Dynamic planar range skyline queries in log logarithmic expected time

K. Doka\textsuperscript{a}, A. Kosmatopoulos\textsuperscript{b,∗}, A. Papadopoulos\textsuperscript{b}, S. Sioutas\textsuperscript{c}, K. Tsichlas\textsuperscript{b}, D. Tsoumakos\textsuperscript{c}

\textsuperscript{a} National Technical University of Athens, Computing Systems Lab, Athens, Greece
\textsuperscript{b} Aristotle University of Thessaloniki, Data Engineering Laboratory, Thessaloniki, Greece
\textsuperscript{c} Ionian University, Information Systems and Databases Laboratory, Corfu, Greece

\section*{Abstract}

The skyline of a set \(P\) of points consists of the "best" points with respect to minimization or maximization of the attribute values. A point \(p\) dominates another point \(q\) if \(p\) is as good as \(q\) in all dimensions and it is strictly better than \(q\) in at least one dimension. In this work, we focus on the 2-d space and provide expected performance guarantees for dynamic (insertions and deletions) 3-sided range skyline queries. We assume that the \(x\) and \(y\) coordinates of the points are drawn from a class of distributions and present the ML-tree (Modified Layered Range-tree), which attains \(O(\log^2 N \log \log N)\) expected update time and \(O(t \log \log N)\) time with high probability for finding planar skyline points in a 3-sided query rectangle \(q = [a, b] \times [d, +\infty)\) in the RAM model, where \(N\) is the cardinality of \(P\) and \(t\) is the answer size.

© 2020 Elsevier B.V. All rights reserved.

\section{Introduction}

In this paper, we study efficient algorithms with non-trivial performance guarantees for dynamic planar skyline processing. Let \(P\) denote the set of points in the data set. Also, let \(p_i\) denote the value of the \(i\)-th coordinate of a point \(p\). A point \(p \in P\) dominates another point \(q \in P\) \((q \prec p)\) when \(\forall i, p_i \geq q_i\) and \(\exists j\) such that \(p_j > q_j\). The skyline of a set of points \(P\) contains the points that are not dominated by any other point.

In this work, we present the ML (Modified Layered Range) tree-structure that provides a loglogarithmic expected solution for finding planar skyline points in a 3-sided query rectangle \([a, b] \times [d, +\infty)\) in the RAM model under point insertions and deletions. This form of query resembles a 3-sided range reporting query with an additional skyline requirement and is handled by the ML-Tree for points drawn from specific distributions in \(O(\log^2 N \log \log N)\) expected update time and \(O(t \log \log N)\) query time w.h.p. The proposed data structure is inspired from the Modified Priority Search Tree presented in [6] that supports 3-sided range reporting queries. However, the modifications to support skyline queries are non-trivial. Note, that if the query range is defined as \([a, b] \times (−∞, d]\) then the problem becomes harder since the skyline can change dramatically based on the choice of \(d\). This is in fact the main reason for which the 4-sided skyline query [1] is more expensive than the respective 3-sided skyline query.

The best previous solution was presented in [1] and supports range skyline queries in \(O(\frac{\log N}{\log \log N} + t)\) worst case time and updates in \(O(\frac{\log N}{\log \log N})\) worst case time using linear space in the RAM model of computation. Al-
though their solution is optimal in the generic case, ML-tree achieves better query time when the size of the reported skyline is small (approximately \( t < \frac{\log N}{\log \log N} \)) and when the point coordinates follow specific class distributions.

Our work is organized as follows. Fundamental concepts are presented in Section 2 while the ML-tree and its dynamic version are given in Sections 3 and 4, respectively.

2. Fundamental concepts

For the remainder of this work we adhere to the RAM model of computation. We denote by \( N \) the number of elements that reside in the data structures and by \( t \) the size of the query.

Furthermore, throughout this work we make use of \((f_1, f_2)\)-smooth distributions for which we provide an intuitive definition: “among a number (measured by \( f_1(n) = n^\alpha, \alpha < 1 \)) of consecutive subsets, each containing consecutive keys from a universe \( U \), no subset containing consecutive keys from \( U \) should be too dense (measured by \( f_2(n) = n^\beta, \beta < 1 \)) compared to the others” (i.e., the distribution does not contain sharp peaks). A detailed description of \((f_1, f_2)\)-smooth probability distributions can be found in [5].

