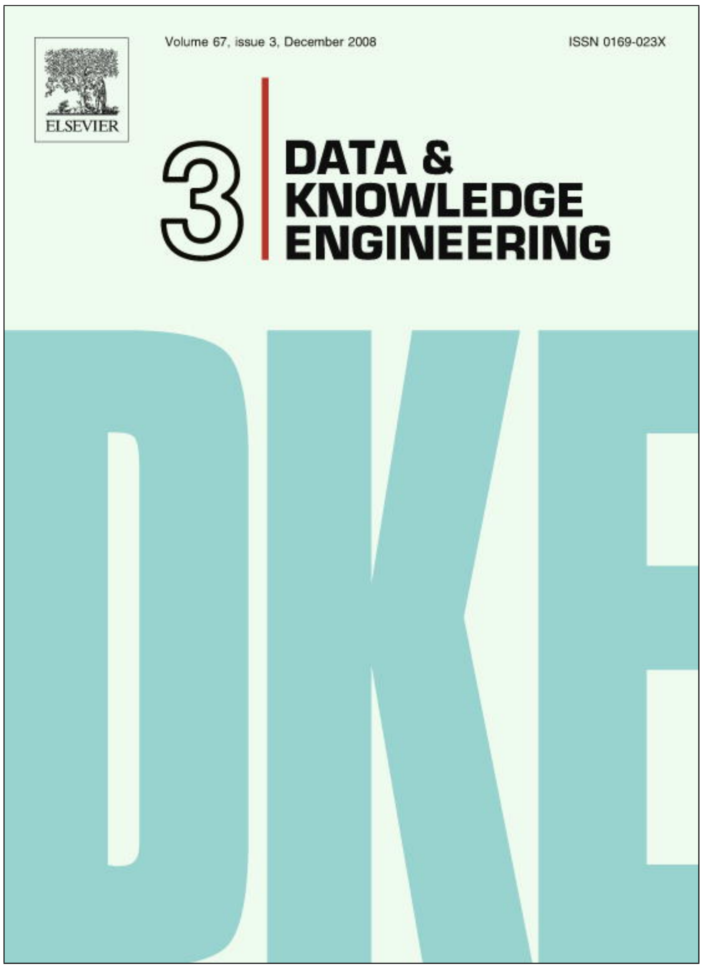

Volume 67, issue 3, December 2008
ISSN 0169-023X
ELSEVIER
DATA & KNOWLEDGE ENGINEERING
DKE

Data & Knowledge Engineering 67 (2008) 362–380

Contents lists available at ScienceDirect

Data & Knowledge Engineering

journal homepage: www.elsevier.com/locate/datak

A new approach on indexing mobile objects on the plane [☆]

S. Sioutas [a,*], K. Tsakalidis [b], K. Tsichlas [c], C. Makris [b], Y. Manolopoulos [c]

[a] *Department of Informatics, Ionian University, Corfu, Greece*
[b] *Department of Computer Engineering and Informatics, University of Patras, Greece*
[c] *Department of Informatics, Aristotle University of Thessaloniki, Greece*

A R T I C L E I N F O

Article history:
Received 16 March 2008
Received in revised form 2 June 2008
Accepted 17 June 2008
Available online 26 June 2008

Keywords:
Spatiotemporal databases
Indexing
Data structures
Computational geometry

A B S T R A C T

We present a set of time-efficient approaches to index objects moving on the plane to efficiently answer range queries about their future positions. Our algorithms are based on previously described solutions as well as on the employment of efficient access methods. Finally, an experimental evaluation is included that shows the performance, scalability and efficiency of our methods.

1. 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. There are two basic approaches used to handle this problem: to assume discrete or to assume continuous movements.

In a discrete environment 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 $t_1 < t_2 < \cdots$, 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 temporal [37] or/and spatial databases [16]; in [25] it is exposed how these indexing techniques can be generalized to handle efficiently queries in a discrete spatiotemporal environment. A plethora of efficient access methods [3,21,29,30,35,36,40] has been proposed to confront the case of continuous movements.

The common thrust behind these indexing structures lies in the idea of abstracting each object's position as a continuous function $f(t)$ of time and updating the database whenever the function parameters change; accordingly an object is modeled as a pair consisting of its extent at a reference time (design parameter) and 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 [32–34]. On the other hand, these structures can be also partitioned into (a) those that are based on geometric duality and represent the stored objects in the dual space [3,21,30], and (b) those

[☆] Preliminary version of this work was presented in the 11th East-European Conference on Advances in Databases and Information Systems (ADBIS2007), and appeared in vol. 4690, pp. 189–204 of Springer LNCS series.
[*] Corresponding author.
E-mail addresses: sioutas@ionio.gr (S. Sioutas), tsakalid@ceid.upatras.gr (K. Tsakalidis), tsichlas@csd.auth.gr (K. Tsichlas), makri@ceid.upatras.gr (C. Makris), manolopo@csd.auth.gr (Y. Manolopoulos).

0169-023X/$ - see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2008.06.009

that leave the original representation intact by indexing data in their native dimensional space [8,29,35,36,40]. The *geometric duality transformation* is a tool heavily used in the computational geometry literature, which maps hyper-planes in $\Re^d$ to points and vice versa. In this paper, we present and experimentally evaluate techniques using the duality transform as in [21,29] to efficiently index future locations of moving points on the plane.

In the next section, we present a short literature survey. In Section 3, we give a formal description of the problem. In Sections 4 and 5, we present our new solutions that compare favorably with the solutions of [21,29], the TPR* index [40] as well as the STRIPES index [30]. In particular, Section 5 presents two alternative solutions, the first one being easily implemented and with many practical merits. In addition, concerning the update performance the first solution is the most efficient. Moreover, with respect to the query performance the first solution outperforms STRIPES (the state of the art as of now) for realistic lengths of the query rectangle. Our second proposed solution has theoretical interest since it uses clever but very complicated access methods, the implementation of which is tedious and, thus, left for future work. Section 5 presents an extended experimental evaluation, and Section 6 concludes the paper.

2. Literature survey – presentation of the most basic methods

In the sequel, let N denote the input size (number of stored objects), B the block size, K the output size and thus $n = N/B$ and $k = K/B$ are the size of the input and output in blocks, respectively.

In [21], a set of indexing techniques was presented, which used the geometric duality transformation at the cost of increasing by one the dimensionality. Hence, for the one-dimensional case they reduced the indexing problem to the two-dimensional simplex range searching problem and they employed external memory partition trees to solve their indexing problem in $O(n)$ space, $O(n^{1/2} + k)$ I/Os query time and $O(\log n)$ I/Os update time.

Partition trees, though having a guaranteed worst-case performance, are generally considered non-practical since they entail large hidden factors. Thus, in [21] two more structures were presented, one based on k-d-trees and a more complex one based on B^+-trees; both structures used linear space and work well on the average. Moreover, they extended their results in the two-dimensional case for two distinct versions of the problem; first, the objects were allowed to move on a network of one-dimensional routes, and, second, the objects were allowed to move arbitrary. The first version reduced to a number of one-dimensional subproblems that use the previously described structures, whereas the second is equivalent (through geometric duality) to simplex range queries in $\Re^3$, which can be solved in $O(n^{2/3} + k)$ I/Os with the use of external memory partition trees.

In [1], the above results were further refined. A new version of partition tree was introduced to handle the indexing problem in the plane in $O(n)$ space, $O(n^{1/2} + k)$ query time, and $O(\log_B n)$ expected amortized update time; the results could apply in higher-dimensional spaces as well, degrading only the update time (it became $O(\log_B^2 n)$ I/Os). If it is assumed that the queries arrive in chronological order, then the query time can be further reduced to $O(\log_B^2 n/\log_B \log_B n)$ I/Os; this is achieved by employing the kinetic data structures framework at external range trees. Moreover, by combining multiversion kinetic data structures with partition trees, they developed an indexing scheme with small query time for near-future queries and increased time for distant in the future queries under the bound of $O(n^{1/2+\epsilon} + k)$ I/Os. Finally, an indexing technique was described for handling δ-approximate queries; the query time was $(n^{1/2+\epsilon}/\delta)$, the expected update time $O(\log_B^2 n/\delta)$ and the space $O(n/\delta)$ disk blocks.

The TPR-tree [35] 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 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. The algorithms for search and update operations 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 one-, two-, and three-dimensional spaces.

The TPR-tree constituted the base structure for further developments in the area [36]. An extension to the TPR-tree was proposed in [40], the so called TPR*-tree. The main improvement lies in the update operations, where it is shown 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. They proposed a novel probabilistic cost model to validate the performance of a spatiotemporal index and analyze with this model the optimal performance for any data-partition index.

Finally, the STRIPES index [30] is based on the application of the duality transformation and employs disjoint regular space partitions (disk based quadtrees [16]); the authors claim, through the use of a series of implementations, that STRIPES outperforms TPR*-trees for both update and query operations.

