(u_i) \cup \cdots \cup S(u_j)$. We store sketches $\Sigma(S(u[i, j]))$ at each internal node $u$, for all $i, j$ satisfying $1 \leq i \leq j \leq f$.

Query. Given an approximate $K$-threshold query with search range $[x_1, x_2]$, first identify, in a standard way, $h = O(\log B N)$ canonical sets $D_1, \ldots, D_h$ of $P \cap [x_1, x_2]$. For each $D_i$, its logarithmic sketch is either available in the form of $\Sigma(S(u[i, j]))$ for some multi-slab $u[i, j]$, or can be computed in $O(1)$ I/Os from a leaf node. Then, apply Lemma 3 to find the result.

Analysis. The query time is $O(\log B N)$, noticing that the canonical sets can be identified in $O(\log B N)$ I/Os, while applying Lemma 3 takes another $O(\log B N)$ time. We now bound the space consumption of the structure. The base tree obviously uses $O(N/B)$ blocks. Next, we analyze the space occupied by the sketches. Define the level of leaf nodes to be 0, and in general, the parent of a level-$l$ node is at level $l + 1$. Since there are at most $N/(B/2)^{h+1}$ nodes at level $l$, each node stores $f^2$ sketches, and each sketch uses $O\left(\frac{1}{B} \log(Bf^2)\right)$ space, the total space occupied by level-$l$ sketches is

$$O\left(\frac{N}{(B/2)^{h+1}} \cdot f^2 \cdot O\left(\frac{1}{B} \log(Bf^2)\right)\right)$$

(as $f = B^{1/5}$) = $O\left(\frac{N}{B} \cdot \frac{f^2}{B^{2h/5}} \cdot \frac{\log B}{(f/2)^2} \cdot \frac{h}{(f/2)^5} = O(N/B) \frac{h}{(f/2)^6}\right)$.

Summing up all levels, overall, the sketches require space:

$$O(N/B) \cdot \sum_{l=1}^{h} \frac{l}{(f/2)^6} = O(N/B)$$

since $B \geq 64$ implies $f/2 > 1$.

3. MAKING THE STRUCTURE DYNAMIC

This section extends our structure to support an update in $O(\log_B N)$ amortized time, while retaining the space and query bounds. We will also show that the new index can be constructed in $O((N/B) \log_M/B(N/B))$ time.

3.1 Approximate sketch

It is obvious that a single insertion in a point set can invalidate its entire logarithmic sketch, thus making the sketch expensive to maintain. Motivated by this, we resort to a looser version of the sketch:

**Definition 2.** An approximate logarithmic sketch $\Pi(D)$ of point set $D$ is a sequence $(y_0, y_1, y_2, \ldots)$ whose length is at least $|\log |D||$ but at most $1 + |\log |D||$. The rank of $y_k$ ($0 \leq k \leq |\Pi(D)|$) in $D$ is in the range $[2^k, 2^{k+1} - 1]$.

Here is a more intuitive interpretation of the definition. First, imagine that the real domain $\mathbb{R}$ is partitioned into intervals by the scores of the points in the static sketch $\Sigma(D) = (D[1], D[2], \ldots, D[2^i], \ldots)$ of $D$. We call these intervals the fragments induced by $D$. Specifically,

- the first fragment is the interval $(-\infty, \text{score}(D[1]))$;
- for $k$ between 0 and $|\log |D|| - 1$, the $(k + 2)$-nd fragment is $[\text{score}(D[2^k]), \text{score}(D[2^{k+1}]))$; and
- the last fragment is $[\text{score}(D[2^{|\log |D||}])), +\infty)$.

