New Results on Two-dimensional Orthogonal Range Aggregation in External Memory

Cheng Sheng
CUHK
Hong Kong
csheng@cse.cuhk.edu.hk

Yufei Tao
CUHK
Hong Kong
taoyf@cse.cuhk.edu.hk

ABSTRACT

We consider the *orthogonal range aggregation* problem. The dataset S consists of N axis-parallel rectangles in $\mathbb{R}^2$, each of which is associated with an integer *weight*. Given an axis-parallel rectangle Q and an aggregate function F, a query reports the aggregated result of the weights of the rectangles in S intersecting Q. The goal is to preprocess S into a structure such that all queries can be answered efficiently. We present indexing schemes to solve the problem in external memory when $F = max$ (hence, *min*) and $F = sum$ (hence, *count* and *average*), respectively. Our schemes have linear or near-linear space, and answer a query in $O(\log_B N)$ or $O(\log_B^2 N)$ I/Os, where B is the disk block size.

Categories and Subject Descriptors

F2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical algorithms and problems—*geometric problems and computations*; H3.1 [**Information storage and retrieval**]: Content analysis and indexing—*indexing methods*

General Terms

Algorithms, theory

Keywords

Indexing, range searching

1. INTRODUCTION

Orthogonal range aggregation is a classic topic in computational geometry and the database area. In this paper, we consider that the aggregate function F is *max* or *sum*, with a particular focus on *max*.

In the *rectangle-intersection-max* problem, the dataset S consists of N rectangles[1] r in $\mathbb{R}^2$, each of which is associated with a *weight* $w(r) \in \mathbb{N}$. Given a rectangle Q, a query reports the maximum weight of all the rectangles in S intersecting Q, namely:

$$\max\{w(r) \mid r \in S \wedge r \cap Q \neq \emptyset\}.$$

Several special instances of the problem have been studied separately:

- *Window-max*, where each rectangle $r \in S$ degenerates to a point. Of special interest to this paper is *3-sided window-max*, which is a restricted version of window-max where Q has the form $[x_1, x_2] \times (-\infty, y]$.
- *Stabbing-max*, where Q degenerates to a point.
- *Segment-intersection-max*, where r (Q) degenerates to a horizontal (vertical) segment.

See Figure 1 for some illustrations.

In a straightforward manner, the above definitions can be adapted to $F = sum$, where the goal becomes finding the total weight of the rectangles in S intersecting Q. The ability of handling *max* and *sum* implies the ability of supporting *min* (which is symmetric to *max*), *count* (a special case of *sum* where all rectangles have a unit weight), and *average* (which can be derived from *count* and *sum*).

1.1 Applications

This subsection gives several applications of orthogonal range aggregation, which sheds some light on its importance in practice. We discuss first $F = max$ and *min*, before extending to other aggregate functions.

In a *spatial database*, each object in S can be the location of a hotel, with the object's weight set to the hotel's rate. A window-min query is *"retrieve the cheapest rate of the hotels in Manhattan"*, where Q is a rectangle describing the area of Manhattan. Sometimes, a data region can be so complex that it needs to be represented as the union of simple geometric shapes like rectangles. An example can be found in the *hurricane inquiry system*, where an object is the region struck by a hurricane. In this case, S consists of the rectangles used to approximate those regions, whereas

[1]All rectangles in this paper are orthogonal.

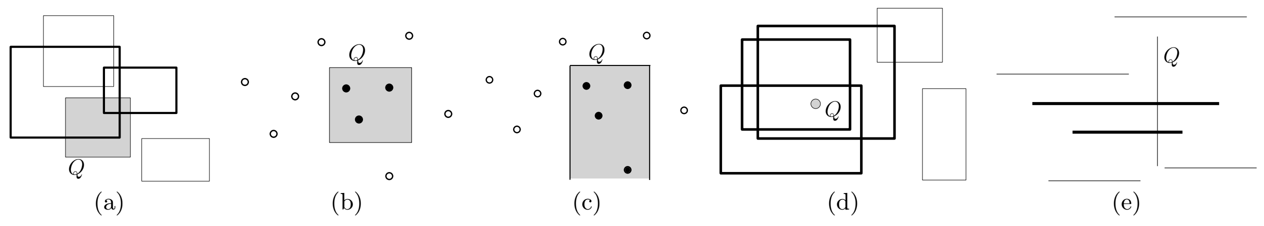

Figure 1: Variants of orthogonal range max search: (a) rectangle-intersection-max, (b) window-max, (c) 3-sided window-max, (d) stabbing max, (e) segment-intersection-max. In each example, the maximum weight of the bold objects is reported.

the weight of a rectangle $r \in S$ is set to the scale of the hurricane hitting the region represented by r. A rectangle-intersection-max query is *"find the most severe scale of all the hurricanes that struck Florida before"*. The query becomes a stabbing-max query if Q is given as a location in Florida.

Segment-intersection-max is useful in *temporal databases*, which manage the historical versions of a dataset evolving with time. Consider a database that stores the balances of bank accounts. Each record has the form $([t_s, t_e], k)$, indicating that a balance started to be k at time t_s and remained so until time t_e. This record can be regarded as a horizontal segment in a 2-d space, where the x- (y-) dimension is time (balance). Let S be a set of such segments, each associated with a weight equal to the monthly premium of medical insurance paid by the account owner. A meaningful query is *"return the highest monthly premium paid by the owners of those accounts whose balances were within* $[\$10k, \$20k]$ *on Dec 1, 2010"*. The query is a vertical segment in the time-balance space.

Rectangle-intersection-max has also been studied in *bi-temporal databases* [17], *OLAP* [15, 16], and *meteorology systems* [21], whereas stabbing-max has been identified as a core operation in *packet classification* [14].

An extensive list of applications for F = *count*, *sum*, and *average* can be obtained by combining the applications in [15, 16, 18, 20]. In fact, all the queries given earlier in italic are still meaningful if they are modified to retrieve the average (e.g., the query in our first application becomes *"retrieve the average rate of the hotels in the Manhattan district"*), whereas computation of the average can be achieved by solving the corresponding *count* and *sum* problems.

1.2 Computation model

Our complexity analysis is under the *external memory* (*EM*) model [5], which has been successful in capturing the I/O characteristics of database algorithms (see a survey in [19]). In this model, the (main) memory has a capacity of M words, whereas the disk has an unbounded size, and is formatted into *blocks* with B words each. We require $B \geq 9$ in our analysis. An I/O transfers a block of data between the disk and memory. Space complexity measures the number of disk blocks occupied, whereas time complexity gauges the number of I/Os performed. *Linear cost* is interpreted as $O(N/B)$ for a dataset cardinality N. In this paper, *poly-logarithmic cost* should be understood as $O(\log_B^c N)$ for some constant c.

We assume that each word has $\log_2 N$ bits, and that each weight fits in $O(1)$ words. These assumptions also exist in the previous work [3, 4, 6, 11, 13, 18] on aggregation problems. We also make the *tall-cache assumption* [3, 8] that $M \geq B^2$. A typical value of B in reality is 1024 (words), in which case the assumption essentially says that the memory should have at least 1 mega words. This is not a demanding requirement for today's machines. In any case, the assumption is needed only to simplify the construction of the proposed structures, while all our space and query bounds hold for any $M \geq 2B$.

1.3 Previous results

There is a rich literature on orthogonal range aggregation in the EM model. When the aggregate function F is *max*, the existing results can be summaized as:

- For window-max, the CRB-tree of Govindarajan et al. [13] answers a query in $O(\log_B^2 N)$ I/Os, and consumes $O(\frac{N}{B} \log_B N)$ space (according to [3]). This has been improved by Agarwal et al. [3], whose structure has linear space, and retains the same query complexity as the CRB-tree. It is unclear whether the two structures can be improved for 3-sided window-max, which can be settled by a modified persistent B-tree of [18, 20] in $O(\log_B N)$ I/Os per query, occupying $O(\frac{N}{B} \log_B N)$ space.
- For stabbing-max, the best linear-space structure is also due to Agarwal et al. [3] and solves a query in $O(\log_B^4 N)$ I/Os. In [4], Agarwal et al. developed an alternative index that has a better query complexity $O(\log_B^2 N)$ but uses $O(\frac{N}{B} \log_B N)$ space.
- For segment-intersection-max, the modified persistent B-tree of [18, 20] can be deployed to process a query in $O(\log_B N)$ I/Os, but its space consumption is $O(\frac{N}{B} \log_B N)$. No linear-space structure is known to have poly-logarithmic query time for this problem.
- Using a reduction explained later, the rectangle-intersection-max problem can be solved in $O(\log_B^2 N)$ I/Os using an indexing scheme with $O(\frac{N}{B} \log_B N)$ space. Again, there is no linear-space structure with poly-logarithmic query cost.