3. 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 velocities bounded by $[u_{\min}, u_{\max}]$ starting from a specific location at a specific time instant. 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 of object z at time t_0. If object z movement is recorded from 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.

For example, in Fig. 1 the lines depict the objects' trajectories on the (t, y) plane. 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 $t_{\text{now}} \leqslant t_{1_q} \leqslant t_{2_q}$), given the current motion information of all objects."

4. Indexing mobile objects in two dimensions

We decompose the 2d motion into two 1d motions on the (t, x) and (t, y) plane, respectively.

4.1. Indexing mobile objects in one dimension

We will try to transform the difficult problem of reporting the trajectory-lines, which are intersected by a rectangle-region, to the simpler one of reporting points, which lie inside a canonical planar polygon (4-sided) region. On this purpose, we will use specific duality transformation methods.

4.1.1. The duality transform

In general, the duality transform maps a hyper-plane h from $\Re^d$ to a point in $\Re^d$ and vice versa. In this subsection, we briefly describe how we can address the problem at hand in a more intuitive way, by using the duality transform on the 1d case.

4.1.2. Hough-X transform

One duality transform for mapping the line with equation $y(t) = ut + a$ to a point in $\Re^2$ 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* [21,29]). Accordingly, the 1d query $[(y_{1_q}, y_{2_q}), (t_{1_q}, t_{2_q})]$ becomes a polygon in the dual space. By using a linear constraint query [17], the query in the dual Hough-X plane (Fig. 2) is expressed as follows: if $u > 0$, then $Q_{\text{Hough-}X} = A_1 \cap A_2 \cap A_3 \cap A_4$, where A_i is defined as follows:

$$\begin{aligned}
A_1 &= u \geqslant u_{\min},\\
A_2 &= u \leqslant u_{\max},\\
A_3 &= a \geqslant y_{1_q} - t_{2_q}u,\\
A_4 &= a \leqslant y_{2_q} - t_{1_q}u.
\end{aligned}$$

Inequalities of the A_1 and A_2 areas are obvious. The inequalities for A_3 and A_4 can be derived as follows: $\forall t \in [t_{1_q}, t_{2_q}] \Rightarrow y_{1_q} \leqslant y \leqslant y_{2_q} \Rightarrow y_{1_q} - t_{2_q}u \leqslant y_{1_q} - tu \leqslant a \leqslant y_{2_q} - tu \leqslant y_{2_q} - t_{1_q}u$, since $t_{1_q} \leqslant t \leqslant t_{2_q}$.

If $u < 0$, then $Q_{\text{Hough-}X} = B_1 \cap B_2 \cap B_3 \cap B_4$, where B_i is defined as follows:

$$\begin{aligned}
B_1 &= u \leqslant -u_{\min},\\
B_2 &= u \geqslant -u_{\max},\\
B_3 &= a \geqslant y_{1_q} - t_{1_q}u,\\
B_4 &= a \leqslant y_{2_q} - t_{2_q}u.
\end{aligned}$$

Inequalities of the B_1 and B_2 areas are obvious. For B_3 and B_4 we are working in the same way as in the case of A_3 and A_4.

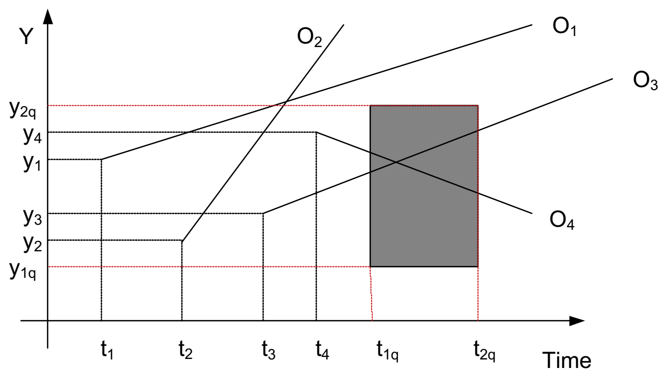


Fig. 1. Trajectories and query in (t, y) plane.

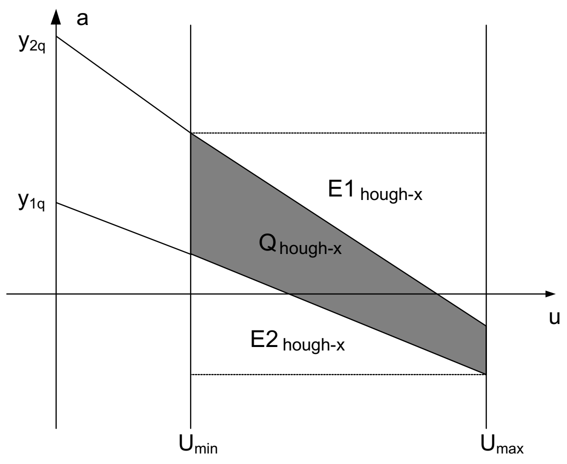


Fig. 2. Query in the Hough-*X* dual plane.

$\forall t \in [t_{1_q}, t_{2_q}] \Rightarrow y_{1_q} \leqslant y \leqslant y_{2_q} \Rightarrow y_{1_q} - t_{1_q}u \leqslant y_{1_q} - tu \leqslant a \leqslant y_{2_q} - tu \leqslant y_{2_q} - t_{2_q}u$, since $0 \leqslant t_{1_q} \leqslant t \leqslant t_{2_q}$ and $u < 0$.

In Fig. 2, the line $a = y_{1_q} - t_{1_q}u$ for $u = u_{\max}$ becomes $a = y_{1_q} - t_{1_q}u_{\max}$ and the line $a = y_{2_q} - t_{2_q}u$ for $u = u_{\min}$ becomes $a = y_{2_q} - t_{2_q}u_{\min}$. Thus, the initial query $[(t_{1_q}, t_{2_q}), (y_{1_q}, y_{2_q})]$ in the (t, y) plane is transformed to the rectangular query $[(u_{\min}, u_{\max}), (y_{1_q} - t_{1_q}u_{\max}, y_{2_q} - t_{2_q}u_{\min})]$ in the (u, a) plane.

4.1.3. Hough-Y transform

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 [21,29]). The point in the dual plane has coordinates (b, w), where $b = -\frac{a}{u}$ and $w = \frac{1}{u}$. Coordinate b is the point where the line intersects the line $y = 0$ in the primal space. Horizontal lines cannot be represented by using this transform. Similarly, the Hough-*X* transform cannot represent vertical lines. Nevertheless, since in our setting lines have a minimum and maximum slope (velocity is bounded by $[u_{\min}, u_{\max}]$), both transforms are valid.

The query in the dual Hough-*Y* plane (Fig. 3) is expressed as follows. If $u > 0$, then $Q_{\text{Hough-}Y} = C_1 \cap C_2 \cap C_3 \cap C_4$, where

$$C_1 = w = \tfrac{1}{u} \geqslant \tfrac{1}{u_{\max}},$$
$$C_2 = w = \tfrac{1}{u} \leqslant \tfrac{1}{u_{\min}},$$
$$C_3 = w \geqslant -\tfrac{1}{y}b + \tfrac{t_{1_q}}{y},$$
$$C_4 = w \leqslant \tfrac{1}{y}b + \tfrac{t_{2_q}}{y}.$$

Inequalities of the C_1 and C_2 areas are obvious. The inequalities for C_3 and C_4 can be derived as follows: $\forall t \in [t_{1_q}, t_{2_q}] \Rightarrow w = -\frac{1}{y}b + \frac{t}{y} \leqslant -\frac{1}{y}b + \frac{t_{1_q}}{y}$ and $w = -\frac{1}{y}b + \frac{t}{y} \leqslant -\frac{1}{y}b + \frac{t_{2_q}}{y}$. Hence, the two lines in Fig. 3 have negative slope and for $b = 0$ intersect the axis b in t_{1_q} and t_{2_q}, respectively. The intersection of the four regions C_1, C_2, C_3 and C_4 form the shaded polygon query of Fig. 3.

If $u < 0$, then $Q_{\text{Hough-}Y} = D_1 \cap D_2 \cap D_3 \cap D_4$, where

$$D_1 = w = \tfrac{1}{u} \leqslant -\tfrac{1}{u_{\max}},$$
$$D_2 = w = \tfrac{1}{u} \geqslant -\tfrac{1}{u_{\min}},$$

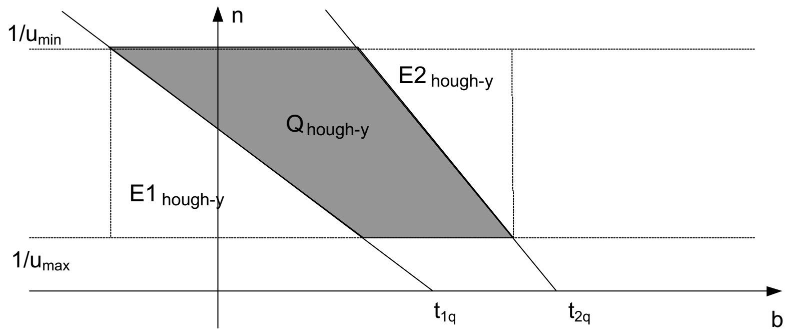


Fig. 3. Query on the Hough-*Y* dual plane.

