I/O-Efficient Planar Range Skyline and Attrition Priority Queues*

Casper Kejlberg-Rasmussen[1] Yufei Tao[2,3] Konstantinos Tsakalidis[4]
Kostas Tsichlas[5] Jeonghun Yoon[3]

[1]MADALGO†, Aarhus University
[2]Chinese University of Hong Kong
[3]Korea Advanced Institute of Science and Technology
[4]Hong Kong University of Science and Technology
[5]Aristotle University of Thessaloniki

ABSTRACT

We study the static and dynamic *planar range skyline reporting problem* in the external memory model with block size B, under a linear space budget. The problem asks for an $O(n/B)$ space data structure that stores n points in the plane, and supports reporting the k maximal input points (a.k.a. *skyline*) among the points that lie within a given query rectangle $Q = [\alpha_1, \alpha_2] \times [\beta_1, \beta_2]$. When Q is *3-sided*, i.e. one of its edges is grounded, two variants arise: *top-open* for $\beta_2 = \infty$ and *left-open* for $\alpha_1 = -\infty$ (symmetrically *bottom-open* and *right-open*) queries.

We present optimal static data structures for *top-open* queries, for the cases where the universe is $\mathbb{R}^2$, a $U \times U$ grid, and rank space $[\mathcal{O}(n)]^2$. We also show that *left-open* queries are harder, as they require $\Omega((n/B)^\epsilon + k/B)$ I/Os for $\epsilon > 0$, when only linear space is allowed. We show that the lower bound is tight, by a structure that supports 4-sided queries in matching complexities. Interestingly, these lower and upper bounds coincide with those of the *planar orthogonal range reporting problem*, i.e., the skyline requirement does not alter the problem difficulty at all!

Finally, we present the first dynamic linear space data structure that supports top-open queries in $O(\log_{2B^\epsilon} n + k/B^{1-\epsilon})$ and updates in $O(\log_{2B^\epsilon} n)$ worst case I/Os, for $\epsilon \in [0, 1]$. This also yields a linear space data structure for 4-sided queries with optimal query I/Os and $\mathcal{O}(\log(n/B))$ amortized update I/Os. We consider of independent interest the main component of our dynamic structures, a new real-time I/O-efficient and catenable variant of the fundamental structure *priority queue with attrition* by Sundar.

*The full version is found on http://arxiv.org under the same title.

†MADALGO (Center for Massive Data Algorithmics – a Center of the Danish National Research Foundation)

PODS'13, June 22–27, 2013, New York, New York, USA.

Categories and Subject Descriptors

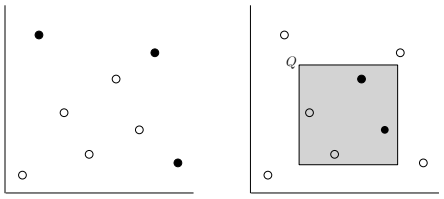
F.2.2 [**Analysis of algorithms and problem complexity**]: Nonnumerical Algorithms and Problems—*computations on discrete structures*; H.3.1 [**Information storage and retrieval**]: Content analysis and indexing—*indexing methods*

Keywords

Skyline, range reporting, priority queues, external memory, data structures

1. INTRODUCTION

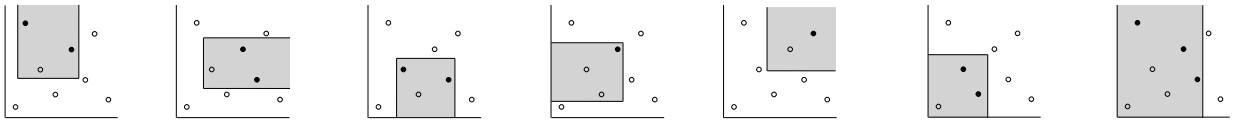
Given two different points $p = (x_p, y_p)$ and $q = (x_q, y_q)$ in $\mathbb{R}^2$, where $\mathbb{R}$ denotes the real domain, we say that p *dominates* q if $x_p \geq x_q$ and $y_p \geq y_q$. Let P be a set of n points in $\mathbb{R}^2$. A point $p \in P$ is *maximal* if it is not dominated by any other point in P. The *skyline* of P consists of all maximal points of P. Notice that the skyline naturally forms an orthogonal staircase where increasing x-coordinates imply decreasing y-coordinates. Figure 1a shows an example where the maximal points are in black.



(a) Skyline (b) Range skyline

Figure 1: Range skyline queries.

Given an axis-parallel rectangle Q, a *range skyline query* (also known as a *range maxima query*) reports the skyline of $P \cap Q$. In Figure 1b, for instance, Q is the shaded rectangle, and the two black points constitute the query result. When Q is a 3-sided rectangle, a range skyline query becomes a *top-open*, *right-open*, *bottom-open* or *left-open* query, as shown in Figures 2a-2d respectively. A *dominance* (resp. *anti-dominance*) query Q is a 2-sided rectangle with both the top and right (resp. the bottom and left) edges grounded, as shown in Figure 2e (resp. 2f). Another well-studied variation is the *contour* query, where Q is a 1-sided rectangle that is the half-plane to the left of a vertical line (Figure 2g).



(a) Top-open (b) Right-open (c) Bottom-open (d) Left-open (e) Dominance (f) Anti-dominance (g) Contour

Figure 2: Variations of range skyline queries (black points represent the query results).

This paper studies linear-size data structures that can answer range skyline queries efficiently, in both the static and dynamic settings. Our analysis focuses on the *external memory* (EM) model [1], which has become the dominant computation model for studying I/O-efficient algorithms. In this model, a machine has M words of memory, and a disk of an unbounded size. The disk is divided into disjoint *blocks*, each of which is formed by B consecutive words. An *I/O* loads a block of data from the disk to memory, or conversely, writes B words from memory to a disk block. The space of a structure equals the number of blocks it occupies, while the cost of an algorithm equals the number of I/Os it performs. CPU time is for free.

By default, the data universe is $\mathbb{R}^2$. Given an integer $U > 0$, $[U]$ represents the set $\{0, 1, \ldots, U-1\}$. All the above queries remain well defined in the universe $[U]^2$. Particularly, when $U = \mathcal{O}(n)$, the universe is called *rank space*. In general, for a smaller universe, it may be possible to achieve better query cost under the same space budget. We consider that P is in general position, i.e., no two points in P have the same x- or y-coordinate (datasets not in general position can be supported by standard tie breaking). When the universe is $[U]^2$, we make the standard assumption that a machine word has at least $\log_2 U$ bits.

1.1 Motivation of 2D Range Skyline

Skylines have drawn very significant attention (see [4, 6, 7, 10–12, 15, 20, 21, 23, 24, 26–31] and the references therein) from the research community due to their crucial importance to multi-criteria optimization, which in turn is vital to numerous applications. In particular, the rectangle of a range skyline query represents range predicates specified by a user. An effective index is essential for maximizing the efficiency of these queries in database systems [24, 28].

This paper concentrates on 2D data for several reasons. First, *planar range skyline reporting* (i.e., our problem) is a classic topic that has been extensively studied in theory [7, 11, 12, 15, 20, 21, 23, 27]. However, nearly all the existing results apply to internal memory (as reviewed in the next subsection), while currently there is little understanding about the characteristics of the problem in I/O environments.

The second, more practical, reason is that many skyline applications are *inherently* 2D. In fact, the special importance of 2D arises from the fact that one often faces the situation of having to strike a balance between a pair of naturally contradicting factors. A prominent example is *price* vs. *quality* in product selection. A range skyline query can be used to find the products that are not dominated by others in both aspects, when the price and quality need to fall in specific ranges. Other pairs of naturally contradicting factors include *space* vs. *query time* (in choosing data structures), *privacy protection* vs. *disclosed information* (the perpetual dilemma in privacy preservation [9]), and so on.

The last reason, and maybe the most important, is that clearly range skyline reporting cannot become easier as the dimensionality increases, whereas even for two dimensions, we will prove a hardness result showing that the problem (unfortunately) is already difficult enough to forbid sub-polynomial query cost under the linear space budget! In other words, the "easiest" dimensionality of 2 is not so easy after all, which also points to the absence of query-efficient structures in any higher dimension when only linear space is permitted.

1.2 Previous Results

Range Skyline in Internal Memory. We first review the existing results when the dataset P fits in main memory. Early research focused on dominance and contour queries, both of which can be solved in $\mathcal{O}(\log n + k)$ time using a structure of $\mathcal{O}(n)$ size, where k is the number of points reported [11, 15, 20, 23, 27]. Brodal and Tsakalidis [7] were the first to discover an optimal dynamic structure for top-open queries, which capture both dominance and contour queries as special cases. Their structure occupies $\mathcal{O}(n)$ space, answers queries in $\mathcal{O}(\log n + k)$ time, and supports updates in $\mathcal{O}(\log n)$ time. The above structures belong to the *pointer machine* model. Utilizing features of the RAM model, Brodal and Tsakalidis [7] also presented an alternative structure in universe $[U]^2$, which uses $\mathcal{O}(n)$ space, answers queries in $\mathcal{O}(\frac{\log n}{\log \log n} + k)$ time, and can be updated in $\mathcal{O}(\frac{\log n}{\log \log n})$ time. In RAM, the static top-open problem can be easily settled using an RMQ (*range minimum queries*) structure (see, e.g., [36]), which occupies $\mathcal{O}(n)$ space and answers queries in $\mathcal{O}(1 + k)$ time.

For general range skyline queries (i.e., 4-sided), all the known structures demand super-linear space. Specifically, Brodal and Tsakalidis [7] gave a pointer-machine structure of $\mathcal{O}(n \log n)$ size, $\mathcal{O}(\log^2 n + k)$ query time, and $\mathcal{O}(\log^2 n)$ update time. Kalavagattu et al. [21] designed a static RAM-structure that occupies $O(n \log n)$ space and achieves query time $\mathcal{O}(\log n + k)$. In rank space, Das et al. [12] proposed a static RAM-structure with $\mathcal{O}(n \frac{\log n}{\log \log n})$ space and $\mathcal{O}(\frac{\log n}{\log \log n} + k)$ query time.

The above results also hold directly in external memory, but they are far from being satisfactory. In particular, all of them incur $\Omega(k)$ I/Os to report k points. An I/O-efficient structure ought to achieve $\mathcal{O}(k/B)$ I/Os for this purpose.

Range Skyline in External Memory. In contrast to internal memory where there exist a large number of results, range skyline queries have not been well studied in external memory. As a naive solution, we can first scan the entire point set P to eliminate the points falling outside the query rectangle Q, and then find the skyline of the remaining points by the fastest skyline algorithm [31] on non-preprocessed input sets. This expensive solution can incur $\mathcal{O}((n/B) \log_{M/B}(n/B))$ I/Os.