Insertions follow a particular distribution among the family of \((f_1, f_2)\)-smooth distributions while deletions of elements are equiprobable. That is, every element present in the data structure is equally likely to be deleted [7]. In the following, we describe the data structures that we use in order to achieve the desired complexities.

Half-Range Minimum/Maximum Queries: The half-Range Maximum Query \((h\text{-RMQ})\) problem asks to preprocess an array \( A \) of size \( N \) such that, given an index range \([r, N]\) where \( 1 \leq r \leq N \), we are asked to report the position of the maximum element in this range on \( A \). Notice that we do not want to change the order of the elements in \( A \), in which case the problem would be trivial. This is a restricted version of the general RMQ problem, in which the range is \([r, r']\), where \( 1 \leq r \leq r' \leq N \). In [4] the RMQ problem is solved in \( O(1) \) time using \( O(N) \) space and \( O(N) \) preprocessing time. We could use this solution for our h-RMQ problem, but in our case the problem can be solved much simpler by maintaining an additional array \( A_{\max} \) of maximum elements for each on the \( N \) positions in the initial array.

\( q^*\text{-heaps} \): The \( q^*\text{-heap} \) [11] is a data structure having the following property: let \( M \) be the current number of elements in the \( q^*\text{-heap} \) and let \( N \) be an upper bound on the maximum number of elements ever stored in the \( q^*\text{-heap} \). Then, update and query operations are carried out in \( O(1 + \frac{\log M}{\log \log M}) \) worst-case time after an \( O(N) \) preprocessing overhead. The \( q^*\text{-heap} \) uses linear space and is constructed in linear time.

Interpolation Search Trees: In [5], a dynamic data structure was presented that supports insertions/deletions in \( O(1) \) time w.w.c. as well as predecessor/predecessor queries in \( O(\log \log N) \) expected time w.h.p., given that the keys are drawn from a \((N^\alpha, N^\beta)\)-smooth distribution, where \( 0 < \alpha, \beta < 1 \). It requires linear space.

3. The static ML-tree

In the following, we describe in detail the indexing scheme, which is termed as the Modified Layered Range Tree (ML-tree). This static data structure for the problem servers only as a step towards the dynamic solution and does not provide better complexities in total for the static case when compared to what is currently known.

3.1. The static non-linear-space ML-tree

The static non-linear ML-tree is stored as an array \((A)\) in memory, yet it can be visualized as a complete binary tree. The static data structure is an augmented binary search tree \( T \) on the set of points \( S \) that resembles a range tree. \( T \) stores all points in its leaves with respect to their \( x\)-coordinate in increasing order. Let \( H \) be the height of tree \( T \). We denote by \( T_v \) the subtree of \( T \) rooted at the internal node \( v \).

Let \( P_q \) be the root-to-leaf path for leaf \( \ell \) of \( T \). We denote by \( P_\ell^r \) the subpath of \( P_q \) consisting of nodes with depth \( \geq r \). Similarly, \( P_\ell^r \leftarrow P_\ell^r \) denotes the set of nodes that are left (right) children of nodes of \( P_\ell^r \) and do not belong to \( P_\ell^r \). \( v \) is the point stored in leaf \( \ell \) of the tree where \( q_v \) is its \( x\)-coordinate and \( q_y \) is its \( y\)-coordinate. Let \( P_q \) denote the search path for \( q_v \), i.e., it is the path from the root to \( \ell \) and it is equal to \( P_\ell \). We augment \( T \) as follows:

- Each internal node \( v \) stores a point \( q_v \), which is the point with the maximum \( y\)-coordinate among all points in its subtree \( T_v \). It also stores its depth.
- Each internal node \( v \) has a secondary data structure \( S_v \), which stores all points in \( T_v \) with respect to \( y\)-coordinate in increasing order. \( S_v \) is implemented with an IS-tree as well as with an h-RMQ structure, where the maximum is w.r.t. the \( x\)-coordinate.
- Each leaf \( \ell \) stores arrays \( L_\ell^r \) and \( R_\ell^r \), where \( 0 \leq r \leq H - 1 \), corresponding to sets \( P_\ell^r \leftarrow \) and \( P_\ell^r \) respectively. More specifically, they contain the points \( q_v \) for each node \( v \) in the corresponding sets. These are sorted with respect to their \( y\)-coordinate and they are implemented with \( q^*\text{-heaps} \) and h-RMQ structures, where the maximum is w.r.t. the \( x\)-coordinate.