$D_3 = w \geqslant -\frac{1}{y}b + \frac{t_{1_q}}{y}$,
$D_4 = w \leqslant -\frac{1}{y}b + \frac{t_{2_q}}{y}$.

The line $w = -\frac{1}{y}b + \frac{t_{1_q}}{y}$ for $w = \frac{1}{u_{\min}}$ implies that

$$b = t_{1_q} - \frac{y}{u_{\min}}. \tag{1}$$

In the same way the line $w = -\frac{1}{y}b + \frac{t_{2_q}}{y}$ for $w = \frac{1}{u_{\max}}$ implies that

$$b = t_{2_q} - \frac{y}{u_{\max}}. \tag{2}$$

However, according to the initial query and Eq. (1), we have

$$y_{1_q} \leqslant y \leqslant y_{2_q} \iff t_{1_q} - \frac{y_{2_q}}{u_{\min}} \leqslant b \leqslant t_{1_q} - \frac{y_{1_q}}{u_{\min}}. \tag{3}$$

Analogously, according to the initial query and Eq. (2), we have

$$y_{1_q} \leqslant y \leqslant y_{2_q} \iff t_{2_q} - \frac{y_{2_q}}{u_{\max}} \leqslant b \leqslant t_{2_q} - \frac{y_{1_q}}{u_{\max}}. \tag{4}$$

According to (3) and (4) the initial query $[(t_{1_q}, t_{2_q}), (y_{1_q}, y_{2_q})]$ in (t, y) plane can be transformed to the following rectangular query in the (b, n) plane:

$$\left[\left(t_{1_q} - \frac{y_{2_q}}{u_{\min}}, t_{2_q} - \frac{y_{1_q}}{u_{\max}}\right), \left(\frac{1}{u_{\max}}, \frac{1}{u_{\min}}\right)\right].$$

4.2. The basic algorithm for indexing mobile objects in two dimensions

In [21,29], 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, minimum 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 $E_{\text{Hough-}X} = (E1_{\text{Hough-}X} + E2_{\text{Hough-}X})$. By $Q_{\text{Hough-}X}$ we denote the actual area of the simplex query. Similarly, on the dual Hough-Y plane, $Q_{\text{Hough-}Y}$ denotes the actual area of the query, and $E_{\text{Hough-}Y}$ denotes the enlargement. According to these observations the solution in [21,29] proposes the choice of that transformation which minimizes the criterion: $c = \frac{E_{\text{Hough-}X}}{Q_{\text{Hough-}X}} + \frac{E_{\text{Hough-}Y}}{Q_{\text{Hough-}Y}}$.

The procedure for building the index follows:

(1) Decompose the 2d motion into two 1d motions on the (t, x) and (t, y) planes.
(2) For each projection, build the corresponding index.

Partition the objects according to their velocity:

- Objects with small velocity are stored using the Hough-X dual transform, whereas the remaining objects are stored using the Hough-Y dual transform.
- Motion information about the other projection is also included.

The outline of the algorithm for answering the exact 2d query follows:

(1) Decompose the query into two 1d queries, for the (t, x) and (t, y) projection.
(2) For each projection get the dual – simplex query.
(3) For each projection calculate the criterion c and choose the one (say p) that minimizes it.
(4) Search in projection p the Hough-X or Hough-Y partition.
(5) 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.

4.3. Our innovative contribution

In [21,29], $Q_{\text{Hough-}X}$ is computed by querying a 2d partition tree, whereas $Q_{\text{Hough-}Y}$ is computed by querying a B^+-tree that indexes the b parameters of Fig. 3. Our construction instead is based

(1) on the use of the Lazy B-tree [19] instead of the B^+-tree when handling queries with the Hough-Y transform, achieving optimal update performance, and
(2) on the employment of a new index that outperforms partition trees in handling polygon queries with the Hough-X transform.

Next we present the main characteristics of our proposed structures.

5. The access methods

In this section we present the efficient access method of *Lazy B-tree*, by simplifying its basic operations into legible pseudocodes as well as a new index method for polygon queries.

5.1. Handling polygon queries when using the Hough-Y transform

As described in [21,29], polygon queries when using the Hough-Y transform can be approximated by a constant number of 1d range queries that can be handled by a classical B-tree [14]. Our construction is based on the use of a B-tree variant, which is called *Lazy B-tree* and has better dynamic performance as well as optimal I/O complexities for both searching and update operations [19]. An orthogonal effort towards developing yet another B-tree variant under the same name has been proposed in [24]. The Lazy B-tree of [19] is a simple but non-trivial externalization of the techniques introduced in [31]. The following theorem provides the complexities of the Lazy B-tree:

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. The Lazy B-tree is a two-level access method as depicted in Fig. 4. The first level consists of an ordinary B-tree, whereas the second one consists of buckets of size $O(\log^2 N)$, where N is approximately equal to the number of elements stored in the access method. Each bucket consists of two list layers, L and L_i, respectively, where $1 \leqslant i \leqslant O(\log N)$, each of which has $O(\log N)$ size. The rebalancing operations are guided by the *global rebalancing lemma* given in [31] (see also [15,23]). In this scheme, each bucket is assigned a *criticality* indicating how close this bucket is to be fused or split. Every $O(\log_B N)$ updates we choose the bucket with the largest criticality and make a rebalancing operation (fusion or split). The update of the Lazy B-tree is performed incrementally (i.e., in a step-by-step manner) during the next $O(\log_B N)$ update operations and until the next rebalancing operation. The global rebalancing lemma ensures that the bucket size will never be larger than $O(\log^2 N)$. □

To realize the Lazy B-tree, the following problems must be tackled: (1) the representation of the criticalities of the buckets; and (2) the representation of each bucket.

To understand how the maintenance of criticalities can be achieved, consider a set S of k objects and assume that each object is assigned a value between 0 and $\log^2 k$. In [19], it is explained how this set can be maintained so that object insertions or deletions are accomplished within $O(\frac{\log k}{B})$ block transfers, how an object value can be incremented by 1 in $O(1)$ block transfers, and how the object with the largest element is located and removed in $O(1)$ block transfers. All bounds are worst-case.

Consider now the representation of each bucket. Since buckets may have up to $O(\log^2 N)$ elements, every bucket is a two-layered structure consisting of lists of size $\log N$. The top layer inside a bucket is a list with size at most $\log N$ that guides the

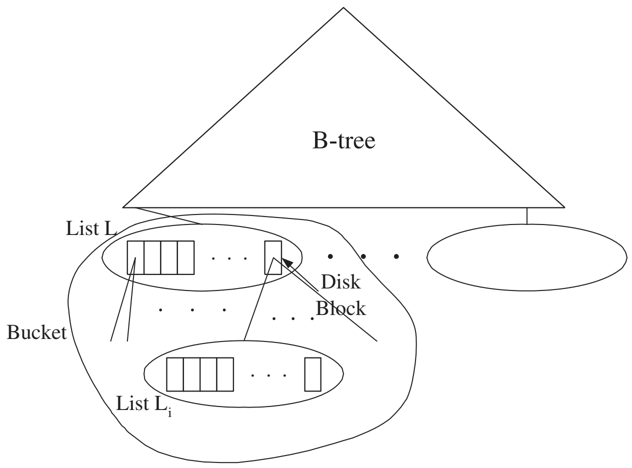


Fig. 4. The Lazy B-tree.

search to the lists of the bottom layer. The lists of the bottom layer inside a bucket store at most $\log N$ elements. These lists are dynamic external sorted lists.

The complicated technical details concerning both the maintenance of criticalities and the representation of buckets, can be found in [19]. Trying to simplify the hidden complicated techniques presented in [19], we provide in Appendix the description of all operations in pseudocode.

5.2. Handling polygon queries when using the Hough-X transform

Our construction is based on an interesting geometric observation that the polygon queries are a special case of the general simplex query and hence can be handled more efficiently without resorting to partition trees.

Let us examine the polygon (4-sided) indexability of Hough-X transformation. Our crucial observation is that the query polygon has the nice property of being divided into orthogonal objects, i.e., orthogonal triangles or rectangles, since the lines $X = U_{\min}$ and $X = U_{\max}$ are parallel.

We depict schematically the three basic cases that validate our observation.

5.2.1. Case I

Fig. 5 depicts the first case where the polygon query has been transformed to four range queries employing the orthogonal triangles $(P_1P_2P_5), (P_2P_7P_8), (P_4P_5P_6), (P_3P_4P_7)$ and one range query for querying the rectangle $(P_5P_6P_7P_8)$.

5.2.2. Case II

The second case is depicted in Fig. 6. In this case the polygon query has been transformed to two range queries employing the orthogonal triangles $(P_1P_4P_5)$ and $(P_2P_3P_6)$ and one range query for querying the rectangle $(P_2P_5P_4P_6)$.