Papadias et al. [28] described a branch-and-bound algorithm when the dataset is indexed by an R-tree [17]. The algorithm is heuristic and cannot guarantee better worst

	space	query	insertion	deletion	remark
top-open in $\mathbb{R}^2$	$\mathcal{O}(n/B)$	$\mathcal{O}(\log_B n + k/B)$	-	-	optimal
top-open in U^2	$\mathcal{O}(n/B)$	$\mathcal{O}(\log\log_B U + k/B)$	-	-	optimal
top-open in $[\mathcal{O}(n)]^2$	$\mathcal{O}(n/B)$	$\mathcal{O}(1 + k/B)$	-	-	optimal
anti-dominance in $\mathbb{R}^2$	$\mathcal{O}(n/B)$	$\Omega((n/B)^\epsilon + k/B)$	-	-	lower bound (indexability)
4-sided in $\mathbb{R}^2$	$\mathcal{O}(n/B)$	$\mathcal{O}((n/B)^\epsilon + k/B)$	-	-	optimal (indexability)
top-open in $\mathbb{R}^2$	$\mathcal{O}(n/B)$	$\mathcal{O}(\log_{2B^\epsilon} n + k/B^{1-\epsilon})$	$\mathcal{O}(\log_{2B^\epsilon} n)$	$\mathcal{O}(\log_{2B^\epsilon} n)$	for any constant $\epsilon \in [0, 1]$
4-sided in $\mathbb{R}^2$	$\mathcal{O}(n/B)$	$\mathcal{O}((n/B)^\epsilon + k/B)$	$\mathcal{O}(\log(n/B))$	$\mathcal{O}(\log(n/B))$	update cost is amortized

Table 1: Summary of our range skyline results (all complexities are in the worst case by default).

case query I/Os than the naive solution mentioned earlier. Different approaches have been proposed for skyline maintenance in external memory under various assumptions on the updates [19, 28, 33, 35]. The performance of those methods, however, was again evaluated only experimentally on certain "representative" datasets. No I/O-efficient structure exists for answering range skyline queries even in sublinear I/Os under arbitrary updates.

Priority Queues with Attrition (PQAs). Let S be a set of elements drawn from an ordered domain, and let $\min(S)$ be the smallest element in S. A PQA on S is a data structure that supports the following operations:

- FindMin: Return $\min(S)$.
- DeleteMin: Remove and return $\min(S)$.
- InsertAndAttrite: Add a new element e to S and remove from S all the elements at least e. After the operation, the new content is $S' = \{e' \in S \mid e' < e\} \cup \{e\}$. The elements $\{e' \in S \mid e' \geq e\}$ are *attrited*.

In internal memory, Sundar [32] described how to implement a PQA that supports all operations in $\mathcal{O}(1)$ worst case time, and occupies $\mathcal{O}(n - m)$ space after n InsertAndAttrite and m DeleteMin operations.

1.3 Our Results

This paper presents external memory structures for solving the planar range skyline reporting problem using only linear space. At the core of one of these structures is a new PQA that supports the extra functionality of catenation. This PQA is a non-trivial extension of Sundar's version [32]. It can be implemented I/O-efficiently, and is of independent interest due to its fundamental nature. Next, we provide an overview of our results.

Static Range Skyline. When P is static, we describe several linear-size structures with the optimal query cost. Our structures also separate the hard variants of the problem from the easy ones.

For top-open queries, we present a structure that answers queries in optimal $\mathcal{O}(\log_B n + k/B)$ I/Os (Theorem 1) when the universe is $\mathbb{R}^2$. To obtain the result, we give an elegant reduction of the problem to *segment intersection*, which can be settled by a *partially persistent B-tree* (PPB-tree) [5]. Furthermore, we show that this PPB-tree is (what we call) *sort-aware build-efficient* (SABE), namely, it can be constructed in linear I/Os, provided that P is already sorted by x-coordinate (Theorem 1). The construction algorithm exploits several intrinsic properties of top-open queries, whereas none of the known approaches [2, 14, 34] for bulkloading a PPB-tree is SABE.

The above structure is *indivisible*, namely, it treats each coordinate as an atom by always storing it using an entire word. As the second step, we improve the top-open query overhead beyond the logarithmic bound when the data universe is small. Specifically, when the universe is $[U]^2$ where U is an integer, we give a *divisible* structure with optimal $\mathcal{O}(\log\log_B U + k/B)$ query I/Os (Corollary 1). In the rank space, we further reduce the query cost again optimally to $\mathcal{O}(1 + k/B)$ (Theorem 2).

Clearly, top-open queries are equivalent to right-open queries by symmetry, and capture dominance and contour queries as special cases, so the results aforementioned are applicable to those variants immediately.

Unfortunately, fast query cost with linear space is impossible for the remaining variants under the well-known *indexability model* of [18] (all the structures in this paper belong to this model). Specifically, for anti-dominance queries, we establish a lower bound showing that every linear-size structure must incur $\Omega((n/B)^\epsilon + k/B)$ I/Os in the worst case (Theorem 5), where $\epsilon > 0$ can be an arbitrarily small constant. Furthermore, we prove that this is tight, by giving a structure to answer a 4-sided query in $\mathcal{O}((n/B)^\epsilon + k/B)$ I/Os (Theorem 6). Since 4-sided is more general than anti-dominance, these matching lower and upper bounds imply that they, as well as left- and bottom-open queries, have exactly the same difficulty.

The above 4-sided results also reveal a somewhat unexpected fact: planar range skyline reporting has precisely the same hardness as *planar range reporting* (where, given an axis-parallel rectangle Q, we want to find all the points in $P \cap Q$, instead of just the maxima; see [3, 18] for the matching lower and upper bounds on planar range reporting). In other words, the extra skyline requirement does not alter the difficulty at all.

Dynamic Range Skyline. The aforementioned static structures cannot be updated efficiently when insertions and deletions occur in P. For top-open queries, we provide an alternative structure with fast worst case update overhead, at a minor expense of query efficiency. Specifically, our structure occupies linear space, is SABE, answers queries in $\mathcal{O}(\log_{2B^\epsilon}(n/B) + k/B^{1-\epsilon})$ I/Os, and supports updates in $\mathcal{O}(\log_{2B^\epsilon}(n/B))$ I/Os, where ϵ can be any parameter satisfying $0 \leq \epsilon \leq 1$ (Theorem 4). Note that setting $\epsilon = 0$ gives a structure with query cost $\mathcal{O}(\log(n/B) + k/B)$ and update cost $\mathcal{O}(\log(n/B))$.

The combination of this structure and our (static) 4-sided structure leads to a dynamic 4-sided structure that uses linear space, answers queries optimally in $\mathcal{O}((n/B)^\epsilon + k/B)$ I/Os, and supports updates in $\mathcal{O}(\log(n/B))$ I/Os amortized (Theorem 6). Table 1 summarizes our structures.

Catenable Priority Queues with Attrition. A central ingredient of our dynamic structures is a new PQA that is more powerful than the traditional version of Sundar [32].

Specifically, besides FindMin, DeleteMin and InsertAndAttrite (already reviewed in Section 1.2), it also supports:

- CatenateAndAttrite: Given two PQAs on sets S_1 and S_2 respectively, the operation returns a single PQA on $S = \{e \in S_1 \mid e < \min(S_2)\} \cup S_2$. In other words, the elements in $\{e \in S_1 \mid e \geq \min(S_2)\}$ are attrited.

We are not aware of any previous work that addressed the above operation, which turns out to be rather challenging even in internal memory.

Our structure, named *I/O-efficient catenable priority queue with attrition* (I/O-CPQA), supports all operations in $\mathcal{O}(1)$ worst case and $\mathcal{O}(1/B)$ amortized I/Os (the amortized bound requires that a constant number of blocks be pinned in main memory, which is a standard and compulsory assumption to achieve $o(1)$ amortized update cost of most, if not all, known structures, e.g., the linked list). The space cost is $\mathcal{O}((n-m)/B)$ after n InsertAndAttrite and CatenateAndAttrite operations, and after m DeleteMin operations.

All the missing proofs of theorems, lemmata and corollaries can be found in the full version.

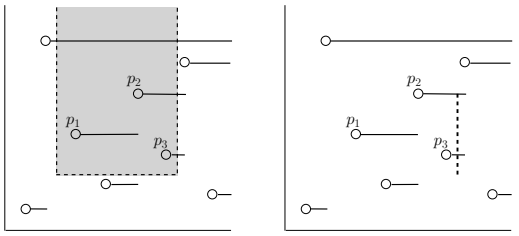
2. SABE TOP-OPEN STRUCTURE

In this section, we describe a structure of linear size to answer a top-open query in $\mathcal{O}(\log_B n + k/B)$ I/Os. The structure is SABE, namely, it can be constructed in linear I/Os provided that the input set P is sorted by x-coordinate.

2.1 Reduction to Segment Intersection

We first describe a simple structure by converting top-open range skyline reporting to the *segment intersection problem*: the input is a set S of horizontal segments in $\mathbb{R}^2$; given a vertical segment q, a query reports all the segments of S intersecting q.

Given a point p in P, denote by $leftdom(p)$ the leftmost point among all the points in P dominating p. If such a point does not exist, $leftdom(p) = nil$. We convert p to a horizontal segment $\sigma(p)$ as follows. Let $q = leftdom(p)$. If $q = nil$, then $\sigma(p) = [x_p, \infty[\times y_p$; otherwise, $\sigma(p) = [x_p, x_q[\times y_p$. Define $\Sigma(P) = \{\sigma(p) \mid p \in P\}$, i.e., the set of segments converted from the points of P. See Figure 3a for an example.



(a) Data conversion (b) Converted query

Figure 3: Reduction.

Now, consider a top-open query with rectangle $Q = [\alpha_1, \alpha_2] \times [\beta, \infty[$. We answer it by performing segment intersection on $\Sigma(P)$. First, obtain β' as the highest y-coordinate of the points in $P \cap Q$. Then, report all segments in $\Sigma(P)$ that intersect the vertical segment $\alpha_2 \times [\beta, \beta']$. An example is shown in Figure 3b. A proof of the correctness of the algorithm can be found in the full version.

We can find β' in $\mathcal{O}(\log_B n)$ I/Os with a *range-max query* on a B-tree indexing the x-coordinates in P. For retrieving the segments intersecting $\alpha_2 \times [\beta, \beta']$, we store $\Sigma(P)$ in a partially persistent B-tree (PPB-tree) [5]. As $\Sigma(P)$ has n segments, the PPB-tree occupies $\mathcal{O}(n/B)$ space and answers a segment intersection query in $\mathcal{O}(\log_B n + k/B)$ I/Os. We thus have obtained a linear-size top-open structure with $\mathcal{O}(\log_B n + k/B)$ query I/Os.

