Journal name
Volume, number, pages

An Experimental Performance Comparison for Indexing Mobile Objects on the Plane

S. Sioutas
Department of Informatics
Ionian University

G. Papaloukopoulos
Department of Computer Engineering and Informatics
University of Patras

K. Tsichlas and Y. Manolopoulos
Department of Informatics
Aristotle University of Thessaloniki

We present a time-efficient approach to index objects moving on the plane to efficiently answer range queries about their future positions. Each object is moving with non small velocity u, meaning that the velocity value distribution is skewed (Zipf) towards u_{min} in some range $[u_{min}, u_{max}]$, where u_{min} is a positive lower threshold. Our algorithm enhances a previously described solution (Sioutas, Tsakalidis, Tsichlas, Makris, & Manolopoulos, 2007) by accommodating the ISB-tree access method as presented in (A. C. Kaporis et al., 2005). Experimental evaluation shows the improved performance, scalability and efficiency of the new algorithm.

Introduction

This paper focuses on the problem of indexing mobile objects in two dimensions and efficiently answering range queries over the objects' future locations. This problem is motivated by a set of real-life applications such as intelligent transportation systems, cellular communications, and meteorology monitoring. The basic approach uses discrete movements, where the problem of dealing with a set of moving objects can be considered as equivalent to a sequence of database snapshots of the object positions/extents taken at time instants $t_1 < t_2 < \ldots$, with each time instant denoting the moment where a change took place. From this point of view, the indexing problems in such environments can be dealt with by suitably extending indexing techniques from the area of spatio-temporal databases (Gaede & Gunther, 1998; Salzberg & Tsotras, 1999). In (Manolopoulos, Theodoridis, & Tsotras, 2000) it is exposed how these indexing techniques can be generalized to handle efficiently queries in a discrete spatio-temporal environment.

The common thrust behind these indexing structures lies in the idea of abstracting each object's position as a continuous function of time, $f(t)$, and updating the database whenever the function parameters change. Accordingly an object is modelled as a pair consisting of its extent at a reference time (design parameter) and of its motion vector. One categorization of the aforementioned structures is according to the family of the underlying access method used. In particular, there are approaches based either on R-trees or on Quadtrees as explained in (Raptopoulou, Vassilakopoulos, & Manolopoulos, 2004, 2006). On the other hand, these structures can be also partitioned into those that: (a) are based on geometric duality and represent the stored objects in the dual space (Agarwal, Arge, & Erickson, 2000; Kollios, Gunopulos, & Tsotras, 1999; Patel, Chen, & Chakka, 2004), and (b) leave the original representation intact by indexing data in their native dimensional space (Beckmann, Begel, Schneider, & Seeger, 1990; Papadopoulos, Kollios, Gunopulos, & Tsotras, 2002; Saltenis, Jensen, Leutenegger, & Lopez, 2000; Saltenis et al., 2001; Tao, Papadias, & Sun, 2003). The *geometric duality transformation* is a tool extensively used in the Computational Geometry literature, which maps hyper-planes in R^d to points and vice-versa. In this paper we present and experimentally evaluate techniques using the duality transform as in (Kollios et al., 1999; Papadopoulos et al., 2002) to efficiently index future locations of moving points on the plane.

In the next section, we present a brief overview of the most basic practical methods. In Section 3 we give a formal description of the problem. In Section 4 we introduce the duality transform methods, in section 5 we briefly present our main contribution whereas in section 6 we present the ISBs access method that compares favourably with the solutions of (Kollios et al., 1999; Papadopoulos et al., 2002), the TPR* index (Tao et al., 2003), the STRIPES index (Patel et al., 2004) and the LBTs index (Sioutas et al., 2007) as well. In simple words, the new solution is the most efficient in terms of update I/O performance. Moreover, with respect to the query I/O performance, solution of ISBs is 4 or 5 faster than LBTs method and outperforms STRIPES (state of the art as of now) in many settings. Section 7 presents a thorough experimental evaluation, whereas Section 8 concludes the paper.

Contact informations: 49100 Corfu, Greece
e-mail:sioutas@ionio.gr

A Brief Overview of the Relevant Methods

The TPR tree (Saltenis et al., 2000) in essence is an R*-tree generalization to store and access linearly moving objects. The leaves of the structure store pairs with the position of the moving point and the moving point id, whereas internal nodes store pointers to subtrees with associated rectangles that minimally bound all moving points or other rectangles in the subtree. The difference with respect to the classical R*-tree lies in the fact that the bounding rectangles are time parameterized (their coordinates are functions of time). It is considered that a time parameterized rectangle bounds all enclosed points or rectangles at all times not earlier than current time. Search and update algorithms in the TPR tree are straightforward generalizations of the respective algorithms in the R*-tree; moreover, the various kinds of spatiotemporal queries can be handled uniformly in 1-, 2-, and 3-dimensional spaces.

The TPR-tree served as the base structure for further developments in the area (Saltenis et al., 2001). *TPR*-tree*, an extension of the TPR-tree, improves the latter in update operations (Tao et al., 2003). The main improvement lies in the fact that local optimization criteria (at each tree node) may degrade seriously the performance of the structure and more particularly in the use of update rules that are based on global optimization criteria. Thus, the authors of (Tao et al., 2003) proposed a novel probabilistic cost model to validate the performance of a spatiotemporal index and analyse with this model the optimal performance for any data-partition index.

The STRIPES index (Patel et al., 2004) is based on the application of the duality transformation and employs disjoint regular space partitions (disk based quadtrees (Gaede & Gunther, 1998)). Through the use of a series of implementations, the authors claim that STRIPES outperforms TPR*-trees for both update and query operations.

Finally, the LBTs index (Sioutas et al., 2007) has the most efficient update performance in all cases. Regarding the query performance, LBTs method prevails as long as the query rectangle length remains in realistic levels (by far superiority in comparison to opponent methods). If the query rectangle length becomes huge in relation to the whole terrain, then STRIPES is the best solution, however, only to a small margin in comparison to LBTs method.

Definitions and Problem Description

We consider a database that records the position of moving objects in two dimensions on a finite terrain. We assume that objects move with a constant velocity vector starting from a specific location at a specific time instant. Thus, we can calculate the future object position, provided that its motion characteristics remain the same. Velocities are bounded by $[u_{min}, u_{max}]$. Objects update their motion information, when their speed or direction changes. The system is dynamic, i.e. objects may be deleted or new objects may be inserted.