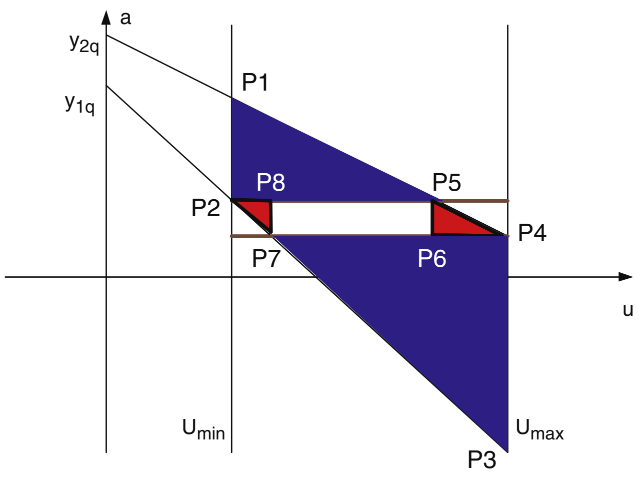


Fig. 5. Orthogonal triangulations: Case I.

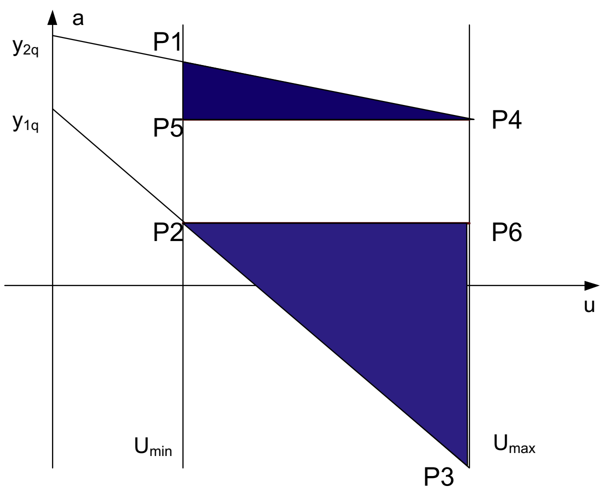


Fig. 6. Orthogonal triangulations: Case II.

5.2.3. Case III

The third case is depicted in Fig. 7. In this case the polygon query has been transformed to two range queries employing the orthogonal triangles $(P_1P_4P_5)$ and $(P_2P_3P_6)$ and one range query for querying the rectangle $(P_2P_1P_5P_6)$.

5.3. Orthogonal 4-sided range queries: dynamic case

The problem of handling orthogonal range searching queries has been handled in [5], where an optimal solution was presented to handle general (4-sided) range queries in $O((N/B)(\log(N/B))\log\log_B N)$ disk blocks and could answer queries in $O(\log_B N + k)$ I/O's; the structure also supports updates in $O((\log_B N)(\log(N/B))/\log\log_B N)$ I/O's.

5.4. Orthogonal triangle queries: static case and dynamization

Let us now consider the problem of devising an access method for handling orthogonal triangle range queries; in this problem we have to determine all the points from a set S of N points on the plane lying inside an orthogonal triangle. Recall that a triangle is orthogonal if two of its edges are axis-parallel. A basic ingredient of our construction will be a structure for handling half-plane range queries, i.e., queries that ask for the reporting all the points in a set S of n points in the plane that lie on a given side of a query line L.

A main memory static solution is presented in [10], which answers half-plane range queries in optimal $O(\log N + K)$ time and linear space using the notion of duality. The main steps of this algorithm are the following:

- Preprocessing
  (1) Partition S into a set of convex layers:
    - Define S_i as the convex hull of all the points currently in S,
    - Remove the vertices of S_i from S,
    - Increment i, repeat the process.

    The time cost is $O(N\log N)$, whereas the space complexity is $O(N)$, using a technique that computes convex hulls in a dynamic environment.

  (2) Augment the set of layers building vertical connections as follows: for each vertex w of layer S_i, keep a pointer to the two edges immediately above and below w. This clearly uses $O(N)$ extra space.

  (3) Using duality, the transformation of each vertex w into its corresponding line maps each layer into another convex polygon. The produced mapping is organized into a point location structure, occupying $O(N)$ space.
- Query processing
  (1) Given a query line L transform it into its corresponding dual point P_L.

  (2) Apply a planar point location algorithm for the point P_L in the properly organized structure. This determines the innermost layer among the layers containing the point P_L. Thus, in the dual mapping it determines the innermost layer among the layers that L intersects. Call this layer *adjacent*. Using an optimal point location algorithm, this costs $O(\log N)$ time.

  (3) Using the pointers mentioned at Step 2 of the preprocessing procedure, it is easy to report one vertex lying at the query half plane for each layer enclosing the adjacent one.

  (4) Traverse each layer from each of the vertices reported across the part of the layer inside the half plane. Report the vertices traversed.

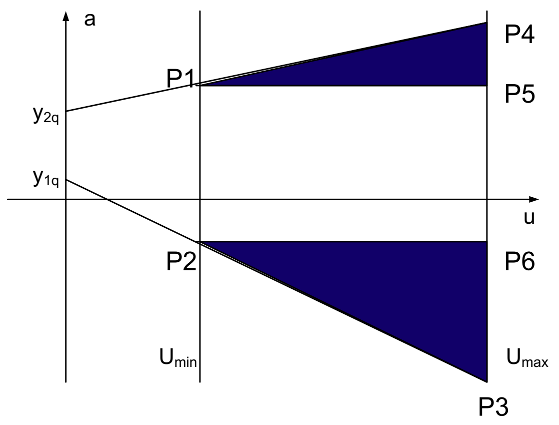


Fig. 7. Orthogonal triangulations: Case III.

Clearly, these steps lead to a static algorithm for answering half-plane range queries using $O(N)$ space and $O(\log N + K)$ query time. In general, known static simplex range searching data structures like the one presented above [10] or other variants, like the algorithms of Clarkson [12] and Matousek [26], can be dynamized easily. Agarwal et al. [2] presented efficient dynamic algorithms for d-dimensional half-space range searching. In particular, they presented a linear access method to answer a half-space range reporting query in time $O(\frac{N}{m^{\lfloor d/2 \rfloor}} \log N + K)$, and insert or delete a point in amortized time $O(m^{1+\epsilon}/N)$, with m a given parameter, $N \leqslant m \leqslant N^{\lfloor d/2 \rfloor}$.

The above static main memory constructions were extended to external memory in [1], where the first optimal access method was presented to answer 2d half-space range queries in the worst-case, based on the geometric technique called *filtering search* [11]. It used $O(n)$ blocks of space and answered a query with $O(\log_B n + k)$ I/Os. The dynamization of the above static solution is viable by using the external version of *logarithmic method* presented in [4,6] (more details about the *internal logarithmic method* can be found in [7,27]). In particular, we have the following:

Theorem 2. *Let P be an external-decomposable problem on a set V of size N. Let D be a linear-space static structure for P that can be constructed in $O((N/B)\log_{M/B}(N/B))$ I/Os, such that queries can be answered in $O(\log_B^{k_q} N)$ I/Os and such that deletes can be performed in $O(\log_B^{k_d} N)$ I/Os, where $k_q \geqslant 0$, M the main memory capacity (for simplicity we make the very realistic assumption that the main memory is capable of holding B^2 elements). There exists a linear-space dynamic access method D′ for P that answers queries in $O(\log_B^{k_q+1} N)$ I/Os, and supports insertions and deletions in $O(\log_B N \cdot \log_{M/B}(N/B) + \log_B^2 N)$ and $O(\log_B N + \log_B^{k_d} N)$ I/Os amortized, respectively.*

Proof. See [6]. □

Lemma 1. *The optimal static access methods presented in [1] requires a construction time of $O(NB^{1/3}(\log_2 N)\log_B^{4/3} n)$ I/Os.*

Proof. See [1]. □

Theorem 3. *The dynamic version of [1] requires $O(\log_B n \log_B N)$ I/Os for answering half-space range queries and $O(B^{4/3} \log N \log_B^{4/3} n \log_B N)$ amortized I/Os for supporting insertions.*

Proof. Theorem 2 implies that we have to pay an extra logarithmic overhead of $O(\log_B N)$ I/Os for query time. Furthermore, the static solution does not support deletions, as a consequence its dynamization supports only insertions. The construction time given in Lemma 1, can be rewritten as follows:

$O((N/B)B^{4/3}\log_2 N \log_B^{4/3} n)$. According to Theorem 2, the I/O amortized complexity for the *insert* operation becomes: $O(B^{4/3}\log_2 N \log_B^{4/3} n \log_B N + \log_B^2 N)$ or $O(B^{4/3}\log_2 N \log_B^{4/3} n \log_B N)$.