Notice that each score in $\Sigma(D)$ belongs to the fragment on its right. Then, except the first fragment and possibly the last one, every other fragment contributes one number to the approximate sketch $\Pi(D)$. The last fragment can contribute either one or no number. More precisely, the $(k + 2)$-nd fragment ($0 \leq k \leq |\log |D||$) contributes, if it does, the $y_k$ of $\Pi(D)$. The value of $y_k$ can be chosen arbitrarily from the fragment, i.e., it does not even need to be the score of a point in $D$. The first two arrays and the axis in Figure 2 illustrate the computation of an approximate sketch $\Pi(D_i)$.
for a point set \( D_i \). For example, \( y_1 \) of \( \Pi(D_i) \), which equals 22, comes from the third fragment \([17, 25]\).

The following lemma indicates that an approximate sketch serves the purpose of Lemma 3 equally well.

**Lemma 4.** Let \( D_1, \ldots, D_h \) be point sets that are mutually score-disjoint. Given their approximate logarithmic sketches and an integer \( K \leq \sum_{i=1}^{h} |D_i| \), we can find in \( O(h) \) I/Os a value \( \tau \) whose rank in \( D_1 \cup \cdots \cup D_h \) is at least \( K \) and at most \( O(K) \).

**Proof.** For each \( i \in [1, h] \), we construct a conceptual array \( A_i \) of size \( |D_i| \), based on the approximate logarithmic sketch \( \Pi(D_i) \). Let \( \Pi(D_i) \) be a sequence \( (y_0, \ldots, y_{|\Pi(D_i)|-1}) \), as in Definition 2. The \( k \)-th entry of \( A_i \) is a dummy point with score\(^4\) \( y_{|\log \tau|} \). Figure 2 illustrates an example. In general, it holds that \( \text{score}(D_i[k]) \leq \text{score}(A_i[k]) \) for all \( k \in [1, |D_i|] \), since the rank of \( y_{|\log \tau|} \) in \( D_i \) is at least \( 2^{|\log \tau|} \geq k \).

As in the proof of Lemma 3, with the same tie-breaking policy, we can apply Lemma 2 on input \( K \) and \( A_1, \ldots, A_h \), without materializing the arrays. The algorithm of Lemma 2 returns, in \( O(h) \) time, a value \( \tau \) whose rank in \( A_1 \cup \cdots \cup A_h \) falls in \([K, cK]\) for some constant \( c \). Next, we will show that

\[
\text{rank}_{A_i}(\tau) \leq \text{rank}_{D_i}(\tau) \leq 4 \text{rank}_{A_i}(\tau)
\]

for every \( i \in [1, h] \), which will conclude the proof since it implies that the rank of \( \tau \) in \( D_1 \cup \cdots \cup D_h \) is covered by \([K, 4cK]\).

The fact \( \text{rank}_{A_i}(\tau) \leq \text{rank}_{D_i}(\tau) \) can be established in exactly the same way as in the proof of Lemma 3. Next, we will focus on proving \( \text{rank}_{D_i}(\tau) \leq 4 \text{rank}_{A_i}(\tau) \). Set \( \alpha = \text{rank}_{A_i}(\tau) \). If \( |D_i| < \log(\alpha + 1) \), rank \( \text{rank}_{D_i}(\tau) \leq 4 \text{rank}_{A_i}(\tau) \) is trivially true. If \( \alpha = 0 \), \( \tau < \text{score}(A_i[1]) = \text{score}(D_i[1]) \), leading to \( \text{rank}_{D_i}(\tau) = 0 \leq 4 \text{rank}_{A_i}(\tau) \). Next, we consider \( |D_i| \geq 4\alpha \) and \( \alpha \geq 1 \). In this case, \( |\log(\alpha + 1)| < |\Pi(D_i)| \) since

\[
|\log(\alpha + 1)| \leq \log(2\alpha) \leq \log(|D_i|/2) \leq |\log |D_i||.
\]

Therefore, the rank of \( \text{score}(A_i[\alpha + 1]) = y_{|\log(\alpha + 1)|} \) in \( D_i \), by Definition 2, is at most

\[
2^{\log(2\alpha + 1)} - 1 \leq 2^{\log(2\alpha + 1)} - 1 = 4\alpha - 1.
\]

\(^4\)In case \( |\log k| \geq |\Pi(D_i)| \), define \( y_{|\log k|} \) to be \( \infty \).