More effort, however, is needed to make the structure SABE. In particular, two challenges are to be overcome. First, we must generate $\Sigma(P)$ in linear I/Os. Second, the PPB-tree on $\Sigma(P)$ must be built with asymptotically the same cost (note that the range-max B-tree is already SABE). We will tackle these challenges in the rest of the section.

2.2 Computing $\Sigma(P)$

$\Sigma(P)$ is not an arbitrary set of segments. We observe:

Lemma 1. *$\Sigma(P)$ has the following properties:*

- **(Nesting)** *for any two segments s_1 and s_2 in $\Sigma(P)$, their x-intervals are either disjoint, or such that one x-interval contains the other.*
- **(Monotonic)** *let ℓ be any vertical line, and $S(\ell)$ the set of segments in $\Sigma(P)$ intersected by ℓ. If we sort the segments of $S(\ell)$ in ascending order of their y-coordinates, the lengths of their x-intervals are non-decreasing.*

We are ready to present our algorithm for computing $\Sigma(P)$, after P has been sorted by x-coordinates. Conceptually, we sweep a vertical line ℓ from $x = -\infty$ to ∞. At any time, the algorithm (essentially) stores the set $S(\ell)$ of segments in a stack, which are en-stacked in descending order of y-coordinates. Whenever a segment is popped out of the stack, its right endpoint is decided, and the segment is output. In general, the segments of $\Sigma(P)$ are output in non-descending order of their right endpoints' x-coordinates.

Specifically, the algorithm starts by pushing the leftmost point of P onto the stack. Iteratively, let p be the next point fetched from P, and q the point currently at the top of the stack. If $y_q < y_p$, we know that $p = leftdom(q)$. Hence, the algorithm pops q off the stack, and outputs segment $\sigma(q) = [x_q, x_p[\times y_q$. Then, letting q be the point that tops the stack currently, the algorithm checks again whether $y_q < y_p$, and if so, repeats the above steps. This continues until either the stack is empty or $y_q > y_p$. In either case, the iteration finishes by pushing p onto the stack. It is clear that the algorithm generates $\Sigma(P)$ in $\mathcal{O}(n/B)$ I/Os.

2.3 Constructing the PPB-tree

Remember that we need a PPB-tree T on $\Sigma(P)$. The known algorithms for PPB-tree construction require super-linear I/Os even after sorting [2, 5, 14, 34]. Next, we show that the two properties of $\Sigma(P)$ in Lemma 1 allow building T in linear I/Os. Let us number the leaf level as *level 0*. In general, the parent of a level-i ($i \geq 0$) node is at level $i + 1$. We will build T in a bottom-up manner, i.e., starting from the leaf level, then level 1, and so on.

Leaf Level. To create the leaf nodes, we need to first sort the left and right endpoints of the segments in $\Sigma(P)$ together by x-coordinate. This can be done in $\mathcal{O}(n/B)$ I/Os as follows. First, P, which is sorted by x-coordinates, gives a sorted list of the left endpoints. On the other hand, our algorithm of the previous subsection generates $\Sigma(P)$ in non-descending

order of the right endpoints' x-coordinates (breaking ties by favoring lower points). By merging the two lists, we obtain the desired sorted list of left and right endpoints combined.

Let us briefly review the algorithm proposed in [5] to build a PPB-tree. The algorithm conceptually moves a vertical line ℓ from $x = -\infty$ to ∞. At any moment, it maintains a B-tree $T(\ell)$ on the y-coordinates of the segments in $S(\ell)$. We call $T(\ell)$ a *snapshot B-tree*. To do so, whenever ℓ hits the left (resp. right) endpoint of a segment s, it inserts (resp. deletes) the y-coordinate of s in $T(\ell)$. The PPB-tree can be regarded as a space-efficient union of all the snapshot B-trees. The algorithm incurs $\mathcal{O}(n \log_B n)$ I/Os because (i) there are $2n$ updates, and (ii) for each update, $\mathcal{O}(\log_B n)$ I/Os are needed to locate the leaf node affected.

When $\Sigma(P)$ is nesting and monotonic, the construction can be significantly accelerated. A crucial observation is that any update to $S(\ell)$ happens only *at the bottom* of ℓ. Specifically, whenever ℓ hits the left/right endpoint of a segment $s \in \Sigma(P)$, s must be the lowest segment in $S(\ell)$. This implies that the leaf node of $T(\ell)$ to be altered must be the leftmost[1] one in $T(\ell)$. Hence, we can find this leaf without any I/Os by buffering it in memory, in contrast to the $\mathcal{O}(\log_B n)$ cost originally needed.

The other details are standard, and are sketched below assuming the knowledge of the classic algorithm in [5]. Whenever the leftmost leaf u of $T(\ell)$ is full, we version copy it to u', and possibly perform a split or merge, if u' strong-version overflows or underflows, respectively[2]. A version copy, split, and merge can all be handled in $\mathcal{O}(1)$ I/Os, and can happen only $\mathcal{O}(n/B)$ times. Therefore, the cost of building the leaf level is $\mathcal{O}(n/B)$.

Internal Levels. The level-1 nodes can be built by exactly the same algorithm, but on a different set of segments Σ_1 which are generated from the leaf nodes of the PPB-tree. To explain, let us first review an intuitive way [13] to visualize a node in a PPB-tree. A node u can be viewed as a rectangle $r(u) = [x_1, x_2[\times [y_1, y_2[$ in $\mathbb{R}^2$, where x_1 (resp. x_2) is the position of ℓ when u is created (resp. version copied), and $[y_1, y_2[$ represents the y-range of u in all the snapshot B-trees where u belongs. See Figure 4.

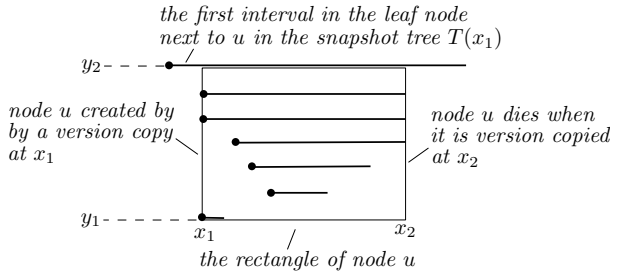


Figure 4: A node in a PPB-tree.

For each leaf node u (already created), we add its the bottom edge of $r(u)$, namely $[x_1, x_2[\times y_1$, into Σ_1. The next lemma points out a crucial fact.

LEMMA 2. *Σ_1 is both nesting and monotonic.*

[1]We adopt the convention that the leaf elements of a B-tree are ordered from left to right in ascending order.

[2]Version copy, strong-version overflow and strong-version underflow are concepts from the terminology of [5].

Our algorithm (for building the leaf nodes) writes the left and right endpoints of the segments in Σ_1 in non-descending order of their x-coordinates (breaking ties by favoring lower endpoints). This, together with Lemma 2, permits us to create the level-1 nodes using the same algorithm in $\mathcal{O}(n/B^2)$ I/Os (as $|\Sigma_1| = \mathcal{O}(n/B)$). We repeat the above process to construct the nodes of higher levels. The cost decreases by a factor of B each level up. The overall construction cost is therefore $\mathcal{O}(n/B)$. Leaving the other details to the full version, we now conclude with the first main result:

THEOREM 1. *There is an indivisible linear-size structure on n points in $\mathbb{R}^2$, such that top-open range skyline queries can be answered in $\mathcal{O}(\log_B n + k/B)$ I/Os, where k is the number of reported points. If all points have been sorted by x-coordinates, the structure can be built in linear I/Os. The query cost is optimal (even without assuming indivisibility).*

3. DIVISIBLE TOP-OPEN STRUCTURE

Theorem 1 holds under the external memory model with the indivisibility assumption. This section eliminates the assumption, and unleashes the power endowed by bit manipulation. As we will see, when the universe is small, it admits linear-size structures with lower query cost.

In Section 3.1, we study a different problem called ray-dragging. Then, in Section 3.2, our ray-dragging structure is deployed to develop a "few-point structure" for answering top-open queries on a small point set. Finally, in Section 3.3, we combine our few-point structure with an existing structure [7] to obtain the final optimal top-open structure.

3.1 Ray Dragging

In the *ray dragging problem*, the input is a set S of m points in $[U]^2$ where $U \geq m$ is an integer. Given a vertical ray $\rho = \alpha \times [\beta, U]$ where $\alpha, \beta \in [U]$, a ray dragging query reports the first point in S to be hit by ρ when ρ moves left. The rest of the subsection serves as the proof for:

LEMMA 3. *For $m = (B \log U)^{\mathcal{O}(1)}$, we can store S in a structure of size $\mathcal{O}(1 + m/B)$ that can answer ray dragging queries in $\mathcal{O}(1)$ I/Os.*

Minute Structure. Set $b = B \log_2 U$. We first consider the scenario where S has very few points: $m \leq b^{1/3}$. Let us convert S to a set S' of points in an $m \times m$ grid. Specifically, map a point $p \in S$ to $p' \in S'$ such that $x_{p'}$ (resp. $y_{p'}$) is the *rank* of x_p (resp. y_p) among the x- (y-) coordinates in S.

Given a ray $\rho = \alpha \times [\beta, \infty[$, we instead answer a query in $[m]^2$ using a ray $\rho' = \alpha' \times [\beta', \infty[$, where α' (resp. β') is the rank of the predecessor of α (resp. β) among the x- (resp. y-) coordinates in S. Create a *fusion tree* [16, 25] on the x- (resp. y-) coordinates in S so that the predecessor of α (resp. β) can be found in $\mathcal{O}(\log_b m) = \mathcal{O}(1)$ I/Os, which is thus also the cost of turning ρ into ρ'. The fusion tree uses $\mathcal{O}(1 + m/B)$ blocks.

We will ensure that the query with ρ' (in $[m]^2$) returns an id from 1 to m that uniquely identifies a point p in S, if the result is non-empty. To convert the id into the coordinates of p, we store S in an array of $\mathcal{O}(1 + m/B)$ blocks such that any point can be retrieved in one I/O by id.