We will use this dynamic method to satisfy orthogonal triangle range queries on points. □

Let us now return to our initial problem, i.e., the devise of a structure suitable for handling orthogonal triangle range queries. Recall, a triangle is orthogonal if two of its edges are axis-parallel. Let T be an orthogonal triangle defined by the point (x_q, y_q) and the line L_q that is not axis-parallel (see Fig. 8). A retrieval query for this problem can be supported efficiently by the following 3-layered access method.

To set-up the access method (see Fig. 9), we first sort the N points according to their x-coordinates and then store the ordered sequence in a leaf-oriented balanced binary search tree of depth $O(\log N)$. This structure answers the query "determine the points having x-coordinates in the range $[x_1, x_2]$ by traversing the two paths to the leaves corresponding to x_1, x_2". The points stored as leaves at the subtrees of the nodes which lie between the two paths are exactly these points in the range $[x_1, x_2]$. For each subtree, the points stored at its leaves are organized further to a second level structure according to their y-coordinates in the same way. For each subtree of the second level structure, the points stored at its leaves are organized further to a third level structure as in [1,10] for half-plane range queries. Thus, each orthogonal triangle range query is performed through the following steps:

(1) In the tree storing the pointset S according to x-coordinates, traverse the path to x_q. All the points having x-coordinate in the range $[x_q, \infty)$ are stored at the subtrees on the nodes that are right sons of a node of the search path and do not belong to the path. There are at most $O(\log N)$ such disjoint subtrees.
(2) For every such subtree traverse the path to y_q. By a similar argument as in the previous step, at most $O(\log N)$ disjoint subtrees are located, storing points that have y-coordinate in the range $[y_q, \infty)$.
(3) For each subtree in Step 2, apply the half-plane range query of [1,10] to retrieve the points that lie on the side of line L_q towards the triangle.

The correctness of the above algorithm follows from the structure used. In each of the first two steps we have to visit $O(\log N)$ subtrees. If in Step 3 we apply the dynamic version of the external memory solution presented in [1], then our method requires $O(\log^2 N \log_B n \log_B N) + K)$ I/Os for answering the query, $O(N \log^2 N)$ disk blocks and $O(B^{4/3}\log^3 N \log_B^{4/3} n \log_B N)$ I/Os for insertion operations in the amortized case.

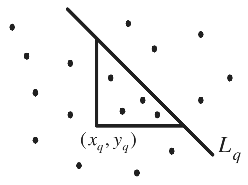


Fig. 8. The query triangle.

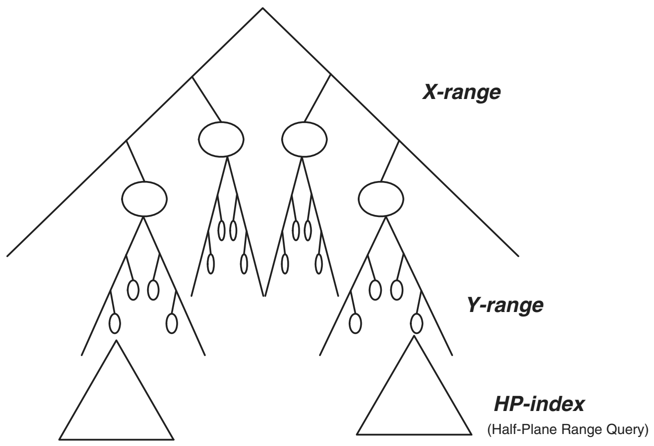


Fig. 9. Orthogonal triangle indexing.

6. Experimental evaluation

The structure presented in [1] is complicated, not easily implemented and, thus, inefficient in practice. In particular, the third complicated component concerning the dynamic half-plane range query in external memory entails large hidden factors and it still remains open how this solution could be practically simplified. For this reason, the solution presented in Section 4.2 is interesting from a theoretical point of view only. On the other hand, as implied by the following experiments, the solution presented in Section 4.1 is very efficient in practice. This section compares the query/update performance of our solution with STRIPES (the best known solution) as well as with those ones that use B^+-trees and TPR*-tree, respectively. We deploy spatiotemporal data that contain insertions at a single timestamp 0. In particular, objects' MBRs are taken from the LA spatial dataset (128971 MBRs).[1] We want to simulate a situation where all objects move in a space with dimensions 100×100 km. 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 movement, (b) the velocity value distribution is skewed (Zipf) towards 0 in range [0,50], and (c) the velocity can be either positive or negative with equal probability. As in [8], we will use a small page size so that the number of index nodes simulates realistic situations. Thus, for all experiments, the page size is 1 Kbyte, the key length is 8 bytes, whereas the pointer length is 4 bytes. Thus, the maximum number of entries ($\langle x \rangle$ or $\langle y \rangle$, respectively) in both Lazy B-trees and B^+-trees is 1024/(8 + 4) = 85. In the same way, the maximum number of entries (2d rectangles or $\langle x_1, y_1, x_2, y_2 \rangle$ 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 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*. The fanout of each non-leaf node is thus 16. For each dataset, all indexes except for STRIPES have similar sizes. Specifically, for LA, each tree has four levels and around 6700 leaves apart from STRIPES index which has a maximum height of seven and consumes about 2.4 times larger disk space. 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.

6.1. Query cost comparison

We measure the performance of our technique earlier described (LBTs, in particular two Lazy B-trees, one for each projection, plus the query processing between the two answers), the traditional technique (B^+-trees, in particular two B^+-trees, one for each projection, plus the query processing between the two answers) presented in [21,29] and TPR*-tree and STRIPES

[1] Downloaded from the Tiger website http://www.census.gov/geo/www/tiger/.

presented in [40,30]], respectively, using the same query workload, after every 10,000 updates. Figs. 9–13 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 figures concerning query costs our solution is almost the same efficient as the solution using B^+-trees. This is an immediate result of the same time complexity of searching procedures in both structures B^+-tree and Lazy B-tree, respectively. In particular, we have to index the appropriate b parameters in each projection and then combine the two answers by detecting and filtering all the pair permutations. Obviously, the required number of block transfers depends on the answer's size and is exactly the same in both solutions for all conducted experiments.

Fig. 10 depicts the efficiency of our solution in comparison to that of the TPR*-tree and STRIPES. In Fig. 9 (top), where the length of the query rectangle is 100 and as a consequence the query's surface is equal to 10,000 m^2 or 1 ha (the surface of the whole spatial terrain is 10^6 ha) the LBTs method is 26.56 times faster than STRIPES and 106.25 times faster than TPR*-tree. Our solution's performance degrades as the query rectangle length grows from 100 to 1000. Thus, in Fig. 9(bottom) where the spatial query's surface is equal to 100 ha, our method is even 2.12 times faster than STRIPES and 8.01 times faster than TPR*-tree.

When the query rectangle length or equivalently the query surface becomes extremely large (e.g. 2000, or equivalently 400 ha), then the STRIPES index shows better performance (see Fig. 11). In particular, our method is 1.9 times faster than the TPR*-tree; however, STRIPES is twice faster than our method. Apparently, while the surface of the query rectangle grows, the answer size in each projection grow as well, thus the performance of the LBTs method that combines and filters the two answers may degrade. In real GIS applications, for a vast spatial terrain of 10^6 ha, e.g. the road

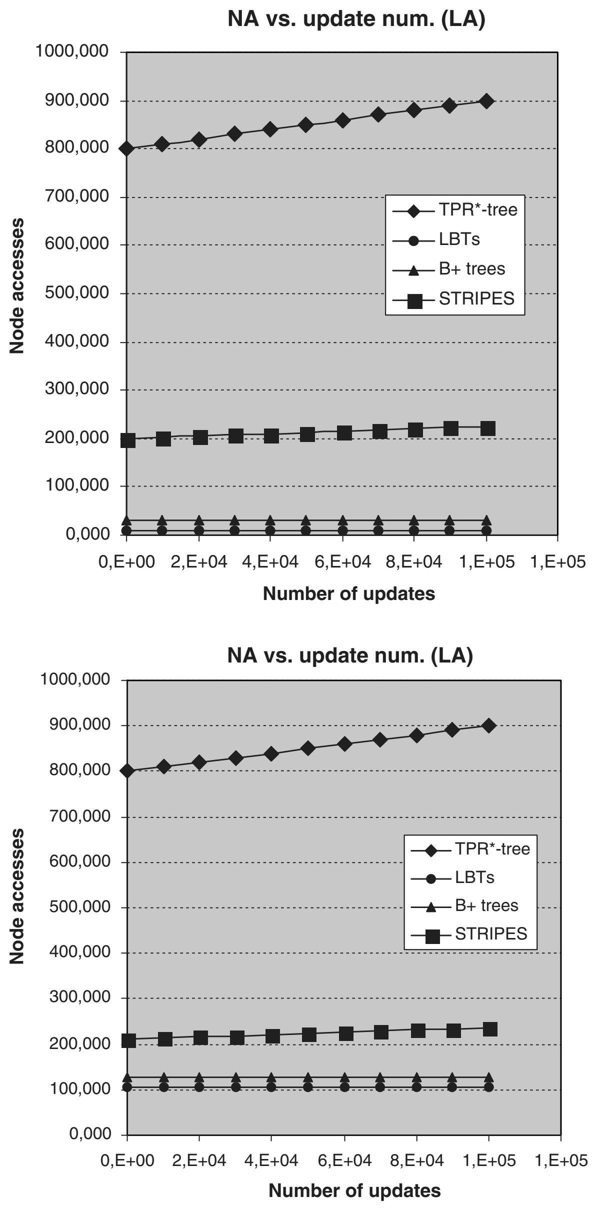


Fig. 10. $q_V len = 5, q_T len = 50, q_R len = 100$ (top), $q_R len = 1000$ (bottom).