When F = *count* and *sum*, the counterparts of the above problems are equivalent to each other [12]. Thus, it suffices to discuss only rectangle-intersection-count and rectangle-intersection-sum. The CRB-tree [13] settles the former in

	previous	**ours**	**remarks**
3-sided window-max	$(N/B, \log_B^2 N)$ [3] $(\frac{N}{B}\log_B N, \log_B N)$ [18, 20]	$(N/B, \log_B N)$	the result of [3] holds for general window-max.
stabbing-max	$(N/B, \log_B^4 N)$ [3] $(\frac{N}{B}\log_B N, \log_B^2 N)$ [4]	$(N/B, \log_B^2 N)$	
segment-intersection-max	$(\frac{N}{B}\log_B N, \log_B N)$ [18, 20]		
rectangle-intersection-max	$(\frac{N}{B}\log_B N, \log_B^2 N)$ [3, 4, 18, 20]		
rectangle-intersection-count	$(N/B, \log_B N)$ [13]	$\left(\frac{N}{B}\max\{1, \log_B \frac{W}{N}\}, \log_B N\right)$	$W=\sum_{r\in S} w(r)$. Hence, $W=N$ for rectangle-intersection-count, for which our space is $O(N/B)$.
rectangle-intersection-sum	$(\frac{N}{B}\log_B \frac{W\log_2 W}{N}, \log_B N)$ [13]		

Table 1: Comparison of the existing and our results. Each result is in the format of (space, query). All complexities are in big-O and worst case.

linear space and logarithmic query cost. For $F = sum$, Govindarajan et al. [13] gave an alternative version of the CRB-tree that has the same query complexity, but requires $O(\frac{N}{B}\log_B \frac{W\log_2 W}{N})$ space, where W is the total weight of all the objects, namely, $W = \sum_{r\in S} w(r)$.

Observe that there is a gap between the space cost of the two CRB-trees. When each data rectangle has a unit weight (i.e., $W = N$), the *sum* CRB-tree has space $O(\frac{N}{B}\log_B \log_2 N)$, which is worse than the linear space of the *count* CRB-tree. This suggests that the space complexity of the sum CRB-tree may be unnecessarily large for small W. It remains open how to close the gap.

Various heuristic access methods [15, 16, 21] are available for orthogonal range aggregation, and have been empirically shown to work well for selected data and query distributions. However, they are not known to carry any interesting performance guarantee in the worst case.

1.4 Our results

Our main results can be summarized in three theorems:

Theorem 1.1. *For 3-sided window-max, there is a linear-space structure that answers a query in $O(\log_B N)$ I/Os. The structure can be built in $O(\frac{N}{B}\log_B N)$ I/Os.*

Theorem 1.2. *For rectangle-intersection-max, there is a linear-space structure that answers a query in $O(\log_B^2 N)$ I/Os. The structure can be built in $O(\frac{N}{B}\log_B N)$ I/Os.*

Theorem 1.3. *For rectangle-intersection-sum, there is a structure that answers a query in $O(\log_B N)$ I/Os, and occupies $O(\frac{N}{B}\max\{1, \log_B \frac{W}{N}\})$ space, where W is the total weight of all the objects. The structure can be built in $O(\frac{N}{B}\log_B N)$ I/Os.*

Table 1 presents a detailed comparison of our and previous results.

Our techniques, except for several technical constructs, revolve around a compressed structure which we call the *bundled compressed B-tree* (*BCB-tree*), and is designed for the so-called *bundled predecessor problem.* While its precise definition will appear in Section 3, informally speaking, in this problem we are given $b = O(B)$ sets of integers: $P_1, ..., P_b$. The goal is to preprocess them so that, given an integer q, we can efficiently find the predecessor of q in every P_i ($i \le b$) at the same time, i.e., b predecessors need to be reported. Let $K = \sum_i |P_i|$. We observe that if every data integer falls in a domain of size $K \cdot B^{O(1)}$, our structure (i.e., the BCB-tree) only needs to consume space *sub-linear* in K, while guaranteeing logarithmic query cost. Another observation made in this paper is that the BCB-tree presents itself as a powerful weapon in approaching several aggregation problems. Note, however, that the idea of solving a single query in multiple datasets simultaneously is not new; see, for example, [1, 2].

Remark 1. In our problem definitions, the weight of each data object is an integer. This is not necessary for $F = max$, and Theorems 1.1 and 1.2 still hold for real-valued weights as well. All that needs to be done is to map each real-valued weight to an integer, and use a B-tree to index the resulting integers so that we can convert each of them back to its real-valued counterpart in $O(\log_B N)$ I/Os. The same trick, however, does not apply to $F = sum$, for which our methods work for integer weights only (this is also true for the CRB-trees of [13]).

Remark 2. As a corollary of Theorem 1.1, we obtain a linear-space structure with logarithmic query time for the following *segment dragging* problem defined in [10]. The dataset S is a set of N points in $\mathbb{R}^2$. The goal is to build an index on S such that, given a horizontal segment s, we can quickly report the first point hit if we move s downwards. This can be converted to the 3-sided window-max problem, where each point in S is associated with a weight equal to its y-coordinate, and each query is replaced by a 3-sided rectangle whose upper boundary is s, and its lower boundary is open.

Remark 3. Theorem 1.2 is clearly applicable to all the special instances of the rectangle-intersection-max problem. Hence, we improve the stabbing-max structure of [3], which incurs $O(\log_B^4 N)$ query cost.

Remark 4. Theorem 1.3 improves the CRB-tree in a small but interesting way. For rectangle-intersection-count (where $W = N$), the space complexity of our structure automatically reduces to $O(N/B)$. We thus close the space gap left by the CRB-tree. Furthermore, the theorem indicates that, if $W = N \cdot B^{O(1)}$, rectangle-intersection-sum is solvable in logarithmic query time by a linear-space index. This feature is absent in the CRB-tree.

2. PRELIMINARIES

We denote by $[x]$ the set of integers $\{1, ..., x\}$. Recall that, in our problems, each data object can be a point, segment, or rectangle. In any case, by standard tie-breaking techniques, we can assume that all the x- (y-) coordinates of the data objects are distinct. Similarly, we assume that their weights are also distinct. The rest of the section gives some basic facts relevant to our discussion.

Reduction for rectangle-intersection-max. For any intersecting rectangles r and Q, at least one of the following occurs: (i) Q contains a corner of r, (ii) r contains a corner of Q, and (iii) an edge of r intersects an edge of Q. Based on the observation, rectangle-intersection-max can be reduced to a collection of window-max, stabbing-max, and segment-intersection-max problems as follows. From the original dataset S, we create a set S_1 of points, and a set S_2 (S_3) of horizontal (vertical) segments by adding, for each rectangle $r \in S$, its four corners to S_1 and horizontal (vertical) edges to S_2 (S_3). Given a rectangle-intersection-max query with rectangle Q, we execute:

- a window-max query on S_1 using Q itself as the search region;
- four stabbing-max queries on S using the corners of Q as the query points;
- two segment-intersection-max queries on S_2 (S_3), using the vertical (horizontal) edges of Q as the query segments.

The final answer is the maximum of all the weights retrieved.

Integer encoding. Our structures require an encoding scheme that compresses a positive integer x into $O(\log x)$ bits, and meanwhile, permits lossless decompression when given a bit-stream that encodes a list of integers. One such scheme, for instance, is the *gamma Elias code*, which converts x into a binary string of $\lfloor \log_2 x \rfloor$ zeros followed by the binary form of x (e.g., 7 is represented as 00111). In other words, the compressed form of x has no more than $1 + 2\log_2 x$ bits. To decompress a bit string σ, we first count the number z of preceding zeros in σ, and then, take the next $z + 1$ bits as a decompressed value. If σ has not been exhausted, the process is repeated to decompress the next value. The above description does not capture $x = 0$, but this can be easily handled by adding another bit.

KL-divergence inequality. We need the following mathematical fact in our analysis:

LEMMA 2.1 (KL-DIVERGENCE INEQUALITY). *Let* $\{x_1, ..., x_b\}$ *and* $\{y_1, ..., y_b\}$ *be two sets of positive values, and* $X = \sum_i x_i$, $Y = \sum_i y_i$. *It holds that:*

$$\sum_i \left(x_i \log_2 \frac{y_i}{x_i} \right) \le X \log_2 \frac{Y}{X}.$$

PROOF. For each $i \in [b]$, introduce $\alpha_i = x_i/X$ and $\beta_i = y_i/Y$. Then:

$$\begin{aligned}\sum_i \left(x_i \log_2 \frac{y_i}{x_i} \right) &= X \sum_i \left(\alpha_i \log_2 \left(\frac{\beta_i}{\alpha_i} \frac{Y}{X} \right) \right) \\ &= X \sum_i \left(\alpha_i \log_2 \frac{\beta_i}{\alpha_i} \right) + \left(X \log_2 \frac{Y}{X} \right)\end{aligned}$$

