Efficient Processing of 3-Sided Range Queries with Probabilistic Guarantees

A. Kaporis
Dep. of Computer Eng. and
Informatics
University of Patras
Patras, Greece
kaporis@ceid.upatras.gr

A. N. Papadopoulos
Dep. of Informatics
Aristotle University
Thessaloniki, Greece
papadopo@csd.auth.gr

S. Sioutas
Dep. of Informatics
Ionian University
Corfu, Greece
sioutas@ionio.gr

K. Tsakalidis
BRICS, MADALGO, Dep. of
Computer Science
University of Aarhus
Aarhus, Denmark
tsakalid@madalgo.au.dk

K. Tsichlas
Dep. of Informatics
Aristotle University
Thessaloniki, Greece
tsichlas@csd.auth.gr

ABSTRACT

This work studies the problem of 2-dimensional searching for the 3-sided range query of the form $[a, b] \times (-\infty, c]$ in both main and external memory, by considering a variety of input distributions. A dynamic linear main memory solution is proposed, which answers 3-sided queries in $O(\log n + t)$ worst case time and scales with $O(\log \log n)$ expected with high probability update time, under continuous μ-random distributions of the x and y coordinates, where n is the current number of stored points and t is the size of the query output. Our expected update bound constitutes a considerable improvement over the $O(\log n)$ update time bound achieved by the classic Priority Search Tree of McCreight [23], as well as over the Fusion Priority Search Tree of Willard [30], which requires $O(\frac{\log n}{\log \log n})$ time for all operations. Moreover, we externalize this solution, gaining $O(\log_B n + t/B)$ worst case and $O(log_B log n)$ amortized expected with high probability I/Os for query and update operations respectively, where B is the disk block size. Then, combining the Modified Priority Search Tree [27] with the Priority Search Tree [23], we achieve a query time of $O(\log \log n + t)$ expected with high probability and an update time of $O(\log \log n)$ expected with high probability, under the assumption that the x-coordinates are continuously drawn from a smooth distribution and the y-coordinates are continuously drawn from a more restricted class of distributions. The total space is linear. Finally, we externalize this solution, obtaining a dynamic data struc-

ture that answers 3-sided queries in $O(\log_B \log n + t/B)$ I/Os expected with high probability, and it can be updated in $O(\log_B \log n)$ I/Os amortized expected with high probability and consumes $O(n/B)$ space, under the same assumptions.

Categories and Subject Descriptors

E.1 [**Data Structures**]: Trees; H.2.2 [**Database Management**]: Physical Design—*access methods*

General Terms

Theory, Algorithms

Keywords

3-sided range queries, probabilistic guarantees, amortized complexity

1. INTRODUCTION

Recently, a significant effort has been performed towards developing worst case efficient data structures for range searching in two dimensions [29]. In their pioneering work, Kanellakis et al. [15] illustrated that the problem of indexing in new data models (such as constraint, temporal and object models), can be reduced to special cases of two-dimensional indexing. In particular, they identified the *3-sided range searching* problem to be of major importance.

The 3-sided range query in the 2-dimensional space is defined by a region of the form $R = [a, b] \times (-\infty, c]$, i.e., an "open" rectangular region, and returns all points contained in R. Figure 1 depicts examples of possible 3-sided queries, defined by the shaded regions. Black dots represent the points comprising the result. In many applications, only positive coordinates are used and therefore, the region defining the 3-sided query always touches one of the two axes, according to application semantics.

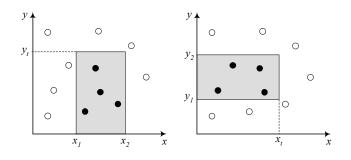Consider a time evolving database storing measurements collected from a sensor network. Assume further, that each

Figure 1: Examples of 3-sided queries.

measurement is modeled as a multi-attribute tuple of the form $<id, a_1, a_2, ..., a_d, time>$, where id is the sensor identifier that produced the measurement, d is the total number of attributes, each a_i, $1 \leq i \leq d$, denotes the value of the specific attribute and finally $time$ records the time that this measurement was produced. These values may relate to measurements regarding temperature, pressure, humidity, and so on. Therefore, each tuple is considered as a point in $\mathbb{R}^d$ space. Let $F: \mathbb{R}^d \rightarrow \mathbb{R}$ be a real-valued ranking function that scores each point based on the values of the attributes. Usually, the scoring function F is monotone and without loss of generality we assume that the lower the score the "better" the measurement (the other case is symmetric). Popular scoring functions are the aggregates sum, min, avg or other more complex combinations of the attributes. Consider the query: "search for all measurements taken between the time instances t_1 and t_2 such that the score is below s". Notice that this is essentially a 2-dimensional 3-sided query with $time$ as the x axis and $score$ as the y axis. Such a transformation from a multi-dimensional space to the 2-dimensional space is common in applications that require a temporal dimension, where each tuple is marked with a timestamp storing the arrival time [24]. This query may be expressed in SQL as follows:

SELECT $id, score, time$
FROM SENSOR_DATA
WHERE $time>=t_1$ AND $time<=t_2$ AND $score<=s$;

It is evident, that in order to support such queries, both search and update operations (i.e., insertions/deletions) must be handled efficiently. Search efficiency directly impacts query response time as well as the general system performance, whereas update efficiency guarantees that incoming data are stored and organized quickly, thus, preventing delays due to excessive resource consumption. Notice that fast updates will enable the support of stream-based query processing [4] (e.g., continuous queries), where data may arrive at high rates and therefore the underlying data structures must be very efficient regarding insertions/deletions towards supporting arrivals/expirations of data. There is a plethora of other applications (e.g., multimedia databases, spatio-temporal) that fit to a scenario similar to the previous one and they can benefit by efficient indexing schemes for 3-sided queries.

Another important issue in such data intensive applica-

tions is memory consumption. Evidently, the best practice is to keep data in main memory if this is possible. However, secondary memory solutions must also be available to cope with large data volumes. For this reason, in this work we study both cases offering efficient solutions both in the RAM and I/O computation models. In particular, the rest of the paper is organized as follows. In Section 3, we discuss preliminary concepts, define formally the classes of used probability distributions and present the data structures that constitute the building blocks of our constructions. Among them, we introduce the External Modified Priority Search Tree. In Section 4 we present the two theorems that ensure the expected running times of our constructions. The first solution is presented in Section 5, whereas our second construction is discussed in Section 6. Finally, Section 7 concludes the work and briefly discusses future research in the area.

2. RELATED WORK AND CONTRIBUTION

The usefulness of 3-sided queries has been underlined many times in the literature [7, 15]. Apart from the significance of this query in multi-dimensional data intensive applications [8, 15], 3-sided queries appear in probabilistic threshold queries in uncertain databases. Such queries are studied in a recent work of Cheng et. al. [7]. The problem has been studied both in main memory (RAM model) and secondary storage (I/O model). A basic solution for the main memory case is provided in [23] by using the Priority Search Tree. A more complicated, but more efficient solution is the Fusion Priority Search Tree of [30].