Let $P_z(t_0) = [x_0, y_0]$ be the initial position at time t_0 of object z. If object z starts moving at time $t > t_0$, its position will be $P_z(t) = [x(t), y(t)] = [x_0 + u_x(t - t_0), y_0 + u_y(t - t_0)]$, where $U = (u_x, u_y)$ is its velocity vector.

We would like to answer queries of the form: "Report the objects located inside the rectangle $[x_{1_q}, x_{2_q}] \times [y_{1_q}, y_{2_q}]$ at the time instants between t_{1_q} and t_{2_q} (where $t_{now} \leq t_{1_q} \leq t_{2_q}$), given the current motion information of all objects."

Indexing mobile objects using duality transformations

In general, the duality transform maps a hyper-plane h from R^d to a point in R^d and vice-versa. One duality transform for mapping the line with equation $y(t) = ut + a$ to a point in R^2 is by using the dual plane, where one axis represents the slope u of an object's trajectory (i.e. velocity), whereas the other axis represents its intercept a. Thus we get the dual point (u, a) (this is the so-called *Hough-X transform* (Kollios et al., 1999; Papadopoulos et al., 2002)). By rewriting the equation $y = ut + a$ as $t = \frac{1}{u}y - \frac{a}{u}$, we arrive to a different dual representation (the so called *Hough-Y transform* in (Kollios et al., 1999; Papadopoulos et al., 2002)). The point in the dual plane has coordinates (b, w), where $b = -\frac{a}{u}$ and $w = \frac{1}{u}$.

In (Kollios et al., 1999; Papadopoulos et al., 2002), motions with small velocities in the Hough-Y approach are mapped into dual points (b, w) having large w coordinates ($w = 1/u$). Thus, since few objects can have small velocities, by storing the Hough-Y dual points in an index such as an R*-tree, Maximum Bounded Rectangles (MBRs) with large extents are introduced, and the index performance is severely affected. On the other hand, by using a Hough-X for the small velocities' partition, this effect is eliminated, since the Hough-X dual transform maps an object's motion to the (u, a) dual point. The query area in Hough-X plane is enlarged by the area E, which is easily computed as $E_{Hough-X} = (E1_{hough-X} + E2_{hough-X})$. By $Q_{Hough-X}$ we denote the actual area of the simplex query. Similarly, on the dual Hough-Y plane, $Q_{Hough-Y}$ denotes the actual area of the query, and $E_{Hough-Y}$ denotes the enlargement. According to these observations the solution in (Kollios et al., 1999; Papadopoulos et al., 2002) proposes the choice of that transformation which minimizes the criterion: $c = \frac{E_{Hough-X}}{Q_{Hough-X}} + \frac{E_{Hough-Y}}{Q_{Hough-Y}}$.

In order to build the index, we first decompose the 2-d motion into two 1-d motions on the (t, x) and (t, y) planes and then we build the corresponding index for each projection. Then we have to partition the objects according to their velocity: Objects with small velocity are stored using the Hough-X dual transform, whereas the rest are stored using the Hough-Y dual transform. Motion information about the other projection is also included.

To answer the exact 2-d query we decompose the query into two 1-d queries, for the (t, x) and (t, y) projection. Then, for each projection, we get the dual-simplex query and calculate the criterion c and choose the one (say p) that minimizes it. We search in projection p the Hough-X or Hough-Y partition and finally we perform a refinement or filtering step "on

the fly", by using the whole motion information. Thus, the result set contains only the objects satisfying the query.

Our contribution

We consider the case, where the objects are moving with non small velocities u, meaning that the velocity value distribution is skewed (Zipf) towards u_{min} in some range $[u_{min}, u_{max}]$ and as a consequence the $Q_{Hough-Y}$ transformation is used (denote that u_{min} is a positive lower threshold). In (Kollios et al., 1999; Papadopoulos et al., 2002) and (Sioutas et al., 2007), $Q_{Hough-Y}$ is computed by querying a B$^+$-tree and LBTs (Lazy B-trees) respectively, each of which indexes the b parameters. Our construction is based on the use of the ISB-tree (A. C. Kaporis et al., 2005) instead of the B$^+$-tree or Lazy B-trees, achieving optimal update performance and near-optimal query performance. Next we describe the main characteristics of the ISB-tree.

The ISB-tree

In the following, we give some basic definitions, we describe the interpolation method of searching, we describe the Interpolation Search Tree as the basic main memory structure and finally we give the required technical details of the external memory Interpolation Search B-Tree (ISB-tree) presented in (A. C. Kaporis et al., 2005).

Basic definitions: regular and smooth input distributions

According to Willard (Willard, 1985), a probability density μ is regular if there are constants b_1, b_2, b_3, b_4 such that $\mu(x) = 0$ for $x < b_1$ or $x > b_2$, and $\mu(x) \geq b_3 > 0$ and $|\mu'(x)| \leq b_4$ for $b_1 \leq x \leq b_2$. This has been further pursued by Mehlhorn and Tsakalidis (Mehlhorn & Tsakalidis, 1993), who introduced the *smooth* input distributions, a notion that was further generalized and refined in (Andersson & Mattsson, 1993). Given two functions f_1 and f_2, a density function $\mu = \mu[a, b](x)$ is (f_1, f_2)-*smooth* (Andersson & Mattsson, 1993) if there exists a constant β, such that for all c_1, c_2, c_3 where $a \leq c_1 < c_2 < c_3 \leq b$, 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 (Andersson & Mattsson, 1993; A. Kaporis et al., 2003). Actually, *any* probability distribution is $(f_1, \Theta(n))$-smooth, for a suitable choice of β.