This implies \( \text{score}(D_i[4\alpha]) > \text{score}(A_i[\alpha + 1]) \geq \tau \). As a result, \( \text{rank}_{D_i}(\tau) < 4\alpha = 4 \text{rank}_{A_i}(\tau) \).

Henceforth, all sketches will refer to approximate logarithmic sketches.

### 3.2 Preliminary: weight-balanced B-tree

We will adopt the weight-balanced B-tree (WBB-tree) of Arge and Vitter [4] as the base tree of our dynamic structure. This subsection reviews the part of the WBB-tree that is sufficient for our discussion.

While an ordinary B-tree determines the balance of a node by its number of children, a WBB-tree determines the balance by its weight, namely, the number of points stored in the subtree rooted at the node. Formally, in a WBB-tree with leaf capacity \( b \) and branching factor \( f \), the weight \( w(u) \) of a level-\( l \) node \( u \) satisfies

\[
bf^l/4 \leq w(u) \leq bf^l
\]

if it is not the root node (recall that leaf nodes are at level 0). If, on the other hand, \( u \) is the root, it requires that \( u \) has at least two children and \( w(u) \leq bf^l \). A node is unbalanced if it violates any of these constraints. By definition, the height of a WBB-tree with \( N \) points is \( O(\log(N/b)) \); an internal node has at least \( f/4 \) and at most \( 4f \) children; and a leaf node stores \( bf \) to \( b \) points.

A crucial property of the WBB-tree is that to make a level-\( l \) node unbalanced, \( \Omega(bf^{l+1}) \) updates must have been performed in its subtree since it was created, if \( u \) is not the root; or \( \Omega(bf^{l+1}) \) if it is the root. This provides considerable convenience in handling the unbalancing of a node \( u \). We adopt the following simple rebalancing strategy: if \( u \) is not the root, we rebuild the whole subtree rooted at its parent, whose weight is at most \( bf^{l+1} \); if \( u \) is the root, we rebuild the whole tree, which contains \( N = O(bf^l) \) points in total. Either way, the strategy guarantees that the cost to reconstruct a subtree of \( Z \) points can be amortized on \( O(Z/f) \) updates; and an update bears at most one such cost at each level. It will be clear later that this suffices to establish the desired bound on the update cost.

### 3.3 Structure

We are now ready to make our structure dynamic. For simplicity, let us assume that \( \log_B N \) does not change; the assumption can be removed by globally rebuilding the whole structure every time \( N \) has been doubled or halved from its value at the moment when the structure was last rebuilt. We also assume that the tree has at least two levels. Given a point set \( D \) and an integer \( k \in [1, |D|] \), define the top-\( k \) points of \( D \) as the \( k \) points in \( D \) with the lowest scores.

The base tree is a WBB-tree with leaf capacity \( b = B\log_B N \) and branching factor \( f = B^{1/5} \). Its height is thus \( O(\log(B^5/N)) = O(\log_B N) \). In each internal node \( u \), we store the following secondary structures:

- \( O(f^2) \) sketches. As before, for each multi-slab \( u[i, j] \), we store a sketch of all points in \( S(u[i, j]) \).
- a multi-way list. It contains the top-\((B^{4/5}/4) \) points from every child of \( u \). Note that a multi-way list can be stored in a single block since \( 4f \cdot B^{4/5}/4 = B \).

![Figure 2: Original array, induced fragments, approximate logarithmic sketch and conceptual array in the proof of Lemma 4 (gray cells represent scores that partition the real domain)](image)
• a one-way list. It contains the top- \((B^{3/5} \log_B N)\)
points of \(S(u)\), in ascending order of scores.