We also use an IS-tree \( T' \) to allow for efficient predecessor/predecessor queries in the leaves of \( T \). Finally, tree \( T \) is preprocessed in order to support Lowest Common Ancestor queries in \( O(1) \) time. Since \( T \) is static, one can use the methods of [3,4] to find the LCA (as well as its depth) of two leaves in \( O(1) \) time by attaching to each node of \( T \) a label. We now move on to the description of the skyline query for a query range \( q = [a, b] \times [d, +\infty) \):

1. We use the IS-tree \( T' \) to find the two leaves \( \ell_a \) and \( \ell_b \) of \( T \) for the search paths \( P_a \) and \( P_b \) respectively. Let \( w \) be the LCA of leaves \( \ell_a \) and \( \ell_b \) and let \( \tau \) be its depth.
2. The successor of \( d \) is located in \( R^\tau(\ell_a) \) and \( L^\tau(\ell_b) \) and let these successors be at positions \( \text{successor}[d] \) and \( \text{successor}[d] \) respectively. In addition, let \( v_1 \) be the node
that has the following property: the y-coordinate of point \( q_{v_1} \) belongs in the range \([d, +\infty)\) and it has the largest \( x \)-coordinate (the \( x \)-coordinate of \( q_{v_1} \) falls in the \([a, b]\) range because of Step 1) among all nodes in \( T_{ls}^{left} \) and \( T_{ls}^{right} \).

3. By executing an h-RMQ in \( T_{ls}^{left} \) and \( T_{ls}^{right} \) arrays for the range \([\text{succ}_L[d], \tau]\) and \([\text{succ}_R[d], \tau]\) node \( v_1 \) is located. The subtree \( T_{v_1} \) stores the point with the maximum \( x \)-coordinate among all points in the query range \([a, b]\) \( \times \) \([d, +\infty)\). By executing a successor query for \( d \) in \( S_{v_1} \), returning the result \( \text{succ}_L[d] \) and then making an h-RMQ in \( S_{v_1} \) for the range \([\text{succ}_L[d], |S_{v_1}|]\), we find and report the required point with the maximum \( x \)-coordinate \( z = (z_x, z_y) \) that belongs to the skyline.

4. The query range now becomes \( q = [a, z_x] \times [z_y, +\infty) \) and we repeat the previous steps until \( S \cap q = \emptyset \).

For each skyline point, we execute \( O(1) \) successor queries in total. Since the \( q^* \)-heap queries and all other steps can be carried out in \( O(1) \) time, the total time cost of the query algorithm is \( O(t \cdot t_{ls}(N)) \) where \( t_{ls}(N) \) is the time required by an IS-tree for a successor query and \( t \) is the answer size. The space cost of the ML-tree is dominated by the space used for implementing the \( L_x^T \), \( R_x^T \) and \( S_y \) sets, which is \( O(N \log^2 N) \) since each point is stored in \( O(\log N) \) \( S_y \) structures and each \( \ell \)-leaf among the \( N \) leaves in total, stores \( O(\log N) \) \( L_x^T \) and \( R_x^T \) sequences each of which has size \( O(\log N) \).

3.2. The main memory static linear-space ML-tree

We reduce the space of the data structure by employing a pruning technique [2,10] as follows: consider the nodes of \( T \) with height \( 2 \log \log N \). These nodes are roots of subtrees of \( T \) of size \( O(\log^2 N) \) and there are \( \Theta \left( \frac{N}{\log^2 N} \right) \) such nodes. Let \( T_1 \) be the tree whose leaves are these nodes and let \( T_2 \) be the subtrees of these nodes for \( 1 \leq i \leq \Theta \left( \frac{N}{\log^2 N} \right) \).