Searching algorithms and interpolation method

Let $S = \{X_i, 1 \leq i \leq n\}$ an ordered set of n elements. The basic routine of a searching operation is described by Algorithm1 as follows:

Algorithm 1 Search(x,S)

```
1: left ← 1
2: right ← n
3: next ← k ∈ [left, right]
4: while x <> S[next] and left < right do
5:    if x ≤ S[next] then
6:       right ← next − 1
7:    else
8:       left ← next + 1
9:    end if
10:   next ← k ∈ [left, right]
11: end while
12: if x = S[next] then
13:   print('Success')
14: else
15:   print('Fail')
16: end if
```

If $next \leftarrow left + 1$ the routine above is a linear searching routine and the worst-case time is $O(n)$. If $next \leftarrow \left\lfloor \frac{right + left}{2} \right\rfloor$ we refer to a binary searching routine with $O(\log n)$ worst-case time. If $next \leftarrow \left\lfloor \frac{x - S[left]}{S[right] - S[left]}(right - left) \right\rfloor + left$ we refer to an interpolation searching routine for which time improvements can be obtained if certain classes of input distributions are considered. In particular, for random data generated according to the uniform distribution, the interpolation searching routine achieves $\Theta(\log \log n)$ expected time. Willard (Willard, 1985) showed that this time bound holds for an extended class of regular distributions, as defined previously. A natural extension is to adapt interpolation search into dynamic data structures, that is, data structures which support insertion and deletion of elements in addition to interpolation search. Thus, the first step was to develop a static tree-like structure, which adapts the method above (the so-called **S**tatic **I**nterpolation **S**earch **T**ree) and the second and final step the dynamization of this tree.

The **I***nterpolation* **S***earch* **T***ree*

The *static interpolation search tree* has been presented in (A. Kaporis et al., 2003). Consider a random file $S = \{X_1, \ldots, X_n\}$, where each key $X_i \in [a, b] \subset \Re$, $1 \leq i \leq n$, obeys an unknown distribution μ. Let $P = \{X_{(1)}, \ldots, X_{(n)}\}$ be an increasing ordering of S. The goal is to find the largest key $X_{(j)} \in P$ that precedes a *target* element x.

A static interpolation search tree (SIST) corresponding to P can be fully characterized by three non-decreasing functions $H(n)$, $R(n)$ and $I(n)$, which are non-decreasing and in-

vertible with a second derivative less than or equal to zero. $H(n)$ denotes the tree height, $R(n)$ denotes the root fan-out, whereas $I(n)$ denotes how fine is the partition of the set of elements. Achieving a height of $H(n)$ dictates that $R(n) = n/H^{-1}(H(n) - 1)$. Moreover, $H(n)$ should be $o(\log n)$ and not $O(1)$, and $H^{-1}(i) \neq 0$, for $1 \leq i \leq H(n) - 1$. To handle an as large as possible class of distributions μ, the approximation of the sample density should be as fine as possible, implying that $I(n)$ should be as large as possible. Since $I(n)$ affects space, it is chosen as $I(n) = n \cdot g(H(n))$, where $\sum_{i=1}^{\infty} g(i) = \Theta(1)$, so that the space of SIST remains linear.

The aforementioned choice of functions $H(n)$, $R(n)$ and $I(n)$ ensures that a SIST on n elements, drawn from a $(n \cdot g(H(n)), H^{-1}(H(n) - 1))$-smooth distribution μ, can be built in $O(n)$ time and space (A. Kaporis et al., 2003).

The root node of SIST corresponds to the ordered file P of size n. Each child corresponds to a part of P of size $\frac{n}{R(n)}$, i.e., the subtree rooted at each child of the root has $n' = n/R(n)$ leaves and height $H(n') = H(n)-1$. The fan-out of each child is $R(n') = \Theta(H^{-1}(H(n) - 1)/H^{-1}(H(n) - 2))$, while $I(n') = n' \cdot g(H(n'))$. In general, consider an internal node v at depth i and assume that n_i leaves (elements of P) are stored in the subtree rooted at v, whose keys take values in $[\ell, u]$. Then, we have that $R(n_i) = \Theta(H^{-1}(H(n) - i + 1)/H^{-1}(H(n) - i))$, and $I(n_i) = n_i \cdot g(H(n) - i)$.

Each internal tree node v at depth i is associated with an array REP[1..$R(n_i)$] of sample elements, containing one sample element for each of its subtrees, and an array ID[1..$I(n_i)$] that stores a set of sample elements approximating the inverse distribution function. The role of the ID array is to partition the interval $[\ell, u]$ into $I(n_i)$ equal parts, each of length $\frac{u-\ell}{I(n_i)}$. The role of the REP array is to partition its associated ordered sub-file of P into $R(n_i)$ equal subfiles, each of size $\frac{n_i}{R(n_i)}$. By using the ID array, we can interpolate the REP array to determine the subtree in which the search procedure will continue. In particular, for the ID[1..$I(n_i)$] array associated with node v, it holds that ID[i] = j iff REP[j] $< \ell + i(u - \ell)/I(n_i) \leq$ REP[$j + 1$]. Let x be the element we seek. To interpolate REP, compute the index $j =$ ID[$\lfloor (I(n_i)(x-\ell)/(u-\ell)) \rfloor$], and then search the REP array from REP[$j + 1$] until the appropriate subtree is located. For each node we explicitly maintain parent and child pointers. The required pointer information can be easily incorporated in the construction of the static interpolation search tree.

The ISB-tree: a dynamic Interpolation Search Tree in external memory

The ISB-tree is a two-level data structure (see Figure 1). The upper level is a non-straightforward externalization of the Static Interpolation Search Tree (SIST).

The lower level of the ISB-tree is a forest of buckets, each one implemented by a new variant of the classical B-tree, the *Lazy B-tree*, introduced in (A. C. Kaporis et al., 2005) and used in (Sioutas et al., 2007). Each bucket contains a subset of the stored elements and is represented by a unique representative. The representatives of the buckets as well as some additional elements are stored in the upper level structure.The following theorem provides the complexities of the Lazy B-tree:

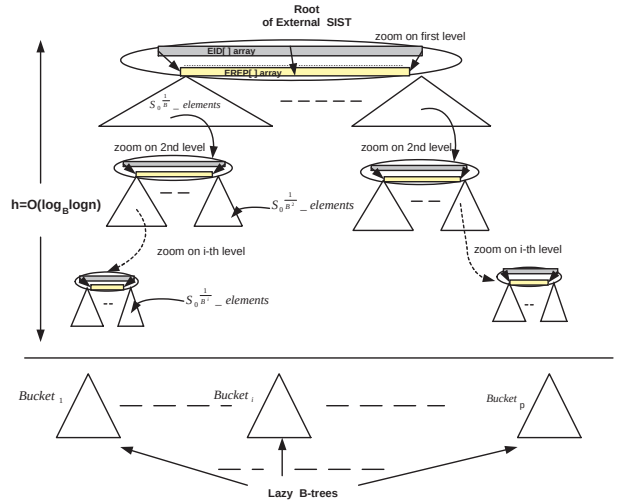

Figure 1. The ISB-tree Index

Theorem 1. The Lazy B-Tree supports the search operation in $O(\log_B N)$ worst-case block transfers and update operations in $O(1)$ worst-case block transfers, provided that the update position is given.

