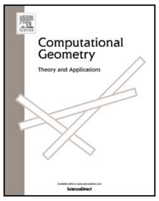
Contents lists available at ScienceDirect
Computational Geometry: Theory and Applications
www.elsevier.com/locate/comgeo



I/O-efficient 2-d orthogonal range skyline and attrition priority queues ☆

Casper Kejlberg-Rasmussen [a], Yufei Tao [b], Konstantinos Tsakalidis [c], Kostas Tsichlas [d], Jeonghun Yoon [e]

[a] *Department of Computer Science, Aarhus University, Denmark*
[b] *Department of Computer Science & Engineering, Chinese University of Hong Kong, Hong Kong*
[c] *Department of Computer Science, University of Liverpool, UK*
[d] *Computer Engineering & Informatics Department, University of Patras, Greece*
[e] *Division of Web Science & Technology, Advanced Institute of Science & Technology, Republic of Korea*

A R T I C L E I N F O

Article history:
Available online xxxx

Keywords:
Computational geometry
Range searching
Skyline
External memory

A B S T R A C T

We present the first static and dynamic external memory data structures for variants of *2-d orthogonal range skyline reporting* with worst-case logarithmic query and update I/O-complexity. The results are obtained by using *persistent data structures* and by extending the *attrition priorities queues* of Sundar (1989) [26] to also support real-time concatenation, a result of independent interest. We show that the problem is as hard as standard 2-d orthogonal range reporting in the indexability model by a lower bound on the I/O-complexity of 2-d orthogonal *anti-dominance* skyline reporting queries.

1. Introduction

We study *orthogonal range skyline reporting* in the I/O model [2] of computation in external memory. The *skyline* or *maximal points* of a planar pointset is the subset of points that are not *dominated* by any other point in the set, i.e. no other point has both coordinates larger than any maximal point. Naturally, the skyline forms an orthogonal *staircase* of the maximal points that appear in decreasing y-order, when considered in increasing x-order. See Fig. 1a for an example. An *orthogonal range skyline reporting query* reports the skyline of the subset of points that are contained in a given *4-sided rectangle*. See Fig. 1b for an example. Standard *2-d orthogonal range reporting queries* [3–7] (where all points in the range are reported, not only the skyline) are harder than the special case of 2-d orthogonal *3-sided* range reporting queries (where the query rectangle is unbounded in one side), *regardless* of which side is unbounded. However for skyline reporting, due to the orientation of the “dominance relation”, we only consider as an easier special case *2-d orthogonal 3-sided range skyline reporting queries* for 3-sided rectangles with their *top side unbounded* (Fig. 1c), since we will show that all other cases of 3-sided rectangles, as well as the 4-sided case, essentially reduce to the harder *2-d orthogonal anti-dominance skyline reporting queries*, where the query rectangle is unbounded in *both the left and bottom sides* (Fig. 1d).

☆ A preliminary version of this work was published in the proceedings of the 32nd symposium on Principles of Database Theory (PODS ’13) [1]. The work of Yufei Tao and Jeonghun Yoon was partially supported by (i) GRF 4166/10, 4165/11, 4164/12 from HKRGC and (ii) project R31-30007 of the World Class University program under the National Research Foundation of Korea, Ministry of Education, Science and Technology of Korea. The work of Konstantinos Tsakalidis was partially supported by NSF grants CCF-1319648 and CCF-1533564.

E-mail addresses: ckr@cs.au.dk (C. Kejlberg-Rasmussen), taoyf@cse.cuhk.edu.hk (Y. Tao), K.Tsakalidis@liverpool.ac.uk (K. Tsakalidis), ktsichlas@ceid.upatras.gr (K. Tsichlas), jeonghun.yoon@kaist.ac.kr (J. Yoon).

https://doi.org/10.1016/j.comgeo.2020.101689
0925-7721/© 2020 Published by Elsevier B.V.

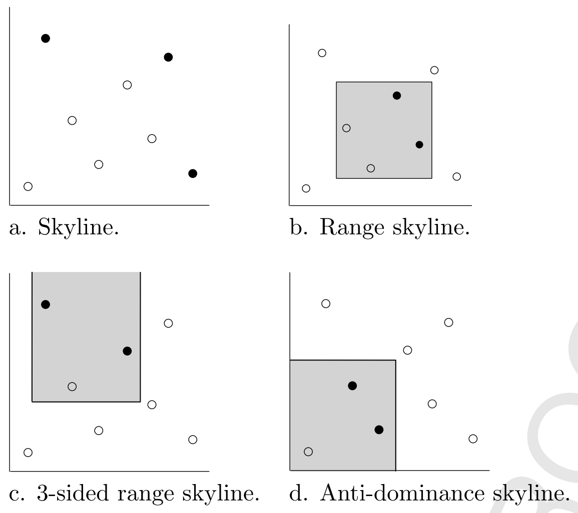


Fig. 1. Variants of 2-d orthogonal range skyline reporting.

1.1. Previous results

Internal memory. In the word-RAM model, Kalavagattu et al. [8] present static *2-d orthogonal range skyline reporting* data structures that support queries in $O(\log n + k)$ time, where n is the number of points and k is the size of the reported skyline, using superlinear $O(n \log n)$ space. For points in *rank-space*, i.e. on an $[O(n)]^2$ grid, the data structures of Das et al. [9] support queries in $O\left(\frac{\log n}{\log\log n} + k\right)$ time, using superlinear $O\left(n\frac{\log n}{\log\log n}\right)$ space. Range skyline queries have been studied in higher dimensions [10].

Brodal and Tsakalidis [11] present *optimal* dynamic 2-d orthogonal *3-sided* range skyline reporting data structures that support queries in $O\left(\frac{\log n}{\log\log n} + k\right)$ time and updates in $O\left(\frac{\log n}{\log\log n}\right)$ time, using linear space, concluding a long series of improvements [12–16] over the initial polylogarithmic worst-case bounds of Overmars and van Leeuwen [17].

External memory. In the I/O model with block size B [2], the optimal pointer machine skyline algorithm [18] has been externalized [19] and I/O-optimal data structures for 2-d orthogonal planar range *skyline counting* queries (that report the skyline size) have been presented [20]. However, despite the abundance of work on *skyline reporting* in various I/O-efficient settings, *no* static [21] or dynamic [22–25] data structures exist for orthogonal range skyline reporting with guaranteed *worst-case I/O-efficiency*, even for planar pointsets. The main focus in literature has been on experimental analyses of average case or adversarially restricted sequences of operations that appear commonly in practice.

1.2. Our contributions

Dynamic orthogonal range skyline reporting. In Section 2 we present the first I/O-efficient dynamic data structures for *2-d orthogonal 3-sided range skyline reporting* that support query and update operations in logarithmic worst-case I/Os, using linear space (Theorem 2.1).

Catenable attrition priority queues. Our dynamic data structures rely heavily on an I/O-optimal generalization of the *attrition priority queues* of Sundar [26] that also supports *real-time concatenation* in constant worst-case time. Specifically, in Subsection 2.1 we present *catenable attrition priority queues* that operate on a set of totally ordered elements and support real-time operations DeleteMin (remove and report the minimum element stored) and CatenateAndAttrite (catenate two attrition priority queues Q_1, Q_1 and *attrite* Q_1, i.e. remove its stored elements smaller than the minimum element in Q_1). We also prove that the operations are actually supported in *subconstant amortized* $O\left(\frac{1}{B}\right)$ I/Os by a detailed amortized analysis (Theorem 2.3).

In more detail, Sundar [26] presents three worst-case-efficient implementations for operations DeleteMin and InsertAndAttrite that use a constant amount of linked lists and additional pointers. In fact, in the third implementation, this amount is a user-defined parameter k_Q. We observe that the third implementation can be modified to further support operation CatenateAndAttrite and we present the detailed implementation in Subsection 2.1, isolating the case where concatenation is actually taking place (see Fig. 3). In the application to 3-sided range skyline reporting, we have that $k_Q = o(B)$, i.e. all additional pointers fit in a block, and hence the attrition priority queue is loaded in main memory without incurring any extra asymptotic cost.

Static orthogonal range skyline reporting. In Section 3 we present *I/O-optimal* static data structures for *2-d orthogonal 3-sided range skyline reporting* in the *indexability model* [5], where a word stores exactly one *indivisible* input coordinate (Theorem 3.2), as well as for *divisible* input coordinates (Theorem 3.5 for points in rank-space and Corollary 3.5.1 for the general case). In Subsection 3.2 we prove a query I/O-complexity lower bound of any static linear-space data structure for *2-d orthogonal range skyline reporting* in the indexability model [4] (Theorem 3.7) and provide a matching data structure that is also dynamic (Corollary 3.7.1). This essentially shows that 2-d orthogonal range skyline reporting is as hard as standard 2-d orthogonal range reporting [3,4,6].