Many external data structures such as grid files, various quad-trees, z-orders and other space filling curves, k-d-B-trees, hB-trees and various R-trees have been proposed. A recent survey can be found in [12]. Often these data structures are used in applications, because they are relatively simple, require linear space and perform well in practice most of the time. However, they all have highly sub-optimal worst case (w.c.) performance, whereas their expected performance is usually not guaranteed by theoretical bounds, since they are based on heuristic rules for the construction and update operations.

Moreover, several attempts have been performed to externalize Priority Search Trees, including [5], [14], [15], [26] and [28], but all of them have not been optimal. The worst case optimal external memory solution (External Priority Search Tree) was presented in [2]. It consumes $O(n/B)$ disk blocks, performs 3-sided range queries in $O(\log_B n + t/B)$ I/Os w.c. and supports updates in $O(\log_B n)$ I/Os amortized.

In this work, we present new data structures for the RAM and the I/O model that improve by a logarithmic factor the update time in an expected sense and attempt to improve the query complexity likewise. The bounds hold with high probability (w.h.p.) under assumptions on the distributions of the input coordinates. We propose two multi-level solutions, each with a main memory and an external memory variant.

For the first solution, we assume that the x and y coordinates are being continuously drawn from an unknown μ-random distribution. It consists of two levels, for both

internal and external variants. The upper level of the first solution consists of a single Priority Search Tree [23] that indexes the structures of the lower level. These structures are Priority Search Trees as well. For the external variant we substitute the structures with their corresponding optimal external memory solutions, the External Priority Search Trees [2]. The internal variant achieves $O(\log n + t)$ w.c. query time and $O(\log \log n)$ expected w.h.p. update time, using linear space. The external solution attains $O(\log_B n + t/B)$ I/Os w.c. and $O(\log_B \log n)$ I/Os amortized expected w.h.p. respectively, and uses linear space.

By the second solution, we attempt to improve the expected query complexity and simultaneously preserve the update and space complexity. In order to do that, we restrict the x-coordinate distribution to be $(f(n), g(n))$-smooth, for appropriate functions f and g depending on the model, and the y-coordinate distribution to belong to a more restricted class of distributions. The smooth distribution is a superset of uniform and regular distributions. The restricted class contains distributions such as the Zipfian and the Power Law. The internal variant consists of two levels, of which the lower level is identical to that of the first solution. We implement the upper level with a static Modified Priority Search Tree [27]. For the external variant, in order to achieve the desired bounds, we introduce three levels. The lower level is again identical to that of the first solution, while the middle level consists of $O(B)$ size buckets. For the upper level we use an External Modified Priority Search Tree, introduced here for the first time. The latter is a straight forward externalization of the Modified Priority Search Tree and is static as well. In order to make these trees dynamic we use the technique of global rebuilding [21]. The internal version reduces the query complexity to $O(\log \log n + t)$ expected with high probability and the external to $O(\log_B \log n + t/B)$ I/Os expected with high probability.

Another more general but less efficient I/O approach was proposed in [6]. In particular, that work studied a constructive and not a probabilistic approach that achieves expected doubly logarithmic complexities for the general case where the x-coordinates are drawn from a smooth distribution and the y-coordinates are arbitrarily distributed. Its main drawbacks appear in the I/O-approach, where the block-size factor B is presented in the second logarithm ($O(\log \log_B n)$).

3. DATA STRUCTURES AND PROBABILITY DISTRIBUTIONS

For the main memory solutions we consider 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. The universe of elements is denoted by S. When we mention that a data structure performs an operation in an *amortized expected with high probability* complexity, we mean the bound is expected to be true with high probability, under a worst case sequence of insertions and deletions of points.

For the external memory solutions we consider the I/O model of computation [29]. That means that the input resides in the external memory in a blocked fashion. Whenever a computation needs to be performed to an element, the block of size B that contains that element is transferred into main memory, which can hold at most M elements. Every computation that is performed in main memory is free, since the block transfer is orders of magnitude more time consuming. Unneeded blocks that reside in the main memory are evicted by a LRU replacement algorithm. Naturally, the number of block transfers (*I/O operation*) consists the metric of the I/O model.

Furthermore, we will consider that the points to be inserted are continuously drawn by specific distributions, presented in the sequel. The term *continuously* implies that the distribution from which we draw the points remains unchanged. Since the solutions are dynamic, the asymptotic bounds are given with respect to the current size of the data structure. Finally, deletions of the elements of the data structures are assumed to be uniformly random. That is, every element present in the data structure is equally likely to be deleted [20].

3.1 Probability Distributions

In this section, we overview the probabilistic distributions that will be used in the remainder of the paper. We will consider that the x and y-coordinates are distinct elements of these distributions and will choose the appropriate distribution according to the assumptions of our constructions.

A probability distribution is *μ-random* if the elements are drawn randomly with respect to a density function denoted by μ. For this paper, we assume that μ is unknown.

Informally, a distribution defined over an interval I is *smooth* if the probability density over any subinterval of I does not exceed a specific bound, however small this subinterval is (i.e., the distribution does not contain sharp peaks). Given two functions f_1 and f_2, a density function $\mu = \mu[a, b](x)$ is (f_1, f_2)-*smooth* [22, 1] if there exists a constant β, such that for all c_1, c_2, c_3, $a \leq c_1 < c_2 < c_3 \leq b$, and all integers n, it holds that:

$$\int_{c_2 - \frac{c_3 - c_1}{f_1(n)}}^{c_2} \mu[c_1, c_3](x)dx \leq \frac{\beta \cdot f_2(n)}{n}$$

where $\mu[c_1, c_3](x) = 0$ for $x < c_1$ or $x > c_3$, and $\mu[c_1, c_3](x) = \mu(x)/p$ for $c_1 \leq x \leq c_3$ where $p = \int_{c_1}^{c_3} \mu(x)dx$. Intuitively, function f_1 partitions an arbitrary subinterval $[c_1, c_3] \subseteq [a, b]$ into f_1 equal parts, each of length $\frac{c_3 - c_1}{f_1} = O(\frac{1}{f_1})$; that is, f_1 measures how fine is the partitioning of an arbitrary subinterval. Function f_2 guarantees that no part, of the f_1 possible, gets more probability mass than $\frac{\beta \cdot f_2}{n}$; that is, f_2 measures the sparseness of any subinterval $[c_2 - \frac{c_3 - c_1}{f_1}, c_2] \subseteq [c_1, c_3]$. The class of (f_1, f_2)-smooth distributions (for appropriate choices of f_1 and f_2) is a superset of both regular and uniform classes of distributions, as well as of several non-uniform classes [1, 17]. Actually, *any* probability distribution is $(f_1, \Theta(n))$-smooth, for a suitable choice of β.