Proof. See (A. C. Kaporis et al., 2005; Sioutas et al., 2007).

The upper level data structure is an external version T of the static interpolation search tree (SIST) (A. Kaporis et al., 2003), with parameters $R(s) = s^{\delta}$, $I(s) = s/(\log\log s)^{1+\epsilon}$, where $\epsilon > 0$, $\delta = 1 - \frac{1}{B}$, and s is the number of stored elements in the tree. The specific choice of δ guarantees the desirable $O(\log_B \log s)$ height of the upper level structure. For each node that stores more than $B^{1+\frac{1}{B-1}}$ elements in its subtree, we represent its REP and ID arrays as static external sorted arrays; otherwise, we store all the elements in a constant number of disk blocks. In particular, let v be a node and n_v be the number of stored elements in its subtree, with $n_v \geq B^{1+\frac{1}{B-1}}$. Node v is associated with two external arrays EREP_v and EID_v that represent the REP_v and ID_v arrays of the original SIST structure. The EID_v array uses $O\left(\frac{I(n_v)}{B}\right)$ contiguous blocks, the EREP_v array uses $O\left(\frac{R(n_v)}{B}\right)$ contiguous blocks, while an arbitrary element of the arrays can be accessed with $O(1)$ block transfers, given its index. Moreover, the choice of the parameter $B^{1+\frac{1}{B-1}}$ guarantees that each of the EREP_v and EID_v arrays contains at least B elements, and hence we do not waste space (in terms of underfull blocks) in the external memory representation.

To insert/delete an element, given the position (block) of the update, we simply have to insert/delete the element to/from the Lazy B-tree storing the elements of the corresponding bucket. Note that the external SIST is not affected by these updates. Each time the number of updates exceeds

cn_0, where c is a predefined constant, the whole data structure is reconstructed.

The search procedure for locating a query element x can be decomposed into two phases: (i) the traversal of internal nodes of the external SIST to locate a bucket $\mathcal{B}_i$, and (ii) the search for x in the Lazy B-tree storing the elements of $\mathcal{B}_i$. For more technical details see (A. C. Kaporis et al., 2005). Algorithms 2-5 provide the description of ISB-tree's basic operations in pseudocode.

Lemma 2: The traversal of internal nodes of the external SIST requires $O(\log_B \log n)$ expected block transfers with high probability.

Proof: See (A. C. Kaporis et al., 2005).

The insertions and deletions of elements into the ISB-tree were simulated by a combinatorial game of balls and bins described in (A. Kaporis et al., 2003) for an internal finger-search data structure. In particular, balls correspond to elements and bins to buckets. Insertions of elements into the ISB-tree were simulated by the insertion of balls into bins according to an unknown smooth probability density μ. Similarly, the deletion of an element from the ISB-tree was simulated by the deletion of an element from a bin uniformly at random. For this process the following has been proven in (A. Kaporis et al., 2003).

Algorithm 2 Sist_Search(x,$SIST$)

1: $v \leftarrow root(SIST)$
2: **while** (v <> $leaf$) **do**
3: $i = \lfloor \frac{x-l_v}{u_v-l_v} R(n_v) \rfloor$ {Let v be a node on the search path for x, n_v the number of leaves in its subtree, l_v and u_v the minimum and maximum element respectively, stored in T_v}
4: Retrieve the $\lceil \frac{i}{B} \rceil$-th block of the EID_v array
5: $j = EID_v[i]$
6: $l = \lceil \frac{j}{B} \rceil$
7: **repeat**
8: $l \leftarrow l + 1$
9: **until** $EREP_v[l] \leq x < EREP_v[l+1]$
10: **end while**
11: follow the pointer from leaf v {Let Bin_i the corresponding bin which is organized as a Lazy B-tree}

Algorithm 3 ISB_Search(x, ISB-tree)

1: Sist_Search($x, SIST$) {Let Bin_i the corresponding Bin of the static interpolation search tree SIST}
2: LazyTree_Search(x, Bin_i) {search in the lazy B-tree was implemented in (Sioutas et al., 2007)}

Theorem 2: Consider the combinatorial game of balls and bins described in (A. Kaporis et al., 2003). Then, the expected number of balls in a bin is $O(\log n)$ with high probability.

The following lemma establishes the searching bound within a bucket of the ISB-tree.

Lemma 3: Searching for an element in a bucket of the ISB-tree takes $O(\log_B \log n)$ expected block transfers with high probability.

Proof: This is an immediate result from Theorem 1 and the size of each bucket, which is determined by Theorem 2.

The main theorem presented in (A. C. Kaporis et al., 2005) follows and holds for the very broad class of $(n/(\log\log n)^{1+\epsilon}, n^{1-\delta})$-smooth densities, where $\delta = 1 - \frac{1}{B}$ and includes the uniform, regular, bounded as well as several non-uniform distributions.

Algorithm 4 ISB_Insert(x, ISB-tree)

1: Sist_Search(x,$SIST$) {Let Bin_i the corresponding Bin}
2: LazyTree_Insert(x, Bin_i) {Bin_i is a lazy B-tree and its insert operation was implemented in (Sioutas et al., 2007)}
3: $numberofupdates \leftarrow numberofupdates + 1$
4: **if** $numberofupdates = cn_0$ **then**
5: Rebuild($SIST$) {n_0 is the total number of elements stored in the initial ISB-tree}
6: **end if**

Algorithm 5 ISB_Delete(x, ISB-tree)

1: Sist_Search($x, SIST$) {Let Bin_i the corresponding Bin}
2: LazyTree_Delete(x, Bin_i) {Bin_i is a lazy B-tree and its delete operation was implemented in (Sioutas et al., 2007)}
3: $numberofupdates \leftarrow numberofupdates + 1$
4: **if** $numberofupdates = cn_0$ **then**
5: Rebuild($SIST$) {n_0 is the total number of elements stored in the ISB-tree at the initial state}
6: **end if**

Theorem 3: Suppose that the upper level of the ISB-tree is an external static interpolation search tree with parameters $R(s_0) = s_0^{\delta}$, $I(s_0) = s_0/(\log\log s_0)^{1+\epsilon}$, where $\epsilon > 0$, $\delta = 1 - \frac{1}{B}$, $s_0 = n_0$, n_0 is the number of elements in the latest reconstruction, and that the lower level is implemented as a forest of Lazy B-trees. Then, the ISB-tree supports search operations in $O(\log_B \log n)$ expected block transfers with high probability, where n denotes the current number of elements, and update operations in $O(1)$ worst-case block transfers, if the update position is given. The worst-case update bound is $O(\log_B n)$ block transfers, and the structure occupies $O(n/B)$ blocks.