2. Dynamic skyline

In this section we prove the following theorem.

Theorem 2.1. *Given n points in the plane and for any constant $\varepsilon \in [0,1]$, there exist data structures that support orthogonal 3-sided range skyline reporting queries in $O\left(\log_{2B^{\varepsilon}} \frac{n}{B} + \frac{k}{B^{1-\varepsilon}}\right)$ I/Os, where k is the size of the reported skyline, and updates in $O\left(\log_{2B^{\varepsilon}} \frac{n}{B}\right)$ amortized I/Os, using $O\left(\frac{n}{B}\right)$ blocks, and preprocessing I/Os on an x-sorted input pointset.*

First we present some necessary preliminaries. Indeed, the following lemma is used in the proofs of both our main Theorems 2.1 and 3.5, respectively, presented in Subsection 2.2 and in Section 3.

Lemma 2.2. *Given a y-coordinate y and access to a node of a B-tree indexing the x-coordinates of a planar pointset, the k skyline points with y-coordinate larger than y in the node's subtree are reported in $O\left(1+\frac{k}{B}\right)$ I/Os.*

There are two complementing ways to obtain this lemma in the pointer machine, i.e. achieve $O(1+k)$ worst-case reporting time, assuming a binary base tree and ignoring rebalancing issues. On the one hand, we can store the skyline points *explicitly* in a list at every node and simply report them in increasing x-order (and thus also decreasing y-order) until the first skyline point with y-coordinate smaller than y is encountered. This is essentially the basic approach adopted by Overmars and van Leeuwen [17], which nevertheless results in slow $O\left(\log^2 n\right)$ update time ($O(\log n)$ lists need to be updated in $O(\log n)$ time each) and superlinear $O(n \log n)$ space. Albeit, the approach is easily externalizable by using I/O-efficient lists. On the other hand, the structure of Brodal and Tsakalidis [11] avoids the explicit storage of every node's skyline by only storing $O(\log n)$ *representative skyline points* per node and corresponding pointers to the node's subtree. Reporting the skyline is done by recursive calls to Lemma 2.2, essentially following pointers within the tree. Unfortunately, this pointer-chasing algorithm makes it difficult to achieve the desired $\frac{1}{B}$-factor I/O-speed-up. Nevertheless, they achieve optimal $O(\log n)$ update time and linear space, due to the efficient implementation of the secondary data structures that maintain the representative points at every node. Specifically, they use *partially persistent* attrition priority queues (where queries on previous *versions* of the structure are supported, as update operations create new versions) that allow for real-time node updates and avoid redundant copies of skyline points.

We prove Theorem 2.1 in Subsection 2.2 by adopting a hybrid approach to these data structures. On the one hand, we implement the lists at every node with I/O-efficient *catenable* attrition priority queues, which we present first in Subsection 2.1. On the other hand, we make the queues *confluently persistent* (where also updates on previous versions are supported, as well as *merging* two versions into a new version is supported by a concatenation update operation) and resolve the technical difficulties to make the whole structure work I/O-efficiently.

2.1. Catenable attrition priority queues

In this subsection we present I/O-efficient *catenable priority queues with attrition* (I/O-CPQAs) that store a set of elements with values from a total order. For the sake of simplicity, we identify an element with its value. An I/O-CPQA Q with smallest stored element $\min(Q)$ supports the following operations:

- 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) concatenates 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 $Q_1' = \{e \in Q_1 \mid e \leq \min(Q_2)\} \cup Q_2$. Q_1 and Q_2 are discarded.
- InsertAndAttrite(Q, e) inserts element e at the end of I/O-CPQA Q, attrites all elements in Q with value larger than the value of e, and returns the resulting I/O-CPQA $Q' = \{e' \in Q \mid e' \leq e\}$. Q is discarded.

Definitions. An I/O-CPQA Q consists of $k_Q + 2$ deques of records, called the *clean* and *buffer* deques $C(Q), B(Q)$ 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 $[b, 4b]$ elements of strictly

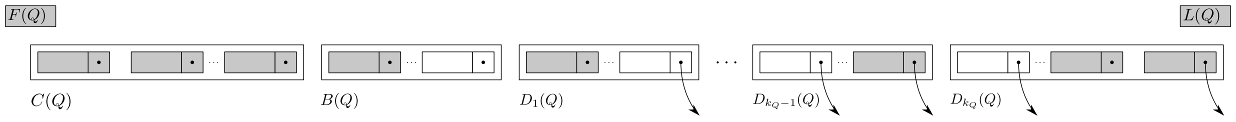


Fig. 2. An I/O-CPQA. Gray records are critical. Only dirty queues contain records with pointers to other I/O-CPQAs.

increasing value and a pointer p to an I/O-CPQA. A record is *simple* when its pointer p is *null*. The clean deque and the buffer deque only contain simple records. The definition of I/O-CPQAs implies an underlying tree structure where pointers are considered as edges and I/O-CPQAs as subtrees. In particular, since the initial I/O-CPQAs Q_1 and Q_2 are discarded after their merge, the merged I/O-CPQA can be seen as a tree, rooted at Q_1 with Q_2 as a child.

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 pointed to by pointer p. We define the queue order of I/O-CPQA Q to be: $C(Q), B(Q), D_1(Q), \ldots, D_{k_Q}(Q)$. It corresponds to an Euler tour over the tree structure. See Fig. 2 for an overview of the structure.

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 record is denoted by first(q) and last(q), respectively. Also, rest(q) denotes all records of the deque q excluding the record first(q). Similarly, front(q) denotes all records of the deque q excluding the record last(q). 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 number of records it contains. The size $|Q|$ of the I/O-CPQA Q is defined to be the number of elements that Q contains. For an I/O-CPQA Q we denote by first(Q) and last(Q), respectively the first and last record out of the records $\cup_q \mathrm{first}(q)$ and $\cup_q \mathrm{last}(q)$ for all deques $q = \{C(Q), B(Q), D_1(Q), \ldots, D_{k_Q}(Q)\}$ 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 points to I/O-CPQA Q', we have

$$\max(l) < \min(Q').$$

I.2) In all deques of Q where record $r_1 = (l_1, p_1)$ precedes record $r_2 = (l_2, p_2)$, we have

$$\max(l_1) < \min(l_2).$$

I.3) For deques $C(Q), B(Q), D_1(Q)$, we have

$$\max(\mathrm{last}(C(Q))) < \min(\mathrm{first}(B(Q))) < \min(\mathrm{first}(D_1(Q))).$$

I.4) Element $\min(\mathrm{first}(D_1(Q)))$ is the smallest element in dirty deques $D_1(Q), \ldots, D_{k_Q}(Q)$.
I.5) All records in the deques $C(Q)$ and $B(Q)$ are simple.
I.6) $|C(Q)| \geq \sum_{i=1}^{k_Q} |D_i(Q)| + k_Q - 1$.
I.7) $|\mathrm{first}(C(Q))| < b$ holds, iff $|Q| < b$.
I.8) $|\mathrm{last}(D_{k_Q}(Q))| < b$ and $|r| \in [b, 5b]$ hold, iff record $\mathrm{last}(D_{k_Q}(Q))$ is simple.

From Invariants I.2, I.3 and I.4, we have that $\min(Q) = \min(\mathrm{first}(C(Q)))$. We say that an operation *improves* or *aggravates* the inequality of Invariant I.6 by a parameter c for I/O-CPQA Q, when the operation increases or decreases $\Delta(Q) = |C(Q)| - \sum_{i=1}^{k_Q} |D_i(Q)| - k_Q + 1$ by c, respectively.

Operations. We present the algorithms implementing the operations supported by the I/O-CPQA Q. Most of the operations call the auxiliary operation Bias(Q), which we present last. Bias improves the inequality of Invariant I.6 for Q by at least 1.

FindMin(Q) returns the value $\min(\mathrm{first}(C(Q)))$.

DeleteMin(Q) removes element $e = \min(\mathrm{first}(C(Q)))$ from record $(l, p) = \mathrm{first}(C(Q))$. After the removal, if $|l| < b$ and $|Q| \geq b$, we do the following. If $b \leq |\mathrm{first}(\mathrm{rest}(C(Q)))| \leq 2b$, then we merge first($C(Q)$) with first(rest($C(Q)$)) into one record which is the new first record. Else if $2b < |\mathrm{first}(\mathrm{rest}(C(Q)))| \leq 3b$ then we take b elements out of first(rest($C(Q)$)) and put them into first($C(Q)$). Else we have that $3b < |\mathrm{first}(\mathrm{rest}(C(Q)))|$, and as a result we take $2b$ elements out of first(rest($C(Q)$)) and put them into first($C(Q)$). If inequality of I.6 for Q is aggravated by 1 we call Bias(Q) once. We return element e.