We call \( T_1 \) the first layer of the structure and the subtrees \( T_2 \) the second layer.

\( T_1 \) and each subtree \( T_2 \) is implemented as a static non-linear space ML-tree. The representative of each tree \( T_2 \) is the point with the maximum \( y \)-coordinate among all points in \( T_2 \). The leaves of \( T_1 \) contain only the representatives of the respective trees \( T_2 \). Each tree \( T_2 \) is further pruned at height \( 2 \log \log N \) resulting in trees \( T_3 \) with \( \Theta(\log^2 \log N) \) elements. Once more, \( T_3 \) contains the representatives of the third layer trees in a similar way as before. Each tree \( T_3 \) is structured as a table that stores all possible precomputed solutions. Specifically, each \( T_3 \) is structured by using a \( q^* \)-heap with respect to the \( x \)-coordinate as well as one with respect to the \( y \)-coordinate. In this way, we can extract the position of the successor in \( T_3 \) with respect to \( x \) and \( y \) coordinates. What is needed to be computed for \( T_3 \) is the point with the maximum \( x \)-coordinate that lies within a 3-sided range region. To obtain this, we use precomputation and tabulation for all possible results.

For the sake of generality, assume that the size of \( T_3 \) is \( k \). Let the points in \( T_3 \) be \( q_1, q_2, \ldots, q_k \) sorted by \( x \)-coordinate. Let their rank according to \( y \)-coordinate be given by the function \( \alpha(i), 1 \leq i \leq k \). Apparently, function \( \alpha \) may generate all possible \( k! \) permutations of the \( k \) points. We make a four-dimensional table \( \text{ANS} \), which is indexed by the number of permutations (one dimension with \( k! \) choices) as well as the possible positions of the successor (3 dimensions with \( k + 1 \) choices for the 3-sided range). Each cell of array \( \text{ANS} \) contains the position of the point with the maximum \( x \)-coordinate for a given permutation that corresponds to a tree \( T_3 \) and the 3-sided range. Each tree \( T_3 \) corresponds to a permutation index that indexes one dimension of table \( \text{ANS} \). The other 3 indices are generated by one predecessor and one successor query on the \( x \)-coordinate and one successor query on the \( y \)-coordinate. The size of \( \text{ANS} \) for each \( T_3 \) is \( O(k!(k+1)^3) \).

Let \( q = [a, b] \times [d, +\infty) \) be the initial range query. To answer this query on the three layered structure we access the layer 3 trees containing \( a \) and \( b \) by using the \( T \)-tree. Then, we locate the subtrees \( T_2 \) and \( T_3 \) containing the representative leaves of the accessed layer 3 trees. The roots of these subtrees are leaves of \( T_1 \). The ML query algorithm described in Section 3.1 is executed on \( T_1 \) with these leaves as arguments. Once we reach the node with the maximum \( x \)-coordinate, we continue in the layer 2 tree corresponding to the representative with the maximum \( x \)-coordinate located in \( T_1 \). The same query algorithm is executed on this layer 2 tree and then we move similarly to a tree \( T_3 \) in the third layer. We make three in total successor queries for \( a, b, \) and \( d \) in \( T_3 \) and we use the \( \text{ANS} \) table to locate the point with the maximum \( x \)-coordinate by retrieving the permutation index of \( T_3 \). Let the point \( z = (z_x, z_y) \) be the desired point at the third layer. We go back to \( T_1 \). The range query now becomes \( q = [a, z_x] \times [z_y, +\infty) \) and we iterate as described in Section 3.1.