Proof [1]: From Lemmas 2 and 3, the searching operation takes $O(\log_B \log n)$ expected number of block transfers with high probability. Considering the update bound, between reconstructions the block transfers for an update are clearly $O(1)$, since we only have to update the appropriate Lazy B-tree which can be done in $O(1)$ block transfers (see Theorem 1). The reconstructions can be easily handled by using the technique of global rebuilding (Levcopoulos & Overmars, 1988). With this technique the linear work spent during a global reconstruction of the upper level structure may

[1] We quote a brief description of the proof presented in (A. C. Kaporis et al., 2005)

be spread out on the updates in such a way that a rebuilding cost of $O(1)$ block transfers is spent at each update.

Experimental Evaluation

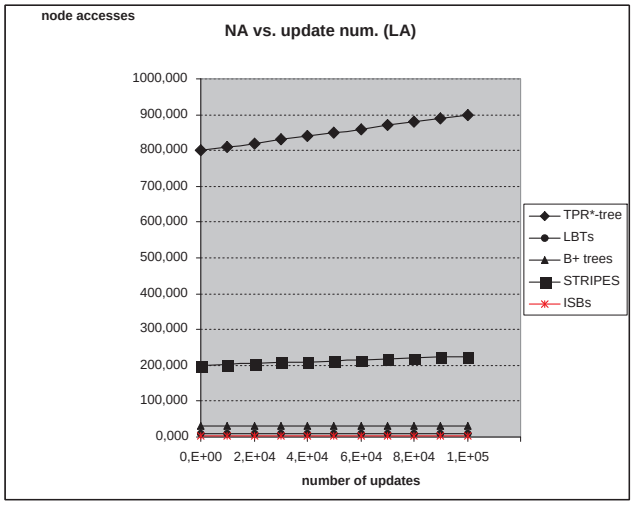

Figure 2. $q_V len = 5, q_T len = 50, q_R len = 100$

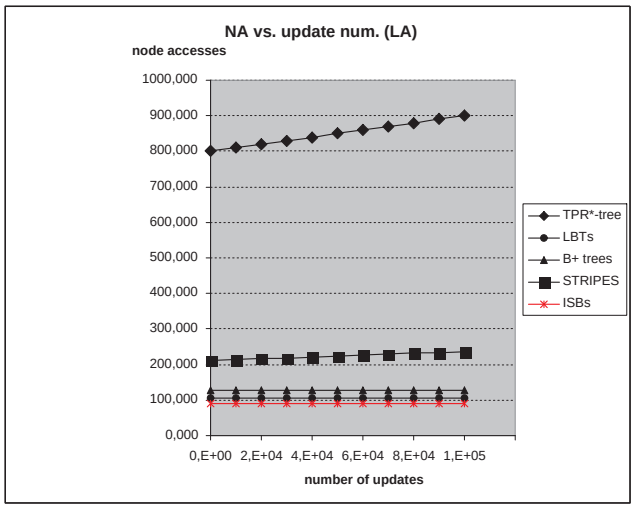

Figure 3. $q_V len = 5, q_T len = 50, q_R len = 1000$

This section compares the query/update performance of our solution with STRIPES as well as with those ones that use B$^+$-trees, Lazy B-trees (LBTs) and TPR*-tree, as well. We deploy spatio-temporal data that contain insertions at a single timestamp 0. In particular, objects' MBRs are taken from the LA spatial dataset[2]. We want to simulate a situation where all objects move in a space with dimensions 100x100 kilometers. For this purpose each axis of the space is normalized to [0,100000]. For the TPR*-tree, each object is associated with a VBR (Velocity Bounded Rectangle) such that (a) the object does not change spatial extents during its

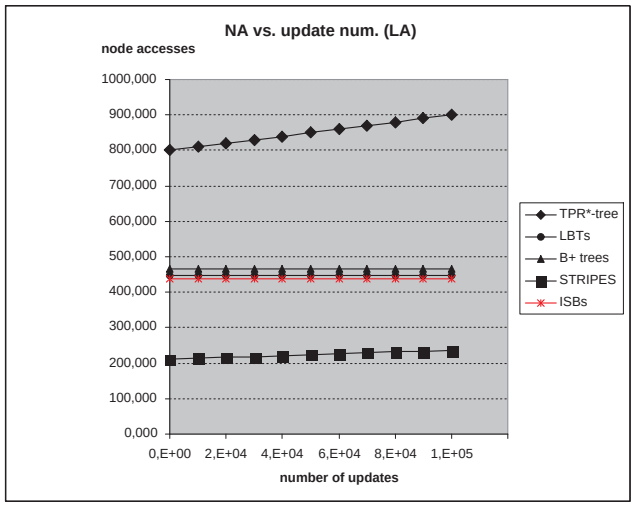

Figure 4. $q_R len = 2000, q_V len = 5, q_T len = 50$

movement, and (b) the velocity value distribution is skewed (Zipf) towards 30 in the range [30,50], and (c) the velocity can be either positive or negative with equal probability.