CatenateAndAttrite(Q_1, Q_2) concatenates Q_2 to the end of Q_1 and removes the elements from Q_1 with value larger than $\min(Q_2)$. To do so, it creates a new I/O-CPQA Q_1' by modifying Q_1 and Q_2, and by calling Bias(Q_1') and Bias(Q_2).

In particular, if $|Q_1| < b$, then Q_1 is only one record $(l_1, \cdot)$, and so we prepend it into the first record $(l_2, \cdot) = \mathrm{first}(Q_2)$ of Q_2. In particular, let l_1' be the non-attrited elements of l_1. If $|l_1'| + |l_2| \leq 4b$, then we prepend l_1' into l_2. Else, we take $2b - |l_1'|$ elements out of l_2, and make them along with l_1' the new first record of Q_2.
Else if $|Q_2| < b$, then Q_2 only consists of one record. We have two cases, depending on how much of Q_1 is attrited by Q_2. Let r_1 be the second last record for Q_1 and let $r_2 = \mathrm{last}(Q_1)$ be the last record. If e attrites all of r_1, then we just pick the

appropriate case among (1–4) below. Else if e attrites partially r_1, but not all of it, then we delete r_2 and merge r_1 and Q_2 into the new last record of Q_1, which cannot be larger than $5b$. Otherwise if e attrites partially r_2, but not all of it, then we simply append the single record of Q_2 into r_2. It will be the new last record of Q_1 and it cannot be larger than $5b$.

We have now dealt with the case where $|Q_1| \geq b$. So in the following, we assume that Q_1 is large. For the remaining cases, let $e = \min(Q_2)$.

1) If $e \leq \min(\text{first}(C(Q_1)))$, we discard I/O-CPQA Q_1 and set $Q_1' = Q_2$.
2) Else if $e \leq \max(\text{last}(C(Q_1)))$, we remove simple record $(l, \cdot) = \text{first}(C(Q_2))$ from $C(Q_2)$, set $C(Q_1') = \emptyset$, $B(Q_1') = C(Q_1)$ and $D_1(Q_1') = (l, p)$, where p points to Q_2, if it exists. This aggravates the inequality of I.6 for Q_2 by at most 1, and gives $\Delta(Q_1') = -1$. Thus, we call Bias(Q_2) once and Bias(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 remove the simple record $(l, \cdot) = \text{first}(C(Q_2))$ from $C(Q_2)$, set $D_1(Q_1') = (l, p)$, and make p point to Q_2, if it exists. If $e \leq \min(\text{first}(B(Q_1)))$, we set $B(Q_1') = \emptyset$. This aggravates the inequality of I.6 for Q_2 by at most 1, and aggravates the inequality of I.6 for Q_1 by at most 1. Thus, we call Bias(Q_2) once and Bias(Q_1') once.
4) Else, let $(l_1, \cdot) = \text{last}(D_{k_{Q_1}})$. We remove $(l_2, \cdot) = \text{first}(C(Q_2))$ from $C(Q_2)$. If $|l_1| < b$, then we remove the record $(l_1, \cdot)$ from $D_{k_{Q_1}}$. Let l_1' be the non-attrited elements under attrition by $e = \min(l_2)$. If $|l_1'| + |l_2| \leq 4b$, then we prepend l_1' into l_2 of record $r_2 = (l_2, p_2)$, where p_2 points to the rest of Q_2. Otherwise, we make a new simple record r_1 with l_1' and $2b$ elements taken out of $r_2 = (l_2, p_2)$. Finally, we put the resulting one or two records r_1 and r_2 into a new deque $D_{k_{Q_1}+1}(Q_1)$. This aggravates the inequality of I.6 for Q_2 by at most 1, and the inequality of I.6 for Q_1 by at most 2. Thus, we call Bias(Q_2) once and Bias(Q_1') twice.

InsertAndAttrite(Q, e) inserts an element e into I/O-CPQA Q and attrites the elements in Q with value larger than e. This is a special case of operation CatenateAndAttrite(Q_1, Q_2), where $Q_1 = Q$ and Q_2 is an I/O-CPQA that only contains one record with the single element e.

Bias(Q) improves the inequality of Invariant I.6 for Q by at least 1.

1) $|B(Q)| > 0$: We remove the first record $\text{first}(B(Q)) = (l_1, \cdot)$ from $B(Q)$ and let $(l_2, p_2) = \text{first}(D_1(Q))$. Let l_1' be the non-attrited elements of l_1 by element $e = \min(l_2)$.

   1) $0 \leq |l_1'| < b$: If $|l_2| \leq 2b$, then we just prepend l_1' onto l_2. Else, we take b elements out of l_2 and append them to l_1'.
   2) $b \leq |l_1'| < 2b$: If $|l_2| \leq 2b$, and if furthermore $|l_1'| + |l_2| \leq 3b$ holds, then we merge l_1' and l_2. Else $|l_1'| + |l_2| > 3b$ holds, so we take $2b$ elements out of l_1' and l_2 and put them into l_1', leaving the rest in l_2.
      Else $|l_2| > 2b$ holds, so we take b elements out of l_2 and put them into l_1'.

   If we did not prepend l_1' onto l_2, we insert l_1' along with any elements taken out of l_2 at the end of $C(Q)$ instead. If $|l_1'| < |l_1|$, we set $B(Q) = \emptyset$. Else, we did prepend l_1' onto l_2, and then we just recursively call Bias. Since $|B(Q)| = 0$ we will not end up in this case again. In all cases the inequality of I.6 for Q is improved by 1.
2) $|B(Q)| = 0$: we have two cases depending on the number of dirty queues, namely cases $k_Q > 1$ and $k_Q = 1$.

   1) $k_Q > 1$: 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.6 for Q by 1.
      If $\min(\text{last}(D_{k_Q-1}(Q))) < e \leq \max(\text{last}(D_{k_Q-1}(Q)))$ holds, on the other hand, 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.

      1) $0 \leq |l_1'| < b$: If $|l_2| \leq 2b$, then we just prepend l_1' onto l_2. Otherwise, we take b elements out of l_2 and append them to l_1'.
      2) If $b \leq |l_1'| < 2b$: If $|l_2| \leq 2b$ and $|l_1'| + |l_2| \leq 3b$, then we merge l_1' and l_2. Else, $|l_1'| + |l_2| > 3b$ holds, so we take $2b$ elements out of l_1' and l_2 and put them into l_1', leaving the rest in l_2.
         Else $|l_2| > 2b$, so we take b elements out of l_2 and put them into l_1'.

      If r_1 still exists, we insert it in the front of $D_{k_Q}(Q)$. Finally, we concatenate $D_{k_Q-1}(Q)$ and $D_{k_Q}(Q)$ into one deque. This improves the inequality of I.6 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.6 for Q by 1.
   2) $k_Q = 1$: In this case Q contains only deques $C(Q)$ and $D_1(Q)$. We remove the record $r = (l, p) = \text{first}(D_1(Q))$ and insert l into a new record at the end of $C(Q)$. This improves the inequality of I.6 for Q by at least 1. If r is not simple, let the pointer p of r point to I/O-CPQA Q'. We restore I.5 for Q by merging I/O-CPQAs Q and Q' into one I/O-CPQA. See Fig. 3. In particular, let $e = \min(\text{first}(D_1(Q)))$.

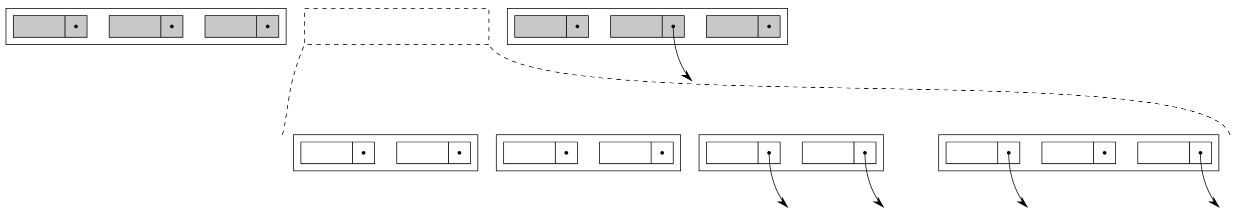


Fig. 3. Merging two I/O-CPQAs occurs only when the left I/O-CPQA Q_1 has $B(Q_1) = \emptyset$ and $k_{Q_1} = 1$.

If $e \le \min(Q')$, we discard Q'. The inequality of I.6 for Q remains unaffected.
Else, if $\min(\text{first}(C(Q'))) < e \le \max(\text{last}(C(Q'))$, we set $B(Q) = C(Q')$ and discard the rest of Q'. The inequality of I.6 for Q remains unaffected.
Else if $\max(\text{last}(C(Q')) < e \le \min(\text{first}(D_1(Q')))$, 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.6 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.6 for Q by $\Delta(Q') \ge 0$, since Q' satisfied Invariant I.6 before the operation.
If $r = \text{first}(Q)$ and $|l| \le 2b$, then we run BIAS recursively to ensure that we get at least two records in $C(Q)$. Let $r' = (l', p') = \text{first}(Q)$ be the first record of Q after the recursive call. If $|l| + |l'| > 3b$, then we take the $2b$ first elements out and make them the new first record of $C(Q)$. Else we merge l into l', so that r is removed and r' is now $\text{first}(Q)$.