The total space required for the data structure depends on the size of each of the three layers. For the first layer, the \( \text{ML} \)-tree on the \( O \left( \frac{N}{\log \log N} \right) \) representatives requires \( O \left( \left( \frac{N}{\log \log N} \right)^2 \frac{N}{\log \log N} \right) = O(N) \) space for the leaf structures (all \( P_\ell \) structures for each leaf \( \ell \) are structured as \( q^* \)-heaps and h-RMQ structures requiring linear space). For the \( S_y \) structures, the total space needed is \( O \left( \frac{N}{\log \log N} \right) = O(N) \). A similar reasoning can be made for the second layer that consists of \( O \left( \frac{N}{\log \log N} \right) \) trees with \( O \left( \left( \log \log N \right)^2 \right) \) representative points of the third layer each for a total space of \( O(\log^2 N) \). In the third layer, we use linear space for the two predecessor data structures (\( q^* \)-heaps) as well as a table of size \( O \left( \frac{N}{\log \log N} \right) \left( \frac{\log \log N + 1}{\log \log N} \right) \), which is \( O(N) \). The construction time of the data structure can be similarly derived taking into account that the \( \text{ANS} \) table can be constructed in \( O(N) \) time. The query time is bounded by the \( O(1) \) number of successor queries per actual resulting skyline point. The following lemma summarizes this dis-
cussion and it will be used to design the dynamic data structure.

**Lemma 1.** Given a set of 2-d N points, we can store them in a static main memory data structure that can be constructed in $O(N \log N)$ time using $O(N)$ space. It supports skyline queries in a 3-sided range in $O(t \cdot T_{15}(N))$ worst-case time, where $t$ is the answer size and $T_{15}(N)$ is the time required by an IS-tree for a predecessor/successor query.

4. The dynamic ML-tree

Making the ML-tree described in Section 3.2 dynamic involves all layers. The following issues must be tackled in order to make the ML-tree dynamic:

1. Use of a dynamic tree structure with care to how rebalancing operations are performed.
2. The layer 3 trees must have variable size within a predefined range, rebuilding them appropriately as soon as they violate this bound (by splitting or merging/sharing with adjacent trees) - similarly, the permutation index must be appropriately defined in order to allow for variable length permutations.
3. All arrays attached to nodes or leaves must be updated efficiently.

We use global rebuilding [9] to maintain the structure. In particular, let $N_0$ be the number of elements stored at the time of the latest reconstruction. At the time when the number of updates exceeds $rN_0$, where $0 < r < 1$ is a constant, the whole data structure is reconstructed taking into account that the number of elements is $rN_0$. In this way, it is guaranteed that the current number of elements $N$ is always within the range $[(1 - r)N_0, (1 + r)N_0]$. We call the time between two successive reconstructions an epoch. The tree structure used for the first two layers is a weight-balanced tree, like the BB[a]-trees [8].

Henceforth, assume for brevity that $k = \log^2 \log N$. We impose that all trees at layer 3 will have size within the range $[k/4, k]$. To compute the permutation index, if the size of the layer 3 tree is $< k$, then we pad the increasing sequence of elements in the tree with $+\infty$ values in order to have exactly size $k$.

Assume that an update operation takes place. The following discussion concerns the case of inserting a new point $q = (q_x, q_y)$ since the case of deleting an existing point $q$ from the structure is symmetric. First, $T'$ is used to locate the predecessor of $q_x$, and in particular to locate the tree $T_{31}$ of layer 3 that contains the predecessor of $q_x$. Then, $T'$ is updated accordingly. The predecessor of $q_x$ in $T_{31}$ is located by using the respective $q^*$-heap. If $|T_{31}| \in [k/4, k]$, then $q_x$ and $q_y$ are inserted in the respective $q^*$-heaps and a new permutation index is computed for $T_{31}$. If $|T_{31}| > k$, then $T_{31}$ is split into two trees with size approximately $\frac{k}{2}$. This means that 4 new $q^*$-heaps must be constructed while two new permutation indices must be computed for the two new trees. Let $T_{32}$ be the layer 2 tree that gets the new leaf. Note that $T_{32}$ is affected either structurally, when one of its leaves $\ell$ at layer 3 splits as in this case ($\ell$ is $T_{32}$) or it is affected without structural changes, when $q_x$ is maximum among all the $y$-coordinates of $T_{32}$ and thus the representative of $T_{32}$ changes. In the latter case, all structures $S_v$ on the path $P_v$ of $T_{32}$ must be updated with the new point. In addition, let $v$ be the highest node with height $h_v$ in $T_{32}$ that has $p_v = q$ (the point with the maximum $y$-coordinate in its subtree changes to $q$). Then, for all leaves $\ell$ in the subtree of the father of $v$, the $q^*$-heaps for $L_{3\ell}$ and $R_{3\ell}$ as well as the $h$-RMQ structures that contain $v$ will be updated, given that $r \geq h_v$. In the former case, we make rebalancing operations on the internal nodes of $T_{32}$ on the path $P_v$. These rebalancing operations result in changing, as in the previous case, the $q^*$-heaps for the $L_{3\ell}$ and $R_{3\ell}$ while the respective $S_v$ structures of the node $v$ that is rebalanced have to be recomputed as well. Similar changes happen to the tree $T_1$ of the first layer given that either a tree of the second layer splits or its maximum element is updated. In case of deleting $x$, the 3 layers of the ML-tree are handled similarly.