The *grid distribution* assumes that the elements are integers that belong to a specific range $[1, M]$.

We define the *restricted class* of distributions as the class that contains distributions used in practice, such as the Zipfian, Power Law, e.t.c..

The *Zipfian* distribution is a distribution of probabilities

of occurrence that follows Zipf's law. Let N be the number of elements, k be their rank and s be the value of the exponent characterizing the distribution. Then Zipf's law is defined as the function $f(k; s, N) = \frac{1/k^s}{\Sigma_{k=1}^{N} 1/n^s}$. Intuitively, few elements occur very often, while many elements occur rarely.

The *Power Law* distribution is a distribution over probabilities that satisfy $Pr[X \geq x] = cx^{-b}$ for constants $c, b > 0$.

3.2 Data Structures

In this section, we describe the data structures that we will combine in order to achieve the desired complexities.

3.2.1 Priority Search Trees

The classic *Priority Search Tree (PST)* [23] stores points in the 2-d space. One of the most important operations that the PST supports is the *3-sided query*. The 3-sided query consists of a half bounded rectangle $[a, b] \times (-\infty, c]$ and asks for all points that lie inside this area. Note that by rotation we can unbound any edge of the rectangle. The PST supports this operation in $O(\log n + t)$ w.c., where n is the number of points and t is the number of the reported points.

The PST is a combination of a search tree and a priority queue. The search tree (an (a, b)-tree suffices) allows the efficient support of searches, insertions and deletions with respect to the x-coordinate, while the priority queue allows for easy traversal of points with respect to their y-coordinate. In particular, the leaves of the PST are the points sorted by x-coordinate. In the internal nodes of the tree there are artificial values which are used for the efficient searching of points with respect to their x-coordinate. In addition, each internal node stores a point that has the minimum y-coordinate among all points stored in its subtree. This corresponds to a tournament on the leaves of the PST. For example, the root of the PST contains a point which has minimum y-coordinate among all points in the plane, as well as a value which is in the interval defined between the x-coordinates of the points stored in the rightmost leaf of the left subtree and the leftmost leaf of the right subtree (this is true in the case of a binary tree). A PST implemented with an red-black tree supports the operations of insertion of a new point, deletion of an existing point and searching for the x-coordinate of a point in $O(\log n)$ worst case time.

Regarding the I/O model, after several attempts, a worst case optimal solution was presented by Arge et al. in [2]. The proposed indexing scheme consumes $O(n/B)$ space, supports updates in $O(\log_B n)$ amortized I/Os and answers 3-sided range queries in $O(\log_B n + t/B)$ I/Os. We will refer to this indexing scheme as the *External Priority Search Tree (EPST)*.

3.2.2 Interpolation Search Trees

In [18], a dynamic data structure based on interpolation search (IS-Tree) was presented, which consumes linear space and can be updated in $O(1)$ time w.c. Furthermore, the elements can be searched in $O(\log \log n)$ time expected w.h.p., given that they are drawn from a (n^α, n^β)-smooth distribution, for any arbitrary constants $0 < \alpha, \beta < 1$.

The externalization of this data structure, called interpolation search B-tree (ISB-tree), was introduced in [16]. It supports update operations in $O(1)$ worst-case I/Os provided that the update position is given and search operations in $O(\log_B \log n)$ I/Os expected w.h.p. The expected search bound holds w.h.p. if the elements are drawn by a $(n/(\log \log n)^{1+\epsilon}, n^{1-\delta})$-smooth distribution, where $\epsilon > 0$ and $\delta = 1 - \frac{1}{B}$ are constants. The worst case search bound is $O(\log_B n)$ block transfers.

3.2.3 Modified Priority Search Trees

A *Modified Priority Search Tree (MPST)* is a static data structure that stores points on the plane and supports 3-sided queries. It is stored as an array (Arr) in memory, yet it can be visualized as a complete binary tree. Although it has been presented in [19], we sketch it here again, in order to introduce its external version.

Let T be a Modified Priority Search Tree (MPST) [27] which stores n points of S. We denote by T_v the subtree of T with root v. Let u be a leaf of the tree. Let P_u be the root-to-leaf path for u. For every u, we sort the points in P_u by their y-coordinate. We denote by P_u^j the subpath of P_u with nodes of depth bigger or equal to j (The depth of the root is 0). Similarly L_u^j (respectively R_u^j) denotes the set of nodes that are left (resp. right) children of nodes of P_u^j and do not belong to P_u^j . The tree structure T has the following properties:

- Each point of S is stored in a leaf of T and the points are in sorted x-order from left to right.

- Each internal node v is equipped with a secondary list $S(v)$. $S(v)$ contains in the points stored in the leaves of T_v in increasing y-coordinate.

- A leaf u also stores the following lists $A(u)$, $P^j(u)$, $L^j(u)$ and $R^j(u)$, for $0 \leq j \leq \log n$. The list $P^j(u)$, $L^j(u)$ and $R^j(u)$ store, in increasing y-coordinate, pointers to the respective internal nodes. $A(u)$ is an array that indexes j.

Note that the first element of the list $S(v)$ is the point of the subtree T_v with minimum y-coordinate. Also note that $0 \leq j \leq \log n$, so there are $\log n$ such sets P_u^j, L_u^j, R_u^j for each leaf u. Thus the size of A is $\log n$ and for a given j, any list $P^j(u)$, $L^j(u)$ or $R^j(u)$ can be accessed in constant time. By storing the nodes of the tree T according to their inorder traversal in an array Arr of size $O(n)$, we can imply the structure of tree T. Also each element of Arr contains a binary label that corresponds to the inorder position of the respective node of T, in order to facilitate constant time lowest common ancestor (LCA) queries.

To answer a query with the range $[a, b] \times (-\infty, c]$ we find the two leaves u, w of Arr that contain a and b respectively. If we assume that the leaves that contain a, b are given, we can access them in constant time. Then, since Arr contains an appropriate binary label, we use a simple LCA (Lowest Common Ancestor) algorithm [11, 13] to compute the depth j of the nearest common ancestor of u, w in $O(1)$ time. That is done by performing the XOR operation between the binary labels of the leaves u and w and finding the position of the first set bit provided that the left-most bit is placed in
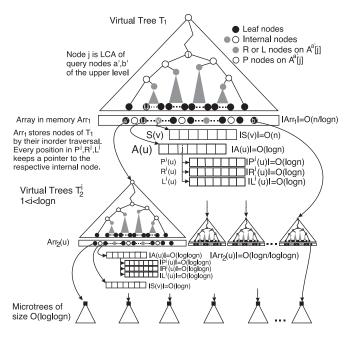
Figure 2: The linear space MPST.

position 0. Afterwards, we traverse $P^j(u)$ until the scanned y-coordinate is not bigger than c. Next, we traverse $R^j(u)$, $L^j(w)$ in order to find the nodes whose stored points have y-coordinate not bigger than c. For each such node v we traverse the list $S(v)$ in order to report the points of Arr that satisfy the query. Since we only access points that lie in the query, the total query time is $O(t)$, where t is the answer size.