Theorem 2.3. *I/O-CPQAs support operations* FINDMIN, DELETEMIN, CATENATEANDATTRITE *and* INSERTANDATTRITE *in $O(1)$ I/Os, using $O(n-m)$ words after n calls to* CATENATEANDATTRITE *and* INSERTANDATTRITE *and m calls to* DELETEMIN.
For any $b \in [1, B]$, a set of ℓ CPQAs support the operations in $O\left(\frac{1}{b}\right)$ amortized I/Os, using $O\left(\frac{n-m}{b}\right)$ blocks, provided a constant number of blocks are loaded in main memory of size $M \ge \ell b$.

Proof. The correctness follows by closely noticing that we maintain Invariants I.1–I.8, and from those we have that DELETEMIN(Q) and FINDMIN(Q) always returns the minimum element of Q.
The constant worst-case I/O-bound is trivial as every operation only accesses $O(1)$ records. Although BIAS is recursive, we 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. Similarly, if $|B(Q)| = 0$ and $k_Q > 1$ there might also be a recursive call to BIAS. However, before the call at least b elements have been taken out of Q, and thus the following recursive call to BIAS will ensure at least b more are taken out and the recursion stops. So the recursion, will have depth at most 3.
To argue about the $O\left(\frac{1}{b}\right)$ amortized I/O-bounds, 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

$$\Phi(Q) = \Phi_F(|\text{first}(Q)|) + |\text{middle}(Q)| + \Phi_L(|\text{last}(Q)|),$$

where

$$\Phi_F(x) = \begin{cases} 3 - \frac{x}{b}, & b \le x < 2b \\ 1, & 2b \le x < 3b \\ \frac{2x}{b} - 5, & 3b \le x \le 4b \end{cases} \quad \text{and} \quad \Phi_L(x) = \begin{cases} 0, & 0 \le x < 4b \\ \frac{3x}{b} - 12, & 4b \le x \le 5b \end{cases}$$

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

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

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.
Operation DELETEMIN. I/Os occur, only if $|\text{first}(C(Q))| = b - 1$. In this case $r = \text{first}(Q)$ has a potential of $\Phi(|r|) \ge 2$, and since we increase the number of elements in r by b to $2b$ elements, the potential of r will then only be $\Phi(|r|) = 1$. Thus, the total potential decreases by at least 1, which also pays for any I/Os including those incurred if BIAS(Q) is invoked.
Operation CATENATEANDATTRITE. When $|Q_1| < b$, if we simply prepend l'_1 into l_2, then the potential $\Phi_S(|l_1|)$ pays for the increase in potential of $\Phi_F(|\text{first}(C(Q_2))|)$. Else, we take $2b - |l'_1|$ elements out of l_2, size $2b$. Thus, $\Phi_F(2b) = 1$ and the

potential drops by 1, which is enough to pay for the I/Os used to flush the old first record of $C(Q_2)$ to disk. When $|Q_2| < b$, if e attrites all of r_1, then we release at least 1 in potential, so all costs in any of the cases (1–4) are paid for. If e attrites partially r_1, then the new record cannot contain more than $5b$ elements, and thus any increase in potential is paid for by the potential of Q_2. Thus, the I/O cost is covered by the decrease of 1 in potential, caused by r_1. If e attrites partially r_2, any increase in potential is paid for by the potential of Q_2. In all the other cases (1–4) both Q_1 and Q_2 are large, hence when we concatenate them we decrease the potential by at least 1, as the number of large I/O-CPQA's decrease by one, which is enough to pay for any I/O's and I/O's charged by the Bias operations.

Operation InsertAndAttrite. Since creating a new I/O-CPQA with only one element and calling CatenateAndAttrite only costs $O\left(\frac{1}{b}\right)$ I/Os amortized, the operation InsertAndAttrite also costs $O\left(\frac{1}{b}\right)$ I/Os amortized.

Operation Bias. Since all I/Os incurred by Bias(Q) are already paid for by the operation that called Bias(Q), we only need to argue that the potential of Q does not increase due to the changes that Bias(Q) makes to Q. When $|B(Q)| > 0$, if $l_1 = \text{first}(Q)$, then after calling Bias we ensure that $2b \le |\text{first}(Q)| \le 3b$, so that the potential of Q does not increase. When $|B(Q)| = 0$ and $k_Q > 1$, if not all of l_1 is attrited then we ensure that its record r_1 has size between $2b$ and $3b$. Thus, if $r_1 = \text{first}(Q)$ holds, we will not have increased the potential of Q. In the cases where all or none of l_1 is attrited, the potential of Q can only be decreased by at least 0. Otherwise, when $k_Q = 1$, since first(Q) is either untouched or left with $2b$ to $3b$ elements, in which case its potential is 1, and since all other changes decrease the potential by at least 0, we have that Bias does not increase the potential of Q. □

Concatenating many I/O-CPQAs. We present the algorithm that executes a possibly unlimited sequence of consecutive CatenateAndAttrite operations on I/O-CPQAs $Q_1, Q_2, \dots, Q_\ell$ to obtain a single I/O-CPQA in $O(1)$ worst case I/Os, given that operation DeleteMin is not included in the sequence. To obtain the result we ensure that operations Bias is never called by imposing two extra assumptions on the I/O-CPQAs. Define the *state* of I/O-CPQA Q to be $\Delta(Q) = |C(Q)| - \sum_{i=1}^{k_Q} |D_i(Q)| - k_Q + 1$, and the *critical records* of Q to be 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.

Lemma 2.4. *The I/O-CPQAs Q_i for $i \in [1, \ell]$ can be concatenated into a single I/O-CPQA by calling only* CatenateAndAttrite *operations without any access to external memory, provided that:*

1. *Every input I/O-CPQA Q_i is in state at least $+2$, unless it contains only one record, in which case its state is at least $+1$.*
2. *The critical records of all input I/O-CPQAs Q_i already reside in main memory.*

Proof. To avoid any I/Os during the sequence of CatenateAndAttrites, we ensure that Bias is not called, and that no more than the critical records need to be already loaded into memory.

To avoid calling Bias we prove by induction the invariant that the temporary I/O-CPQAs $Q_{i\dots\ell}, i \in [1, \ell]$ constructed during the sequence are in state at least $+1$. Let the invariant hold for $Q_{i+1\dots\ell}$ and let $Q_{i\dots\ell}$ be constructed by calling the operation CatenateAndAttrite$(Q_i, Q_{i+1\dots\ell})$. If Q_i contains at most two records, which both reside in deque $C(Q_i)$, we only need to access record first$(C(Q_{i+1\dots\ell}))$ and the at most two records of Q_i. The invariant holds for $Q_{i\dots\ell}$, since it holds inductively for $Q_{i+1\dots\ell}$ and the new records were added at $C(Q_{i+1\dots\ell})$. As a result, the inequality of I.6 for $Q_{i+1\dots\ell}$ can only be improved. If $Q_{i+1\dots\ell}$ consists of only one record, then either one of the following cases applies or we follow the steps described in operation CatenateAndAttrite. In the second case, there is no aggravation of inequality 6 and only critical records are used.

In the following, we can safely assume that Q_i has at least three records and its state is at least $+2$. We parse the cases of the CatenateAndAttrite algorithm assuming that $e = \min(Q_{i+1\dots\ell})$.

Case 1 The invariant holds trivially since Q_i is discarded and no change happens to $Q_{i\dots\ell} = Q_{i+1\dots\ell}$. Bias is not called.

Cases 2, 3 The algorithm checks whether the first two records of $C(Q_i)$ are attrited by e. If this is the case, we continue as denoted at the start of this proof. Otherwise, case 2 of CatenateAndAttrite is applied as is. $Q_{i+1\dots\ell}$ is in state 0 after the concatenation and $Q_{i\dots\ell}$ is in state $+1$. Thus the invariant holds, and Bias is not called. Note that all changes take place at the critical records of Q_i and $Q_{i+1\dots\ell}$.