The benefit of working with S' is that each coordinate in $[m]^2$ requires fewer bits to represent (than in $[U]^2$), that is, $\log_2 m$ bits. In particular, we need $3 \log_2 m$ bits in total to represent a point's x-, y-coordinates, and id. Since $|S'| = m$, the

storage of the entire S' demands $3m \log m = \mathcal{O}(b^{1/3} \log_2 b)$ bits. If $B \geq \log_2 U$, then $b^{1/3} \log_2 b = \mathcal{O}((B^2)^{1/3} \log_2(B^2)) = \mathcal{O}(B)$. On the other hand, if $B < \log_2 U$, then $b^{1/3} \log_2 b = \mathcal{O}((\log_2^2 U)^{1/3} \log_2(\log_2^2 U)) = \mathcal{O}(\log_2 U)$. In other words, we can always store the entire set S' in a single block! Given a query with ρ', we simply load this block into memory, and answer the query in memory with no more I/O.

We have completed the description of a structure that uses $\mathcal{O}(1 + m/B)$ blocks, and answers queries in constant I/Os when $m \leq b^{1/3}$. We refer to it as a *minute structure*.

Proof of Lemma 3. We store S in a B-tree that indexes the x-coordinates of the points in S. We set the B-tree's leaf capacity to B and internal fanout to $f = b^{1/3}$. Note that the tree has a constant height.

Given a node u in the tree, define $Y_{max}(u)$ as the highest point whose x-coordinate is stored in the subtree of u. Now, consider u to be an internal node with child nodes $v_1, ..., v_f$. Define $Y^*_{max}(u) = \{Y_{max}(v_i) \mid 1 \leq i \leq f\}$. We store $Y^*_{max}(u)$ in a minute structure. Also, for each point $p \in Y^*_{max}(u)$, we store an index indicating the child node whose subtree contains the x-coordinate of p. A child index requires $\log_2 b^{1/3} = \mathcal{O}(\log_2 m) = \mathcal{O}(\log U)$ bits, which is no more than the length of a coordinate. Hence, we can store the index along with p in the minute structure without increasing its space by more than a constant factor. For a leaf node z, define $Y^*_{max}(z)$ to be the set of points whose x-coordinates are stored in z.

Since there are $\mathcal{O}(1 + m/(b^{1/3}B))$ internal nodes and each minute structure demands $\mathcal{O}(1+b^{1/3}/B)$ space, all the minute structures occupy $\mathcal{O}((1 + \frac{m}{b^{1/3}B})(\frac{b^{1/3}}{B} + 1)) = \mathcal{O}(1 + m/B)$ blocks in total. Therefore, the overall structure consumes linear space.

We answer a ray-dragging query with ray $\rho = \alpha \times [\beta, U]$ as follows. First, descend a root-to-leaf path π to the leaf node containing the predecessor of α among the x-coordinates in S. Let u be the *lowest* node on π such that $Y^*_{max}(u)$ has a point that can be hit by ρ when ρ moves left. For each node $v \in \pi$, whether $Y^*_{max}(v)$ has such a point can be checked in $\mathcal{O}(1)$ I/Os by querying the minute structure over $Y^*_{max}(v)$. Hence, u can be identified in $\mathcal{O}(h)$ I/Os where h is the height of the B-tree. If u does not exist, we return an empty result (i.e., ρ does not hit any point no matter how far it moves).

If u exists, let p be the first point in $Y^*_{max}(u)$ hit by ρ when it moves left. Suppose that the x-coordinate of p is in the subtree of v, where v is a child node of u. The query result must be in the subtree of v, although it may not necessarily be p. To find out, we descend another path from v to a leaf. Specifically, we set u to v, and find the first point p in $Y^*_{max}(u)$ $(= Y^*_{max}(v))$ that is hit by ρ when it moves left (notice that p has changed). Now, letting v be the child node of u whose subtree p is from, we repeat the above steps. This continues until u becomes a leaf, in which case the algorithm returns p as the final answer. The query cost is $\mathcal{O}(h) = \mathcal{O}(1)$. This completes the proof of Lemma 3. We will refer to the above structure as a *ray-drag tree*.

3.2 Top-Open Structure on Few Points

Next, we present a structure for answering top-open queries on small P, called henceforth the *few-point structure*. Remember that P is a set of n points in $[U]^2$ for some integer $U \geq n$, and a query is a rectangle $Q = [\alpha_1, \alpha_2] \times [\beta, U]$ where $\alpha_1, \alpha_2, \beta \in [U]$.

LEMMA 4. *For $n \leq (B \log U)^{\mathcal{O}(1)}$, we can store P in a structure of $\mathcal{O}(1 + n/B)$ space that answers top-open range skyline queries with output size k in $\mathcal{O}(1 + k/B)$ I/Os.*

PROOF. Consider a query with $Q = [\alpha_1, \alpha_2] \times [\beta, U]$. Let p be the first point hit by the ray $\rho = \alpha_2 \times [\beta, U]$ when ρ moves left. If p does not exist or is out of Q (i.e., $x_p < \alpha_1$), the top-open query has an empty result. Otherwise, p must be the lowest point in the skyline of $P \cap Q$.

The subsequent discussion focuses on the scenario where $p \in Q$. We index $\Sigma(P)$ with a PPB-tree T, as in Theorem 1. Recall that the top-open query can be solved by retrieving the set S of segments in $\Sigma(P)$ intersecting the vertical segment $\psi = \alpha_2 \times [\beta, \beta']$, where β' is the highest y-coordinate of the points in $P \cap Q$. To do so in $\mathcal{O}(1 + k/B)$ I/Os, we utilize the next two observations. (see the full version for their proofs):

OBSERVATION 1. All segments of S intersect $\psi' = x_p \times [y_p, \beta']$.

OBSERVATION 2. Let $T(\ell)$ be the snapshot B-tree in T when ℓ is at the position $x = x_p$. Once we have obtained the leaf node in $T(\ell)$ containing y_p, we can retrieve S in $\mathcal{O}(1 + k/B)$ I/Os without knowing the value of β'.

We now elaborate on the structure of Lemma 4. Besides T, also create a structure of Lemma 3 on P. Moreover, for every point $p \in P$, keep a pointer to the leaf node of T that (i) is in the snapshot B-tree $T(\ell)$ when ℓ is at $x = x_p$, and (ii) contains y_p. Call the leaf node the *host leaf* of p. Store the pointers in an array of size n to permit retrieving the pointer of any point in one I/O.

The query algorithm should have become straightforward from the above two observations. We first find in $O(1)$ I/Os the first point p hit by ρ when ρ moves left. Then, using p, we jump to the host leaf of p. Next, by Observation 2, we retrieve S in $O(1 + k/B)$ I/Os. The total query cost is $\mathcal{O}(1 + k/B)$. □

3.3 Final Top-Open Structure

We are ready to describe our top-open structure that achieves sub-logarithmic query I/Os for arbitrary n. For this purpose, we externalize an internal-memory structure of [7]. The structure of [7], however, has logarithmic query overhead, which we improve with new ideas based on the few-point structure in Lemma 4. Delegating the details to the full version, we now state our main results in rank space and universe $[U]^2$:

THEOREM 2. *There is a linear-size structure on n points in rank space such that top-open range skyline queries can be answered optimally in $\mathcal{O}(1 + k/B)$ I/Os, where k is the number of reported points.*

COROLLARY 1. *There is a linear-size structure on a set of n points in $[U]^2$ (where $U \geq n$ is an integer) such that a top-open range skyline query can be answered optimally in $\mathcal{O}(\log \log_B U + k/B)$ I/Os, when k points are reported.*

4. DYNAMIC TOP-OPEN STRUCTURE

In this section, we present a dynamic data structure, which is SABE, that uses linear space, and supports top-open queries in $\mathcal{O}(\log_{2B^\epsilon}(n/B) + k/B^{1-\epsilon})$ I/Os and updates in $\mathcal{O}(\log_{2B^\epsilon}(n/B))$ I/Os, for any parameter $0 \leq \epsilon \leq 1$. We are

inspired by the approach of Overmars and van Leeuwen [27] for maintaining the planar skyline in the pointer machine. As a brief review, a dynamic binary base tree indexes the x-coordinates of P, and every internal node stores the skyline of the points in its subtree using a secondary search tree. More specifically, the skyline of an internal node is $(L \backslash L') \cup R$, where L (resp. R) is the skyline of its left (resp. right) child node, and L' is the set of points in L dominated by the leftmost (and thus also highest) point of R.

Our approach is based on I/O-CPQAs, which are described in Section 4.1. We observe that attrition can be utilized to maintain the internal node skylines in [27], after mirroring the y-axis. To explain this, let us first map the input set P to its mirrored counterpart $\widetilde{P} = \{(x_p, -y_p) \mid (x_p, y_p) \in P\}$. In the context of PQAs, we will interpret each point $(\widetilde{x}_p, \widetilde{y}_p) \in \widetilde{P}$ as an *element* with "key" value $\widetilde{y}_p$ that is inserted at "time" $\widetilde{x}_p$. To formalize the notion of time, we define the $<_x$-ordering of two elements $\widetilde{p}, \widetilde{q} \in \widetilde{P}$ to be $\widetilde{p} <_x \widetilde{q}$, if and only if $\widetilde{x}_p < \widetilde{x}_q$ holds. It is easy to see that element $\widetilde{p} \in \widetilde{P}$ is attrited by element $\widetilde{q} \in \widetilde{P}$, if and only if point $p \in P$ is dominated by point $q \in P$. See Figure 5 for a geometric illustration of the mirroring transformation and the effects of attrition.

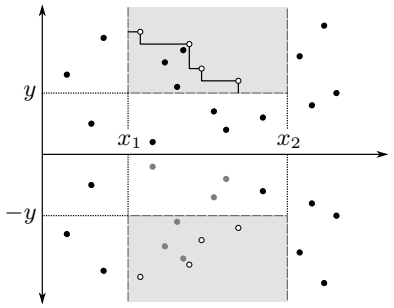


Figure 5: The skyline problem (above) mirrored to the attrition problem (below). White points are reported for the gray query area $[x_1, x_2] \times [y, \infty[$, while gray elements are attrited within $[x_1, x_2]$.

Thus, we index the $<_x$-ordering of $\widetilde{P}$ in a $(2B^\epsilon, 4B^\epsilon)$-tree, for a parameter $0 \leq \epsilon \leq 1$, and employ I/O-CPQAs as secondary structures, such that the I/O-CPQA at an internal node is simply the concatenation of its children's I/O-CPQAs. To obtain logarithmic query and update I/Os, this sequence of consecutive CatenateAndAttrite operations at an internal node must be performed in $\mathcal{O}(1)$ I/Os (Lemma 5). The presented I/O-CPQAs are *ephemeral* (not persistent), and thus the supported operations are *destructive*, as they destroy the initial configuration of the structure. This only allows operating on the I/O-CPQA that is the final result of all concatenations and resides at the root of the base tree. However, in order to support top-open queries efficiently, accessing I/O-CPQAs at the internal nodes is required. This is made possible by non-destructive operations. Therefore, we render the I/O-CPQAs confluently persistent by merely replacing the catenable deques, which are used as black boxes in our ephemeral construction, with real-time purely functional catenable deques [22]. Since the imposed overhead is $\mathcal{O}(1)$ worst case I/Os, confluently persistent I/O-CPQAs ensure the same I/O bounds as their ephemeral counterparts. Section 4.2 describes our dynamic data structure in detail.