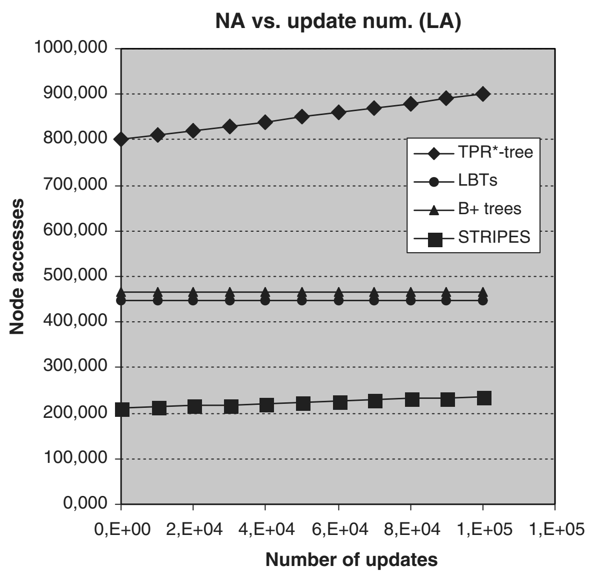
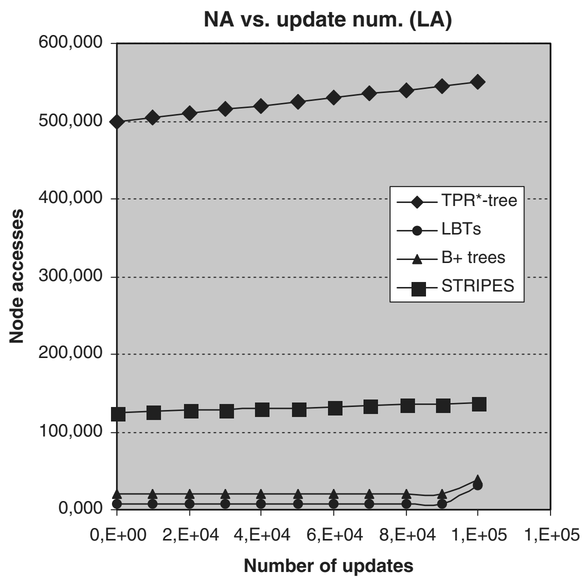


Fig. 11. $q_R len = 2000, q_V len = 5, q_T len = 50$.



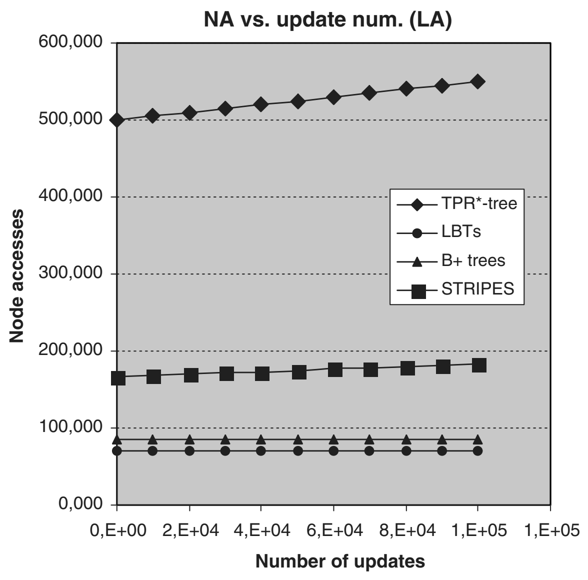
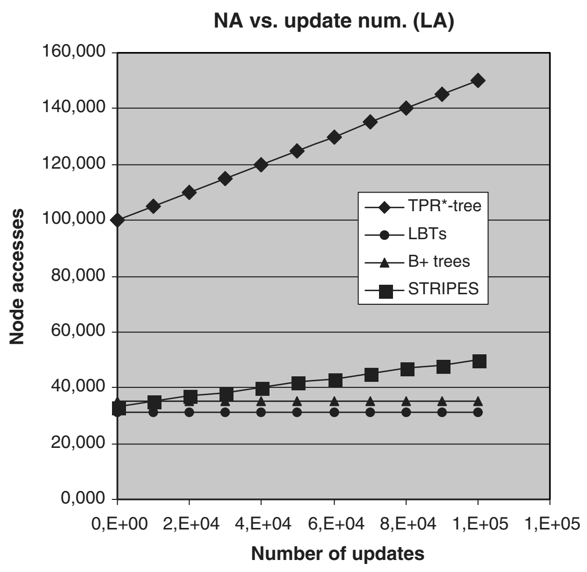


Fig. 12. $q_V len = 10, q_T len = 50, q_R len = 400$ (top), $q_R len = 1000$ (bottom).



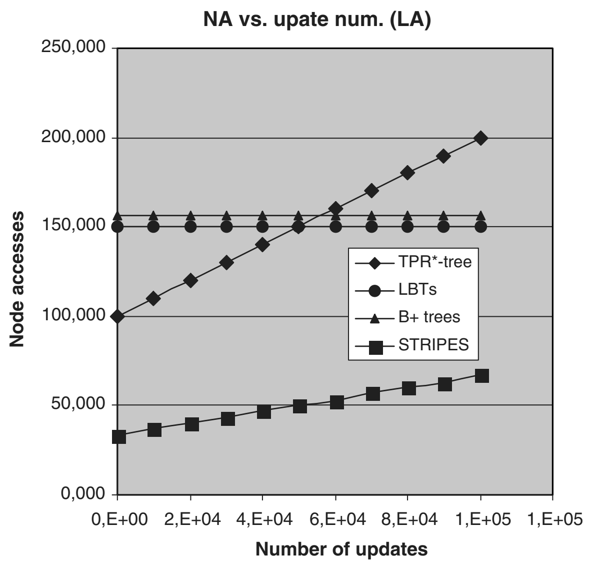


Fig. 13. $q_V len = 5$, $q_T len = 1$, $q_R len = 400$ (top), $q_R len = 1000$ (bottom).

network of a big town where each road square covers no more than 1 ha (or 10,000 m^2) the most frequent queries consider spatial query's surface no more than 100 road squares (or 100 ha) and future time interval no larger than 100 s. This is what we consider as *realistic* scenarios.

Fig. 12 depicts the performance of all methods for a growing velocity vector. In particular, in Fig. 11(top) the LBTs method is 16.5 times faster than STRIPES and 68.75 times faster than TPR*-tree. The performance of our solution degrades as the query rectangle length grows from 400 (16 ha of query surface) to 1000 (100 ha of query surface). Even in the latter case, our method is 2.5 times faster than STRIPES and 7.85 times faster than TPR*-tree. Obviously, the velocity factor is very important for TPR-like solutions, but not for the other methods, for LBTs in particular, which depend exclusively on the query surface.

Fig. 13 depicts the performance of all methods in case the time interval length degrades to value 1. Even in this case (Fig. 12(top)), the LBTs method is 1.35 times faster than STRIPES and 4.03 times faster than TPR*-tree. As query rectangle length grows from 400 to 1000, the LBTs method advantage decreases; from Fig. 12(bottom) we remark that STRIPES is 3 times faster, whereas our method has exactly the same performance with the TPR*-trees.

Fig. 14 depicts the efficiency of our solution in comparison to that of TPR*-trees and STRIPES, respectively in case the time interval length enlarges to 100. In particular, the LBTs method is 5.37 times faster than STRIPES and 18.75 times faster than TPR*-tree. Apparently, the query surface remains at *realistic* levels (16 ha).

6.1.1. Update cost comparison

Fig. 15 compares the average cost (amortized over insertions and deletions) as a function of the number of updates. The LBTs method (Lazy *B*-trees for the *x*- and *y*-projections) has optimal update performance and consistently outperforms the TPR*-tree by a wide margin as well as the STRIPES index by a narrow margin. In particular, our method requires a constant

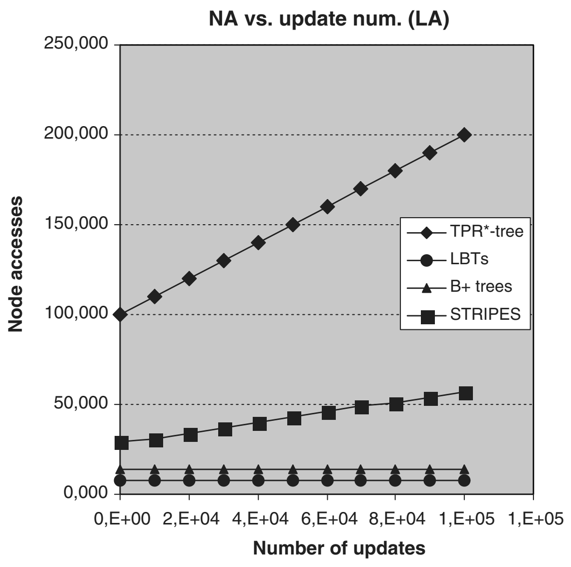


Fig. 14. $q_R len = 400, q_V len = 5, q_T len = 100$.