Case 4 The algorithm works exactly as in case 4 of CatenateAndAttrite, with the following exception. At the end, $Q_{i\dots\ell}$ will be in state 0, since we added the deque $D_{k_{Q_{i+1\dots\ell}}+1}(Q_{i+1\dots\ell})$ with a new record and the inequality of I.6 is aggravated by 2. To restore the invariant we apply case 2(1) of Bias. This step requires access to records last$(D_{k_{Q_{i\dots\ell}}-1}(Q_{i\dots\ell}))$ and first$(D_{k_{Q_{i\dots\ell}}}(Q_{i\dots\ell}))$. These records are both critical, since the former corresponds to last$(D_{k_{Q_{i+1\dots\ell}}}(Q_{i+1\dots\ell}))$ and the latter to first$(C(Q_{i+1\dots\ell}))$. In addition, Bias$(Q_{i+1\dots\ell})$ needs not be called, since by the invariant, $Q_{i+1\dots\ell}$ was in state $+1$ before the removal of first$(C(Q_{i+1\dots\ell}))$. In this way, we improve the inequality for $Q_{i\dots\ell}$ by 1 and the invariant holds.

This concludes the proof. □

2.2. 3-sided range skyline reporting

In this subsection we prove Theorem 2.1.

Data structure. The data structure consists of B-tree with leaf capacity B and internal fanout $2B^{\varepsilon}$ that indexes the x-coordinates of the points, for some constant $\varepsilon \in [0,1]$. At every node, we store the points in its subtree in a confluently persistent I/O-CPQA with buffer size $b := B^{1-\varepsilon}$. Although there is no general technique to obtain efficient confluently persistent data structures, fortunately there does exist an efficient confluently persistent variant of the basic building block of I/O-CPQAs, namely real-time confluently persistent *catenable double-ended queues* [27].

Preprocessing algorithm. After constructing the base tree, we augment it with secondary confluently persistent I/O-CPQAs in a bottom-up manner, as following. At every leaf, we construct a confluently persistent I/O-CPQA by calling InsertAndAttrite on its points (x, y) considered from left to right as elements inserted at "time" x with value $-y$. This reflects the "dominance relation" between two points into an "attrition operation" between their corresponding elements, preventing non-skyline points to be returned by DeleteMin during query operations. Also, we call Bias enough times for the state of the constructed I/O-CPQA to satisfy Lemma 2.4. In a second pass over the leaves, we gather the critical records of every $O\left(2B^{\varepsilon}\right)$ consecutive I/O-CPQAs into a *representative block* and continue recursively up the tree, with the difference that for internal nodes we call (from left to right) CatenateAndAttrite on the I/O-CPQAs of its children nodes. At the end of the construction, we obtain *implicitly* the representation of a single confluently persistent I/O-CPQA, accessible by the version at the root node. Every internal node stores explicitly only a representative block for some version of the I/O-CPQA and all versions form a connected directed acyclic *version graph* over the base tree with edges directed from child to parent node.

Since every leaf contains $O(B)$ elements, the base tree has $O\left(\frac{n}{B}\right)$ leaves and thus also $O\left(\frac{n}{B}\right)$ internal nodes. Every internal node has $\Theta\left(B^{\varepsilon}\right)$ children, each associated with an I/O-CPQA with $O(1)$ critical records of size $O\left(B^{1-\varepsilon}\right)$. Thus the representative blocks stored in every internal node take $O(1)$ blocks and in total $O\left(\frac{n}{B}\right)$ blocks and also I/Os to construct, since Lemma 2.4 ensures Bias incurs no extra I/Os. A leaf I/O-CPQA is constructed in $O(1)$ I/Os and thus we need in total $O\left(\frac{n}{B}\right)$ preprocessing I/Os.

Update algorithm. To insert or delete a point, we first search for the leaf with the point's x-predecessor. At every node on the root-to-leaf search path, we remove its I/O-CPQA by discarding its representative blocks. Specifically, the operations that created the I/O-CPQA are execute in reverse [11]. We rebalance the tree and reconstruct the I/O-CPQAs along the search path in a bottom-up manner, as described above.

The total update I/Os are $O\left(\log_{2B^{\varepsilon}} \frac{n}{B}\right)$, since we spend $O(1)$ I/Os to reconstruct every I/O-CPQA (Lemma 2.4) and to rebalance the path's nodes.

Query algorithm. To report the skyline points of P that reside within a given top-open query range $[x_l, x_r] \times [y_b, \infty)$, we first traverse top-down the two search paths π_l and π_r from the root of the base tree to the leaves ℓ_l and ℓ_r that contain the x_l-successor and x_r-predecessor points. Let node u be on the path $\pi_l \cap \pi_r$, and let $c(u)$ be the children nodes of u whose subtrees are fully contained within $[x_l, x_r]$. 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 2.4. We consider the temporary I/O-CPQAs of nodes u and the I/O-CPQAs of the leaves ℓ_l and ℓ_r from left to right, and we CatenateAndAttrite them into a single auxiliary I/O-CPQA. To report the skyline points within the query range, we call DeleteMin at the auxiliary I/O-CPQA, until a point with y-coordinate smaller than y_b is encountered. This implements Lemma 2.2, since only the leaves ℓ_l and ℓ_r may contain $o(B)$ points with x-coordinate outside $[x_l, x_r]$.

There are $O\left(\log_{2B^{\varepsilon}} \frac{n}{B}\right)$ nodes on $\pi_l \cap \pi_r$ and we spend $O(1)$ I/Os to access the representative block of each node and to construct the auxiliary I/O-CPQAs. Reporting the k output points costs $O\left(\frac{k}{B^{1-\varepsilon}} + 1\right)$ worst-case I/Os by Lemma 2.2 and since the I/O-CPQAs at the leaves ℓ_l and ℓ_r take only $O(1)$ extra I/Os to process. Thus the query takes $O\left(\log_{2B^{\varepsilon}} \frac{n}{B} + \frac{k}{B^{1-\varepsilon}}\right)$ I/Os in total.

3. Static skyline

In this section we present *I/O-optimal* static data structures for *2-d orthogonal 3- and 4-sided range skyline reporting* for both divisible input coordinates, as well as for indivisible coordinates (*indexability model* [4]), where we also prove a tight lower bound.

3.1. 3-sided range skyline reporting

Indexability model. We reduce the problem to a special case of *orthogonal segment intersection reporting* on n horizontal segments by replacing every input point (x, y) with a horizontal line segment $\left[x, x'\right) \times y$, where $x' > x$ is the x-coordinate of the leftmost input point that dominates (x, y), if it exists, and $+\infty$ otherwise. Given a 3-sided query rectangle $Q :=$

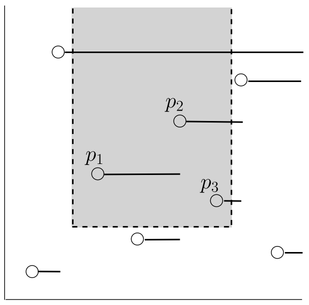


Fig. 4. Reduction to nesting and monotonic orthogonal line segment intersection.

$[x_l, x_r] \times [y_b, +\infty)$, we report the horizontal segments that intersect the vertical query segment $Q' := x_r \times \left[y_b, y'\right]$, where y' is the largest y-coordinate of the input points in Q. An example is shown in Fig. 4.

Correctness follows from the observation that, on the one hand, any skyline point $(x, y) \in Q$, when know that $y \le y'$ and also that its segment's right endpoint $\left(x', y\right) \notin Q$, and thus $x' > x_r$. Hence the corresponding segment intersects Q'. On the other hand, for any input point (x, y) that corresponds to a segment intersecting Q', we know that $x \le x_r < x'$ and $y_b \le y \le y'$. Thus, to show that $(x, y) \in Q$, it suffices to prove that $x_l \ge x$. Let $\left(x'', y'\right) \in Q$ be the input point with the largest y-coordinate among the points in Q. Indeed, if $y \le y'$ holds, then also $x \ge x''$ holds (otherwise $\left(x'', y'\right)$ would dominate (x, y), which would in turn violate the definition of $\left(x', y\right)$, since we have that $x'' \le x_r < x'$), and we are done, since $x'' \ge x_l$. Finally, no input point in Q dominates (x, y), because $x_r < x'$ implies that the leftmost point dominating (x, y) lies outside Q.

Lemma 3.1. *The generated set of horizontal segments is:*

- **Nesting:** *Any two segments have either disjoint x-intervals, or one x-interval contains the other.*
- **Monotonic:** *Ordering the segments that intersect any vertical line by increasing y-coordinates, also orders them in increasing length of their x-intervals.*

Proof. *Nesting:* Let p_1 and p_2 be the input points corresponding to the two segments, and without loss of generality let $x_1 < x_2$. If $y_1 < y_2$, p_1 is dominated by p_2 and thus its x-interval ends before x_2, i.e., the two x-intervals disjoint. Otherwise $y_1 \ge y_2$, and let p_1' be the right endpoint of the segment for p_1. If $x_1' < x_2$, then the two x-intervals are again disjoint. Otherwise $x_1' \ge x_2$, and thus p_1' also dominates p_2, i.e., the x-interval of p_1 contains that of p_2.