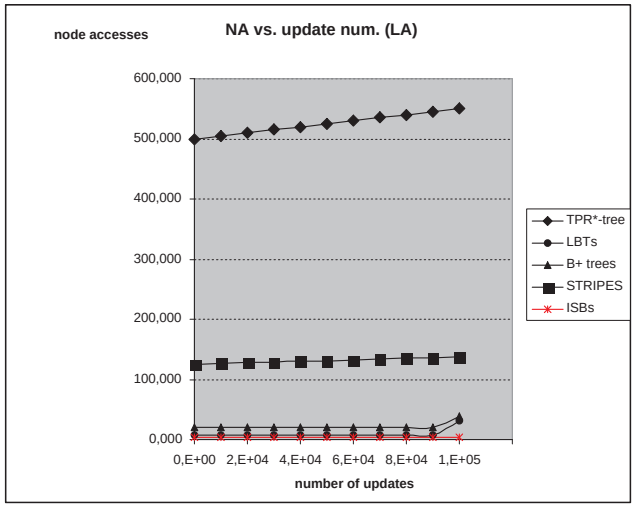

Figure 5. $q_V len = 10, q_T len = 50, q_R len = 400$

We will use a page size of 1 Kbyte so that the number of index nodes simulates realistic situations. Also, for all experiments, the key length is 8 bytes, whereas the pointer length is 4 bytes. Thus, the maximum number of entries ($< x >$ or $< y >$, respectively) in both Lazy B-trees and B$^+$-trees is 1024/(8+4)=85. In the same way, the maximum number of entries (2-d rectangles or $< x1, y1, x2, y2 >$ tuples) in TPR*-tree is 1024/(4*8+4)=27. On the other hand, the STRIPES index maps predicted positions to points in a dual

[2] Downloaded 128.971 MBRs from `http://www.census.gov/geo/www/tiger/`

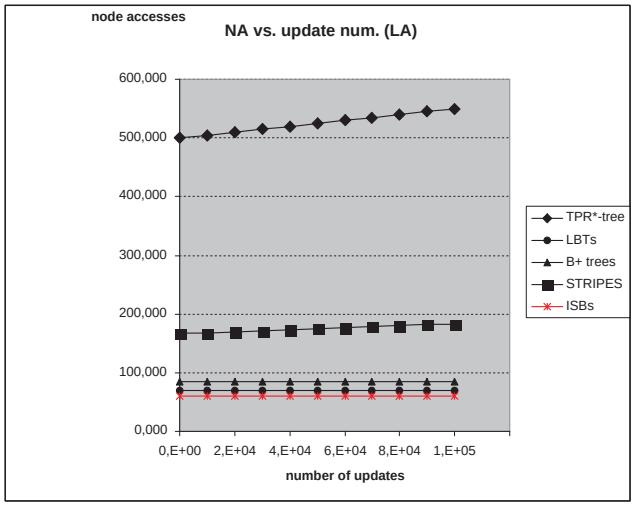

Figure 6. $q_V len = 10, q_T len = 50, q_R len = 1000$

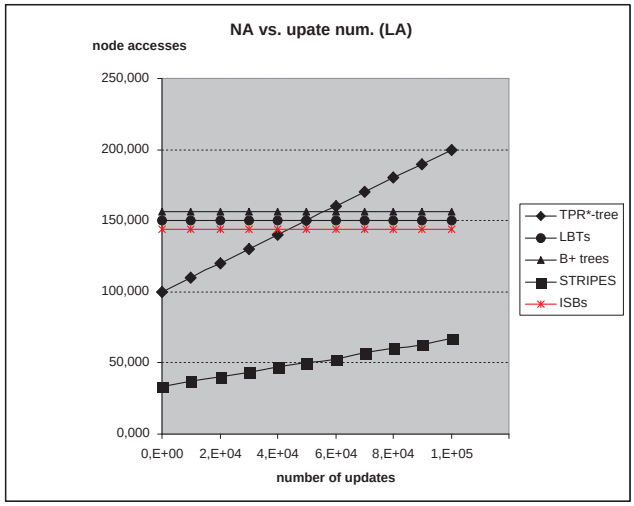

Figure 8. $q_V len = 5, q_T len = 1, q_R len = 1000$

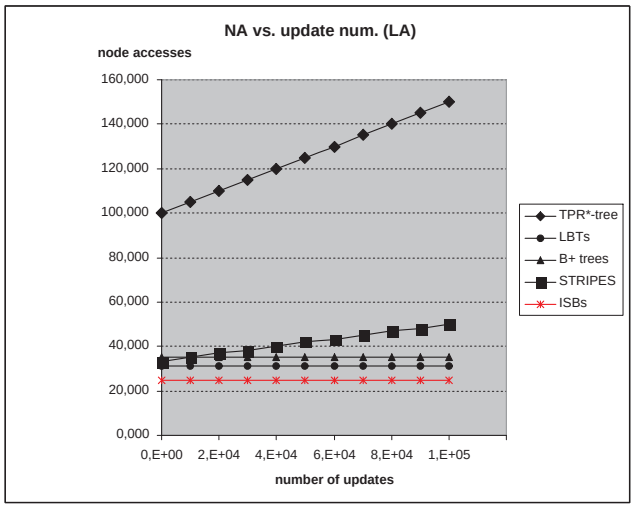

Figure 7. $q_V len = 5, q_T len = 1, q_R len = 400$

transformed space and indexes this space using a disjoint regular partitioning of space. Each of the two dual planes, are equally partitioned into four quads. This partitioning results in a total of $4^2 = 16$ partitions, which we call *grids*. Thus, the fan-out of each internal node is 16. The ISB-tree[3] has an exponentially decreased fan-out and 2 levels at most.

For each dataset, all indexes except for STRIPES and ISBs have similar sizes. Specifically, for LA, each tree has 4 levels and around 6700 leaves with the exception of: (a) the STRIPES index which has a maximum height of seven and consumes about 2.4 times larger disk space, and (b) the ISB index which has a maximum height of 2. Each query q has three parameters: $q_R len$, $q_V len$, and $q_T len$, such that: (a) its MBR q_R is a square, with length $q_R len$, uniformly generated in the data space, (b) its VBR is $q_V = -q_V len/2, q_V len/2, -q_V len/2, q_V len/2$, and (c) its query interval is $q_T = [0, q_T len]$. The query cost is measured as the average number of node accesses in executing a workload of 200 queries with the same parameters. Implementations were carried out in C++ including particular libraries from SECONDARY LEDA v4.1.