For each leaf node \(v\), store the points of \(S(v)\) in ascending order of scores. For convenience, sometimes we will refer to the first \(B^{3/5} \log_B N\) points of this ordering as the one-way list of \(v\).

Furthermore, we maintain an external priority search tree [3] on the 2d point set converted from \(P\) in the way as explained in Section 1.3. That is, each point \(x \in S\) is mapped to a 2d point \((x, \text{score}(x))\). We refer to the 2d point as the image of \(x\).

**Space.** The WBB-tree and the external priority search tree use \(O(N/B)\) space. For sketches, we follow a similar argument as in the static case: since (i) each internal node carries \(O(f^2)\) sketches, (ii) each level-\(l\) sketch occupies \(O((1/B) \cdot \log(bf^l))\) space, and (iii) the number of level-\(l\) nodes is at most \(2^l/\lfloor Bf \rfloor\), the space consumption of all sketches is

\[
O(\log_B N) \sum_{l=1} \frac{N}{bf^l/4} \cdot O(f^2) \cdot O\left(\frac{1}{B} \log(bf^l)\right)
\]

\[
= O(N/B) \cdot \sum_{l=1} O(\log_B N) \left(\frac{f^2 \log b}{b} + \frac{f^2 \log f}{b} \cdot \frac{l}{f^l}\right)
\]

\[
= O(N/B) \cdot O\left(\frac{f^2 \log b}{b} + \frac{f^2 \log f}{b}\right) = o(N/B)
\]

(as \(f = B^{3/5} > 1\) and \(b = B \log_B N\)).

As the multi-way and one-way lists of an internal node together consume \(O(1 + (1/B) \cdot B^{3/5} \log_B N)\) space, and the number of internal nodes is \(O(N/\log_B N)\), all the lists occupy

\[
O(1 + (1/B) \cdot B^{3/5} \log_B N) \cdot O(N/\log_B N)
\]

\[
= O\left(\frac{N}{bf} + O\left(\frac{NB^{3/5} \log_B N}{Bf}\right)\right)
\]

(as \(b = B \log_B N\))

\[
= O(N/B)
\]

space. Therefore, our structure consumes linear space overall.

**Query.** An approximate \(K\)-threshold query with search range \([x_1, x_2]\) can be answered by the same procedure as in Section 2. First, identify \(O(\log_B N)\) canonical sets of

\[P \cap [x_1, x_2].\]

Sketches are available for all but at most two of the canonical sets at the leaf level. Hence in the second step, for these two canonical sets, we compute their sketches in \(O(\log_B N)\) time by scanning the corresponding leaf nodes once. Finally, apply Lemma 4 to obtain the query result. As each step takes \(O(\log_B N)\) time, the query cost is \(O(\log_B N)\).

### 3.4 Update

This subsection explains how to support insertions and deletions in our structure. We will first discuss how to modify sketches before elaborating the entire update algorithm.

**Updating sketches.** For each number in sketch \(\Pi_D\), we store a counter to indicate its real rank in \(D\). Next, we describe how to update a sketch \(\Pi(S(u[i,j]))\) if a point \(x\) is inserted in or deleted from a multi-slab \(u[i,j]\). First, scan the whole sketch to update the counters. Recall that \(\Pi(S(u[i,j]))\) is a sequence \((y_1, y_2, y_3, \ldots)\) of numbers, where \(y_i\) comes from the \((k + 2)\)-nd fragment induced by \(S(u[i,j])\). For each \(y_i\) in the sequence, increase or decrease its counter by one if \(\text{score}(x) \leq y_i\). Then, if its rank constraint is violated (i.e., the counter is no longer in the range \([2^k, 2^{k+1} - 1]\)), check if the \((k + 2)\)-nd fragment is the last fragment, i.e., \(k = \lfloor \log |S(u[i,j])| \rfloor\). If yes, fix the violation by simply discarding \(y_i\), since the last fragment is allowed to contribute no number to the sketch. Otherwise, recompute \(y_i\) by retrieving a super-set of the top- \((1.5 \times 2^k)\) points of the multi-slab. The super-set has size \(O(2^k)\), with its points fetched in an arbitrary order. After that, select the point with the \((1.5 \times 2^k)\)-th lowest score in the super-set, which can be done in \(O(2^k/B)\) time using the algorithm of [2]. The score of this point becomes the new \(y_i\). Assuming that the multi-way list of node \(u\) is already in memory, the aforementioned super-set can be produced in \(O(2^k/B^{3/5})\) I/Os as follows.