Monotonic: The left endpoints of the segments intersecting a given vertical line x, constitute the skyline of the input points that are contained in the 1-side query range $(-\infty, x] \times (-\infty, \infty)$. Therefore, enumerating these segments in ascending y-order, also enumerates the x-coordinates of their left endpoints in decreasing order. The nesting property implies that the corresponding x-intervals appear in non-decreasing length. □

Theorem 3.2. *Given n points in the plane, there exist data structures that support orthogonal 3-sided range skyline reporting queries in $O\left(\log_B n + \frac{k}{B}\right)$ I/Os, where k is the size of the reported skyline, using $O\left(\frac{n}{B}\right)$ blocks, and preprocessing I/Os on an x-sorted input pointset.*

Proof. To report the skyline of the points in a given 3-sided query range Q, first we find the largest y-coordinate y' in Q by a *range-max query* $[x_l, x_r]$ on a B-tree storing the input x-coordinates, in $O\left(\log_B n\right)$ I/Os. To report the segments intersecting the vertical query segment Q', we execute a *1-d range query* $\left[y_b, y'\right]$ on the x_r-th version of a *partially persistent B-tree* [28] that stores the horizontal segments, in $O\left(\log_B n\right)$ I/Os, and report the k intersecting horizontal segments by increasing y-order in $O\left(1 + \frac{k}{B}\right)$ extra I/Os.

Two obstacles prevent us from achieving $O\left(\frac{n}{B}\right)$ blocks and preprocessing I/Os on an x-sorted input pointset: generating the set of segments and constructing the partially persistent B-tree. To generate the set, we sweep a vertical line x from $-\infty$ to ∞ maintaining an auxiliary I/O-efficient stack where the encountered points are pushed in decreasing y-order. If the encountered point at "time" x has larger y-coordinate than that of point $\left(x', y'\right)$ at the top of the stack, then it is its leftmost dominating point, so we pop the top point and generate the horizontal segment $\left[x', x\right] \times y'$. The nesting and monotonic properties of the generated segment-set ensure that only the bottommost segments intersecting the sweep line are being updated, and thus also only the leftmost leaves of the partially persistent B-tree. To achieve linear I/O-complexity, it suffices to keep these leaves loaded in main memory and adopt a bottom-up rebalancing scheme for partially persistent B-trees [28] by standard modifications. □

Divisible input. We present I/O-optimal data structures for input points in $[U]^2$ (for integer $U \geq n$) and in rank-space (where $U = O(n)$) with *divisible* coordinates. We show that packing multiple points into a single word allows for query operations with real-time searching algorithms, by reducing the case of *few points* to real-time *vertical ray dragging queries* on $(B \log U)^{O(1)}$ planar points (report the first point hit by moving a given vertical query ray $x \times [y, +\infty)$ to the left).

Lemma 3.3. *Given $n \leq (B \log U)^{O(1)}$ divisible points in $[U]^2$, there exist data structures that support real-time vertical ray dragging queries, using $O\left(1 + \frac{n}{B}\right)$ blocks.*

Proof. We store the x-coordinates of the points in a B-tree with leaf capacity and internal fanout B. For an internal node v with children nodes $v_1, \ldots, v_B$, we denote by $Y_{max}(v)$, the largest point in its subtree. We define for an internal node v, $Y^*_{max}(v) = \{Y_{max}(v_i) \mid 1 \leq i \leq B\}$, and for a leaf v $Y^*_{max}(v)$ to be the set of all points stored in it. To answer a ray-dragging query with ray $x \times [x, U][\beta, U]$, we descend down the path from the root to the leaf containing the predecessor of x, until we reach the *lowest* node v on the path (if it exists), where $Y^*_{max}(v)$ contains a point p hit by the ray (when it moves to the left). Let $p = Y_{max}(v_i)$ for some child v_i of v. Point p is indeed the answer to the query, unless there exists another point in the subtree of v_i that lies to the left of p and has y-coordinate higher than y. To find out, we reset v to v_i and process $Y^*_{max}(v)$ recursively, essentially descending a v-to-leaf path.

Let h denote the height of the B-tree. The query algorithm takes in total $O(h) = O(\log_B n)$ I/Os, since for every accessed node v, we can load $Y^*_{max}(v)$ into main memory in $O(1)$ I/Os. Note that if $B \geq \sqrt{\log U}$, then $O(\log_B n) = O(\log_B B \log U) = O(1)$ I/Os, thus it suffices to assume henceforth that $B < \sqrt{\log U}$. We set parameter $b := B \log_2 U < \log^{3/2} U$ and consider the case first of a few points, namely where $n \leq \sqrt{b} < \log^{3/4} U$. We map the input points (x, y) to an $n \times n$ grid, replacing their coordinates with their respective *ranks*, i.e., the number of points whose x-coordinates are no greater than x (respectively, for y). Every mapped point is associated with an id, namely a unique integer in $[1, n]$. We store the id's in an array of $O\left(\frac{n}{B}\right)$ blocks in order to convert the id's to the original coordinates in $O(1)$ I/Os. Finally, we store the x- and y-coordinates of the points in two *fusion trees* [29,30] in order to support predecessor searches in $O\left(\log_b n\right) = O(1)$ I/Os, using $O\left(\frac{n}{B}\right)$ blocks. A ray dragging query with ray $x \times [y, +\infty)$ can now be answered in $O(1)$ I/Os, by a ray dragging query with the mapped ray $x' \times \left[y', +\infty\right)$ on the grid, where x', y' are the x- and y-ranks of x and y, respectively, which are retrieved by querying the fusion trees first. The benefit of this approach is that we can store the mapped pointset in a single word, since we need only $3 \log_2 n$ bits to represent the ranks and id of every mapped point, and thus use at most $3n \log n = O\left(\log^{3/4} U \cdot \log \log U\right) = o(\log U)$ bits in total.

For the remaining case, where $n = b^{O(1)}$, we modify the described B-tree by setting the internal fanout to b and using the described grid-structure to store $Y^*_{max}(v)$ for every internal node v, using $O\left(\frac{b}{B}\right)$ blocks per internal node. Hence, a ray dragging query takes $O(1)$ I/Os, since the height of the tree becomes $h = O(\log_B n) = O(1)$, and the $O\left(\frac{n}{bB}\right)$ internal nodes use $O\left(\frac{b}{B} \cdot \frac{n}{bB}\right) = O\left(\frac{n}{B}\right)$ blocks in total. □

Lemma 3.4. *Given $n \leq (B \log U)^{O(1)}$ divisible points in $[U]^2$, there exist data structures that support orthogonal 3-sided range skyline reporting queries in $O\left(1 + \frac{k}{B}\right)$ I/Os, where k is the size of the reported skyline, using $O\left(1 + \frac{n}{B}\right)$ blocks.*

Proof. Given a 3-sided range $[x_l, x_r] \times [y_b, U]$, let (x, y) be the first point hit by the ray $x_r \times [y_b, U]$ when it moves to the left. Unless the range contains no points, this is the lowest skyline point in the range. We store the points in a structure of Theorem 3.2. Recall that it supports orthogonal segment intersection queries with the vertical query line segment $x_r \times \left[y_b, y'\right]$, given that we know the largest y-coordinate y' of the points in the range. To support queries in $O\left(1 + \frac{k}{B}\right)$ I/Os, we prove the next two observations:

(i) The output contains exactly the horizontal line segments intersecting another vertical segment $x \times \left[y, y'\right]$.
(ii) The output can be reported without knowing y', given access to the leaf containing y in the version of the partially persistent B-tree at position x.

(i) The horizontal line segment corresponding to (x, y) is the lowest among the output segments, therefore all output segments intersect the query segment $x_r \times \left[y_b, y'\right]$, if and only if they intersect segment $x_r \times \left[y, y'\right]$, and thus also segment $x_r \times \left[y, y'\right]$, because the set of segments is nesting and monotonic. (ii) The segments intersecting the ray $x \times [y, U]$ can be reported in increasing y-coordinate, by following sibling pointers from the accessed leaf. The nesting and monotonicity properties ensure that the segments are also reported in decreasing x-coordinate of their left endpoints, which allows us to stop reporting as soon as we exceed the query coordinate x_l. The query takes $O\left(1 + \frac{k}{B}\right)$ I/Os, since we report $\Omega(1 + B)$ segments per accessed leaf. □

Theorem 3.5. *Given n divisible points in rank-space, there exist data structures that support orthogonal 3-sided range skyline reporting queries in $O\left(1 + \frac{k}{B}\right)$ I/Os, where k is the size of the reported skyline, using $O\left(\frac{n}{B}\right)$ blocks.*

Proof. Consider the $[U]^2$ grid and let both $\lambda := B\log^2 U$ and $\frac{U}{\lambda}$ be integers. We divide the x-dimension into $\frac{U}{\lambda}$ x-intervals (*slabs*), assign each input point to the unique slab that contains it (some slabs may be empty) and build a complete binary search tree on the slabs. At every internal node u, we store $\mathrm{high}(u)$, i.e., the B largest skyline points among the points in its subtree. Moreover, if the B-th stored skyline point exists and is contained in slab z, we also store at u, the set $\mathrm{MAX}(u)$, i.e., the (x-sorted) skyline points among the points in $\mathrm{high}(v)$, for all right sibling nodes v (if they exist) of the nodes on the u-to-z path, essentially implementing Lemma 2.2. At every slab z, we store the points it contains in a few-points structure of Lemma 3.4, and moreover for every proper ancestor node u of z, we also store the sets $\mathrm{rmax}(z,u)$ and $\mathrm{lmax}(z,u)$, defined as MAX for the right and left sibling nodes of the u-to-z path, respectively.