We will prove the lemma by showing that $\sum_i (\alpha_i \log_2 \frac{\beta_i}{\alpha_i})$ is always non-positive. Regard $(\alpha_1, ..., \alpha_b)$ as the pdf of a (discrete) random variable α, and $(\beta_1, ..., \beta_b)$ as the pdf of random variable β. Observe that:

$$\begin{aligned}\sum_i \left(\alpha_i \log_2 \frac{\beta_i}{\alpha_i} \right) &= -\sum_i \left(\alpha_i \log_2 \frac{\alpha_i}{\beta_i} \right) \\ &= D_{KL}(\alpha \| \beta)\end{aligned}$$

where $D_{KL}(\alpha \| \beta)$ is the KL-divergence from α to β, and (as a well-known fact) is always non-negative. This completes the proof. $\square$

COROLLARY 2.2. *Let* $x_1, ..., x_b$ *be* b *positive values, and* $X = \sum_i x_i$. *For any* $C > 0$, *it holds that:*

$$\sum_i \left(x_i \log_2 \frac{C}{x_i} \right) \le X \log_2 \frac{Cb}{X}.$$

3. BUNDLED COMPRESSED B-TREE

This section discusses the following *bundled predecessor* problem. Let $P_1, ..., P_b$ be $b \le B$ sets of integer *keys* in $[D]$ where $D = N^{O(1)}$. Each key $k \in P_i$ ($i \in [b]$) is associated with a *label* $\ell(k, i) \in [L]$, where $L = N^{O(1)}$. These labels have a *monotonicity property*: for any keys $k < k'$ in the same P_i, it always holds that $\ell(k, i) \le \ell(k', i)$. Given a value $q \in [D]$, a bundled-predecessor query reports, for every $i \in [b]$, the label associated with the largest key in P_i that is no greater than q (if such a key exists). We refer to $\{P_1, ..., P_b\}$ as a *bundle*, each P_i as a *category*, and D as the *domain size* of the bundle.

Next we describe a structure named *bundled compressed B-tree* (BCB-tree) to solve the above problem. For each key $k \in P_i$ ($i \in [b]$), define $\delta(k, i) = \ell(k, i) - \ell(k', i)$, where k' is the integer in P_i preceding k. If k' does not exist, $\delta(k, i) = \ell(k, i)$. Set $K = \sum_i |P_i|$, and let P be the multi-set that unions all of $P_1, ..., P_b$ (i.e., a key appears in P as many times as the number of categories containing it). Let $k_1, ..., k_K$ be the keys of P in non-descending order. Define $\hat{\delta}(1) = k_1$, and $\hat{\delta}(j) = k_j - k_{j-1}$ for $j > 1$. We create a list Δ of length K as follows. The j-th entry of Δ is a triple $(\hat{\delta}(j), i, \delta(k_j, i))$, where i is such that $k_j \in P_i$. Since there are totally b categories, i can be represented with $\lceil \log_2 b \rceil$ bits. We store $\hat{\delta}(j)$ and $\delta(k_j, i)$ using the gamma Elias code (see Section 2).

Δ precisely captures $P_1, ..., P_b$, but we must scan it from the beginning to restore any key/label of any P_i. To remedy the drawback, we materialize Δ in a blocked manner. Let us define a *fat block* to be 4 consecutive blocks (i.e., $4B$ words). Recall that each triple in Δ corresponds to a key in P, so the tuples can be grouped by their corresponding keys. We make sure that each group (which has at most b tuples) resides in a fat block as follows: adjacent groups are always placed in the same fat block, as long as it still has enough space; otherwise, we put the succeeding group in a new fat block. Since each group occupies at most $3b \leq 3B$ words, at least B words are used in every fat block, except possibly the last one. Denote by $\mathcal{I}(v)$ the minimal interval enclosing all the keys in a fat block v. Clearly all the $\mathcal{I}(v)$ are disjoint.

Each fat block v is associated with a *relay set*, denoted as $\texttt{relay}(v)$, which contains $b+1$ values. To explain, assume $\mathcal{I}(v) = [\alpha, \beta]$. The i-th ($i \leq b$) value in $\texttt{relay}(v)$ equals $\ell(k, i)$, where k is the greatest integer in P_i smaller than α (if k does not exist, store 0 instead). Refer to $\ell(k, i)$ as the *relay-label* of P_i. The last value in $\texttt{relay}(v)$ is α. Note that the relay set allows us to convert each triple $(\hat{\delta}(j), i, \delta(k_j, i))$ in v to $(k_j, i, \ell(k_j, i))$ accurately. As $b \leq B$, $\texttt{relay}(v)$ can be stored in $O(1)$ blocks. The first address of these blocks is kept in v, so that we can load $\texttt{relay}(v)$ after having accessed v. Finally, we create a B-tree[2] on the $\mathcal{I}(v)$ of all fat blocks v. The B-tree, the fat blocks and their relay sets constitute the BCB-tree.

To answer a bundled-predecessor query q, first descend to the fat block v whose $\mathcal{I}(v) = [\alpha, \beta]$ covers q. Then, we load $\texttt{relay}(v)$, and set $l[i]$ to the relay-label of P_i ($i \in [b]$) in $\texttt{relay}(v)$. Next we scan all the triples in v that correspond to keys at most q. For each triple $(\hat{\delta}(j), i, \delta(k_j, i))$ scanned, increase $l[i]$ by $\delta(k_j, i)$. Finally, $l[1], ..., l[b]$ are returned as the answers.

LEMMA 3.1. *For $b \leq B$, the BCB-tree consumes*

$$O\left(\frac{K}{B \log N}\left(1 + \log \frac{\max\{L, D, K\}}{K} + \log b\right)\right)$$

space, and answers a bundled-predecessor query in $O(\log_B K)$ I/Os. If all the keys have been sorted, the tree can be built in $O(K/B)$ I/Os.

PROOF. It suffices to consider $L \geq K$ and $D \geq K$. Let us first bound the number of bits in Δ. Recall that each triple in Δ has the form $(\hat{\delta}(j), i, \delta(k_j, i))$. Apparently, $O(K \log_2 b)$ bits are required to encode the i-fields of all the triples. In the sequel, we focus on the other fields.

Consider any P_i for some $i \in [b]$; and set $n_i = |P_i|$. Let $k'_1 < ... < k'_{n_i}$ be the keys in P_i. Hence, $\delta(k'_1, i) = \ell(k'_1, i)$, and $\delta(k'_j, i) = \ell(k'_j, i) - \ell(k'_{j-1}, i)$ for $j > 1$. The number of bits required to store the $\delta(k'_j, i)$ of all $j \in [n_i]$ equals $O(n_i + \sum_j \log_2 \delta(k'_j, i))$. We have

$$\begin{aligned}\sum_j \log_2 \delta(k'_j, i) &= \log_2 \prod_j \delta(k'_j, i) \\ &\leq \log_2 (L/n_i)^{n_i} = n_i \log_2(L/n_i)\end{aligned}$$

where the inequality used the fact that $\sum_j \delta(k'_j, i) = \ell(k'_{n_i}, i) \leq L$.

Hence, the number of bits to encode the $\delta(k'_j, i)$ of all j, i is bounded by $O(\sum_i (n_i + n_i \log_2(L/n_i)))$. As $\sum_i n_i = K$, a direct application of Corollary 2.2 gives:

$$\sum_i (n_i + n_i \log_2(L/n_i)) = O(K(1 + \log(Lb/K))).$$

An analogous argument shows that $O(K(1 + \log(D/K)))$ bits are sufficient to encode the $\hat{\delta}(j)$ of all j. As each fat block (except possibly the last one) packs at least $B \log N$ bits of Δ, the total number of blocks is

$$O\left(\frac{K}{B \log N}\left(1 + \log \frac{\max\{L, D\}}{K} + \log b\right)\right).$$

The relay sets, as well as the B-tree, increase the space by only a constant factor.

As both L and D are $N^{O(1)}$, the height of the BCB-tree is bounded by

$$O\left(\log_B\left(\frac{K}{B \log N}\log(LDb/K)\right)\right) = O(\log_B K).$$

This is also the query cost. Finally, it is straightforward to build the structure bottom-up in $O(K/B)$ I/Os. $\square$

Remark 1. When $L = B^{O(1)}K$ and $D = B^{O(1)}K$, the space complexity is bounded by $O(\frac{K}{B \log_B N})$.

Remark 2. In the above discussion, we assumed that each key of a category is associated with a single label. Sometimes, it may be useful to associate $k \in P_i$ with a constant number c of labels $\ell_1(k, i), ..., \ell_c(k, i)$, where $\ell_j(k, i) \in [L_j]$ for each $j \in [c]$, and $L_j = N^{O(1)}$. In this case, given an integer $q \in [D]$, a bundled-predecessor query reports, for each $i \in [b]$, all the c labels associated with the largest integer in each P_i that is no greater than q. The BCB-tree can be easily extended to support such queries. Lemma 3.1 still holds, except that L should be set to $\max_{i=1}^{c} L_i$.

4. THREE-SIDED WINDOW-MAX

This section tackles the 3-sided window-max problem. Let S be a set of N points in $\mathbb{R}^2$. Each point $p \in S$ is associated with a weight $w(p) \in \mathbb{N}$. Given a 3-sided rectangle $Q : [x_1, x_2] \times (-\infty, y]$, the goal is to report the maximum $w(p)$ of all points $p \in S \cap Q$.