The total size of the lists $S(u)$ for each level of T is $O(n)$. Each of the $O(n)$ leaves stores $\log n$ lists P_j, L_j and R_j, each of which consumes $O(\log n)$ space. Thus the space for these lists becomes $O(n \log^2 n)$. By implementing these lists as partially persistent sorted lists [3], their total space becomes $O(n \log n)$, resulting in a total space of $O(n \log n)$ for these lists. Thus, the total space occupied by T is $O(n \log n)$.

We can reduce the space of the structure by pruning as in [9, 25]. However, pruning alone does not reduce the space to linear. We can get better but not optimal results by applying pruning recursively. To get an optimal space bound we will use a combination of pruning and table lookup. The pruning method is as follows: Consider the nodes of T, which have height $\log \log n$. These nodes are roots of subtrees of T of size $O(\log n)$ and there are $O(n/\log n)$ such nodes. Let T_1 be the tree whose leaves are these nodes and let T_2^i be the subtrees of these nodes for $1 \le i \le O(n/\log n)$. We call T_1 the first layer of the structure and the subtrees T_2^i the second layer. T_1 and each subtree T_2^i is by itself a Modified Priority Search Tree. Note that T_1 has size $O(n/\log n) = O(n)$. Each subtree T_2^i has $O(\log n/\log \log n)$ leaves and depth $O(\log \log n)$. The space for the second layer is $O(n \log n)$. By applying the pruning method to all the trees of the second layer we get a third layer which consists of $O(n/\log \log n)$ modified priority search trees each of size $O(\log \log n)$. Ignoring the third layer, the second layer needs now linear space, while the

$O(n \log n)$ space bottleneck is charged on the third level. If we use table lookup [10] to implement the modified priority search trees of the third layer we can reduce its space to linear, thus consuming linear space in total.

In order to answer a query on the three layered structure we access the microtrees that contain a and b and extract in $O(1)$ time the part of the answer that is contained in them. Then we locate the subtrees T_2^i, T_2^j that contain the representative leaves of the accessed microtrees and extract the part of the answer that is contained in them by executing the query algorithm of the MPST. The roots of these subtrees are leaves of T_1. Thus we execute again the MPST query algorithm on T_1 with these leaves as arguments. Once we reach the node with y-coordinate bigger than c, we continue in the same manner top down. This may lead us to subtrees of the second layer that contain part of the answer and have not been accessed yet. That means that for each accessed tree of the second layer, we execute the MPST query algorithm, where instead of a and b, we set as arguments the minimum and the maximum x-coordinates of all the points stored in the queried tree. The argument c remains, of course, unchanged. Correspondingly, in that way we access the microtrees of the third layer that contain part of the answer. We execute the top down part of the algorithm on them, in order to report the final part of the answer.

LEMMA 3.1. *Given a set of n points on the plane we can store them in a static data structure with $O(n)$ space that allows three-sided range queries to be answered in $O(t)$ worst case, where t is the answer size.*

PROOF. See [27]. $\square$

The *External Modified Priority Search Tree (EMPST)* is similar to the MPST, yet we store the lists in a blocked fashion. In order to attain linear space in external memory we prune the structure k times, instead of two times. The pruning terminates when $\log^{(k)} n = O(B)$. Since computation within a block is free, we do not need the additional layer of microtrees. By that way we achieve $O(n/B)$ space.

Assume that the query algorithm accesses first the two leaves u and v of the k-th layer of the EMPST, which contain a and b respectively. If they belong to different EMPSTs of that layer, we recursively take the roots of these EMPSTs until the roots r_u and r_v belong to the same EMPST, w.l.o.g. the one on the upper layer. That is done in $O(k) = O(1)$ I/Os. Then, in O(1) I/Os we access the j-th entry of $A(r_u)$ and $A(r_v)$, where j is the depth of $LCA(r_u, r_v)$, thus also the corresponding sublists $P^j(r_u), R^j(r_u), L^j(r_u)$ and $P^j(r_v), R^j(r_v), L^j(r_v)$. Since these sublists are y-ordered, by scanning them in t_1/B I/Os we get all the t_1 pointers to the S-lists that contain part of the answer. We access the S-lists in t_1 I/Os and scan them as well in order to extract the part of the answer (let's say t_2) they contain. We then recursively access the t_2 S-lists of the layer below and extract the part t_3 that resides on them. In total, we consume $t_1/B + t_1 \cdot t_2/B + ... + t_{i-1} \cdot t_i/B + ... + t_{k-1} \cdot t_k/B$ I/Os. Let p_i the probability that $t_i = t^{p_i}$ where t is the total size of the answer and $\sum_{i=1}^{k} p_i = 1$. Thus, we need $t^{p_1}/B + \sum_{i=1}^{k-1} \frac{t^{p_i}}{B} \cdot t^{p_{i+1}}$ I/Os or $t^{p_1}/B + \sum_{i=1}^{k-1} \frac{t^{(p_i+p_{i+1})}}{B}$ I/Os. Assuming w.h.p. an

equally likely distribution of answer amongst the k layers, we need $t^{\frac{1}{k}}/B + \sum_{i=1}^{k-1} \frac{t^{\frac{1}{k}+\frac{1}{k}}}{B}$ expected number of I/Os or $t^{\frac{1}{k}}/B + \sum_{i=1}^{k-1} \frac{t^{\frac{2}{k}}}{B}$. Since $k >> 2$, we need totally $O(t/B)$ expected w.h.p. number of I/Os.

LEMMA 3.2. *Given a set of n points on the plane we can store them in a static data structure with $O(n/B)$ space that allows three-sided range queries to be answered in $O(t/B)$ expected w.h.p. case, where t is the size of the answer.*

4. EXPECTED FIRST ORDER STATISTIC OF UKNOWN DISTRIBUTIONS

In this section, we prove two theorems that will ensure the expected running times of our constructions. They are multilevel data structures, where for each pair of levels, the upper level indexes representative elements (in our case, point on the plane) of the lower level buckets. We call an element *violating* when its insertion to or deletion from the lower level bucket causes the representative of that bucket to change, thus triggering an update on the upper level. We prove that for an epoch of $O(\log n)$ updates, the number of violating elements is $O(1)$ if they are continuously being drawn from a μ-random distribution. Secondly, we prove that for a broader epoch of $O(n)$ updates, the number of violating elements is $O(\log n)$, given that the elements are being continuously drawn from a distribution that belongs to the restricted class. Violations are with respect to the y-coordinates, while the distribution of elements in the buckets are with respect to x-coordinates.