4.1 I/O-Efficient Catenable Attrition Priority Queues

Here we present ephemeral *I/O-efficient catenable priority queues with attrition (I/O-CPQAs)* that store a set of elements from a total order and support all operations in $\mathcal{O}(1)$ I/Os. Also the operations take $\mathcal{O}(1/b)$ amortized I/Os, when a constant number of blocks are already loaded into main memory for every root I/O-CPQA, for any parameter $1 \leq b \leq B$. We call these preloaded records *critical records*. For the sake of simplicity, we identify an element with its value. Denote by Q an I/O-CPQA and by $\min(Q)$ the smallest element stored in Q. We denote by Q also the set of elements in I/O-CPQA Q. Next, we re-state the supported operations in the context of I/O-CPQAs:

- FindMin(Q) returns $\min(Q)$.
- DeleteMin(Q) returns $\min(Q)$ and removes it from Q. The resulting I/O-CPQA is $Q' = Q \backslash \{\min(Q)\}$, and Q is discarded.
- CatenateAndAttrite(Q_1, Q_2)[3] catenates I/O-CPQA Q_2 to the end of another I/O-CPQA Q_1, removes all elements in Q_1 that are larger than or equal to $\min(Q_2)$ (attrition), and returns the result as a combined I/O-CPQA $Q_1' = \{e \in Q_1 \mid e < \min(Q_2)\} \cup Q_2$. The old I/O-CPQAs Q_1 and Q_2 are discarded.

An I/O-CPQA Q consists of two sorted buffers, called the first buffer $F(Q)$ with $[b, 4b]$ elements and the last buffer $L(Q)$ with $[0, 4b]$ elements, and $k_Q + 2$ deques of records, called the *clean* deque $C(Q)$, the *buffer* deque $B(Q)$ and the *dirty* deques $D_1(Q), \ldots, D_{k_Q}(Q)$, where $k_Q \geq 0$. A *record* $r = (l, p)$ consists of a buffer l of $[b, 4b]$ sorted elements and a pointer p to an I/O-CPQA. A record is *simple* when its pointer p is *nil*. The definition of I/O-CPQAs implies an underlying tree structure when pointers are considered as edges and I/O-CPQAs as subtrees. We define the ordering of the elements in a record r to be all elements of its buffer l followed by all elements in the I/O-CPQA referenced by pointer p. We define the queue order of I/O-CPQA Q to be F, $C(Q)$, $B(Q)$ and $D_1(Q), \ldots, D_{k_Q}(Q)$ and L. It corresponds to an Euler tour over the tree structure. See Figure 6 for an overview of the structure.

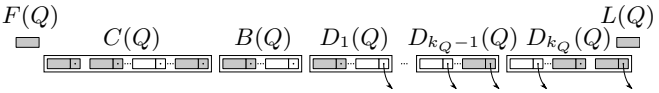


Figure 6: I/O-CPQA Q. Critical records are shown in gray.

Given a record $r = (l, p)$, the minimum and maximum elements in the buffers of r, are denoted by $\min(r) = \min(l)$ and $\max(r) = \max(l)$, respectively. They appear respectively first and last in the queue order of l, since the buffer of r is sorted by value. Given a deque q, the first and the last records are denoted by $\text{first}(q)$ and $\text{last}(q)$, respectively. Also, $\text{rest}(q)$ denotes all records of the deque q excluding the record $\text{first}(q)$. Similarly, $\text{front}(q)$ denotes all records of the deque q excluding the record $\text{last}(q)$. The size $|F|$ ($|L|$) of the buffer F (L) is defined to be the number of elements in F (L). The size $|r|$ of a record r is defined to be the number of elements in its buffer. The size $|q|$ of a deque q is defined to be the

[3] InsertAndAttrite(Q, e) corresponds to CatenateAndAttrite(Q_1, Q_2), where Q_2 contains only element e.

number of records it contains. The size $|Q|$ of the I/O-CPQA Q is defined to be the number of elements (both attrited and non-attrited) that Q contains. For an I/O-CPQA Q we denote by first(Q) and last(Q), respectively the first and last records out of all the records of all the deques $C(Q), B(Q), D_1(Q), \ldots, D_{k_Q}(Q)$ that exist in Q. For an I/O-CPQA Q we maintain the following invariants:

I.1) For every record $r = (l, p)$ where pointer p references I/O-CPQA Q', $\max(l) < \min(Q')$ holds.

I.2) In all deques of Q where record $r_1 = (l_1, p_1)$ precedes record $r_2 = (l_2, p_2)$: $\max(l_1) < \min(l_2)$ holds.

I.3) For the buffer $F(Q)$ and deques $C(Q), B(Q), D_1(Q)$: $\max(F(Q)) < \min(\text{first}(C(Q))) < \max(\text{last}(C(Q))) < \min(\text{first}(B(Q))) < \min(\text{first}(D_1(Q)))$ holds.

I.4) Element $\min(\text{first}(D_1(Q)))$ is the smallest element in the dirty deques $D_1(Q), \ldots, D_k(Q)$.

I.5) $\min(\text{first}(D_1(Q))) < \min(L(Q))$.

I.6) All records in the deques $C(Q)$ and $B(Q)$ are simple.

I.7) $|C(Q)| \geq \sum_{i=1}^{k_Q} |D_i(Q)| + k_Q$.

I.8) $|F(Q)| < b$ holds iff $|Q| < b$ holds.

I.9) If Q is a child of another I/O-CPQA then $F(Q) = \emptyset$ and $L(Q) = \emptyset$ holds.

From Invariants I.2, I.3, I.4 and I.5, we have that $\min(Q) = \min(F(Q))$. We say that an operation *improves* or *aggravates* the inequality of Invariant I.7 by a parameter c for I/O-CPQA Q, when the operation, respectively, increases or decreases by c the *state* of Q:

$$\Delta(Q) = |C(Q)| - \sum_{i=1}^{k_Q} |D_i(Q)| - k_Q$$

To argue about the $\mathcal{O}(1/b)$ amortized I/O bounds we need more definitions. The *critical records* of I/O-CPQA Q are first($C(Q)$), first(rest($C(Q)$)), last($C(Q)$), first($B(Q)$), first($D_1(Q)$), last($D_{k_Q}(Q)$) and last(front($D_{k_Q}(Q)$)), if it exists. Otherwise last($D_{k_Q-1}(Q)$) is critical. By records(Q) we denote all records in Q and the records in the I/O-CPQAs pointed to by Q and its descendants. We call an I/O-CPQA Q *large* if $|Q| \geq b$ and *small* otherwise. We define the following potential functions for large and small I/O-CPQAs. In particular, for large I/O-CPQAs Q the potential $\Phi(Q)$ is defined as

$$\Phi(Q) = \Phi_F(|F(Q)|) + |\text{records}(Q)| + \Phi_L(|L(Q)|),$$

where

$$\Phi_F(x) = \begin{cases} 5 - \frac{2x}{b}, & b \leq x < 2b \\ 1, & 2b \leq x < 3b \\ \frac{2x}{b} - 5, & 3b \leq x \leq 4b \end{cases}$$

and

$$\Phi_L(x) = \begin{cases} \frac{x}{b}, & 0 \leq x < b \\ 1, & b \leq x \leq 3b \\ \frac{2x}{b} - 5, & 3b < x \leq 4b \end{cases}$$

For small I/O-CPQAs Q, the potential $\Phi(Q)$ is defined as

$$\Phi(Q) = \frac{3|Q|}{b}$$

The total potential Φ_T is defined as

$$\Phi_T = \sum_Q \Phi(Q) + \sum_{Q, b \leq |Q|} 1,$$

where the first sum is the total potential of all I/O-CPQAs Q and the second sum counts the number of large I/O-CPQAs Q.

Operations. In the following, we describe the algorithms that implement the operations supported by the I/O-CPQA Q. Most of the operations call the auxiliary operations Bias(Q) and Fill(Q), which we describe last. Bias improves the inequality of I.7 for Q by at least 1 if Q contains any records. Fill(Q) ensures I.8.

FindMin(Q) returns the value $\min(F(Q))$.

DeleteMin(Q) removes element $e = \min(F(Q))$ from the first buffer $F(Q)$, calls Fill(Q) and returns e.

CatenateAndAttrite(Q_1, Q_2) creates a new I/O-CPQA Q_1' by modifying Q_1 and Q_2, and by calling Bias(Q_1'), Bias(Q_2), Fill(Q_1') and Fill(Q_2).

If $|Q_1| < b$ holds, then Q_1 consists only of the first buffer $F(Q_1)$. Let $F'(Q_1)$ be the non-attrited elements of $F(Q_1)$, under attrition by $\min(F(Q_2))$. Prepend $F'(Q_1)$ onto the first buffer $F(Q_2)$ of Q_2. If this prepend causes $F(Q_2) > 4b$, then we take the last $2b$ elements out of $F(Q_2)$, make a new record out of them and we prepend it onto the deque $C(Q_2)$.

If $|Q_2| < b$ holds, then Q_2 only consists of $F(Q_2)$. If $|Q_1| < b$ then we delete attrited elements in $F(Q_1)$ and append $F(Q_2)$ to $F(Q_1)$. We now assume that $|Q_1| \geq b$. We have three cases, depending on how much of Q_1 is attrited by Q_2. Let $r = (l, \cdot) = \text{last}(Q_1)$ and let $e = \min(Q_2)$.