4.1 The first structure

This subsection describes a structure with query overhead $O(\log_B^2 N)$, which will be improved in Section 4.2.

Structure. The base tree is a B-tree $\mathcal{T}$ on the x-coordinates of the points in S. Let $root(\mathcal{T})$ be the root of $\mathcal{T}$. Given a node u of $\mathcal{T}$, denote by $\rho(u)$ the parent node of u (if

[2]All B-trees in this paper have a leaf capacity of B. Unless otherwise stated, the internal fanout is also B.

$u = root(\mathcal{T})$, $\rho(u) = \emptyset$). Let S_u be the set of data points in the subtree of u. Set $K_u = |S_u|$.

Given the weight w of a point in S_u, we define the *u-rank* of w as λ, if w is the λ-th smallest (among the weights of the points) in S_u. For a point set X and a point p, let $Y(X, p)$ be the set of points in X whose y-coordinates are at most that of p. Also, we may refer to the largest weight of all the points in X simply as *the maximum weight in* X.

Each internal node u is associated with two BCB-trees Γ_u and Λ_u, except for $root(\mathcal{T})$, where only $\Gamma_{root(T)}$ is needed. Γ_u is employed to find the maximum weight, say w, of the points in $S_u \cap Q$. The retrieved value of w, however, is its u-rank. The usage of Λ_u is to convert the u-rank of w to its $\rho(u)$-rank.

Let $u_1, ..., u_B$ be the child nodes of u. Γ_u is built on a bundle $\{P_1(u), ..., P_B(u)\}$ that has domain size K_u. Consider any value $k \in [K_u]$. Let p be the point in S_u satisfying $|Y(S_u, p)| = k$, i.e., p has the k-th smallest y-coordinate in S_u. Assume that p is in the subtree of u_i for some $i \in [B]$. We assign k to category $P_i(u)$. Meanwhile, k is associated with two labels $\ell_{\texttt{rank}}(k, i)$ and $\ell_{\texttt{y}}(k, i)$. Specifically, $\ell_{\texttt{rank}}(k, i)$ is the u-rank of the maximum weight in $Y(S_{u_i}, p)$, while $\ell_{\texttt{y}}(k, i)$ is set to $|Y(S_{u_i}, p)|$. In the sequel, when referring to a label, we will omit the category if it can be inferred from the context. For example, $\ell_{\texttt{rank}}(k, i)$ and $\ell_{\texttt{y}}(k, i)$ will be abbreviated as $\ell_{\texttt{rank}}(k)$ and $\ell_{\texttt{y}}(k)$, respectively.

Λ_u is created on a special bundle that has only a single category with domain size K_u. Each $k \in [K_u]$ has a label $\ell_{\texttt{p-rank}}(k)$ that equals the $\rho(u)$-rank of w, where w is the k-th smallest weight in S_u. Finally, for each point $p \in S$ in a leaf node z of $\mathcal{T}$, we store the $\rho(z)$-rank of $w(p)$ along with p.

Query. To answer a query $Q : [x_1, x_2] \times (-\infty, y]$, our algorithm first executes a *downward* step, followed by an *upward* step. In the downward step, we traverse (at most) two root-to-leaf paths, and compute a candidate weight at each node accessed. The upward step then ascends the same paths to merge those candidate weights into the final result.

The downward phase maintains an invariant:

> *Prior to accessing a node u, we should have obtained the number $C(u)$ of points in S_u whose y-coordinates are no greater than y.*

At the beginning, $C(root(\mathcal{T}))$ is the total number of points in S with y-coordinates at most y, and can be determined in $O(\log_B N)$ I/Os using a B-tree. In general, let u be an internal node being accessed. Naturally, u corresponds to an interval $\mathcal{I}(u) \subseteq \mathbb{R}$, which encloses all the real values that may reside in the subtree of u. Let $u_1, ..., u_B$ be the child nodes of u. Assume that x_1 and x_2 are covered by $\mathcal{I}(u_\alpha)$ and $\mathcal{I}(u_\beta)$, respectively (we assume that both α and β exist; otherwise, the algorithm can be modified in a straightforward manner). Perform a bundled-predecessor query with search key $C(u)$ on Γ_u, which returns a pair of labels $(l_{\texttt{rank}}[i], l_{\texttt{y}}[i])$ for each category $P_i(u)$ $(i \in [B])$. Set $\gamma(u) = \max_{\alpha < i < \beta} l_{\texttt{rank}}[i]$. In case $\beta <= \alpha + 1$, $\gamma(u) = 0$. Note that $\gamma(u)$ is the u-rank of the maximum weight of the points in $(S_{u_{\alpha+1}} \cup ... \cup S_{u_{\beta-1}}) \cap Q$. Setting $C(u_\alpha) = l_{\texttt{y}}[\alpha]$ and $C(u_\beta) = l_{\texttt{y}}[\beta]$, we descend to u_α and u_β, respectively. The downward step terminates when the leaf level is reached.

The upward step also keeps an invariant:

> *Prior to backtracking from u to $\rho(u)$, $\gamma(u)$ must be equal to the $\rho(u)$-rank of the maximum weight in $S_u \cap Q$.*

This step starts from the (at most) two leaf nodes where the downward phase ended. For each such leaf node z, simply check all the points in $S_z \cap Q$, and set $\gamma(z)$ to the $\rho(z)$-rank of the maximum weight of those points (recall that the $\rho(z)$-rank is explicitly stored in z). In general, consider that the upward step has backtracked to an internal node u from two child nodes u_α and u_β. We update $\gamma(u) = \max\{\gamma(u), \gamma(u_\alpha), \gamma(u_\beta)\}$. Note that $\gamma(u)$ now equals the u-rank of the maximum weight w in $S_u \cap Q$. So if $u = root(\mathcal{T})$, $\gamma(u)$ can be used to restore w in $O(\log_B N)$ I/Os with a B-tree. Otherwise, $\gamma(u)$ needs to be converted to the $\rho(u)$-rank of w. For this purpose, we query Λ_u using $\gamma(u)$ as the search key, and set $\gamma(u)$ to the label retrieved from Λ_u. The algorithm then backtracks to $\rho(u)$.

As $O(\log_B N)$ I/Os are performed to search the BCB-trees at each level of $\mathcal{T}$, the query complexity is $O(\log_B^2 N)$.

4.2 The improved structure

To obtain logarithmic query cost, we aim at spending only constant I/Os at each level of $\mathcal{T}$. Naturally, we resort to fractional cascading, whose application, however, results in super-linear space. To keep the space linear, we must squeeze $\omega(B)$ pointers in a block. Fortunately, this can be achieved this by utilizing BCB-trees for pointer compression.

Structure. We now clarify the necessary modification on the structure of the previous subsection. Let u be an internal node $\mathcal{T}$ with child nodes $u_1, ..., u_B$. Recall that, in the BCB-tree Γ_u, each $k \in [K_u]$ has been associated with labels $\ell_{\texttt{rank}}(k)$ and $\ell_{\texttt{y}}(k)$. We give it two more labels $\ell_{\texttt{c-addr}}(k)$ and $\ell_{\texttt{addr}}(k)$ defined as below. Assume that the point $p \in S_u$ with $|Y(S_u, p)| = k$ is in the subtree of u_i for some $i \in [B]$. If u_i is a leaf node, $\ell_{\texttt{c-addr}}(k)$ equals the address of u_i. Otherwise, $\ell_{\texttt{c-addr}}(k)$ is the address of the (unique) fat block v in Γ_{u_i} such that $\mathcal{I}(v)$ covers $\ell_{\texttt{y}}(k)$ (recall that $\mathcal{I}(v)$ is the minimal interval enclosing all the keys in v). As for $\ell_{\texttt{addr}}(k)$, it is the address of the fat block v in Λ_u whose $\mathcal{I}(v)$ covers $\ell_{\texttt{rank}}(k)$. Specially, if $u = root(\mathcal{T})$, label $\ell_{\texttt{addr}}(k)$ is unnecessary.

We also need to slightly augment Λ_u if $\rho(u) \neq root(\mathcal{T})$. In Λ_u, each $k \in [K_u]$ currently has only a label $\ell_{\texttt{p-rank}}(k)$. We associate it with another label $\ell_{\texttt{p-addr}}(k)$, which equals the address of the fat block v in $\Lambda_{\rho(u)}$ whose $\mathcal{I}(v)$ covers $\ell_{\texttt{p-rank}}(k)$.

Finally, for each leaf node z of $\mathcal{T}$, we augment each point p in z with the address of the fat block v in $\Lambda_{\rho(z)}$ whose $\mathcal{I}(v)$ covers the $\rho(z)$-rank of $w(p)$.

Query. We continue to elaborate the changes to the query algorithm. The downward step now maintains an extra invariant:

Prior to accessing an internal node u, we should know the address of the fat block v in Γ_u whose $\mathcal{I}(v)$ covers $C(u)$. Denote that address as $A_\Gamma(u)$.