But first, the proof of an auxiliary lemma is necessary. Assume a sequence $\mathcal{S}$ of distinct numbers generated by a continuous distribution $\mu = \mathcal{F}$ over a universe $\mathcal{U}$. Let $|\mathcal{S}|$ denote the size of $\mathcal{S}$. Then, the following holds:

LEMMA 4.1. *The probability that the next element q drawn from $\mathcal{F}$ is less than the minimum element s in $\mathcal{S}$ is equal to $\frac{1}{|\mathcal{S}|+1}$.*

PROOF. Suppose that we have n random observations $X_1,\dots,X_n$ from an unknown continuous probability density function $f(X)$, with cumulative distribution $\mu = F(X)$, $X \in [a,b]$. We want to compute the probability that the $(n+1)-th$ observation is less than $min\{X_1,\dots,X_n\}$. Let $X_{(1)} = min\{X_1,\dots,X_n\}$. Therefore, $P\{X_{n+1} < X_{(1)}\} = \sum_x P\{X_{n+1} < X_{(1)}/X_{(1)} = x\} \cdot P\{X_{(1)} = x\}\ (\alpha)$.

It is easy to see that $P\{X_{n+1} < X_{(1)}/X_{(1)} = x\} = F(X)$ $= P\{X_{n+1} < x\}\ (\beta)$. Also $P\{X_{(k)} = x\} = n \cdot f(x) \cdot \binom{n-1}{k-1} \cdot F(X)^{k-1} \cdot (1 - F(X))^{n-k}\ (\gamma)$, where $X_{(k)}$ is the $k-th$ smallest value in $\{X_1,\dots,X_n\}$.

In our case $k = 1$, which intuitively means that we have n choices for one in $\{X_1,\dots,X_n\}$ being the smallest value. This is true if all the rest $n-1$ are more than x, which occurs with probability: $(1 - F(X))^{n-1} = (1 - P\{X < x\})^{n-1}$. By (β) and (γ), expression (α) becomes: $P\{X_{n+1} < X_{(1)}\} = \int_b^a n \cdot f(X) \binom{n-1}{k-1} \cdot F(X) \cdot (1 - F(X))^{n-1}$ dX. After some mathematical manipulations, we have that: $P\{X_{n+1} < X_{(1)}\} = \int_b^a n \cdot f(X) \cdot (1 - F(X))^{n-1} \cdot F(X) dX =$ $\int_b^a [-(1 - F(X))^n]' F(X)\,dX = \int_b^a [-(1 - F(X))^n \cdot F(X)]'$ $dX + \int_b^a (1 - F(X))^n \cdot F'(X) dX = \{-(1 - F(X))^n \cdot F(x)|_a^b\}$

$+ \int_b^a - \left[\frac{(1-F(X))^{n+1}}{n+1}\right]' dX = -(1 - F(b))^n \cdot F(b) + (1 - F(a))^n \cdot$ $F(a) - \left\{\frac{(1-F(X))^{n+1}}{n+1}|_a^b\right\} = -\left\{\frac{(1-F(b))^{n+1}}{n+1} - \frac{(1-F(a))^{n+1}}{n+1}\right\} =$ $\frac{1}{n+1}$ $\square$

Apparently, the same holds if we want to maintain the maximum element of the set $\mathcal{S}$.

PROPOSITION 4.2. *Suppose that the input elements have their x-coordinate generated by an arbitrary continuous distribution μ on $[a, b] \subseteq \Re$. Let n be the elements stored in the data structure at the latest reconstruction. An epoch starts with $\log n$ updates. During the i-th update let $N(i) \in [n, r \cdot n]$, with constant $r > 1$, denote the number of elements currently stored into the $\frac{n}{\log n}$ buckets that partition $[a, b] \subseteq \Re$. Then the $N(i)$ elements remain μ randomly distributed in the buckets per i-th update.*

PROOF. The proof is analogous to [17, Lem. 2] and is omitted. $\square$

THEOREM 4.3. *For a sequence of $O(\log n)$ updates, the expected number of violating elements is $O(1)$, assuming that the elements are being continuously drawn from a μ-random distribution.*

PROOF. According to Prop. 4.2, there are $N(i) \in [n, r \cdot n]$ (with constant $r > 1$) elements with their x-coordinates μ-randomly distributed in the buckets $j = 1, \dots, \frac{n}{\log n}$, that partition $[a, b] \subseteq \Re$. By [17, Th. 4], with high probability, each bucket j receives an x-coordinate with probability $p_j = \Theta(\frac{\log n}{n})$. It follows that during the i-th update operation, the elements in bucket j is a Binomial random variable with mean $p_j \cdot N(i) = \Theta(\log n)$.

The elements with x-coordinates in an arbitrary bucket j are $\alpha N(i)$ with probability $\binom{N(i)}{\alpha N(i)} p_j^{\alpha N(i)} (1 - p_j)^{(1-\alpha)N(i)} \sim$ $\left[\left(\frac{p_j}{\alpha}\right)^\alpha \left(\frac{1-p_j}{1-\alpha}\right)^{1-\alpha}\right]^{N(i)}$. In turn, these are $\leq \alpha N(i) = \frac{p_j}{2} N(i)$ (less than half of the bucket's mean) with probability

$$\leq \frac{p_j N(i)}{2} \cdot \left[\left(\frac{p_j}{\alpha}\right)^\alpha \left(\frac{1-p_j}{1-\alpha}\right)^{1-\alpha}\right]^{N(i)} \to 0 \qquad (1)$$

as $n \to \infty$ and $\alpha = \frac{p_j}{2}$.

Suppose that an element is inserted in the i-th update. It induces a violation if its y-coordinate is strictly the minimum element of the bucket j it falls into.

- If the bucket contains $\geq \frac{p_j}{2} \log N(i) \geq \frac{p_j}{2} \log n$ coordinates then by Lemma 4.1 element y incurs a violation with probability $O(\frac{1}{\log n})$.

- If the bucket contains $< \frac{p_j}{2} \log N(i)$ coordinates, which is as likely as in Eq. (1), then element y may induce ≤ 1 violation.

Putting these cases together, element y expectedly induces at most $O(\frac{1}{\log n})$+Eq. (1)= $O(\frac{1}{\log n})$ violations. We conclude that during the whole epoch of $\log n$ insertions the expected number of violations are at most $\log n \cdot O(\frac{1}{\log n})$ plus $\log n \cdot$ Eq. (1) which is $O(1)$. $\square$

THEOREM 4.4. *For a sequence of $O(n)$ updates, the expected number of violating elements is $O(\log n)$, assuming that $x-$ coordinates are drawn from a continuous smooth distribution and the $y-$ coordinates are drawn from the restricted class of distributions (power-law or zipfian).*

PROOF. Suppose an element is inserted, with its y-coordinate following a discrete distribution (while its x-coordinate is arbitrarily distributed) in the universe $\{y_1, y_2, \ldots\}$ with $y_i < y_{i+1}, \forall i \geq 1$. Also, let $q = \Pr[y > y_1]$ and y_j^* the min y-coordinate of the elements in bucket j as soon as the current epoch starts. Clearly, the element just inserted incurs a violation when landing into bucket j with probability $\Pr[y < y_j^*]$.