Query cost comparison

We measure the performance of our technique described previously (in particular one ISB-tree for each projection, plus the query processing between the two answers), in comparison to that of the LBTs method (Sioutas et al., 2007), the traditional technique presented in (Kollios et al., 1999; Papadopoulos et al., 2002), the TPR*-tree (Tao et al., 2003) and the STRIPES method (Patel et al., 2004), using the same query workload, after every 10000 updates. Figures 2-6 show the query cost (for datasets generated from LA as described above) as a function of the number of updates, using workloads with different parameters. In these figures our solution is almost 4-5 times more efficient (in terms of the number of I/Os) than the solution using LBTs and B^+-trees. This fact is an immediate result of the sublogarithmic I/O searching complexity of ISB-tree in comparison to the logarithmic I/O searching complexities of both structures B^+-tree and Lazy B-tree. In particular, we have to index the appropriate b parameters in each projection and then to combine the two answers by detecting and filtering all pair permutations. As a consequence, the ISBs method is significantly faster than LBTs and traditional B^+-trees methods.

Figures 2 and 3 depict the efficiency of our solution in comparison to that of the TPR*-tree and STRIPES. In figure 2, where the length of the query rectangle is 100 and as a consequence the query's surface is equal to $10000m^2$ or 1 hectare (the surface of the whole spatial terrain is 10^6 hectares) the

[3] source code of ISB-tree access method is available at http://www.ionio.gr/~sioutas/New-Software.htm

ISBs method is consistently about 53 times faster than the STRIPES method, 212 times faster than the TPR*-tree, 7.5 times faster than the B+-trees method and 2 times faster than the LBTs method. The superiority of our solution decreases as the query rectangle length grows from 100 to 1000. Thus, in figure 3, where the spatial query's surface is equal to 100 hectares, again our method is faster about 2.2 times with respect to the STRIPES method, 8.3 times wrt the TPR*-tree, 1.25 times wrt the B+-trees methods and 1.05 times wrt the LBTs method.

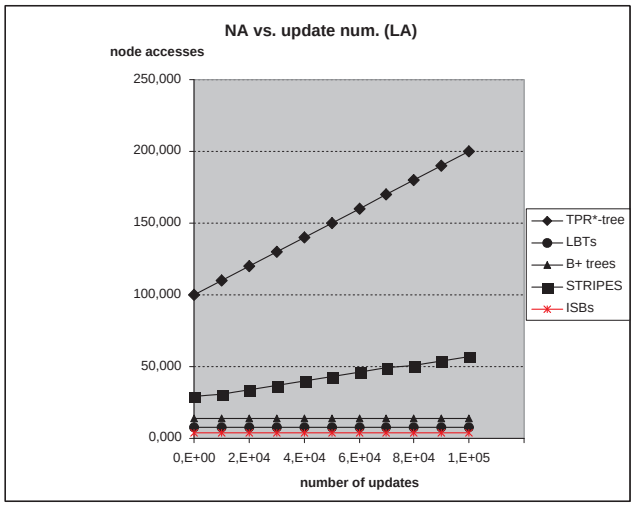

Figure 9. $q_R len = 400$, $q_V len = 5$, $q_T len = 100$

In real GIS applications, for a vast spatial terrain of 10^6 hectares, e.g. the road network of a big town where each road square covers no more than 1 hectare (or $10.000m^2$) the most frequent queries consider spatial query's surface no more than 100 road squares (or 100 hectares) and future time interval no larger than 100 seconds. However, to test the methods' performance in extreme cases we conducted the following experiment. When the query rectangle length or equivalently the query surface becomes extremely large (e.g. 2000, or equivalently 400 hectares), then the STRIPES index shows better performance as depicted in Figure 4. In particular, our method is still 1.9 times faster than the TPR*-tree, however, the STRIPES method is twice faster than our one. The apparent explanation is that as the surface of the query rectangle grows, the answer size in each projection grows as well, thus the performance of the ISBs method that combines and filters the two answers becomes less attractive. However, we do not consider such extreme case as *realistic* scenarios. Figures 5 and 6 depict the performance of all methods for a growing velocity vector. In particular, in figure 5 the ISBs method consistently prevails about 33 times in comparison to the STRIPES method, 137 times in comparison to the TPR*-tree, 5 times in comparison to the B+-trees and 2 times in comparison to the LBTs method. The superiority of our solution becomes less strong as the query rectangle length grows from 400 (16 hectares of query surface) to 1000 (100 hectares of query surface). However, notably even in the latter case (see Figure 6), our method is about 2.7 times faster with respect to the STRIPES method, 8.3 times wrt the TPR*-tree, 1.3 times wrt the B+-trees and 1.06 times wrt the LBTs method. Obviously, the velocity factor is very important for TPR-like solutions, but not for the other methods, for LBTs and ISBs in particular, which depend exclusively on the query surface. Figures 7 and 8 depict the performance of all methods when the time interval length approaches the 1 value. However, notably even in this case (see Figure 7), the ISBs method is about 1.6 times faster with respect to the STRIPES method, 4.6 times faster wrt the TPR*-tree, 1.3 times faster wrt the B+-trees and 1.2 times faster wrt the LBTs method. As query rectangle length grows from 400 to 1000, the ISBs method advantage decreases; from the bottom figure, we remark that STRIPES is about 3 times faster, whereas our method is 1.03 times faster than the TPR*-tree, 1.07 times faster than the B+-trees and 1.03 times faster than the LBTs method. Figure 9 depicts the efficiency of our solution in comparison to that of TPR*-trees and STRIPES when the time interval length reaches the value of 100. In particular, the ISBs method is consistently about 10 times faster than STRIPES, 37 times faster than TPR*-tree, 3.5 times faster than the B+-trees and 2 times faster than the LBTs method. As required in practice, the query surface remains in *realistic* levels (16 hectares).