1. If \(1.5 \times 2^k \leq B^{4/5}/4\), obtain the super-set via accessing the multi-way list of \(u\) with no cost, since the multi-way list is in memory.

2. If \(B^{4/5}/4 < 1.5 \times 2^k \leq B^{3/5} \log_B N\), fetch the super-set from the one-way lists of the children of \(u\) whose subtrees compose multi-slab \(u[i,j]\); First, apply Lemma 2 on those one-way lists to obtain a value \(\tau\) whose rank in \(S(u[i,j])\) is at least \(1.5 \times 2^k\) and at most \(O(2^k)\). This can be done in \(O(f)\) I/Os. Then, generate the super-set by collecting all the points of each one-way list whose scores are at most \(\tau\). This costs \(O(f + 1.5 \times 2^k/B) = O(2^k/B^{3/5})\) time, since \(1.5 \times 2^k/B^{3/5} > (B^{4/5}/4)/B^{3/5} = O(f)\).

3. If \(1.5 \times 2^k > \max\{B^{4/5}/4, B^{3/5} \log_B N\}\), issue a 3-sided range query on the external priority search tree. This query retrieves all the points in \(P\) whose images fall in \([x_1, x_2] \times (-\infty, y_{k+1})\), where \([x_1, x_2]\) is the \(x\)-range of multi-slab \(u[i,j]\). Here, \(y_{k+1}\) is the number that succeeds \(y_i\) in \(\Pi(S(u[i,j]))\); in case \(y_i\) is the last number of the sketch, set \(y_{k+1} = \infty\). The 3-sided query returns a super-set as needed because the rank of \(y_{k+1}\) in \(S(u[i,j])\) is at least \(2^{k+1} - 1\) (if \(y_{k+1} \neq \infty\)) or at least \(1.5 \times 2^k\) (if \(y_{k+1} = \infty\)), and (ii) at most \(2^{k+1} - 1\). The cost is \(O(\log_B N + 2^{k+2}/B) = O(2^k/B^{3/5})\).

We have finished explaining how to recompute existing numbers in the sketch. Finally, check if we need to add a new number to \(\Pi(S(u[i,j]))\). Specifically, let \(y_i\) be the last number in \(\Pi(S(u[i,j]))\). If the \((k + 3)\)-rd fragment is no longer the last fragment induced by \(S(u[i,j])\), i.e., \(k + 1 < \lfloor \log |S(u[i,j])| \rfloor\), add \(y_{k+1}\) to the sketch. The value of \(y_{k+1}\) can also be computed with the above procedure.

Now, we analyze the cost of updating sketches. As will be clear shortly, inserting/deleting a point can affect the sketches of \(O(\log_B N)\) nodes, such that \(O(f^2)\) sketches at each node can be modified. There are two types of sketch updates: counter update and number recomputation. At

\[\text{As a special case, when } k = 0, \text{ we replace } 1.5 \times 2^k \text{ with } 1.\]

The same convention is adopted in the sequel.
each node, the counters of the affected sketches can be updated in $O((f^2 \log N)/B) = O(\log_B N)$ time by scanning the $O(f^2)$ sketches once.