- If the bucket contains $\geq \frac{p_j}{2} \log N(i) \geq \frac{p_j}{2} \log n$ coordinates, then coordinate y incurs a violation with probability $\leq q^{\frac{p_j}{2} \log n}$. (In other words, a violation may happens when at most all the $\Omega(\log n)$ coordinates of the elements in bucket j are $> y_1$, that is, when $y_j^* > y_1$.)

- If the bucket contains $< \frac{p_j}{2} \log N(i)$ coordinates, which is as likely as in Eq. (1) then coordinate y may induces ≤ 1 violation.

All in all, y coordinate expectedly induces $\leq q^{\Omega(\log n)} +$ Eq. (1) violations. Thus, during the whole epoch of n insertions the expected number of violations are at most $n \cdot \left(q^{\Omega(\log n)} \right) + n \cdot$ Eq. (1) $= nq^{\Omega(\log n)} + o(1)$ violations. This is at most $c \cdot \log n = O(\log n)$ if $q \leq \left(\frac{c \log n}{n} \right)^{(\log n)^{-1}} \to e^{-1}$ as $n \to \infty$. $\square$

REMARK 4.5. *Note that* Power Law *and* Zipfian *distributions have the aforementioned property that* $q \leq \left(\frac{c \log n}{n} \right)^{(\log n)^{-1}}$ $\to e^{-1}$ *as* $n \to \infty$.

5. THE SOLUTION FOR RANDOM DISTRIBUTIONS

In this section, we present the construction that works under the assumptions that the x and y-coordinates are continuously drawn by an unknown μ-random distribution.

The structure we propose consists of two levels, as well as an auxiliary data structure. All of them are implemented as PSTs. The lower level partitions the points into *buckets* of almost equal logarithmic size according to the x-coordinate of the points. That is, the points are sorted in increasing order according to x-coordinate and then divided into sets of $O(\log n)$ elements each of which constitutes a bucket. A bucket C is implemented as a PST and is represented by a point C^{min} which has the smallest y-coordinate among all points in it. This means that for each bucket the cost for insertion, deletion and search is equal to $O(\log \log n)$, since this is the height of the PST representing C.

The upper level is a PST on the representatives of the lower level. Thus, the number of leaves in the upper level is $O\left(\frac{n}{\log n}\right)$. As a result, the upper level supports the operations of insert, delete and search in $O(\log n)$ time. In addition, we keep an extra PST for insertions of violating points. Under this context, we call a point p *violating*, when

its y-coordinate is less than C^{min} of the bucket C in which it should be inserted. In the case of a violating point we must change the representative of C and as a result we should make an update operation on the PST of the upper level, which costs too much, namely $O(\log n)$.

We assume that the x and y-coordinates are drawn from an unknown μ-random distribution and that the μ function never changes. Under this assumption, according to the combinatorial game of bins and balls, presented in Section 5 of [17], the size of every bucket is $O(\log^c n)$, where $c > 0$ is a constant, and no bucket becomes empty w.h.p. We consider epochs of size $O(\log n)$, with respect to update operations. During an epoch, according to Theorem 4.3, the number of violating points is expected to be $O(1)$ w.h.p. The extra PST stores exactly those $O(1)$ violating points. When a new epoch starts, we take all points from the extra PST and insert them in the respective buckets in time $O(\log \log n)$ expected w.h.p. Then we need to incrementally update the PST of the upper level. This is done during the new epoch that just started. In this way, we keep the PST of the upper level updated and the size of the extra PST constant. As a result, the update operations are carried out in $O(\log \log n)$ time expected w.h.p., since the update of the upper level costs $O(1)$ time w.c.

The 3-sided query can be carried out in the standard way. Assume the query $[a, b] \times (-\infty, c]$. First we search down the PST of the upper level for a and b. Let P_a be the search path for a and P_b for b respectively. Let $P_m = P_a \cap P_b$. Then, we check whether the points in the nodes on $P_a \cup P_b$ belong to the answer by checking their x-coordinate as well as their y-coordinate. Then, we check all right children of $P_a - P_m$ as well as all left children of $P_b - P_m$. In this case we just check their y-coordinate since we know that their x-coordinate belongs in $[a, b]$. When a point belongs in the query, we also check its two children and we do this recursively. After finishing with the upper level we go to the respective buckets by following a single pointer from the nodes of the upper level PST of which the points belong in the answer. Then we traverse in the same way the buckets and find the set of points to report. Finally, we check the extra PST for reported points. In total the query time is $O(\log n + t)$ w.c.

Note that deletions of points do not affect the correctness of the query algorithm. If a non violating point is deleted, it should reside on the lower level and thus it would be deleted online. Otherwise, the auxiliary PST contains it and thus the deletion is online again. No deleted violating point is incorporated into the upper level, since by the end of the epoch the PST contains only inserted violating points.

THEOREM 5.1. *There exists a dynamic main memory data structure that supports 3-sided queries in $O(\log n + t)$ w.c. time, can be updated in $O(\log \log n)$ expected w.h.p. and consumes linear space, under the assumption that the x and y-coordinates are continuously drawn from a μ-random distribution.*

If we implement the above solution by using EPSTs [2], instead of PSTs, then the solution becomes I/O-efficient, however the update cost is amortized instead of worst case. Thus we get that:
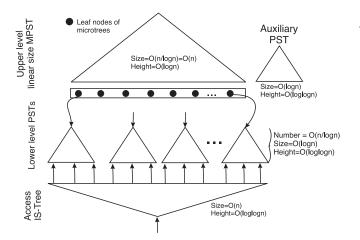
Figure 3: The internal memory construction for the restricted distributions

THEOREM 5.2. *There exists a dynamic external memory data structure that supports 3-sided queries in $O(\log_B n + t/B)$ w.c. time, can be updated in $O(\log_B \log n)$ amortized expected w.h.p. and consumes linear space, under the assumption that the x and y-coordinates are continuously drawn from a μ-random distribution.*

6. THE SOLUTION FOR THE SMOOTH AND THE RESTRICTED DISTRIBUTIONS

We would like to improve the query time and simultaneously preserve the update time. For this purpose we will incorporate to the structure the MPST, which is a static data structure. We will dynamize it by using the technique of global rebuilding [21], which unfortunately costs $O(n)$ time.