1. $e \leq \min(r)$: Delete r. We now have four cases:
   1) If $e \leq \min(F(Q_1))$ holds, we discard I/O-CPQA Q_1 and set $Q_1' = Q_2$.
   2) Else if $e \leq \max(\text{last}(C(Q_1)))$ holds, we prepend $F(Q_1)$ onto $C(Q_1)$, set $F(Q_1') = \emptyset$, $C(Q_1') = \emptyset$, $B(Q_1') = C(Q_1)$, $k_{Q_1'} = 0$ and $L(Q_1') = F(Q_2)$. We call Bias(Q_1') once to restore I.7 and then call Fill(Q_1') once to restore Invariant I.8.
   3) Else if $e \leq \min(\text{first}(B(Q_1)))$ or $e \leq \min(\text{first}(D_1(Q_1)))$ holds, we set $Q_1' = Q_1$ and $k_{Q_1'} = 0$ and set $L(Q_1') = F(Q_2)$. If $e \leq \min(\text{first}(B(Q_1)))$ holds, we set $B(Q_1') = \emptyset$, else we set $B(Q_1') = B(Q_1)$.
   4) Else, let $L'(Q_1)$ be the non-attrited elements under attrition by $\min(F(Q_2))$. If $|L'(Q_1)| + |F(Q_2| \leq 4b$ then append $F(Q_2)$ to $L'(Q_1)$, else $|L'(Q_1)| + |F(Q_2| > 4b$ so take the first $4b$ elements of $L'(Q_1)$ and $F(Q_2)$ and make into a new record in a new last dirty queue of Q_1', leave the rest in $L(Q_1')$, set $k_{Q_1'} = k_{Q_1} + 1$ and call Bias(Q_1') twice to restore I.7.
2. Else if $e \leq \min(L(Q_1))$, we set $Q_1' = Q_1$ and $L(Q_1') = F(Q_2)$.
3. Else $\min(L(Q_1)) < e$: Let l' be the non-attrited elements of l, under attrition by $\min(L(Q_1))$, and $L'(Q_1)$ be the non-attrited elements, under attrition by e. If

$|L'(Q_1)| + |F(Q_2)| > 4b$ holds, we do the following: if $|l'| < |l|$ holds, we put the first $4b - |l'|$ elements of $L'(Q_1)$ and $F(Q_2)$ into l along with l'. Moreover, if we still have more than $3b$ elements left in $L'(Q_1)$ and $F(Q_2)$, we put the first $3b$ elements into a new last record of $D_{k_{Q_1}}(Q_1)$. Finally, we leave the remaining elements in $L(Q_1)$. If we added a new last record to $D_{k_{Q_1}}(Q_1)$, we also call BIAS(Q) once.

We have now entirely dealt with the cases where $|Q_1| < b$ or $|Q_2| < b$ holds, so in the following we assume that $|Q_1| \geq b$ and $|Q_2| \geq b$ hold, i.e. any I/Os incurred in the cases (1–4) below are already paid for, since the total number of large I/O-CPQAs decreases by one. Let $e = \min(Q_2)$.

1) If $e \leq \min(F(Q_1))$ holds, we discard I/O-CPQA Q_1 and set $Q'_1 = Q_2$.

2) Else if $e \leq \max(\text{last}(C(Q_1)))$ holds, we prepend $F(Q_1)$ onto $C(Q_1)$ and $F(Q_2)$ onto $C(Q_2)$. We remove the simple record $(l, \cdot) = \text{first}(C(Q_2))$ from $C(Q_2)$, set $Q'_1 = Q_1$, $F(Q'_1) = \emptyset$, $C(Q'_1) = \emptyset$, $B(Q'_1) = C(Q_1)$, $D_1(Q'_1) = (l, p)$, $k_{Q'_1} = 1$, $L(Q'_1) = L(Q_2)$ and $L(Q'_2) = \emptyset$, where p points to Q'_2 if it exists. This gives $\Delta(Q'_1) = -2$, thus we call BIAS(Q'_1) twice and FILL(Q'_1) once.

3) Else if $e \leq \min(\text{first}(B(Q_1)))$ or $e \leq \min(\text{first}(D_1(Q_1)))$ holds, we prepend $F(Q_2)$ onto $C(Q_2)$ and remove the simple record $(l, \cdot) = \text{first}(C(Q_2))$ from $C(Q_2)$, set $Q'_1 = Q_1$, $D_1(Q'_1) = (l, p)$, $k_{Q'_1} = 1$, $L(Q'_1) = L(Q_2)$, $L(Q'_2) = \emptyset$ and set p to point to Q'_2, if it exists. If $e \leq \min(\text{first}(B(Q_1)))$ holds, we set $B(Q'_1) = \emptyset$, else we set $B(Q'_1) = B(Q_1)$. This gives $\Delta(Q'_1) = -2$ in the worst case, thus we call BIAS(Q'_1) twice.

4) Else let $L'(Q_1)$ be the non-attrited elements of $L(Q_1)$, under attrition by $F(Q_2)$. If $|L'(Q_1)| + |F(Q_2)| \leq 4b$ holds, then we make $L'(Q_1)$ and $F(Q_2)$ into the first record of $C(Q_2)$. Else we make them into the first two records of $C(Q_2)$ of size $\lfloor (|L'(Q_1)| + |F(Q_2)|)/2 \rfloor$ and $\lceil (|L(Q_1)| + |F(Q_2)|)/2 \rceil$ each. We set $Q'_1 = Q_1$, $F(Q'_2) = \emptyset$, $L(Q'_1) = L(Q_2)$, $L(Q'_2) = \emptyset$, remove $(l_2, \cdot) = \text{first}(C(Q_2))$ from $C(Q_2)$. Moreover, we add (l_2, p) as a new record in $D_{k_{Q_1}+1}(Q'_1)$, where p points to the rest of Q'_2, if it exists, and set $k_{Q'_1} = k_{Q_1} + 1$. All this aggravates the inequality of I.7 for Q'_1 by at most 2, so we call BIAS(Q'_1) twice.

FILL(Q) restores Invariant I.8, if it is violated. In particular, if $|F(Q)| < b$ and $|Q| \geq b$, let $r = (l, \cdot) = \text{first}(C(Q))$. If $|l| \geq 2b$ holds, then we take the b first elements of l and append them to $F(Q)$. Else $|l| < 2b$ holds, so we append l to $F(Q)$, discard r and call BIAS(Q) once.

BIAS(Q) improves the inequality of I.7 for Q by at least 1 if Q contains any records. It also ensures that Invariant I.8 is maintained. We distinguish two basic cases with respect to $|B(Q)|$, namely $|B(Q)| = 0$ and $|B(Q)| > 0$.

1) $|B(Q)| > 0$: We have two cases depending on if $k_Q \geq 1$ or $k_Q = 0$.

   1) $k_Q = 0$: Let $e = \min(L(Q))$, if it exists. We remove the first record $r_1 = (l_1, \cdot) = \text{first}(B(Q))$ from $B(Q)$. Let l'_1 be the non-attrited elements of l_1, under attrition by element e. If $|l'_1| = |l_1|$ holds nothing is attrited, so we just add $r_1 = (l_1, \cdot)$ at the end of $C(Q)$.
   Else $|l'_1| < |l_1|$ holds, so we set $B(Q) = \emptyset$. If $|l'_1| \geq b$ holds, then we make record r_1 with buffer l'_1 into the new last record of $C(Q)$. Else $|l'_1| < b$ holds, so if $|l'_1| + |L(Q)| \leq 3b$ also holds, we add l'_1 to $L(Q)$ and discard r_1. Else $|l'_1| + |L(Q)| > 3b$ also holds, so we take the $2b$ first elements of l'_1 and $L(Q)$ and put them into r_1, making it the new last record of $C(Q)$.

   2) $k_Q \geq 1$: Let $e = \min(\text{first}(D_1(Q)))$. We remove the first record $r_1 = (l_1, \cdot) = \text{first}(B(Q))$ from $B(Q)$. Let l'_1 be the non-attrited elements of l_1, under attrition by element e.
   If $|l'_1| = |l_1|$ or $b \leq |l'_1| < |l_1|$ holds, we just add $r_1 = (l'_1, \cdot)$ at the end of $C(Q)$. Else $|l'_1| < b$ and $|l'_1| < |l_1|$ hold. We set $B(Q) = \emptyset$. Let $r_2 = (l_2, p_2) = \text{first}(D_1(Q))$. If $|l'_1| + |l_2| \leq 4b$ holds, we discard r_1 and prepend l'_1 onto l_2 of r_2. Else $|l'_1| + |l_2| > 4b$ holds, so we take the first $2b$ elements of l'_1 and l_2 and put them in r_1, making it the new last record of $C(Q)$. If this causes $\min(L(Q)) \leq \min(\text{first}(D_1(Q)))$, we discard all dirty queues.

   If r_1 was discarded, then we have that $|B(Q)| = 0$ and we call BIAS recursively, which will not invoke this case again. In all cases the inequality of I.7 for Q is improved by 1.

2) $|B(Q)| = 0$: we have three cases depending on the number of dirty queues, namely cases $k_Q > 1$, $k_Q = 1$ and $k_Q = 0$.

   1) $k_Q > 1$: If $\min(L(Q)) \leq \min(\text{first}(D_{k_Q}(Q)))$ holds, we set $k_Q = k_Q - 1$ and discard $D_{k_Q}(Q)$. This improves the inequality of I.7 for Q by at least 2. Else let $e = \min(\text{first}(D_{k_Q}(Q)))$.
   If $e \leq \min(\text{last}(D_{k_Q-1}(Q)))$ holds, we remove the record $\text{last}(D_{k_Q-1}(Q))$ from $D_{k_Q-1}(Q)$. This improves the inequality of I.7 for Q by 1.
   If $\min(\text{last}(D_{k_Q-1}(Q))) < e \leq \max(\text{last}(D_{k_Q-1}(Q)))$ holds, we remove record $r_1 = (l_1, p_1) = \text{last}(D_{k_Q-1}(Q))$ from $D_{k_Q-1}(Q)$, and let $r_2 = (l_2, p_2) = \text{first}(D_{k_Q}(Q))$. We delete any elements in l_1 that are attrited by e, and let l'_1 denote the set of non-attrited elements. If $|l'_1| + |l_2| \leq 4b$ holds, we prepend l'_1 onto l_2 of r_2 and discard r_1. Else we take the first $\lfloor (|l'_1| + |l_2|)/2 \rfloor$ elements of l'_1 and l_2 and replace r_1 of $D_{k_Q-1}(Q)$ with them. Finally, we concatenate $D_{k_Q-1}(Q)$ and $D_{k_Q}(Q)$ into a single deque. This improves the inequality of I.7 for Q by at least 1.
   Else $\max(\text{last}(D_{k_Q-1}(Q))) < e$ holds and we just concatenate the deques $D_{k_Q-1}(Q)$ and $D_{k_Q}(Q)$, which improves the inequality of I.7 for Q by 1.

   2) $k_Q = 1$: In this case Q contains only deques $C(Q)$ and $D_1(Q)$. Let $r = (l, p) = \text{first}(D_1(Q))$. If $\min(L(Q)) \leq \min(\text{first}(\text{rest}(D_1(Q))))$ holds, we discard all dirty queues, except for record r of $D_1(Q)$.
   If $\min(L(Q)) \leq \max(l)$ holds, we discard all the dirty deques and let l' be the non-attrited elements of l. If $|l'| + |L(Q)| \leq 3b$ holds, we prepend l' onto $L(Q)$. Else $|l'| + |L(Q)| > 3b$ holds, so we take the first $2b$ elements of l' and $L(Q)$ and make them the new last record of $C(Q)$ and leave the rest in $L(Q)$. This improves the inequality of I.7 for Q by 1.