Next, we bound the cost of number recomputation. Recall, once again, that a sketch is a sequence $(y_0, y_1, \ldots)$ of numbers. Let us focus on one specific number, say $y_i$. Since the last recomputation of $y_i$, at least $0.5 \times 2^k$ updates must have happened before we need to reconstruct it again. As mentioned earlier, each recomputation requires $O(2^k/B^{3/5})$ I/Os. We amortize this cost over those $0.5 \times 2^k$ updates, so that each update accounts for only $O(1/B^{3/5})$ I/Os. As a sketch has $O(\log N)$ numbers, each update can be charged $O(\log N)$ times with respect to one sketch. Hence, the update can be charged $O(f^2 \log N)$ times with respect to a node. Finally, an update needs to bear cost for $O(\log_B N)$ nodes. Therefore, overall, the amortized cost of an update is

$$O(f^2 \log N) \cdot O(1/B^{3/5}) \cdot O(\log_B N) = O \left( \frac{\log N}{B^{3/5}} \right) \cdot O(\log_B N) = O(\log_B^2 N).$$

**Full update steps.** We now give the complete update algorithm. An insertion/deletion of a point $x$ is carried out in five steps:

1. Insert/delete $x$ in the WBB-tree. We use the term update path to refer to the path from the root to the leaf node containing $x$.

2. Only nodes on the update path may have become unbalanced. Find the highest unbalanced node $u^*$, and follow the rebalancing strategy as in Section 3.2: if $u^*$ is the root, reconstruct the whole tree; otherwise, reconstruct the subtree rooted at the parent of $u^*$. We defer the construction algorithm to Section 3.5.

   If $u^*$ or its parent is the root, the update is complete because the whole tree has been reconstructed. Otherwise, let $U$ be the set of nodes on the update path that have not been reconstructed. Specifically, if $u^*$ does not exist, $U$ includes all the nodes on the update path. If $u^*$ exists, $U$ includes all the proper ancestors of the parent of $u^*$. Perform the following steps on the nodes $u \in U$ in the bottom-up order.

3. Recompute the one-way list of $u$ by merging the one-way lists of its children.

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

5. Load the multi-way list of $u$ into memory. Then, update the sketches of $u$ as described previously.

**Analysis.** To analyze the update cost, we will use in advance the fact that we can construct a subtree in $O((Z/B) \log_{M/B}(Z/B))$ time if the subtree contains $Z$ points (the fact will be proved in the next subsection). Steps 1 and 5 consume $O(\log_B N)$ and $O(\log_B^2 N)$ amortized time, respectively. Next, we will show that Steps 2 to 4 incur $O(\log_B N)$ amortized cost per level. This will establish that the overall update time is $O(\log_B^2 N)$ amortized.

- As mentioned in Section 3.2, if the reconstruction of Step 2 involves $Z$ points, we can amortize the cost over $\Omega(Z/f)$ updates, so that each update bears cost

  $$O \left( \frac{(Z/B) \log_{M/B}(Z/B)}{\Omega(Z/f)} \right) = O \left( \frac{\log_{M/B}(Z/B)}{B^{4/5}} \right) = O(\log_B N).$$

- As each one-way list contains $B^{3/5} \log_B N$ points, the $f$-way merge of Step 3 can be performed in time

  $$O \left( \frac{f B^{3/5} \log_B N \log_{M/B} f}{B} \right) = O \left( \frac{\log_B N \log_{M/B} f}{B^{1/5}} \right) = O(\log_B N).$$

- Finally, Step 4 takes one I/O.

### 3.5 Construction

This subsection describes an $O((Z/B) \log_{M/B}(Z/B))$-time algorithm to reconstruct the subtree of a node, where $Z$ is the number of points in the subtree. Given those $Z$ points, the algorithm outputs a new subtree $T$ in which all the secondary structures have been properly constructed. As $T$ is itself a WBB-tree, its nodes can be built in $O((Z/B) \log_{M/B}(Z/B))$ I/Os [4]. In the sequel, we focus on building the secondary structures.