At $root(\mathcal{T})$, we find $A_\Gamma(root(\mathcal{T}))$ by simply searching Γ_u with $C(root(\mathcal{T}))$ in $O(\log_B N)$ I/Os. In general, given $A_\Gamma(u)$, we proceed as follows at an internal node u with child nodes $u_1, ..., u_B$. First, we retrieve, in $O(1)$ I/Os (by accessing the fat block of Γ_u at address $A_\Gamma(u)$ and its relay set), directly the result of the bundled-predecessor query on Γ_u with search key $C(u)$. The result contains four labels $l_{\texttt{rank}}[i], l_{\texttt{y}}[i], l_{\texttt{c-addr}}[i], l_{\texttt{addr}}[i]$ for each $P_i(u)$ ($i \in [B]$). As before, set $\gamma(u) = \max_{\alpha<i<\beta} l_{\texttt{rank}}[i]$ (see Section 4.1 for the meanings of α and β). We also need to remember an address $A_\Lambda(u) = l_{\texttt{addr}}[i^*]$, where $i^* = \arg\max_{\alpha<i<\beta} l_{\texttt{rank}}[i]$ (in case $\beta \le \alpha + 1$, $A_\Lambda(u) = \emptyset$). Note that $A_\Lambda(u)$ references the fat block v in Λ_u whose $\mathcal{I}(v)$ covers $\gamma(u)$. At this moment, we are ready to descend to u_α and u_β, setting $A_\Gamma(u_\alpha) = l_{\texttt{c-addr}}[\alpha]$ and $A_\Gamma(u_\beta) = l_{\texttt{c-addr}}[\beta]$, respectively.

The upward step also keeps one more invariant:

Prior to backtracking from u to $\rho(u) \ne root(\mathcal{T})$, $A_\Lambda(u)$ must have been set to the address of the fat block v in $\Lambda_{\rho(u)}$ whose $\mathcal{I}(v)$ covers $\gamma(u)$ (review Section 4.1 for the meaning of $\gamma(u)$ at this stage).

This can be trivially done if z is a leaf node, where the required address is explicitly stored. In general, suppose we have backtracked to u from child nodes u_α and u_β. If u is the root of $\mathcal{T}$, the algorithm continues in the way as described in Section 4.1. Otherwise, as before, set $\gamma(u) = \max\{\gamma(u), \gamma(u_\alpha), \gamma(u_\beta)\}$; in case now $\gamma(u)$ equals $\gamma(u_\alpha)$ (or $\gamma(u_\beta)$), we reset $A_\Lambda(u)$ to $A_\Lambda(u_\alpha)$ (or $A_\Lambda(u_\beta)$). Then, we retrieve, using O(1) I/Os directly the result of the bundled-predecessor query on Λ_u with search key $\gamma(u)$, i.e., a pair of labels $(l_{\texttt{p-rank}}, l_{\texttt{p-addr}})$. After setting $\gamma(u) = l_{\texttt{p-rank}}$ and $A_\Lambda(u) = l_{\texttt{p-addr}}$, we backtrack to $\rho(u)$.

Analysis. The query cost is $O(\log_B N)$ because $O(1)$ I/Os are spent at each level, except for $root(\mathcal{T})$ where $O(\log_B N)$ I/Os are performed. As for the space, $\mathcal{T}$ itself obviously has linear size. The subsequent discussion will show that the BCB-trees of all nodes at the same level of $\mathcal{T}$ occupy $O(\frac{N}{B \log_B N})$ space in total, which implies that all the secondary structures require only linear space.

Consider an internal node u with child nodes $u_1, ..., u_B$. It suffices to prove that (i) Γ_u, and (ii) $\Lambda_{u_1}, ..., \Lambda_{u_B}$ together consume $O(\frac{K_u}{B \log_B N})$ space. Let us first observe a trivial space bound of $O(K_u/B)$ for both (i) and (ii).

In Γ_u, each $k \in [K_u]$ is associated with labels: $\ell_{\texttt{rank}}(k)$, $\ell_{\texttt{y}}(k), \ell_{\texttt{c-addr}}(k)$, and $\ell_{\texttt{addr}}(k)$. It is easy to see that $\ell_{\texttt{rank}}(k) \in [K_u]$, and $\ell_{\texttt{addr}}(k)$ belongs to a domain of size $O(K_u/B)$. As the space of $\Gamma_{u_1}, ..., \Gamma_{u_B}$ is bounded by $O(K_{u_1}/B), ..., O(K_{u_B}/B)$ respectively, $\ell_{\texttt{c-addr}}(k)$ falls in a domain of size $\sum_i O(K_{u_i}/B) = O(K_u/B)$. Similarly, $\ell_{\texttt{y}}(k)$ is in a domain of size $\max_i O(K_{u_i}) = O(K_u)$. Therefore, by Lemma 3.1, the space of Γ_u is bounded by $O(\frac{K_u}{B \log_B N})$.

In Λ_{u_i} ($i \in [B]$), each $k \in [K_{u_i}]$ is associated with two labels: $\ell_{\texttt{p-rank}}(k)$ and $\ell_{\texttt{p-addr}}(k)$. Clearly, $\ell_{\texttt{p-rank}}(k) \in [K_u]$ and $\ell_{\texttt{p-addr}}(k)$ belongs to a domain of size $O(K_u/B)$. Hence, by Lemma 3.1, the space of all $\Lambda_{u_1}, ..., \Lambda_{u_B}$ is at most

$$O\left(\sum_i \frac{K_{u_i}}{B \log N}\left(\log \frac{K_u}{K_{u_i}}\right)\right)$$

which, by Corollary 2.2, is bounded by $O(\frac{K_u}{B \log_B N})$.

Construction. We will complete the proof of Theorem 1.1 by explaining how to build our structure in $O(\frac{N}{B}\log_B N)$ I/Os, using B blocks of memory. The construction of $\mathcal{T}$ is straightforward. Hence, we focus on the BCB-trees Γ_u and Λ_u of the internal nodes u of $\mathcal{T}$. Let u_1, ..., u_B be the child nodes of u.

To create Λ_u, we arrange S_u into a list J_u such that (i) the points in J_u are sorted in ascending order of their weights, and (ii) if $u \ne root(\mathcal{T})$, each point $p \in J_u$ has a label equal to the $\rho(u)$-rank of $w(p)$. At $root(\mathcal{T})$, $J_{root(\mathcal{T})}$ is obtained by sorting in $O(\frac{N}{B}\log_B N)$ I/Os. In general, after J_u is ready, we can produce $J_{u_1}, ..., J_{u_B}$ by partitioning J_u in $O(K_u/B)$ I/Os. $\Lambda_{u_1}, ..., \Lambda_{u_B}$ can then be built in $O(K_u/B)$ I/Os by a synchronous scan of $J_{u_1}, ..., J_{u_B}$ and, if $u \ne root(\mathcal{T})$, also Λ_u. Totally, $O(N/B)$ I/Os are performed at each non-root level of $\mathcal{T}$.

Γ_u is constructed in two phases. The first one arranges S_u into a list J'_u where points are sorted in ascending order of their y-coordinates. Each $p \in J'_u$ is associated with:

- a label $\sigma_u(p)$ that equals the u-rank of the maximum weight in $Y(J'_u, p)$;
- if $u \ne root(\mathcal{T})$, another label $\hat{\sigma}_u(p)$ that equals the $\rho(u)$-rank of the maximum weight in $Y(J'_u, p)$.

$J'_{root(\mathcal{T})}$ can be computed in $O(\frac{N}{B}\log_B N)$ I/Os. First, sort the points of S_u by their weights, to obtain the u-rank of $w(p)$ for every $p \in S_u$. Then, sort S_u another time but in ascending order of y-coordinates. Finally, scan S_u in the new sorted order. As we go, monitor the greatest u-rank λ of all the points already seen, and assign λ to the $\sigma_u(p)$ of the last point p scanned.

In general, after J'_u is available, $J'_{u_1}, ..., J'_{u_B}$ can be generated in $O(K_u/B)$ I/Os as follows:

1. Create an array MAP_u of size K_u. For each $j \in [K_u]$, $MAP_u[j]$ equals the u_i-rank of $w(p)$, where p is the point having the j-th smallest weight in S_u, and u_i is the child node that contains p in its subtree. This array can be obtained in $O(K_u/B)$ I/Os by a scan of J_u (which was produced in building Λ_u).
2. Divide J'_u into $J'_{u_1}, ..., J'_{u_B}$ with $O(K_u/B)$ I/Os. Points of each J'_{u_i} are now in ascending order of y-coordinates, but their σ_u and $\hat{\sigma}_u$ labels are not ready yet.
3. Scan each J'_{u_i} (in its order) separately to decide the $\hat{\sigma}_{u_i}(p)$ of each $p \in J'_{u_i}$. Specifically, during the scan, we monitor the largest u-rank of the points already encountered (the u-ranks were inherited from J'_u), and assign it to the $\hat{\sigma}_{u_i}(p)$ of the last point p seen.