Recall that the time complexity of the update operation supported by the IS-tree and $q^*$-heap is $O(1)$. The change of the point with the maximum $y$-coordinate can always propagate from $T_{32}$ to the root of $T_1$. $T_{32}$ can be updated in $O(|T_{32}|)$ time since the two updates in $q^*$-heaps cost $O(1)$ while the computation of the permutation index costs $O(|T_{32}|)$. Let the respective tree in the second layer be $T_{32}$. Then, the cost for changing the point with the maximum $y$-coordinate in each node on the path from the leaf to the root of $T_{32}$ is related to the update cost for the $L_{3\ell}$ and $R_{3\ell}$ lists as well as for the $S_v$ structures. In particular, all $O(|T_{32}| \log |T_{32}|)$ lists $L_{3\ell}$ and $R_{3\ell}$ are updated (deletion of the previous point and insertion of the new one in a $q^*$-heap) in $O(|T_{32}| \log |T_{32}|)$ time. Similarly, a deletion and an insertion is carried out in each $S_v$ structure in $O(\log |T_{32}|)$ total time. The same holds for the tree $T_1$ netting a total complexity of $O(|T_1| \log |T_1|)$.

Rebalancing operations on the level 2 trees as well as on the level 1 tree of the structure may be applied when splits or fusions of leaves of level 2 trees take place. Since level 2 trees are exponentially smaller than the level 1 tree, the cost is dominated by the rebalancing operations at $T_1$. Assume an update operation at a leaf $\ell$ of $T_1$. In the worst case, each $S_v$ structure may have to be rebuilt and similarly to the previous paragraph the $L_{1\ell}$ and $R_{1\ell}$ structures need to be updated. The total cost is equal to $O(|T_1| \log^2 |T_1|)$ for the $O(|T_1| \log |T_1|)$ lists while it is $O(|T_1|)$ for the $S_v$ structures since the reconstruction of the $S_v$ structure of the root $r$ dominates the cost. One can similarly reason for level 2 trees. However, the amortized cost is way lower for two reasons: 1. A leaf of $T_1$ is updated roughly every $O(\log^2 N)$ update operations and 2. The weight property of the tree structures guarantees that costly operations are rare. By using a standard weight property argument along with the above two reasons we get that the amortized rebalancing cost is $O(\log^2 N + \frac{|T_{32}| \log |T_{32}|}{N})$. This amortized cost is dominated by the cost to update the maximum element, in which
case the worst-case as well as the amortized case coincide. Thus, we obtain the following theorem:

**Theorem 1.** Given a set of \( N \) points we can store them in a dynamic main memory data structure that uses \( O(N) \) space and supports update operations in \( O\left(\frac{N}{\log N}\right) \) time in the worst case. It supports skyline queries in a 3-sided range in \( O\left(t \cdot T_3(N)\right) \) worst-case time, where \( t \) is the answer size and \( T_3(N) \) is the time required by an IS-tree for a predecessor query.

Although rebalancing operations are efficient in an amortized sense, the change of maximum depends on the user and in the worst-case this change can propagate to the root in each update operation. We overcome this problem by making a rather strong assumption about the distribution of the points.