In order to retain the update time in the same sublogarithmic levels, we must ensure that at most a logarithmic number of lower level structures will be violated in a broader epoch of $O(n)$ updates. Since the violations concern the y-coordinate we will restrict their distribution to the more restricted class, since Theorem 4.4 ensures exactly this property. Thus, the auxiliary PST consumes at most $O(\log n)$ space during an epoch.

Moreover, we must waive the previous assumption on the x-coordinate distribution, as well. Since the query time of the previous solution was $O(\log n)$ we could afford to pay as much time in order to locate the leaves containing a and b. In this case, though, this blows up our complexity. If, however, we assume that the x-coordinates are drawn from a (n^α, n^β)-smooth distribution, we can use an IS-tree to index them, given that $0 < \alpha, \beta < 1$. By doing that, we pay w.h.p. $O(\log \log n)$ time to locate a and b.

When a new epoch starts we take all points from the extra PST and insert them in the respective buckets in time $O(\log \log n)$ w.h.p. During the epoch we gather all the violating points that should access the MPST and the points that belong to it and build in parallel a new MPST on them. At the end of the $O(n)$ epoch, we have built the updated version of the MPST, which we use for the next epoch that

just started. By this way, we keep the MPST of the upper level updated and the size of the extra PST logarithmic. By incrementally constructing the new MPST we spend $O(1)$ time worst case for each update of the epoch. As a result, the update operation is carried out in $O(\log \log n)$ time expected with high probability.

For the 3-sided query $[a, b] \times (-\infty, c]$, we first access the leaves of the lower level that contain a and b, through the IS-tree. This costs $O(\log \log n)$ time w.h.p. Then the query proceeds bottom up in the standard way. First it traverses the buckets that contain a and b and then it accesses the MPST from the leaves of the buckets' representatives. Once the query reaches the node of the MPST with y-coordinate bigger than c, it continues top down to the respective buckets, which contain part of the answer, by following a single pointer from the nodes of the upper level MPST. Then we traverse top down these buckets and complete the set of points to report. Finally, we check the auxiliary PST for reported points. The traversal of the MPST is charged on the size of the answer $O(t)$ and the traversal of the lower level costs $O(\log \log n)$ expected with high probability. Due to Theorem 4.4, the size of the auxiliary PST is with high probability $O(\log n)$, thus the query spends $O(\log \log n)$ expected with high probability for it. Hence, in total the query time is $O(\log \log n + t)$.

THEOREM 6.1. *There exists a dynamic main memory data structure that supports 3-sided queries in $O(\log \log n + t)$ time expected w.h.p., can be updated in $O(\log \log n)$ expected w.h.p. and consumes linear space, under the assumption that the x-coordinates are continuously drawn from a μ-random distribution and the y-coordinates are drawn from the restricted class of distributions.*

In order to extend the above structure to work in external memory we will follow a similar scheme with the above structure. We use an auxiliary EPST and index the leaves of the main structure with and ISB-tree. This imposes that the x-coordinates are drawn from a $(n/(\log \log n)^{1+\epsilon}, n^{1-\delta})$-smooth distribution, where $\epsilon > 0$ and $\delta = 1 - \frac{1}{B}$, otherwise the search bound would not be expected to be doubly logarithmic. Moreover, the main structure consists of three levels, instead of two. That is, we divide the n elements into $n' = \frac{n}{\log n}$ buckets of size $\log n$, which we implement as EP-STs (instead of PSTs). This will constitute the lower level of the whole structure. The n' representatives of the EPSTs are again divided into buckets of size $O(B)$, which constitute the middle level. The $n'' = \frac{n'}{B}$ representatives are stored in the leaves of an external MPST (EMPST), which constitutes the upper level of the whole structure. In total, the space of the aforementioned structures is $O(n' + n'' + n'' \log^{(k)} n'') = O(\frac{n}{\log n} + \frac{n}{B \log n} + \frac{n}{B \log n} B) = O(\frac{n}{\log n}) = O(\frac{n}{B})$, where k is such that $\log^{(k)} n'' = O(B)$ holds.

The update algorithm is similar to the variant of internal memory. The query algorithm first proceeds bottom up. We locate the appropriate structures of the lower level in $O(\log_B \log n)$ I/Os w.h.p., due to the assumption on the x-coordinates. The details for this procedure in the I/O model can be found in [16]. Note that if we assume that the x-coordinates are drawn from the *grid distribution* with parameters $[1, M]$, then this access step can be
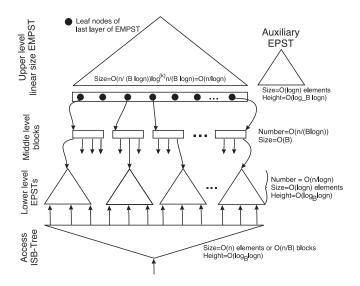
Figure 4: The external memory construction for the restricted distributions

realized in $O(1)$ I/Os. That is done by using an array A of size M as the access data structure. The position $A[i]$ keeps a pointer to the leaf with x-coordinate not bigger than i [27]. Then, by executing the query algorithm, we locate the at most two structures of the middle level that contain the representative leaves of the EPSTs we have accessed. Similarly we find the representatives of the middle level structures in the EMPST. Once we reached the node whose minimum y-coordinate is bigger than c, the algorithm continues top down. It traverses the EMPST and accesses the structures of the middle and the lower level that contain parts of the answer. The query time spent on the EMPST is $O(t/B)$ I/Os. All accessed middle level structures cost $O(2 + t/B)$ I/Os. The access on the lower level costs $O(\log_B \log n + t/B)$ I/Os. Hence, the total query time becomes $O(\log_B \log n + t/B)$ I/Os expected with high probability. We get that:

THEOREM 6.2. *There exists a dynamic external memory data structure that supports 3-sided queries in $O(\log_B \log n + t/B)$ expected w.h.p., can be updated in $O(\log_B \log n)$ expected w.h.p. and consumes $O(n/B)$ space, under the assumption that the x-coordinates are continuously drawn from a smooth-distribution and the y-coordinates are drawn from the restricted class of distributions.*

7. CONCLUSIONS

We considered the problem of answering three sided range queries of the form $[a, b] \times (-\infty, c]$ under sequences of inserts and deletes of points, trying to attain linear space and doubly logarithmic expected w.h.p. operation complexities, under assumptions on the input distributions. We proposed two solutions, which we modified appropriately in order to work for the RAM and the I/O model. Both of them consist of combinations of known data structures that support the 3-sided query operation.

The internal variant of the first solution combines Priority Search Trees [23] and achieves $O(\log \log n)$ expected

w.h.p. update time and $O(\log n + t)$ w.c. query time, using linear space. Analogously, the external variant of the first solution combines External Priority Search Trees [2] and achieves the update operation in $O(\log_B \log n)$ I/Os expected w.h.p. and the query operation in $O(\log_B n + t/B)$ I/Os amortized expected w.h.p., using linear space. The bounds are true under the assumption that the x and y-coordinates are drawn continuously from μ-random distributions.