For a node $u$ in $T$, define the ranked list of $u$, denoted as $rlist(u)$, to be the list of points from $S(u)$ in ascending order of their scores (recall that $S(u)$ is the set of points in the subtree of $u$). 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 $T$. Given $rlist(u)$, we next show how to compute the $O(f^2)$ sketches of $u$ in two steps. The first one scans $rlist(u)$ to generate all the sketch entries of $u$. At this point, those entries are not necessarily grouped by the sketches they belong to. Then, the second step achieves the grouping by sorting. Here 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 $u[i,j]$ ($1 \leq i \leq j \leq 4f$). The counter of $u[i,j]$ indicates how many points in the multi-slab have already been scanned in $rlist(u)$. Every time the counter reaches $1.5 \times 2^k$ for some $k \geq 0$, a tuple $(i, j, k, y)$ is output, where $y$ is the score of the last point scanned. Reading the ranked list obviously takes $O(|rlist(u)|/B)$ I/Os. In addition, we have to write in total $O(f^2 \log Z)$ tuples to the disk, whose overhead is

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

  $$= O(1 + (1/B^{2/5}) \log_B Z).$$

- Given a tuple $(i, j, k, y)$, refer to $(i, j)$ as its key. The second step groups the $O(f^2 \log Z)$ tuples by their keys. Since the number of the distinct keys is $O(f^2)$,
this can be done using the distribution sort algorithm of [2] in time
\[ O \left( 1 + \frac{f^2 \log Z}{B} \log_{M/B}(f^2) \right) \]
\[ = O(1 + (1/B^{2/5}) \log_B Z). \]  
(3)

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

Therefore, the sketches of \( u \) can be computed in \( O(|rlist(u)|/B) + O(1/B^{2/5}) \log_B Z \) I/Os. 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 \( T \) in a bottom-up manner, where the ranked list of a node is obtained by an \( f \)-way merge which combines the ranked lists of its children. All the \( f \)-way merges that take place at the same level perform \( O((Z/B) \log_{M/B} f) \) I/Os. As there are \( O(\log f Z) \) levels, all the ranked lists can be produced in \( O(Z/B) \log_{M/B}(Z/B) \) time. Then, for each internal node \( u \), compute its sketches with sketch-build. Since \( |rlist(u)|/B = \Omega(bf)/B = \Omega(\log_B N) \), the computation of those sketches requires
\[ O(|rlist(u)|/B) + O(1/B^{2/5}) \log_B Z \]
\[ = O(|rlist(u)|/B) + O(\log_B N) = O(|rlist(u)|/B) \]  
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 \( rlist(u) \). Thus, the construction of all the secondary structures is no more expensive than the generation of all the ranked lists. Therefore, the algorithm entails in total \( O((Z/B) \log_{M/B}(Z/B)) \) 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 \( \Omega(\log f Z) \), can be much greater than \( \log_{M/B}(Z/B) \), 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 \( O(\log_{M/B}(Z/B)) \) levels, and deploying each ranked list to build secondary structures for multiple nodes of different levels. In general, suppose that we have obtained all the ranked lists at some level \( l \). We will proceed to compute the ranked lists for the nodes at level \( l + \lambda \), where \( \lambda \) is the maximum integer satisfying
\[ 4f^\lambda < \frac{M}{2B} \cdot \min\{1, f - 1\}. \]  
(5)

As \( B \geq 64, f - 1 = \Omega(1) \), meaning that the right hand side of the above inequality is \( \Omega(M/B) = \Omega(B) \). This guarantees the existence of a valid \( \lambda \). For each level \( l + \lambda \) node \( u \), its ranked list \( rlist(u) \) can be obtained by merging the ranked lists of the descendants of \( u \) at level \( l \). Node \( u \) has at most \( 4f^\lambda \) descendants at level \( l \) because, a node at level \( l \) has at least \( bf^l/4 \) points in its subtree, whereas \( u \) has at most \( bf^{l+\lambda} \) 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 \( 4f^\lambda \) \( M/(2B) \) descendants. After that, the secondary structures of \( u \) can be created as discussed before with \( O(|rlist(u)|/B) \) I/Os, as shown in (4).