4. Go through $J'_{u_1}, ..., J'_{u_B}$ and MAP_u once to determine all $\sigma_{u_i}(p)$, where $p \in J'_{u_i}$ and $i \in [B]$. For each J'_{u_i}, monitor the maximum $\hat{\sigma}_{u_i}(p')$ of all the points $p' \in J'_{u_i}$ already seen. Use λ_i to represent that maximum. For the last point $p \in J'_{u_i}$ scanned, set $\sigma_{u_i}(p) = MAP_u[\lambda_i]$. Synchronize the accesses to $J'_{u_1}, ..., J'_{u_B}$ and MAP_u appropriately so that the entire process finishes in $O(K_u/B)$ I/Os.

Thus, the first phase performs $O(N/B)$ I/Os for each level of $\mathcal{T}$, entailing $O(\frac{N}{B} \log_B N)$ I/Os overall.

The second phase constructs Γ_u bottom-up. If $u_1, ..., u_B$ are leaf nodes, Γ_u can be built by one scan of the leaf level of Λ_u, while keeping $u_1, ..., u_B$ in memory. Otherwise, we create Γ_u in $O(K_u/B)$ I/O by synchronously scanning $\Gamma_{u_1}, ..., \Gamma_{u_B}$, $J'_{u_1}, ..., J'_{u_B}$, and Λ_u (ignore Λ_u if $u = \mathit{root}(\mathcal{T})$). The cost in either case is $O(K_u/B)$ I/Os. Therefore, the second step performs $O(N/B)$ I/Os at each level of $\mathcal{T}$, and thus, $O(\frac{N}{B} \log_B N)$ I/Os in total.

5. SEGMENT-INTERSECTION-MAX

Let S be a set of N horizontal segments in $\mathbb{R}^2$. Each segment $s \in S$ carries a weight $w(s) \in \mathbb{N}$. Given a vertical segment $Q : x \times [y_1, y_2]$, the goal is to report the maximum weight of the segments in S intersecting Q.

Bundled 1-d window-max. Let $V_1, ..., V_b$ be b sets of 1-d points in $\mathbb{R}$. Set $K = \sum_i |V_i|$. Each point $p \in V_i$ is associated with an integer weight $w(p)$. Given an interval $[x_1, x_2] \subseteq \mathbb{R}$, a *bundled window-max* query reports, for each $i \in [b]$, the maximum weight of the points of $V_i \cap [x_1, x_2]$.

LEMMA 5.1. *For $b = B^{1-\epsilon}$ (where ϵ is a constant satisfying $0 < \epsilon < 1$), there is a linear-space structure that answers a bundled 1-d window-max query in $O(\log_B K)$ I/Os. The structure can be constructed in $O(K/B)$ I/Os if all the points in $V_1 \cup ... \cup V_b$ have been sorted.*

PROOF. Let $V = V_1 \cup ... \cup V_b$. Build a B-tree $\mathcal{T}$ with fanout $f = B^{\epsilon/2}$ on V. Each node u of $\mathcal{T}$ naturally corresponds to an interval $\mathcal{I}(u) \subseteq \mathbb{R}$ (defined in the same way as in Section 4.1). At the leaf level, keep with each point p the i such that $p \in V_i$. Consider u as an internal node of $\mathcal{T}$ with child nodes $u_1, ..., u_f$. Let set S_u include the points of V in the subtree of u, and $S_u(i, j) = S_{u_i} \cup ... \cup S_{u_j}$ where $1 \le i \le j \le f$. For each such i, j, store an array $A[i, j]$ of size b, where the k-th ($k \in [b]$) entry $A[i, j, k]$ equals the maximum weight of the points in $S_u(i, j) \cap V_k$. These arrays have no more than $f^2 b = B$ values in total and hence can be stored in 1 block.

Given a query interval $[x_1, x_2]$, first initiate in memory a size-b array γ with $\gamma[k] = -\infty$ for all $k \in [b]$. Then, visit two root-to-leaf paths of $\mathcal{T}$ to reach the leaf nodes z_1, z_2 such that $x_1 \in \mathcal{I}(z_1)$ and $x_2 \in \mathcal{I}(z_2)$. Let u be an internal node accessed. Assume $x_1 \in \mathcal{I}(u_\alpha)$ and $x_2 \in \mathcal{I}(u_\beta)$ for some $\alpha, \beta \in [f]$. In case α (β) does not exist, set it to 0 ($f + 1$). If $\beta \ge \alpha + 2$, retrieve array $A[\alpha + 1, \beta - 1]$, and set $\gamma[k] = \max\{\gamma[k], A[\alpha + 1, \beta - 1, k]\}$ for each $k \in [b]$. At z_1 (the case of z_2 is similar), if a point $p \in V_k$ (for some $k \in [b]$) has weight greater than $\gamma[k]$, set $\gamma[k] = w(p)$. The final $\gamma[1], ..., \gamma[b]$ are returned.

The total space is linear because each internal node of $\mathcal{T}$ is associated with only one extra block. The height of $\mathcal{T}$ is $O(\log_B K)$. A query performs $O(1)$ I/Os at each level, resulting in $O(\log_B K)$ I/Os overall. If V has been sorted, $\mathcal{T}$ (as well as the arrays of the internal nodes) can be constructed bottom-up in $O(K/B)$ I/Os. $\square$

Segment-intersection-max. Our structure is similar to the *external interval tree* of [9], but with two main differences. First, the base structure $\mathcal{T}$ has an (internal) fanout $f = B^{1/4}$. Second, the secondary structures of the nodes in $\mathcal{T}$ are the indexes developed earlier in this paper.

Specifically, $\mathcal{T}$ is a B-tree on the x-coordinates of the endpoints of the segments in S. Associate each endpoint with the segment it belongs to. As before, each node u of $\mathcal{T}$ naturally corresponds an interval $\mathcal{I}(u)$, which in the context here represents a vertical *slab* $\mathcal{I}(u)$ in $\mathbb{R}^2$. Assign each segment $s \in S$ to the lowest node u whose $\mathcal{I}(u)$ covers s. Denote by G_u the set of segments assigned to u.

Let $u_1, ..., u_f$ be the child nodes of u. Define a *multi-slab* $\mathcal{I}(u[i, j]) = \mathcal{I}(u_i) \cup ... \cup \mathcal{I}(u_j)$ for i, j satisfying $1 \le i \le j \le f$. Process each segment $s \in G_u$ as follows. Let α, β be such that $\mathcal{I}(u_\alpha)$ and $\mathcal{I}(u_\beta)$ contain the left and right endpoints of s, respectively. Then:

- If $\beta \ge \alpha + 2$, add the y-coordinate of s to a set $G_u(\alpha + 1, \beta - 1)$, after associating the coordinate with $w(s)$.
- Insert the left and right endpoints of s to sets $G_u^{\sqsupset}(\alpha)$ and $G_u^{\sqsubset}(\beta)$ respectively, after associating both endpoints with $w(s)$.

There are less than $f^2 = \sqrt{B}$ sets $G_u(i, j)$ ($1 \le i \le j \le f$), on which we create a structure $\mathcal{M}_u$ of Lemma 5.1. For each $i \in [f]$, build a structure $\mathcal{L}_u(i)$ of Theorem 1.1 on $G_u^{\sqsupset}(i)$ to support 3-sided window-max queries whose search regions have the form $(-\infty, x] \times [y_1, y_2]$. Symmetrically, construct another structure $\mathcal{R}_u(i)$ on $G_u^{\sqsubset}(i)$ to support queries of the form $[x, \infty) \times [y_1, y_2]$.

Each segment $s \in S$ generates $O(1)$ information in at most 3 secondary structures. Since each secondary structure has linear size, the overall space is linear. To construct the structure, we first build $\mathcal{T}$, and then create the G_u of each node u top-down in $O(\frac{N}{B} \log_B N)$ I/Os. After this, the secondary structures are constructed in $O(\frac{N}{B} \log_B N)$ I/Os (see Theorem 1.1 and Lemma 5.1).

To answer a query $Q : x \times [y_1, y_2]$, our algorithm initiates a value $\gamma = -\infty$, and then follows a root-to-leaf path of $\mathcal{T}$ to the leaf node z whose $\mathcal{I}(z)$ covers x. Let u be an internal node on the path. Let $\alpha \in [f]$ be such that $\mathcal{I}(u_\alpha)$ covers x. Perform a bundled window-max query $[y_1, y_2]$ on $\mathcal{M}_u$, which reports a weight from each $G_u(i, j)$ of all $1 \le i \le j \le f$. Let w_1 be the maximum of the weights from the $G_u(i, j)$ satisfying $i \le \alpha \le j$. Next, perform a 3-sided window-max query $(-\infty, x] \times [y_1, y_2]$ on $\mathcal{L}_u(\alpha)$, and a query $[x, \infty) \times [y_1, y_2]$ on $\mathcal{R}_u(\alpha)$. Let w_2 and w_3 be their results, respectively. Set $\gamma = \max\{\gamma, w_1, w_2, w_3\}$. Finally, at z, simply check all the segments in z, and identify the maximum weight w of those segments intersecting Q. Return $\max\{w, \gamma\}$ as the final result. By Theorem 1.1 and Lemma 5.1, $O(\log_B N)$ I/Os are performed at each level of $\mathcal{T}$, resulting in $O(\log_B^2 N)$ I/Os overall.