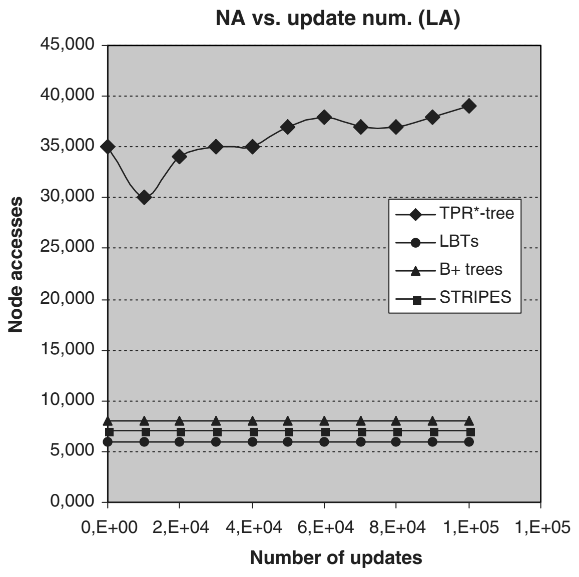


Fig. 15. Update cost comparison.

number of 6 block transfers (3 block transfers for each projection, for details see [19]) and this update performance is independent of the dataset size. On the other hand, the other three solutions do not have constant update performance; instead their performance depends on the dataset size even if as in the experiment of Fig. 15 B^+-trees and STRIPES reach the optimal performance of LBTs method requiring 8 and 7 block transfers, respectively (TPR*-tree requires 35 block transfers in average).

According to our theoretical outcomes, the solution of LBTs outperforms the update performance of B^+-trees by a logarithmic factor; however, this is not depicted clearly in Fig. 15 due to small datasets. For this reason we performed additional experiments with gigantic synthetic datasets of size $N_0 \in [10^6, 10^{12}]$. In particular, we initially have 10^6 mobile objects and during the experiment we continuously insert new objects until their number became 10^{12}. For each object, we considered a synthetic linear function where the velocity value distribution was skewed (zipf) towards 30 in the range [30, 50]. The velocity was either positive or negative with equal probability. For simplicity, all objects were stored using the Hough-*Y* dual transform. This assumption is also realistic, since in practice the number of mobile objects, which are moving with very small velocities, is negligible.

Due to the gigantic synthetic dataset we increased the page size from 1 KB to 4 KB. Since the length of each key is 8 bytes and the length of each pointer is 4 bytes the block capacity became 341. In particular, Fig. 16 establishes that in our solution, the number of block transfers for the update operations will remain constant even for gigantic datasets. This fact is an immediate result of the time complexity of update procedures in the Lazy *B*-tree.

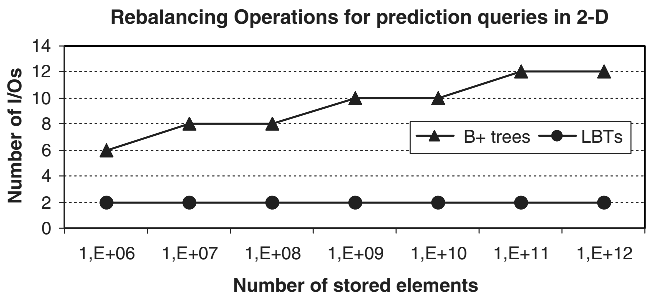


Fig. 16. Rebalancing operations comparison between solutions that use B^+- and Lazy B-trees, respectively for the particular problem of 2d prediction queries.

7. Conclusions

We presented access methods for indexing mobile objects that move on the plane to efficiently answer range queries about their future location. Concerning our first solution, it has been proved that its update performance is the most efficient in all cases. Regarding the query performance, the superiority of our structure has been shown as far as the query rectangle length remains in realistic levels (by far outperformance in comparison to opponent methods). If the query rectangle length becomes extremely huge in relation to the whole terrain, then STRIPES is better than any other solution, however, only to a small margin in comparison to our method. Finally, the second presented solution, although complicated, has been proved to be theoretically efficient. Thus, our future plan is to simplify it into an implementable version and as a consequence a practically applicable structure.

Acknowledgements

This work has been partially funded by the *Greek-Bulgarian Bilateral Scientific Protocol.*

Appendix

For the better understanding of the pseudocodes below, see also the Fig. 17

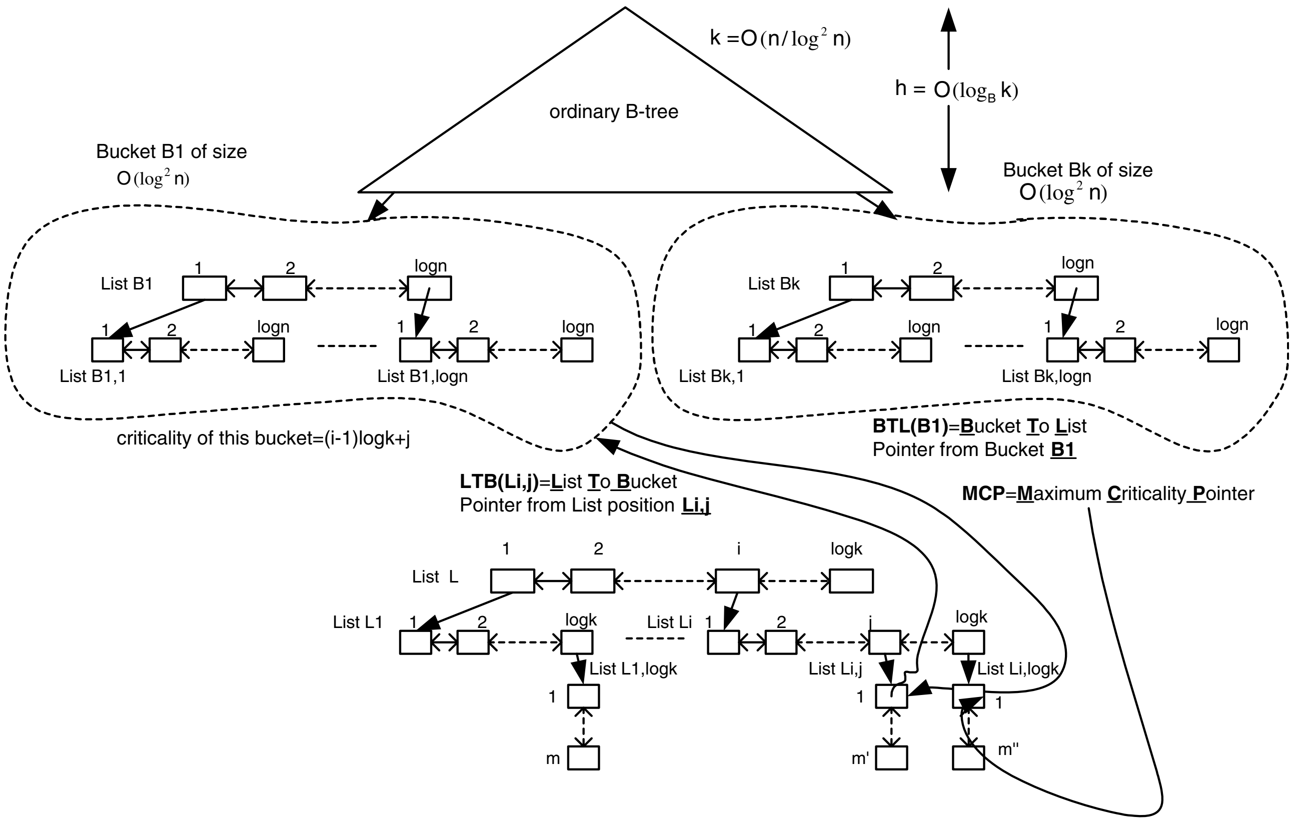

k = O(n/log² n)
h = O(log_B k)
ordinary B-tree
Bucket B1 of size O(log² n)
Bucket Bk of size O(log² n)
List B1
List B1,1
List B1,logn
List Bk
List Bk,1
List Bk,logn
1
2
logn
criticality of this bucket=(i-1)logk+j
BTL(B1)=Bucket To List Pointer from Bucket B1
LTB(Li,j)=List To Bucket Pointer from List position Li,j
MCP=Maximum Criticality Pointer
List L
List L1
List Li
List L1,logk
List Li,j
List Li,logk
i
logk
m
m'
m''

Fig. 17.