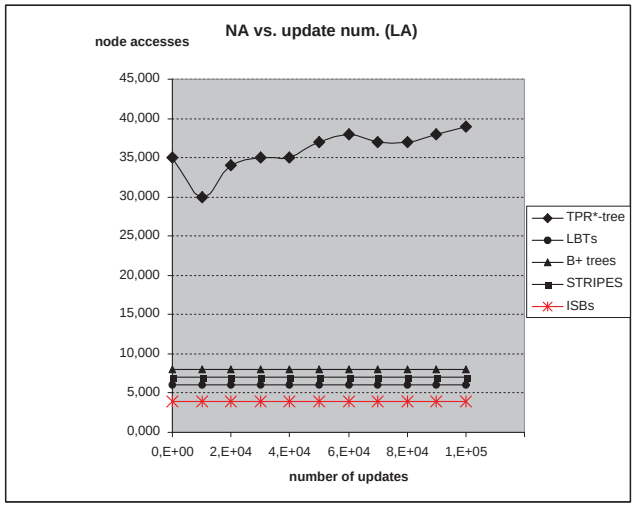

Figure 10. Update Cost Comparison

Scalability and update cost comparison

Figure 10 compares the average cost (amortized over insertions and deletions) as a function of the number of updates. ISBs and LBTs methods have optimal update performance and consistently outperform the TPR*-tree by a wide margin as well as the STRIPES index by a narrow margin. In particular, ISBs and LBTs methods require a constant number of 4 and 6 block transfers respectively(3 and 2 block transfers respectively for each projection, for details see (A. C. Kaporis et al., 2005)) and this update performance

is independent of the dataset size. On the other hand, the other 3 solutions do not have constant update performance; instead their performance depends on the dataset size even if as in the experiment of Figure 10 B^+-trees and STRIPES reach the optimal performance of ISBs and LBTs methods requiring 8 and 7 block transfers respectively (TPR^*-tree requires 35 block transfers in average). The experiments above show that ISBs method achieves a near optimal performance for the most cases. This stems from the fact that the MBRs' projections from the LA spatial datasets follow an almost uniform (the most popular density of smooth family) distribution, due to the almost uniform decomposition of spatial maps. In particular, LA dataset constitutes a dense spatial map and hence the derived one-dimensional data produce densely populated elements. Thus, the interpolation technique of ISBs method works very well and its expected excellent behavior follows with high probability.

Conclusions

We have used a new access method for indexing moving objects on the plane to efficiently answer range queries about their future location. Its update performance is the most efficient in all cases with no exception. Regarding the query performance, the superiority of our structure has been shown as long as the query rectangle length remains in realistic levels, thus by far outperforming the opponent methods. If the query rectangle length becomes extremely huge in relation to the whole terrain (which apparently is a non-practical instance), then the STRIPES method is the best solution, however, only to a small margin in comparison to our method. We anticipate that for synthetic gigantic datasets the ISBs method will be superior in any case.

References

Agarwal, P. K., Arge, L., & Erickson, J. (2000). Indexing moving points. In *Proceedings of 19th symposium on principles of database systems* (pp. 175–186). ACM.

Andersson, A., & Mattsson, C. (1993). Dynamic interpolation search in o(log log n) time. In *Proceedings of 20th international colloquium on automata, languages and programming* (pp. 15–27). Springer.

Beckmann, N., Begel, H.-P., Schneider, R., & Seeger, B. (1990). The r*-tree: an efficient and robust access method for points and rectangles. *ACM SIGMOD Record*, *19*(2), 322–331.

Gaede, V., & Gunther, O. (1998). Multidimensional access methods. *ACM Computing Surveys*, *30*(2), 170–231.

Kaporis, A., Makris, C., Sioutas, S., Tsakalidis, A., Tsichlas, K., & Zaroliagis, C. (2003). Improved bounds for finger search on a ram. In *Proceedings of 12th annual european symposium on algorithms* (pp. 325–336). Springer.

Kaporis, A. C., Makris, C., Mavritsakis, G., Sioutas, S., Tsakalidis, A. K., Tsichlas, K., et al. (2005). Isb-tree: A new indexing scheme with efficient expected behaviour. In *Proceedings of 13th international symposium on algorithms and computation* (pp. 318–327). Springer.

Kollios, G., Gunopulos, D., & Tsotras, V. J. (1999). On indexing mobile objects. In *Proceedings of 18th symposium on principles of database systems* (pp. 261–272). ACM.

Levcopoulos, C., & Overmars, M. H. (1988). A balanced search tree with o(1) worst case update time. *Acta Inf.*, *26*(3), 269–277.

Manolopoulos, Y., Theodoridis, Y., & Tsotras, V. J. (2000). *Advanced database indexing*. Kluwer Academic Publishers.

Mehlhorn, K., & Tsakalidis, A. K. (1993). Dynamic interpolation search. *J. ACM*, *40*(3), 621-634.

Papadopoulos, D., Kollios, G., Gunopulos, D., & Tsotras, V. J. (2002). Indexing mobile objects on the plane. In *Proceedings of 13th international workshop on database and expert systems applications* (pp. 693 – 697). IEEE Computer Society.

Patel, J. M., Chen, Y., & Chakka, V. P. (2004). Stripes: An efficient index for predicted trajectories. In *Proceedings of the 2004 acm sigmod international conference on management of data* (pp. 637–646). ACM.

Raptopoulou, K., Vassilakopoulos, M., & Manolopoulos, Y. (2004). Towards quadtree-based moving objects databases. In *Proceedings of 8th east-european conference on advanced databases and information systems* (pp. 230–245). Springer.

Raptopoulou, K., Vassilakopoulos, M., & Manolopoulos, Y. (2006). On past-time indexing of moving objects. *Journal of Systems and Software*, *79*(8), 1079-1091.

Saltenis, S., Jensen, C. S., (codirector), C. S. J., Bohlen, M. H., Gregersen, H., Pfoser, D., et al. (2001). Indexing of moving objects for location-based services. In *Proceedings of 18th ieee international conference on data engineering* (pp. 463–472). IEEE Computer Society.

Saltenis, S., Jensen, C. S., Leutenegger, S. T., & Lopez, M. A. (2000). Indexing the positions of continuously moving objects. *ACM SIGMOD Record*, *29*(2), 331–342.

Salzberg, B., & Tsotras, V. J. (1999). Comparison of access methods for time-evolving data. *ACM Comput. Surv.*, *31*(2), 158–221.

Sioutas, S., Tsakalidis, K., Tsichlas, K., Makris, C., & Manolopoulos, Y. (2007). Indexing mobile objects on the plane revisited. In *Proceedings of 11th east european conference on advances in databases and information systems* (pp. 189–204). Springer.

Tao, Y., Papadias, D., & Sun, J. (2003). The tpr*-tree: an optimized spatio-temporal access method for predictive queries. In *Proceedings of 29th international conference on very large data bases* (pp. 790–801). VLDB Endowment.

Willard, D. E. (1985). Searching unindexed and nonuniformly generated files in log log n time. *SIAM J. Comput.*, *14*(4), 1013-1029.