### 4.1. Exploiting the distribution of the elements

To reduce the huge worst-case update cost of Theorem 1 we have to tackle the propagation of maximum elements. To accomplish this we assume that the coordinates of the points are generated by discrete distributions. The result can also be attained by minor modifications for the case of continuous distributions. Assume that a new point \( q \equiv (q_x, q_y) \) is to be inserted in the ML-tree. Let \( q \) be stored in level 2 tree \( T_2 \) based on \( q_x \). We call the point \( q \) violating if \( q_x \) is the maximum \( x \)-coordinate among all \( y \)-coordinates of the points in \( T_2 \). When a new point is violating it means that a costly update operation must be performed on \( T_1 \). In the following, we show that under assumptions on the generating distributions of the \( x \) and \( y \) coordinates of points we can prove that during an epoch only \( O(\log N) \) violations will happen.

We assume that all points have their \( x \)-coordinate generated by the same discrete distribution \( \mu \) that is \( (f_1(N) = \frac{N}{(\log \log N)^{2+\epsilon}}, f_2(N) = N^{1-\delta}) \)-smooth distribution, where \( \epsilon > 0 \) and \( \delta \in (0, 1) \) are constants. We also assume that the \( y \) coordinates of all points are generated by a restricted set of discrete distributions \( \mathcal{Y} \) on the sample space \( \{y_1, y_2, \ldots\} \) such that \( y_i < y_{i+1}, \forall i \geq 1 \). In particular, let an arbitrary point \( p = (p_x, p_y) \) and let \( \alpha = \Pr[p_y > y_1] \).

**Theorem 2.** The construction of the terminating subranges defining the level 2 trees can be performed in \( O(N) \) time in expectation with high probability. Each level 2 tree has \( \Theta(\log^2 N) \) points in expectation with high probability during an epoch.

The above theorem guarantees that the size of the buckets is not expected to change considerably and as a result we are allowed to assume that no update operations will happen on \( T_1 \). This is the result of assuming that the \( x \)-coordinates of the points inserted are generated by an \((\log N)^{2+\epsilon}, N^{1-\delta}) \)-smooth distribution.

The reduction of the number of violating points during an epoch is attributed to our assumption that the \( y \) coordinates follow a distribution that belongs to the \( \mathcal{Y} \) family of distributions. All violating points are stored explicitly and since there are only a few in expectation during an epoch, we can easily support the query operation. After the end of the epoch, the new structure has no violating points stored explicitly. The following theorem from [6] guarantees the small number of violating points during an epoch:

**Theorem 3.** For a sequence of \( \Theta(n) \) updates, the expected number of violations is \( O(\log n) \), assuming that \( x \) coordinates are drawn from an \((N/(\log \log N)^{1+\epsilon}, N^{1-\delta}) \)-smooth distribution, where \( \epsilon > 0 \) and \( \delta \in (0, 1) \) are constants, and the \( y \) coordinates are drawn from the restricted class of distributions \( \mathcal{Y} \) with sample space \( \{y_1, y_2, \ldots\} \), where \( y_i < y_{i+1}, \forall i \geq 1 \), such that it holds that \( \alpha \leq \left(\frac{\log N}{N}\right)^{\frac{1}{\delta}} \rightarrow e^{-1} \), where \( \alpha = \Pr[p_y > y_1] \) for an arbitrary point \( p = (p_x, p_y) \).

The theorem that describes the result attained in this paper for 3-sided dynamic skyline queries follows:

**Theorem 4.** Given a set of \( N \) 2-d points, whose \( x \) coordinates are generated by an \((N/(\log \log N)^{1+\epsilon}, N^{1-\delta}) \)-smooth distribution, where \( \epsilon > 0 \) and \( \delta \in (0, 1) \) are constants, and the \( y \) coordinates are drawn from the restricted class of distributions \( \mathcal{Y} \), we can store them in a dynamic main memory data structure that uses \( O(N) \) space and supports update operations in \( O(\log^2 N \log \log N) \) expected time with high probability. It supports skyline queries in a 3-sided range in \( O(t \log \log N) \) worst-case time, where \( t \) is the answer size.

### Declaration of competing interest

The authors declare that they have no conflict of interest.
References