In order to improve exponentially on the query complexity, we proposed a second solution with stronger assumptions on the coordinate distributions. We restricted the y-coordinates to be continuously drawn from a restricted distribution and the x- coordinates to be drawn from $(f(n), g(n))$-smooth distributions, for appropriate functions f and g, depending on the model. The internal variant of this solution can be accessed by a IS-tree [18], incorporates the Modified Priority Search Tree [19] and decreases the query complexity to $O(\log \log n + t)$ expected w.h.p., preserving the update and space complexity. The external variant combines the External Modified Priority Search Tree, which was presented here, with External Priority Search Trees [2] and is accessed by an ISB-tree [16]. The update time is $O(\log_B \log n)$ I/Os expected w.h.p., the query time is $O(\log_B \log n + t/B)$ I/Os and the space is linear.

The proposed solutions are practically implementable. Thus, we leave as a future work an experimental performance evaluation, in order to prove in practice the improved query performance and scalability of the proposed methods.

8. REFERENCES

[1] A. Andersson and C. Mattson. "Dynamic Interpolation Search in $o(\log \log n)$ Time", *Proceedings of ICALP*, pp.15-27, 1993.

[2] L. Arge, V. Samoladas, J.S. Vitter, "On Two-Dimensional Indexability and Optimal Range Search Indexing", *Proceedings of PODS*, pp. 346-357, 1999.

[3] B. Becker, S. Gschwind, T. Ohler, B. Seeger and P. Widmayer, "An asymptotically optimal multiversion B-tree", *The VLDB Journal*, pp.264-275, 1996.

[4] Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: "Models and Issues in Data Stream Systems", *Proceedings of PODS*, pp.1-16, 2002.

[5] G. Blankenagel and R.H. Gueting, "XP-trees-External priority search trees", Technical report, FernUniversitB Hagen, Informatik-Bericht, Nr.92, 1990.

[6] G.S. Brodal, A.C. Kaporis, S.Sioutas, K. Tsakalidis, and K. Tsichlas, "Dynamic 3-sided Planar Range Queries with Expected Doubly Logarithmic Time", *Proceedings of ISAAC*, 2009 (to appear).

[7] R. Cheng, Y. Xia, S. Prabhakar, R. Shah, J. S. Vitter, "Efficient Indexing Methods for Probabilistic Threshold Queries over Uncertain Data", *Proceedings of VLDB*, 2004.

[8] M. deBerg, M. van Kreveld, M. Overmars, O. Schwarzkopf, "*Computational Geometry, algorithms and applications*", Springer, 1998.

[9] O. Fries, K. Mehlhorn, S. Naher and A. Tsakalidis, "A loglogn data structure for three sided range queries",

Information Processing Letters, 25, pp.269-273, 1987.

[10] H. N. Gabow and R. E. Tarjan, "A linear-time algorithm for a special case of disjoint set union", *Journal of Computer and System Sciences*, 30, pp.209-221, 1985.

[11] D. Gusfield, "*Algorithms on Strings, Trees and Sequences, Computer Science and Computational Biology*", Cambridge University Press,1994.

[12] V. Gaede and 0. Gfinther, "Multidimensional access methods", *ACM Computing Surveys*, 30(2), pp.170-231, 1998.

[13] D. Harel, R. E. Tarjan, "Fast algorithms for finding nearest common ancestor", *SIAM Journal of Computing*, 13, pp.338-355, 1984.

[14] C. Icking, R. Klein, and T. Ottmann, "Priority search trees in secondary memory", *Proceedings of Graph-Theoretic Concepts in Computer Science*, LNCS 314, pp.84-93, 1987.

[15] P.C. Kanellakis, S. Ramaswamy, D.E. Vengroff, and J.S. Vitter, "Indexing for data models with constraints and classes", *Journal of Computer and System Sciences*, 52(3), pp.589-612, 1996.

[16] A. Kaporis, C. Makris, G. Mavritakis, S. Sioutas, A. Tsakalidis, K. Tsichlas, and C. Zaroliagis, "ISB-Tree: A New Indexing Scheme with Efficient Expected Behaviour", *Proceedings of ISAAC*, pp. 318-327, 2005.

[17] A.C. Kaporis, C. Makris, S. Sioutas, A. Tsakalidis, K. Tsichlas, and C. Zaroliagis, "Improved Bounds for Finger Search on a RAM", *Proceedings of ESA*, pp. 325-336, 2003.

[18] A. Kaporis, C. Makris, S. Sioutas, A. Tsakalidis, K. Tsichlas, C. Zaroliagis, "Dynamic Interpolation Search Revisited", *Proceedings of ICALP*, pp.382-394, 2006.

[19] N. Kitsios, C. Makris, S. Sioutas, A. Tsakalidis, J. Tsaknakis, B. Vassiliadis, "2-D Spatial Indexing Scheme in Optimal Time", *Proceedings of ADBIS-DASFAA*, pp.107-116, 2000.

[20] D.E. Knuth, "Deletions that preserve randomness", *IEEE Transactions on Software Engineering*, 3, pp.351-359, 1977.

[21] C. Levcopoulos and M.H. Overmars, "Balanced Search Tree with O(1) Worst-case Update Time", *Acta Informatica*, 26, pp.269-277, 1988.

[22] K. Mehlhorn and A. Tsakalidis, "Dynamic Interpolation Search", *Journal of the ACM*, 40(3), pp.621-634, 1993.

[23] E. McCreight, "Priority search trees", *SIAM Journal of Computing*, 14(2), pp.257-276, 1985.

[24] Mouratidis, K., Bakiras, S., Papadias, D.: "Continuous Monitoring of Top-k Queries over Sliding Windows", *In Proc. of SIGMOD*, 635-646, 2006.

[25] M. H. Overmars, "Efficient data structures for range searching on a grid", *Journal of Algorithms*, 9, pp.254-275, 1988.

[26] S. Ramaswamy and S. Subramanian, "Path caching: A technique for optimal external searching", *Proceedings of PODS*, pp.25-35, 1994.

[27] S. Sioutas, C. Makris, N. Kitsios, G. Lagogiannis, J. Tsaknakis, K. Tsichlas, B. Vassiliadis, "Geometric Retrieval for Grid Points in the RAM Model", J. UCS 10(9), pp.1325-1353, 2004.

[28] S. Subramanian and S. Ramaswamy, "The P-range tree: A new data structure for range searching in secondary memory", *Proceedings of SODA*, pp.378-387, 1995.

[29] J.S. Vitter, "External memory algorithms and data structures: dealing with massive data", *ACM Computing Surveys*, 33(2), pp.209-271, 2001.

[30] D.E. Willard, "Applications of the fusion tree method to computational geometry and searching", *Proceedings of SODA*, pp.286-295, 1992.