LEMMA 5.2. *There is a linear-space structure that answers a segment-intersection-max query in $O(\log_B^2 N)$ I/Os. The structure can be constructed in $O(\frac{N}{B}\log_B N)$ I/Os.*

6. STABBING-MAX

Let S be a set of N rectangles in $\mathbb{R}^2$. Each rectangle $r \in S$ is associated with a weight $w(r) \in \mathbb{N}$. Given a query point $Q : (x, y)$, the goal is to report the maximum $w(r)$ of the rectangles $r \in S$ that cover Q.

6.1 Ray-segment-max

We first tackle the following *ray-segment-max* problem that is a special instance of segment-intersection-max, and lies at the heart of stabbing-max. Let H be a set of horizontal segments in $\mathbb{R}^2$. Each segment $s \in H$ is associated with a weight $w(s) \in \mathbb{R}$. Given a vertical ray $Q : x \times (-\infty, y]$, a query reports the maximum $w(s)$ of the segments $s \in H$ intersecting Q. We will solve the problem with a linear-space structure that guarantees logarithmic query cost.

Structure. Our index combines the external segment tree [7] with techniques developed in Sections 3 and 4. The base tree $\mathcal{T}$ is a B-tree with fanout $f = \sqrt{B}$ on the x-coordinates of the endpoints of the segments in H. Each node u of $\mathcal{T}$ corresponds to a vertical slab $\mathcal{I}(u)$ in $\mathbb{R}^2$. We associate u with a set H_u of segments s such that s has at least an endpoint in $\mathcal{I}(u)$. Observe that s appears in the H_u of at most two u at each level of $\mathcal{T}$. If w is the λ-th smallest weight (of the segments) in H_u, we say that the *u-rank* of w is λ. At a leaf node z of $\mathcal{T}$, associate each endpoint in z with the segment s the endpoint belongs to, as well as the $\rho(z)$-rank of $w(s)$, where $\rho(z)$ is the parent of z.

Given a segment-set X and a segment s (all horizontal), let $Y(X, s)$ be the set of segments in X whose y-coordinates are at most that of s. Now consider u as an internal node with $u_1, ..., u_f$ as its child nodes. Define a multi-slab $\mathcal{I}(u[i, j]) = \mathcal{I}(u_i) \cup ... \cup \mathcal{I}(u_j)$ for i, j satisfying $1 \le i \le j \le f$. Let $H_u(i, j)$ be the set of segments s in H_u such that $\mathcal{I}(u[i, j])$ is the maximal multi-slab spanned by s.

We build two BCB-trees Γ_u and Λ_u (only Γ_u if u is the root of $\mathcal{T}$). Γ_u indexes a bundle of domain size $K_u = |H_u|$ with $f(f+1)/2+f \le B$ categories (recall $B \ge 9$). Precisely, there is a category $P_u^{\texttt{multi}}(i, j)$ for each multi-slab $\mathcal{I}(u[i, j])$, and a category $P_u^{\texttt{slab}}(i)$ for each slab $\mathcal{I}(u_i)$. Λ_u indexes a bundle of domain size K_u with a single category.

Next, we clarify the labels in the BCB-trees, starting with Γ_u. Consider a key $k \in [K_u]$, and the segment s having the k-th smallest y-coordinate in H_u. We decide the categories and labels of k as follows:

- If the left endpoint of s falls in the $\mathcal{I}(u_\alpha)$ of some α, assign k to $P_u^{\texttt{slab}}(\alpha)$ with two labels. The first one $\ell_{\text{y}}(k)$ equals $|Y(H_{u_\alpha}, s)|$. The second $\ell_{\texttt{c-addr}}(k)$ references the fat block v in Γ_{u_i} whose $\mathcal{I}(v)$ covers $\ell_{\text{y}}(k)$. Repeat the same with respect to the right endpoint of s.
- If $s \in H_u(\alpha, \beta)$ for some α and β, assign k to $P_u^{\texttt{multi}}(\alpha, \beta)$ also with two labels. The first one $\ell_{\texttt{rank}}(k)$ equals the u-rank of the maximum weight of the points in $Y(H_u(\alpha, \beta), s)$. The second $\ell_{\texttt{addr}}(k)$ references the fat block v in Λ_u whose $\mathcal{I}(v)$ encloses $\ell_{\texttt{rank}}(k)$.

Finally, in Λ_u, every $k \in [K_u]$ has two labels:

- $\ell_{\texttt{p-rank}}(k)$, which equals the $\rho(u)$-rank of the k-th smallest weight in H_u;
- $\ell_{\texttt{p-addr}}(k)$, which references the fat block v in $\Lambda_{\rho(u)}$ whose $\mathcal{I}(v)$ contains $\ell_{\texttt{p-rank}}(k)$. If $\rho(u)$ is the root, $\ell_{\texttt{p-addr}}(k)$ is undefined.

Query. The query algorithm involves a downward step and an upward step. Given a ray $Q : x \times (-\infty, y]$, the downward step searches a root-to-leaf path Π to reach the leaf node z such that $x \in \mathcal{I}(z)$. Let u be an internal node on Π, u_α ($\alpha \in [f]$) the child node of u on Π, and $C(u)$ the number of segments in H_u whose y-coordinates are at most y. Perform a bundled-predecessor query on Γ_u with $C(u)$. Let $P_u^{\texttt{multi}}(i^*, j^*)$ be the category whose $\ell_{\texttt{rank}}$-label retrieved by the query is the largest among all categories $P_u^{\texttt{multi}}(i, j)$ satisfying $i \le \alpha \le j$. Set $\gamma(u)$ to that label, and $A_\Lambda(u)$ to the retrieved $\ell_{\texttt{addr}}$-label of $P_u^{\texttt{multi}}(i^*, j^*)$. Before descending to u_α, set $C(u_\alpha)$ and $A_\Gamma(u_\alpha)$ to the ℓ_{y}- and $\ell_{\texttt{c-addr}}$-labels fetched from category $P_u^{\texttt{slab}}(\alpha)$, respectively. With $C(u_\alpha)$ and $A_\Gamma(u_\alpha)$, the bundled-predecessor query on Γ_{u_α} can be answered in $O(1)$ I/Os. In other words, only $\Gamma_{root(\mathcal{T})}$ is searched completely.

The upward step starts from z, and follows the same path back to the root. Along the way, we make use of the $\gamma(u)$, $A_\Lambda(u)$, and Λ_u of each internal node $u \in \Pi$ to determine the answer in the way explained in Section 4.2.

Analysis. The key to proving the linear size of our structure is to show that all the BCB-trees require linear space in total. We focus on Γ_u (where u is a node of $\mathcal{T}$) because the analysis of Λ_u is the same as in Theorem 1.1. It suffices to argue that Γ_u occupies $O(\frac{K_u}{B \log_B N})$ space. Let $u_1, ..., u_f$ ($f \le \sqrt{B}$) be the child nodes of u. Each key $k \in [K_u]$ indexed by Γ_u is assigned to constant categories. Hence, the total number of keys of all categories is $O(K_u)$. Γ_u has four types of labels: $\ell_{\texttt{rank}}(k)$, $\ell_{\text{y}}(k)$, $\ell_{\texttt{c-addr}}(k)$, and $\ell_{\texttt{addr}}(k)$. We know:

- $\ell_{\texttt{rank}}(k) \in [K_u]$;
- the domain of $\ell_{\text{y}}(k)$ has size $\sum_i K_{u_i} = O(K_u)$;
- the domain of $\ell_{\texttt{c-addr}}(k)$ has size $\sum_i O(K_{u_i}/B) = O(K_u/B)$;
- the domain of $\ell_{\texttt{addr}}(k)$ has size $O(K_u/B)$.

By Lemma 3.1, Γ_u occupies $O(\frac{K_u \log f}{B \log N}) = O(\frac{K_u}{B \log_B N})$ space.

A query performs $O(\log_B N)$ I/Os at the root of $\mathcal{T}$, and $O(1)$ I/Os at every other level. The construction algorithm in Theorem 1.1 can be adapted to build the structure in $O(\frac{N}{B}\log_B N)$ I/Os. Omitting the tedious details, we conclude:

LEMMA 6.1. *There is a linear-space structure that answers a ray-segment-max query in $O(\log_B N)$ I/Os. The structure can be built in $O(\frac{N}{B}\log_B N)$ I/Os.*

6.2 Stabbing-max

Assume that the ray-segment-max problem can be settled by a linear-space structure that has query cost $T_Q(N, B)$, and can be built in $T_{build}(N, B)$ I/Os. Agarwal et al. [3] showed how to obtain a linear-space structure that answers a stabbing-max query in $O(T_{query}(N, B) \log_B N + \log_B^2 N)$ I/Os, and takes $O(T_{build}(N, B) + \frac{N}{B} \log_B N)$ I/Os to construct. Combining their technique with Lemma 6.1 gives:

LEMMA 6.2. *There is a linear-space structure that answers a stabbing-max query in $O(\log_B^2 N)$ I/Os. The structure can be built in $O(\frac{N}{B} \log_B N)$ I/Os.*

Theorem 1.2 can now be established from the reduction in Section 2, the window-max result of [3], and Lemmas 5.2 and 6.2.