Let $h = O(\log U)$ be the height of the tree. For each of the $O\left(\frac{U}{\lambda}\right)$ internal nodes u, $\mathrm{MAX}(u)$ uses $O(h)$ blocks, for a total of $O\left(h\cdot\frac{U}{\lambda}\right) = O\left(\frac{U}{B}\right) = O\left(\frac{n}{B}\right)$ blocks, since we work on rank-space, where $U = O(n)$. The few-point structures at $O\left(\frac{U}{\lambda}\right)$ slabs use $O\left(\frac{U}{\lambda}+\frac{n}{B}\right) = O\left(\frac{n}{B}\right)$ blocks in total. Finally, all sets rmax and lmax use $O\left(h^2\cdot\frac{U}{\lambda}\right) = O\left(\frac{n}{B}\right)$ blocks in total.

To report the skyline of the points in a given 3-sided range $[x_l, x_r]\times[y_b, U]$, where $x_l, x_r, y_b \in [U]$, first we find the slabs z_l, z_r containing x_l, x_r, respectively, by dividing the x-coordinates with λ in $O(1)$ I/Os. Let u be the lowest common ancestor of z_l and z_r in the tree, also found in $O(1)$ I/Os since the tree in complete. Unless the z_l, z_r are not identical or consecutive (in which case we simply use Lemma 3.4), the query algorithm proceeds in the following sequence:

1. Query the few-points structure at z_r with $[1, x_r]\times[y_b, U]$, and let y' be the largest reported y-coordinate.
2. Report the points in $\mathrm{rmax}(z_l, u)$ and $\mathrm{lmax}(z_r, u)$ with y-coordinate higher than y'. Let v_i denote the i-th right or left sibling node, respectively to the u-to-z_l and u-to-z_r path, with all B skyline points reported. Report the points in $\mathrm{MAX}(v_i)$ with y-coordinate larger than the largest y-coordinate in v_{i+1}.
3. Query the few-points structure at z_l with $[x_l, U]\times\left[y_b, y'\right]$, where y' is the largest reported y-coordinate.

Lemmata 2.2 and 3.4 imply real-time reporting. □

Predecessor search on coordinates from a static universe $U \geq n$ incurs a slow-down to our query algorithm [31].

Corollary 3.5.1. *Given n divisible points in $[U]^2$, there exist data structures that support orthogonal 3-sided range skyline reporting queries in $O\left(\log\log_B U + \frac{k}{B}\right)$ I/Os, where k is the size of the reported skyline, using $O\left(\frac{n}{B}\right)$ blocks.*

3.2. 4-sided range skyline reporting

In this subsection we prove a *tight* query I/O-complexity lower bound for any linear-space data structure in the indexability model [4] that supports 2-d orthogonal *anti-dominance* range skyline reporting queries that report the skyline of the points dominated by a given query point (x, y). This is the simplest special case of the general planar 4-sided ranges that requires superlinear space to support queries in polylogarithmic I/Os.

Lemma 3.6. *For any integers $\omega, \lambda \geq 1$, there exists a set of ω^λ points and a set of $\lambda\omega^{\lambda-1}$ anti-dominance ranges in the plane, such that:*

(i) every range contains ω skyline points,
(ii) 2 different ranges contain at most one common skyline point.

Proof. In the context of Chazelle and Liu [3,6], this queryset is $(2,\omega)$-*favorable* for the input pointset. To construct the pointset, for every integer $i \in [1,\omega]$, denote by $i_j^{(\omega)}$ its j-th digit in base ω, and define the point:

$$\left(\left(i_0^{(\omega)}\right)\ldots\left(i_{\lambda-1}^{(\omega)}\right), \left(\omega - i_{\lambda-1}^{(\omega)} - 1\right)\ldots\left(\omega - i_0^{(\omega)} - 1\right)\right)$$

with x-coordinate integer i written with λ digits in base ω and y-coordinate the integer obtained after reversing the digits' order and taking their complement in base ω. To construct the queryset, we define a *trie* of depth λ, where leaves correspond to the input points' x-coordinates and the edge of an internal node at depth $d \in [0, \lambda-1]$ to its parent has y-label $\omega - i_d^{(\omega)} - 1$. For every internal node at depth d, we define the anti-dominance query range $\left(-\infty, i_\omega^{(\omega)}\right)\times\left(-\infty, i_1^{(\omega)}\right)$ that contains every $\omega^{\lambda-d-1}$-th y-highest point in the node's subtree. See Fig. 5 for an example.

(i) Indeed, every range contains exactly ω points that are actually skyline points, since they differ among each other only at the $\omega^{\lambda-d-1}$-th digit and are thus arranged in both increasing x- and decreasing y-coordinate. (ii) For any two query ranges, their corresponding nodes in the trie have either disjoint subtrees or the one is a proper ancestor of the other. In the former case, the ranges contain no common points, since every point is encoded only once in the trie. In the latter case, they contain at most one common point, since the only digit that changes to define the points in one range is fixed for some point in the other range. □

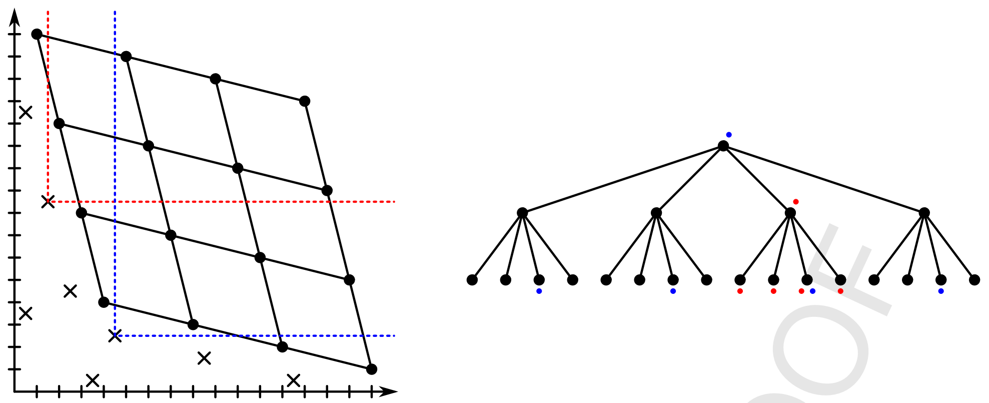


Fig. 5. The constructed pointset (dots) and queryset (crosses) for $\omega = 4$ and $\lambda = 2$ and the corresponding trie. The points are aligned to a Fibonacci lattice on the plane with reversed x and y directions, where the query points represent dominance ranges.

Theorem 3.7. *Any static anti-dominance range skyline reporting data structure in the indexability model that uses at most $c\frac{n}{B}$ blocks, for some constant $c \geq 1$, supports the queries in $\Omega\left(\left(\frac{n}{B}\right)^{1/25c} + \frac{k}{B}\right)$ I/Os, where k is the size of the reported skyline.*

Proof. The *indexability theorem* [4, Theorem 1] defines *access overhead* A for any data structure on an (ω, λ)-input of Lemma 3.6 that reports ω points in $O\left(A\frac{\omega}{B}\right)$ I/Os, and claims that if $\omega \geq \frac{B}{2}$ and $A \leq \frac{\sqrt{B}}{4}$, the structure uses at least $\frac{\lambda\omega^\lambda}{12B}$ blocks. Therefore, a structure with query I/O-complexity $O\left(\left(\frac{n}{B}\right)^{1/25c} + \frac{k}{B}\right)$, reports ω points on a $(B, 12c + 1.1)$-input in at most:

$$\alpha\left(\left(\frac{\omega^\lambda}{B}\right)^{1/25c} + \frac{\omega}{B}\right) = \alpha\left(B^{\frac{12c+0.1}{25c}} + 1\right) \leq \alpha\left(B^{\frac{121}{25}} + 1\right) \text{ I/Os},$$

for a constant $\alpha > 0$. Thus, for large enough B, we get $A \leq \alpha\left(B^{\frac{121}{25}} + 1\right) < \frac{\sqrt{B}}{4}$ and hence need at least $\frac{\lambda\omega^\lambda}{12B} = \left(c + \frac{1.1}{12}\right)\frac{n}{B} > c\frac{n}{B}$ blocks. $\square$

Matching data structure. We obtain a data structure matching the bounds in Theorem 3.7 by implementing the structure of Theorem 3.2 with a B-tree with internal fanout $\left(\frac{n}{B\log^{1-\varepsilon}\frac{n}{B}}\right)^{\varepsilon}$, for some constant $\varepsilon > 0$, and thus of constant height. The structure also supports updates, when using a weight-balanced B-tree.