Next, we will explain how to build, simultaneously, the secondary structures for all the descendants of \( u \) at levels from \( l + 1 \) to \( l + \lambda - 1 \). Let \( t \) be the number of such descendants, denoted as \( v_1, \ldots, v_t \). Since each level \( (l + i) \) node has at least \( bf^{l+i}/4 \) points in its subtree, the number of level \( (l + i) \) descendants of \( u \) is at most \( 4f^{\lambda+i} \), meaning that
\[ t \leq \sum_{i=1}^{\lambda-1} 4f^{\lambda+i} < 4f^\lambda(f^{-1}) < M/(2B). \]

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 two 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 ascending order of their scores. 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 time of reading \( rlist(u) \) and that of outputting the \( O(f^2 \log Z) \) tuples for each \( v_i \), we know from (2) that the total cost is
\[ O(|rlist(u)|/B) + t \cdot O \left( 1 + (1/B^{2/5}) \log_B Z \right). \]  
(6)

Now, we can carry out the second step of sketch-build for each \( v_i \) *individually*. By (3), doing so for all \( v_i \) requires
\[ t \cdot O \left( 1 + (1/B^{2/5}) \log_B Z \right) \]
I/Os in total. Therefore, the overall cost of computing the sketches of \( v_1, \ldots, v_t \) is dominated by (6). Since \( v_1, \ldots, v_t \) are internal nodes in the subtree of \( u \) and the subtree contains \( |rlist(u)| \) points, \( t \) is bounded by \( O(|rlist(u)|/(bf)) \), meaning that
\[ t \cdot O \left( 1 + (1/B^{2/5}) \log_B Z \right) \leq t \cdot O(\log_B N) \]
\[ = O(|rlist(u)|/(bf)) \cdot O(\log_B N) \]
\[ = O(|rlist(u)|/(Bf)). \]

Therefore, (6) = \( O(|rlist(u)|/B) \).

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 \( rlist(u) \). Similarly, the multi-way lists of all \( v_1, \ldots, v_t \) can be computed with another scan (recall that each multi-way list can be stored in one block). At this point, we have finished constructing the secondary structures of \( v_1, \ldots, v_t \) in totally \( O(|rlist(u)|/B) \) time.

The above analysis shows that, in \( O(|rlist(u)|/B) \) I/Os, we can build the secondary structures for \( u \) and all of its descendants at level \( l + 1 \) or above. This implies that the secondary structures of all the nodes from levels \( l + 1 \) to
$t + \lambda$ can be computed in $O(Z/B)$ time. As the height of the subtree $T$ we are reconstructing is $O(\log_B Z)$, we need to pay the $O(\log_B Z)/\lambda$ times. Since $\lambda$ is the maximum integer satisfying (5), $4\lambda + 1 = \Omega(M/B)$. This means that $\lambda = \Omega(\log_{f}(M/B)) = \Omega(\log_B(M/B))$, indicating $O(\log_B Z)/\lambda = O(\log_{M/B}(Z/B))$. Therefore, the total reconstruction cost of $T$ is $O((Z/B) \log_{M/B}(Z/B))$ for the case $M = \Omega(B^2)$.

**Remark.** As a direct corollary, our structure can be built from scratch in $O((N/B) \log_{M/B}(N/B))$ time. We thus have completed the proof of Lemma 1, and hence, Theorem 1.

**Acknowledgements**

This work was supported in part by (i) projects GRF 4169/09, 4166/10, 4165/11 from HKRGC, and (ii) the WCU (World Class University) program under the National Research Foundation of Korea, and funded by the Ministry of Education, Science and Technology of Korea (Project No: R31-30007).

**References**