Else $\max(\ell) < \min(L(Q))$ holds, so we remove r and insert buffer l into a new record at the end of $C(Q)$. This improves the inequality of I.7 for Q by at least 1. If r is not simple, let the pointer p of r reference I/O-CPQA Q'. We restore I.6 for Q by *merging* I/O-CPQAs Q and Q' into one I/O-CPQA; see Figure 7. In particular, let $e = \min(\min(\text{first}(D_1(Q))), \min(L(Q)))$.

We proceed as follows: If $e \leq \min(Q')$ holds, we discard Q'. Else if $\min(\text{first}(C(Q'))) < e \leq \max(\text{last}(C(Q'))$ holds, we set $B(Q) = C(Q')$ and discard the rest of Q'. In both cases, the inequality of I.7 for Q remains unaffected.

Else if $\max(\text{last}(C(Q')) < e \leq \min(\text{first}(D_1(Q')))$ holds, we concatenate the deque $C(Q')$ at the end of $C(Q)$. If moreover $\min(\text{first}(B(Q'))) < e$ holds, we set $B(Q) = B(Q')$. Finally, we discard the rest of Q'. This improves the inequality of I.7 for Q by $|C(Q')|$.

Else $\min(\text{first}(D_1(Q'))) < e$ holds. We concatenate the deque $C(Q')$ at the end of $C(Q)$, we set $B(Q) = B(Q')$, we set $D_1(Q'), \ldots, D_{k_{Q'}}(Q')$ as the first $k_{Q'}$ dirty queues of Q and we set $D_1(Q)$ as the last dirty queue of Q. This improves the inequality of I.7 for Q by $\Delta(Q') \geq 0$, since Q' satisfied Invariant I.7 before the operation.

3) $k_Q = 0$: If all deques are empty, $L(Q) \neq \emptyset$ and $|F(Q)| \leq 2b$ hold, we take the first b elements of $L(Q)$ and append to $F(Q)$. The inequality of I.7 for Q remains $\Delta(Q) = 0$.

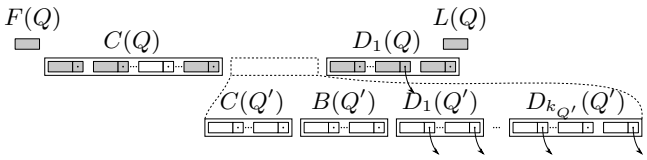


Figure 7: Merging I/O-CPQAs Q and Q'. This case can only occur when $B(Q) = \emptyset$ and $k_Q = 1$.

THEOREM 3. *An I/O-CPQA supports* FINDMIN, DELETEMIN, CATENATEANDATTRITE *and* INSERTANDATTRITE *in $\mathcal{O}(1)$ I/Os per operation. It occupies $\mathcal{O}((n-m)/B)$ blocks after calling* CATENATEANDATTRITE *and* INSERTANDATTRITE *n times and* DELETEMIN *m times, respectively.*

All operations are supported by a set of ℓ I/O-CPQAs in $\mathcal{O}(1/b)$ amortized I/Os, when $M = \Omega(\ell b)$, using $\mathcal{O}((n-m)/b)$ blocks of space, for any parameter $1 \leq b \leq B$.

PROOF. (Sketch) The correctness follows by closely noticing that we maintain Invariants I.1–I.9, which in turn imply that DELETEMIN(Q) and FINDMIN(Q) always return the minimum element of Q. The $\mathcal{O}(1)$ worst case I/O bound is trivial as every operation only accesses $\mathcal{O}(1)$ records. Although BIAS is recursive, notice that in the case where $|B(Q)| > 0$, BIAS only calls itself after making $|B(Q)| = 0$, so it will not end up in this case again. We elaborate on all the operations that modify the I/O-CPQA in order to argue for the amortized bounds:

DELETEMIN: If after the call $|F(Q)| \geq b$ holds, no I/Os are incurred and the amortized cost of $\leq \frac{3}{b}$ pays for increasing the potential. Otherwise $\Phi_F(|F(Q)|) \geq 3$ pays for any I/Os to call FILL and BIAS.

CATENATEANDATTRITE: $|Q_1| < b$: If $|F'(Q_1)| + |F(Q_2)| \leq 4b$ holds, $\Phi(|F(Q_1)|)$ pays for any increase in potential. Else the new record of $C(Q_2)$ is paid for by $\Delta(\Phi_T) > 1$.

$|Q_2| < b$: In cases (1) and (2) the potential decreases. In case (3), the potential does not change if $|L'(Q_1)| + |F(Q_2)| > 4b$. If $L'(Q_1)$ and $F(Q_2)$ still contain $> 3b$ elements, the change in potential is paid for by $\Delta(\Phi_T) > 0$.

In the following cases, both Q_1 and Q_2 are large. Since concatenating them decreases by one the number of large I/O-CPQA's, the potential decreases by at least 1, which is enough to pay for any other I/Os also incurred by BIAS and FILL. So we only need to argue that the potential does not increase in any of the cases. In fact, in cases (1 - 3) the potential only decreases. In case (4), if we make one record, it is paid for by $\Phi_F(|F(Q_2)|) \geq 1$. Otherwise the second record is paid for by $\Phi_L(|L(Q_1)|) \geq 1$ if moreover $|L'(Q_1)| + |F(Q_2)| > 4b$ holds, or by $\Phi_L(|L(Q_1)|) + \Phi_F(|F(Q_2)|) > 2$ otherwise.

All I/Os in FILL and BIAS have been paid for by a decrease in potential caused by their caller. Thus, it suffices to argue that these operations do not increase the potential.

FILL: Indeed, $\Phi_F(|F(Q)|)$ only decreases, when $|F(Q)| < b$ and $|Q| \geq b$ hold.

BIAS: Indeed, cases (1) and (2.1) do not create new records. Similarly for (2.2), unless $|l'| + |L(Q)| \leq 3b$ holds, where r pays for increasing the potential by 1. In (2.3) $\Phi_F(|F(Q)|)$ or $\Phi_L(|L(Q)|)$ decreases. □

Catenating a set of I/O-CPQAs. The following lemma is required by the dynamic structure of the next section.

LEMMA 5. *A set of I/O-CPQAs Q_i for $i \in [1, \ell]$ can be concatenated into a single I/O-CPQA without any access to external memory, by calling only* CATENATEANDATTRITE *operations, provided that for all i:*

1. *$\Delta(Q_i) \geq 2$ holds, unless Q_i contains only one record, in which case $\Delta(Q_i) \geq 1$ suffices.*
2. *The critical records of Q_i are loaded in main memory.*

4.2 Final Dynamic Top-Open Structure

The data structure consists of a base tree, implemented as a dynamic $(a, 2a)$-tree where the leaves store between k and $2k$ elements. We set $a = \lceil 2B^\epsilon \rceil$ and $k = B$, for a given $0 \leq \epsilon \leq 1$. The base tree indexes the $<_x$-ordering of $\widetilde{P}$, and is augmented with confluently persistent I/O-CPQAs with buffer size $b = B^{1-\epsilon}$ as secondary structures. In particular, after constructing the base tree, we augment it with secondary I/O-CPQAs in a bottom-up manner, as follows. For every leaf we make one I/O-CPQA over its elements, and execute an appropriate amount of BIAS operations, such that the state of the I/O-CPQA satisfies Lemma 5. We associate the I/O-CPQA with the leaf. In a second pass over the leaves, we gather its critical records into a *representative block* in its parent. The procedure continues one level above. For every internal node u, we access the representative blocks that contain the critical records of the children I/O-CPQAs of u, and CATENATEANDATTRITE them into a new I/O-CPQA as implied by Lemma 5. We execute BIAS on the I/O-CPQA enough times such that its state also satifies Lemma 5. We associate the I/O-CPQA with u. After the level has been processed, we create the representative blocks for I/O-CPQAs associated with the nodes of the level, in the

same way as described above. The augmentation ends at the root node of the base tree. We will ensure that our algorithms access the I/O-CPQA associated with a node through the representative block stored at the parent of the node. Thus, it will suffice to explicitly store only the representative blocks in every internal node and not its associated I/O-CPQA.

Since every leaf contains $\mathcal{O}(B)$ elements, the base tree has $\mathcal{O}(n/B)$ leaves and thus also $\mathcal{O}(n/B)$ internal nodes. Every internal node has $\Theta(B^\epsilon)$ children, each associated with an I/O-CPQA with $\mathcal{O}(1)$ critical records of size $\mathcal{O}(B^{1-\epsilon})$. Thus the representative blocks stored in the internal node occupy $\mathcal{O}(1)$ blocks of space. Thus the structure occupies $\mathcal{O}(n/B)$ blocks in total. Assume that $\widetilde{P}$ is already sorted by the $<_x$-ordering. The leaves' I/O-CPQAs are created in $\mathcal{O}(1)$ I/Os, since they contain at most $\mathcal{O}(B)$ elements. All representative blocks are created in $\mathcal{O}(n/B)$ I/Os. To create the internal nodes' I/O-CPQAs, we need only $\mathcal{O}(1)$ I/Os to access the representative blocks and to execute Bias on the resulting I/O-CPQA. Its representative blocks residing in memory thus are written on disk in $\mathcal{O}(1)$ I/Os. Thus the total preprocessing cost is $\mathcal{O}(n/B)$; the structure is SABE.

Updates. To insert (resp. delete) a point p into (resp. from) P, we insert (resp. delete) $\widetilde{p} = (\widetilde{x}_p, \widetilde{y}_p)$ in the structure. In particular, we first find the leaf to insert (resp. delete) that contains the predecessor of $\widetilde{x}_p$ (resp. contains $\widetilde{x}_p$), by a top-down traversal of the path from the root of the base tree. For every node u on the path, we also discard the part of its representative block corresponding to the child that the search path goes into, and u's associated I/O-CPQA by executing in reverse the operations that created it. Next we insert (resp. delete) $\widetilde{p}$ into (from) the accessed leaf, and rebalance the base tree by executing the appropriate splits and merges on the nodes along the path in a bottom-up manner. Moreover, we recompute the I/O-CPQA of every accessed node on the path, as described above. The total update I/Os are $\mathcal{O}(\log_{2B^\epsilon}(n/B))$ in the worst case, since we spend $\mathcal{O}(1)$ I/Os to rebalance every accessed node and to recompute its secondary structures.