Corollary 3.7.1. *Given n points in the plane and for any constant $\varepsilon \in [0, 1]$, there exist data structures that support orthogonal 4-sided range skyline reporting queries in $O\left(\left(\frac{n}{B}\right)^{\varepsilon} + \frac{k}{B}\right)$ I/Os, where k is the size of the reported skyline, and updates in $O\left(\log\frac{n}{B}\right)$ amortized I/Os, using $O\left(\frac{n}{B}\right)$ blocks.*

4. Conclusion

We have presented the first *worst-case* I/O-efficient data structures for skyline range reporting in 2-d. It would be interesting to prove the I/O-optimality of our dynamic 2-d structure in the *dynamic indexability model* [7]. Recently, significantly sublogarithmic update time was achieved for standard dynamic 2-d orthogonal range reporting in the word-RAM [32]. Our results suggest that similar improvements are possible for dynamic 2-d orthogonal range skyline reporting. Polylogarithmic worst-case bounds for *dynamic 3-d orthogonal range skyline reporting* are still unknown even in internal memory.

Declaration of competing interest

None declared.

References

[1] C. Kejlberg-Rasmussen, Y. Tao, K. Tsakalidis, K. Tsichlas, J. Yoon, I/O-efficient planar range skyline and attrition priority queues, in: Proc. of 32nd ACM Symposium on Principles of Database Systems (PODS), 2013, pp. 103–114.
[2] L. Arge, J.S. Vitter, Optimal external memory interval management, SIAM J. Comput. 32 (6) (2003) 1488–1508, https://doi.org/10.1137/S009753970240481X.

[3] B. Chazelle, Lower bounds for orthogonal range searching: I. The reporting case, J. ACM 37 (2) (1990) 200–212, https://doi.org/10.1145/77600.77614.
[4] L. Arge, V. Samoladas, J.S. Vitter, On two-dimensional indexability and optimal range search indexing, in: Proc. of 18th ACM Symposium on Principles of Database Systems (PODS), 1999, pp. 346–357.
[5] J.M. Hellerstein, E. Koutsoupias, D.P. Miranker, C.H. Papadimitriou, V. Samoladas, On a model of indexability and its bounds for range queries, J. ACM 49 (1) (2002) 35–55, https://doi.org/10.1145/505241.505244.
[6] B. Chazelle, D. Liu, Lower bounds for intersection searching and fractional cascading in higher dimension, J. Comput. Syst. Sci. 68 (2) (2004) 269–284, https://doi.org/10.1016/j.jcss.2003.07.003.
[7] K. Yi, Dynamic indexability and the optimality of B-trees, J. ACM 59 (4) (2012) 21, https://doi.org/10.1145/2339123.2339129.
[8] A.K. Kalavagattu, A.S. Das, K. Kothapalli, K. Srinathan, On finding skyline points for range queries in plane, in: Proc. of 23rd Canadian Conf. on Computational Geometry (CCCG), 2011.
[9] A. Das, P. Gupta, A. Kalavagattu, J. Agarwal, K. Srinathan, K. Kothapalli, Range aggregate maximal points in the plane, in: Proc. of 6th Int. Workshop on Algorithms and Computation (WALCOM), vol. 7157, 2012, pp. 52–63.
[10] S. Rahul, R. Janardan, Algorithms for range-skyline queries, in: Proc. of 20th Int. Conf. on Advances in Geographic Information Systems (SIGSPATIAL), 2012, pp. 526–529.
[11] G.S. Brodal, K. Tsakalidis, Dynamic planar range maxima queries, in: 38th Int. Colloquium on Automata, Languages and Programming (ICALP), 2011, pp. 256–267.
[12] M.H. Overmars, D. Wood, On rectangular visibility, J. Algorithms 9 (3) (1988) 372–390, https://doi.org/10.1016/0196-6774(88)90028-4.
[13] G.N. Frederickson, S. Rodger, A new approach to the dynamic maintenance of maximal points in a plane, Discrete Comput. Geom. 5 (1990) 365–374, https://doi.org/10.1007/BF02187797.
[14] R. Janardan, On the dynamic maintenance of maximal points in the plane, Inf. Process. Lett. 40 (2) (1991) 59–64, https://doi.org/10.1016/0020-0190(91)90010-F.
[15] F. d'Amore, P.G. Franciosa, R. Giaccio, M. Talamo, Maintaining maxima under boundary updates, in: Italian Conference on Algorithms and Complexity, 1997, pp. 100–109.
[16] S. Kapoor, Dynamic maintenance of maxima of 2-d point sets, SIAM J. Comput. 29 (2000) 1858–1877, https://doi.org/10.1137/S0097539798348365.
[17] M.H. Overmars, J. van Leeuwen, Maintenance of configurations in the plane, J. Comput. Syst. Sci. 23 (2) (1981) 166–204, https://doi.org/10.1016/0022-0000(81)90012-X.
[18] H.T. Kung, F. Luccio, F.P. Preparata, On finding the maxima of a set of vectors, J. ACM 22 (4) (1975) 469–476, https://doi.org/10.1145/321906.321910.
[19] C. Sheng, Y. Tao, On finding skylines in external memory, in: Proc. of 30th ACM Symposium on Principles of Database Systems (PODS), 2011, pp. 107–116.
[20] G.S. Brodal, K.G. Larsen, Optimal planar orthogonal skyline counting queries, in: Proceedings of Scandinavian Symposium and Workshops on Algorithm Theory (SWAT), 2014, pp. 110–121.
[21] C. Kalyvas, T. Tzouramanis, A survey of skyline query processing, CoRR arXiv:1704.01788, http://arxiv.org/abs/1704.01788.
[22] D. Papadias, Y. Tao, G. Fu, B. Seeger, Progressive skyline computation in database systems, ACM Trans. Database Syst. 30 (1) (2005) 41–82, https://doi.org/10.1145/1061318.1061320.
[23] Z. Huang, H. Lu, B.C. Ooi, A.K.H. Tung, Continuous skyline queries for moving objects, IEEE Trans. Knowl. Data Eng. 18 (12) (2006) 1645–1658, https://doi.org/10.1109/TKDE.2006.185.
[24] Y. Tao, D. Papadias, Maintaining sliding window skylines on data streams, IEEE Trans. Knowl. Data Eng. 18 (3) (2006) 377–391, https://doi.org/10.1109/TKDE.2006.48.
[25] P. Wu, D. Agrawal, Ö. Egecioglu, A.E. Abbadi, Deltasky: optimal maintenance of skyline deletions without exclusive dominance region generation, in: Proc. of 23rd Int. Conf. on Data Engineering (ICDE), 2007, pp. 486–495.
[26] R. Sundar, Worst-case data structures for the priority queue with attrition, Inf. Process. Lett. 31 (1989) 69–75, https://doi.org/10.1016/0020-0190(89)90071-9.
[27] H. Kaplan, R.E. Tarjan, Purely functional, real-time deques with catenation, J. ACM 46 (5) (1999) 577–603, https://doi.org/10.1145/324133.324139.
[28] B. Becker, S. Gschwind, T. Ohler, B. Seeger, P. Widmayer, An asymptotically optimal multiversion B-tree, VLDB J. 5 (4) (1996) 264–275, https://doi.org/10.1007/s007780050028.
[29] M.L. Fredman, D.E. Willard, Surpassing the information theoretic bound with fusion trees, J. Comput. Syst. Sci. 47 (3) (1993) 424–436, https://doi.org/10.1016/0022-0000(93)90040-4.
[30] K.G. Larsen, R. Pagh, I/O-efficient data structures for colored range and prefix reporting, in: Proc. of 23rd ACM-SIAM Symposium on Discrete Algorithms (SODA), 2012, pp. 583–592.
[31] M. Pătraşcu, M. Thorup, Time-space trade-offs for predecessor search, in: Proc. of 38th ACM Symposium on Theory of Computing (STOC), 2006, pp. 232–240.
[32] T.M. Chan, K. Tsakalidis, Dynamic orthogonal range searching on the ram, revisited, J. Comput. Geom. 9 (2) (2018) 45–66, https://doi.org/10.20382/jocg.v9i2a5.

Sponsor names

Do not correct this page. Please mark corrections to sponsor names and grant numbers in the main text.

National Research Foundation of Korea, *country*=South Korea, *grants*=R31-30007

NSF, *country*=United States, *grants*=CCF-1319648, CCF-1533564