7. RECTANGLE-INTERSECTION-SUM

Let S be a set of N points in $\mathbb{R}^2$. Each point $p \in S$ is associated with a weight $w(p) \in \mathbb{N}$. Given a 2-sided rectangle $Q : (-\infty, x] \times (-\infty, y]$, the goal is to find the sum of the weights of all the points covered by Q. Once the above *2-sided window-sum* problem is settled, rectangle-intersection-sum can be solved with asymptotically the same space, query, and preprocessing cost (see [12]).

Structure. The base tree of our structure is still a B-tree $\mathcal{T}$ on the x-coordinates of the points in S. Define $root(\mathcal{T})$, $Y(X, p)$, K_u, S_u and $\mathcal{I}(u)$ (where u is a node in $\mathcal{T}$) as in Section 4. Consider an internal node u with child nodes $u_1, ..., u_B$. We associate u with a BCB-tree Γ_u indexing a bundle $\{P_1(u), ..., P_B(u)\}$ with domain size K_u. Key $k \in [K_u]$ is assigned to category $P_i(u)$ if the point p, which has the k-th smallest y-coordinate in S_u, is in the subtree of u_i for some $i \in [B]$. We assign to k three labels $\ell_{\texttt{sum}}(k)$, $\ell_{\texttt{y}}(k)$, and $\ell_{\texttt{c-addr}}(k)$, where $\ell_{\texttt{sum}}(k)$ is the sum of the weights in $Y(S_{u_i}, p)$, and both $\ell_{\texttt{y}}(k)$ and $\ell_{\texttt{c-addr}}(k)$ are as defined in Section 4.

Query. We answer a query with $Q : (-\infty, x] \times (-\infty, y]$ by accessing a root-to-leaf path of $\mathcal{T}$. The algorithm keeps an invariant that, prior to visiting a node u, we know (i) the number $C(u)$ of points in S_u whose y-coordinates are at most y, and (ii) the address $A(u)$ of the fat block v in Γ_u whose $\mathcal{I}(v)$ covers $C(u)$. $C(root(\mathcal{T}))$ and $A(root(\mathcal{T}))$ can be obtained in $O(\log_B N)$ I/Os as in Section 4.

The algorithm maintains at any moment the current result γ, which equals 0 at the beginning. Assume, in general, that we are visiting a node u. If u is a leaf node, simply add to γ the total weight of all the points in u that fall in Q. The final γ is then returned. For an internal u, let $u_1, ..., u_B$ be the child nodes of u, and α an integer such that $x \in \mathcal{I}(u_\alpha)$. In a constant number of I/Os, we (utilizing $A(u)$) answer the bundled-predecessor query on Γ_u with search key $C(u)$. The result contains a label-set $(l_{\texttt{sum}}[i], l_{\texttt{y}}[i], l_{\texttt{c-addr}}[i])$, retrieved from category $P_i(u)$, for each $i \in [B]$. We increase γ by $\sum_{i=1}^{\alpha-1} l_{\texttt{sum}}[i]$. After setting $C(u_\alpha)$ to $l_{\texttt{y}}[\alpha]$ and $A(u_\alpha)$ to $l_{\texttt{c-addr}}[\alpha]$, the algorithm descends to u_α.

Analysis. It is easy to see that the query cost is $O(\log_B N)$. As for space consumption, we focus on bounding the size of BCB-trees. Our analysis below utilizes the fact that W, which is the total weight of all the objects in S, is $N^{O(1)}$, as each weight fits in a constant number of words.

Consider an internal node u with child nodes $u_1, ..., u_B$. Γ_u has three types of labels $\ell_{\texttt{sum}}(k)$, $\ell_{\texttt{y}}(k)$, and $\ell_{\texttt{c-addr}}(k)$. Let W_u be the sum of the weights of all the points in S_u. Each $\ell_{\texttt{sum}}(k)$ is in a domain with size $\sum_i W_{u_i} = W_u$. The sizes of the domains for $\ell_{\texttt{y}}(k)$ and $\ell_{\texttt{c-addr}}(k)$ are as explained in Section 4.2. By Lemma 3.1, Γ_u occupies $O(\frac{K_u}{B \log N}(\log \frac{W_u}{K_u} + \log B))$ space.

Consider any particular level of $\mathcal{T}$, and denote by X the set of all nodes at this level. The space of the BCB-trees associated with the nodes in X is

$$\begin{aligned} & O\left(\sum_{u \in X} \frac{K_u}{B \log N}\left(\log \frac{W_u}{K_u} + \log B\right)\right) \\ = \; & O\left(\frac{1}{B \log N}\left(\sum_{u \in X}\left(K_u \log \frac{W_u}{K_u}\right) + \log B \sum_{u \in X} K_u\right)\right). \end{aligned}$$

As $\sum_{u \in X} K_u = N$ and $\sum_{u \in X} W_u = W$, Lemma 2.1 shows that the above equation is bounded by

$$\begin{aligned} & O\left(\frac{1}{B \log N}\left(N \log \frac{W}{N} + N \log B\right)\right) \\ = \; & O\left(\frac{N \max\{1, \log_B \frac{W}{N}\}}{B \log_B N}\right). \end{aligned}$$

Hence, the BCB-trees at all $O(\log_B N)$ levels of $\mathcal{T}$ consume totally $O(\frac{N}{B} \max\{1, \log_B \frac{W}{N}\})$ space. Our structure can be easily constructed bottom-up in $O(\frac{N}{B} \log_B N)$ I/Os. This proves Theorem 1.3.

8. ACKNOWLEDGEMENTS

This work was supported by grants GRF 4173/08, GRF 4169/09, and GRF 4166/10 from HKRGC.

9. REFERENCES

[1] P. Afshani, L. Arge, and K. D. Larsen. Orthogonal range reporting in three and higher dimensions. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 149–158, 2009.

[2] P. Afshani, L. Arge, and K. D. Larsen. Orthogonal range reporting: query lower bounds, optimal structures in 3-d, and higher-dimensional improvements. In *Symposium on Computational Geometry (SoCG)*, pages 240–246, 2010.

[3] P. K. Agarwal, L. Arge, J. Yang, and K. Yi. I/O-efficient structures for orthogonal range-max and stabbing-max queries. In *Proceedings of European Symposium on Algorithms (ESA)*, pages 7–18, 2003.

[4] P. K. Agarwal, L. Arge, and K. Yi. An optimal dynamic interval stabbing-max data structure? In

Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pages 803–812, 2005.

[5] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM (CACM)*, 31(9):1116–1127, 1988.

[6] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 198–207, 2000.

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

[8] L. Arge and J. Vahrenhold. I/O-efficient dynamic planar point location. *Computational Geometry*, 29(2):147–162, 2004.

[9] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proceedings of Annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pages 560–569, 1996.

[10] B. Chazelle. An algorithm for segment-dragging and its implementation. *Algorithmica*, 3:205–221, 1988.

[11] B. Chazelle. A functional approach to data structures and its use in multidimensional searching. *SIAM Journal of Computing*, 17(3):427–462, 1988.

[12] H. Edelsbrunner and M. H. Overmars. On the equivalence of some rectangle problems. *Information Processing Letters (IPL)*, 14(3):124–127, 1982.

[13] S. Govindarajan, P. K. Agarwal, and L. Arge. CRB-tree: An efficient indexing scheme for range-aggregate queries. In *Proceedings of International Conference on Database Theory (ICDT)*, pages 143–157, 2003.

[14] H. Kaplan, E. Molad, and R. E. Tarjan. Dynamic rectangular intersection with priorities. In *Proceedings of ACM Symposium on Theory of Computing (STOC)*, pages 639–648, 2003.

[15] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. In *Proceedings of ACM Management of Data (SIGMOD)*, pages 401–412, 2001.

[16] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *Proceedings of Symposium on Advances in Spatial and Temporal Databases (SSTD)*, pages 443–459, 2001.

[17] B. Salzberg and V. J. Tsotras. Comparison of access methods for time-evolving data. *ACM Computing Surveys*, 31(2):158–221, 1999.

[18] Y. Tao and D. Papadias. Range aggregate processing in spatial databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, 16(12):1555–1570, 2004.

[19] J. S. Vitter. Algorithms and data structures for external memory. *Foundation and Trends in Theoretical Computer Science*, 2(4):305–474, 2006.

[20] D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger. On computing temporal aggregates with range predicates. *ACM Transactions on Database Systems (TODS)*, 33(2), 2008.

[21] D. Zhang and V. J. Tsotras. Optimizing spatial min/max aggregations. *The VLDB Journal*, 14(2):170–181, 2005.