Queries. To report the skyline points of P that reside within a given top-open query range $[\alpha_1, \alpha_2] \times [\beta, \infty[$, we first traverse top-down the two search paths $\widetilde{\pi_1} = \pi\pi_1$ and $\widetilde{\pi_2} = \pi\pi_2$ from the root of the base tree to the leaves ℓ_1 and ℓ_2 that contain points of $\widetilde{P}$ whose $<_x$-ordering succeed and precede the query parameters α_1 and α_2, respectively. Let node u be on the path $\pi_1 \cup \pi_2$, and let $c(u)$ be the children nodes of u whose subtrees are fully contained within $[\alpha_1, \alpha_2]$. For every u, we load its representative block into memory in order to access the critical records of the I/O-CPQAs associated with $c(u)$ and to CatenateAndAttrite them into a temporary I/O-CPQA, as implied by Lemma 5. We consider the temporary I/O-CPQAs of nodes u and the I/O-CPQAs of the leaves ℓ_1 and ℓ_2 from right to left, and we CatenateAndAttrite them into one auxiliary I/O-CPQA. The I/O-CPQAs for ℓ_1 and ℓ_2 are created only on the points within the x-range $[\alpha_1, \alpha_2]$ in $\mathcal{O}(1)$ I/Os.

To report the skyline points within the query range, we call DeleteMin on the auxiliary I/O-CPQA. The procedure stops as soon as a point with $\widetilde{y}_p > -\beta$ is returned, or when the auxiliary I/O-CPQA becomes empty.

There are $\mathcal{O}(\log_{2B^\epsilon}(n/B))$ nodes on $\pi_1 \cup \pi_2$ and we spend $\mathcal{O}(1)$ I/Os to access the representative block of each node. After this, the construction of the auxiliary I/O-CPQA costs $\mathcal{O}(\log_{2B^\epsilon}(n/B))$ I/Os. Reporting the k output points costs $\mathcal{O}(\frac{k}{B^{1-\epsilon}} + 1)$ I/Os. Therefore the query takes $\mathcal{O}(\log_{2B^\epsilon}(n/B) + \frac{k}{B^{1-\epsilon}})$ I/Os in total. We conclude that:

Theorem 4. *There is an indivisible linear-size dynamic data structure on n points in $\mathbb{R}^2$ that supports top-open range skyline queries in $\mathcal{O}(\log_{2B^\epsilon}(n/B) + k/B^{1-\epsilon})$ I/Os when k points are reported, and updates in $\mathcal{O}(\log_{2B^\epsilon}(n/B))$ I/Os for any parameter $0 \leq \epsilon \leq 1$. The structure can be constructed in $\mathcal{O}(n/B)$ I/Os, assuming an initial sorting on the input points' x-coordinates.*

5. GENERAL RANGE SKYLINE QUERIES

We now move on to discuss the other variants of range skyline reporting that are neither symmetric to nor subsumed by top-open queries. It would be nice if they could be answered in $\mathcal{O}(\log_B n + k/B)$ I/Os by a linear-size structure. Unfortunately, we will prove its impossibility. In fact, even sub-polynomial query cost is already unachievable for anti-dominance queries, let alone left-open and 4-sided queries. In fact, anti-dominance, left-open and 4-sided are just as hard as each other. Next, we will formally establish these facts. Refer to the full version for the proofs.

5.1 A Query Lower Bound

By making a crucial observation on a variant of the low-discrepancy point set proposed by Chazelle and Liu [8], we manage to prove the next geometric fact:

Lemma 6. *For any integer $\omega \geq 1$ and $\lambda \geq 1$, there is a set P of ω^λ points in $\mathbb{R}^2$ and a set G of $\lambda\omega^{\lambda-1}$ anti-dominance queries such that (i) each query in G retrieves d points of P, and (ii) at most one point in P is returned by two different queries in G simultaneously.*

We use the term (ω, λ)-*input* to refer to the point set P obtained in Lemma 6 after ω and λ have been fixed. We deploy such input sets to derive:

Lemma 7. *Regarding anti-dominance queries on n points in $\mathbb{R}^2$, any structure (in the indexability model) of at most cn/B blocks must incur $\Omega((n/B)^{1/(25c)} + k/B)$ I/Os to answer a query in the worst case, where $c \geq 1$ is a constant and k is the result size.*

Theorem 5. *Regarding anti-dominance queries on n points, any linear-size structure under the indexability model must incur $\Omega((n/B)^\epsilon + k/B)$ I/Os answering a query in the worst case, where $\epsilon > 0$ can be an arbitrarily small constant, and k is the result size.*

Remarks. In the full version, we utilize Lemma 6 to prove that any internal memory pointer-based data structure that supports anti-dominance queries in $\mathcal{O}(\log^{\mathcal{O}(1)} + k)$ time requires $\Omega(n\frac{\log n}{\log\log n})$ space. Thus, the dynamic structure of [7] for 4-sided queries occupies optimal space within a $\mathcal{O}(\log\log n)$ factor, for the attained query time.

5.2 Query-Optimal Structure

The above lower bound is tight. In fact, we are able to prove a stronger fact: a *4-sided* query can be answered in $\mathcal{O}((n/B)^\epsilon + k/B)$ I/Os by a linear-size dynamic structure. Deferring the details to the full version, we claim:

THEOREM 6. *There is an indivisible linear-size structure on n points in $\mathbb{R}^2$ such that, 4-sided range skyline queries can be answered in $\mathcal{O}((n/B)^\epsilon + k/B)$ I/Os, where k is the number of reported points. The query cost is optimal under the indexability model. The structure can be updated in $\mathcal{O}(\log(n/B))$ amortized I/Os.*

ACKNOWLEDGEMENTS

The work of Yufei Tao and Jeonghun Yoon was supported in part by (i) projects GRF 4166/10, 4165/11, and 4164/12 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).

6. REFERENCES

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

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

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

[4] I. Bartolini, P. Ciaccia, and M. Patella. Efficient sort-based skyline evaluation. *TODS*, 33(4), 2008.

[5] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion B-tree. *VLDB J.*, 5(4):264–275, 1996.

[6] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.

[7] G. Brodal and K. Tsakalidis. Dynamic planar range maxima queries. In *ICALP*, pages 256–267, 2011.

[8] B. Chazelle and D. Liu. Lower bounds for intersection searching and fractional cascading in higher dimension. *JCSS*, 68(2):269 – 284, 2004.

[9] B.-C. Chen, R. Ramakrishnan, and K. LeFevre. Privacy skyline: Privacy with multidimensional adversarial knowledge. In *VLDB*, pages 770–781, 2007.

[10] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang. Skyline with presorting: Theory and optimizations. In *Intelligent Information Systems*, pages 595–604, 2005.

[11] F. d'Amore, P. G. Franciosa, R. Giaccio, and M. Talamo. Maintaining maxima under boundary updates. In *Italian Conference on Algorithms and Complexity*, pages 100–109, 1997.

[12] A. Das, P. Gupta, A. Kalavagattu, J. Agarwal, K. Srinathan, and K. Kothapalli. Range aggregate maximal points in the plane. In *WALCOM: Algorithms and Computation*, volume 7157, pages 52–63, 2012.

[13] J. V. den Bercken and B. Seeger. Query processing techniques for multiversion access methods. In *VLDB*, pages 168–179, 1996.

[14] J. V. den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *VLDB*, pages 406–415, 1997.

[15] G. N. Frederickson and S. Rodger. A new approach to the dynamic maintenance of maximal points in a plane. *Discrete & Computational Geometry*, 5:365–374, 1990.

[16] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *JCSS*, 47(3):424 – 436, 1993.

[17] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *SIGMOD*, pages 47–57, 1984.

[18] J. M. Hellerstein, E. Koutsoupias, D. P. Miranker, C. H. Papadimitriou, and V. Samoladas. On a model of indexability and its bounds for range queries. *JACM*, 49(1):35–55, 2002.

[19] Z. Huang, H. Lu, B. C. Ooi, and A. K. H. Tung. Continuous skyline queries for moving objects. *TKDE*, 18(12):1645–1658, 2006.

[20] R. Janardan. On the dynamic maintenance of maximal points in the plane. *IPL*, 40(2):59 – 64, 1991.

[21] A. K. Kalavagattu, A. S. Das, K. Kothapalli, and K. Srinathan. On finding skyline points for range queries in plane. In *CCCG*, 2011.

[22] H. Kaplan and R. E. Tarjan. Purely functional, real-time deques with catenation. *JACM*, 46(5):577–603, 1999.

[23] S. Kapoor. Dynamic maintenance of maxima of 2-d point sets. *SIAM J. of Comp.*, 29:1858–1877, 2000.

[24] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.

[25] K. G. Larsen and R. Pagh. I/O-efficient data structures for colored range and prefix reporting. In *SODA*, pages 583–592, 2012.

[26] M. D. Morse, J. M. Patel, and H. V. Jagadish. Efficient skyline computation over low-cardinality domains. In *VLDB*, pages 267–278, 2007.

[27] M. H. Overmars and J. van Leeuwen. Maintenance of configurations in the plane. *JCSS*, 23(2):166 – 204, 1981.

[28] D. Papadias, Y. Tao, G. Fu, and B. Seeger. Progressive skyline computation in database systems. *TODS*, 30(1):41–82, 2005.

[29] A. D. Sarma, A. Lall, D. Nanongkai, and J. Xu. Randomized multi-pass streaming skyline algorithms. *PVLDB*, 2(1):85–96, 2009.

[30] M. Sharifzadeh, C. Shahabi, and L. Kazemi. Processing spatial skyline queries in both vector spaces and spatial network databases. *TODS*, 34(3), 2009.

[31] C. Sheng and Y. Tao. On finding skylines in external memory. In *PODS*, pages 107–116, 2011.

[32] R. Sundar. Worst-case data structures for the priority queue with attrition. *IPL*, 31:69–75, 1989.

[33] Y. Tao and D. Papadias. Maintaining sliding window skylines on data streams. *TKDE*, 18(3):377–391, 2006.

[34] J. S. Vitter. Algorithms and data structures for external memory. *Found. Trends Theor. Comput. Sci.*, 2(4):305–474, 2008.

[35] P. Wu, D. Agrawal, Ö. Egecioglu, and A. E. Abbadi. Deltasky: Optimal maintenance of skyline deletions without exclusive dominance region generation. In *ICDE*, pages 486–495, 2007.

[36] H. Yuan and M. J. Atallah. Data structures for range minimum queries in multidimensional arrays. In *SODA*, pages 150–160, 2010.