Algorithm 1: List_Search(x, E)

```
1: Linear Scan of list $E$
2: Return the block $b$ containing the item $x$
```

Algorithm 2: List_Insert(x, E))

```
1: List_Search($x$,$E$)) {Locate the corresponding block $b$}
2: if $|b| \leqslant B$ then
3:    insert $x$ into $b$
4: else
5:       create a new adjacent block $b'$
6:       move half the elements of $b$ to $b'$
7: endif
```

Algorithm 3: List_Delete(x, E))

```
1: List_Search($x$,$E$)) {Locate the corresponding block $b$}
2: remove $x$ from $b$
3: if $|b| \leqslant B/2$ then
4:       Let $b'$ the adjacent block
5:       if $|b + b'| \leqslant B$ then
6:           $fuse(b, b')$
7:       else
8:           transfer some elements from $b'$ to $b$
9:       endif
10: endif
```

Algorithm 4: List_Inc($x, L_{i,j}$)

```
1: if $j < \log k$ then
2:    move $x$ from $L_{i,j}$ to $L_{i,j+1}$
3: endif
4: if $j = \log k$ then
5:    move $x$ to $L_{i+1,1}$
6: endif
```

Algorithm 5: Bucket_Add(B_i, L)

```
1: Allocate a new pointer $BTL(B_i)$ {from bucket $B_i$}
2: $BTL(B_i) = L_{1,1}$ {To the head of list $L$}
3: Allocate a new pointer $LTB(L_{1,1})$ {from the head of list}
4: $LTB(L_{1,1}) = B_i$ {to bucket $B_i$}
```

Algorithm 6: Bucket_Remove(B_i, L)

```
1: free the pointer $BTL(B_i)$
2: $BTL(B_i) = NULL$
3: free the pointer $LTB(L_{i,j})$ {Let $L_{i,j}$ the list position which represents the current criticality of bucket $B_i$}
4: $LTB(L_{i,j}) = NULL$
```

Algorithm 7: Bucket_RemoveMax(L)

```
1: follow the $MCP$ pointer {Let $L_{\log k,j}$ the corresponding list position}
2: follow the $LTB(L_{\log k,j})$ pointer {Let $B_i$ the corresponding bucket into which we must apply the rebalancing operations}
3: if $B_i$ and an adjacent $B_i'$ are fused then
4:       Bucket_Remove($B_i, L$)
5: endif
6: if there is a transfer between $B_i$ and its adjacent $B_i'$ then
7:       compute the new criticalities of $B_i$ and $B_i'$, respectively
8:       update the corresponding $BTL$ and $LTB$ pointers from and to $B_i$ and $B_i'$, respectively
9: endif
10: if $B_i$ is split to $B_i$ and $B_i'$ then
11:       Bucket_Add($B_i', L$)
12: endif
```

Algorithm 8: Bucket_Rebalancing(B_i)

```
1: $\Phi(B_i) = \frac{|B_i|}{\log^2 n}$
2: if $\Phi(B_i) > 1.8$ then
3:     split the bucket $B_i$ into two parts of approximately equal size, $B_i$ and $B_i'$
4: endif
5: if $\Phi(b) < 0.7$ then
6:     if one of its adjacent buckets $b'$ has $\Phi(b') \geqslant 1$
7:         transfer elements from $b'$ to $b$
8:     else
9:         fuse with an adjacent bucket $b'$
10:    endif
11: endif
12: Bucket_RemoveMax($L$)
```

Algorithm 9: LazyTree_Insert(x, LBT)

```
1: search($x$,$B$-tree) {Let $B_l, 1 \leqslant l \leqslant k$ the corresponding bucket and $B_l$ the corresponding 1st layer list}
2: Bucket_Search($x, B_l$) {Let $B_{l,r}$ the corresponding 2nd layer list}
3: List_Insert($x, B_{l,r}$)
4: $\gamma(B_l, n) = \frac{1}{\alpha \log n}$ max$\{0.7\log^2 n - |B_l|, |B_l| - 1.8\log^2 n\}$
5: if $\gamma(B_l, n)$ has been increased by one then
6:     follow the $BTL(B_l)$ pointer {Let $L_{i,j}$ the corresponding list}
7: List_Inc($\gamma(B_l, n), L_{i,j}$)
8: endif
9: numberofupdates $\leftarrow$ numberofupdates $+ 1$
10: if numberofupdates $= \alpha\log_B n$ then
11:     Bucket_Rebalancing($LBT$) {$n$ is the number of total elements at the beginning of the epoch}
12: endif
```

Algorithm 10: LazyTree_Delete(x, LBT)

```
1: search($x$,$B$-tree) {Let $B_l, 1 \leqslant l \leqslant k$ the corresponding bucket and $B_l$ the corresponding 1st layer list}
2: Bucket_Search($x, B_l$) {Let $B_{l,r}$ the corresponding 2nd layer list}
3: List_Delete($x, B_{l,r}$)
4: $\gamma(B_l, n) = \frac{1}{\alpha \log n}$ max$\{0.7\log^2 n - |B_l|, |B_l| - 1.8\log^2 n\}$
5: if $\gamma(B_l, n)$ has been increased by one then
6:     follow the $BTL(B_l)$ pointer {Let $L_{i,j}$ the corresponding list}
7:     List_Inc($\gamma(B_l, n), L_{i,j}$)
8: endif
9: number of updates $\leftarrow$ number of updates $+ 1$
10: if number of updates $= \alpha\log_B n$ then
11:     Bucket_Rebalancing($LBT$) {$n$ is the number of total elements at the beginning of the epoch}
12: endif
```

Algorithm 11: LazyTree_Search(x,Bin_i)

```
1: search($x$,$B$-tree) {Let $B_l, 1 \leqslant l \leqslant k$ the corresponding bucket and $B_l$ the corresponding 1st layer list}
2: List_Search($x, B_l$) {Let $B_{l,r}$ the corresponding 2nd layer list}
3: Linear Scan of list $B_{l,r}$
4: Return the corresponding block $b$
```

References

[1] P.K. Agarwal, L. Arge, J. Erickson, P.G. Franciosa, J.S. Vitter, Efficient searching with linear constraints, Journal of Computer and System Sciences 61 (2) (2000) 194–216.

[2] P.K. Agarwal, D. Eppstein, J. Matousek, Dynamic half-space reporting, geometric optimization, and minimum spanning trees, in: Proceedings of the 33rd Annual Symposium on Foundations of Computer Science (FOCS), Pittsburgh, PA, 1992, pp. 80–89.

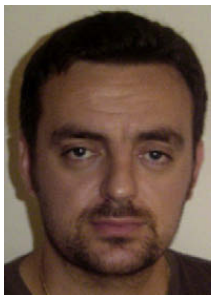
[3] P.K. Agarwal, L. Arge, J. Erickson, Indexing moving points, Proceedings of the 19th ACM Symposium on Principles of Database Systems (PODS), Dallas, TX, 2000, pp. 175–186.

[4] J. Abello, P.M. Pardalos, M.G.C. Resende (Eds.), Handbook of Massive Datasets, Kluwer Academic Publishers, 2001 (Chapter 1).

[5] L. Arge, V. Samoladas, J.S. Vitter, On two-dimensional indexability and optimal range search indexing, in: Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS), Philadelphia, PA, 1999, pp. 346–357.

[6] L. Arge, J. Vahrenhold, I/O-efficient dynamic planar point location, Computational Geometry 29 (2) (2004) 147–162.

[7] J.L. Bentley, Decomposable searching problems, Information Processing Letters 8 (5) (1979) 244–251.
[8] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger, The R^*-tree: an efficient and robust access method for points and rectangles, in: Proceedings of the ACM International Conference on Management of Data (SIGMOD), Atlantic City, NJ, 1990, pp. 322–331.
[10] B. Chazelle, L. Guibas, D.L. Lee, The power of geometric duality, in: Proceedings of the 24th IEEE Annual Symposium on Foundations of Computer Science (FOCS), Tucson, AZ, 1983, pp. 217–225.
[11] B. Chazelle, Filtering search: a new approach to query answering, SIAM Journal on Computing 15 (3) (1986) 703–724.
[12] K.L. Clarkson, New applications of random sampling in computational geometry, Discrete Computational Geometry 2 (1987) 195–222.
[14] D. Comer, The ubiquitous *B*-tree, ACM Computing Surveys 11 (2) (1979) 121–137.
[15] P. Dietz, R. Raman, A constant update time finger search tree, Information Processing Letters 52 (3) (1994) 147–154.
[16] V. Gaede, O. Gunther, Multidimensional access methods, ACM Computing Surveys 30 (2) (1998) 170–231.
[17] J. Goldstein, R. Ramakrishnan, U. Shaft, J.B. Yu, Processing queries by linear constraints, in: Proceedings of the 16th ACM Symposium on Principles of Database Systems (PODS), Tucson, AZ, 1997, pp. 257–267.
[19] A. Kaporis, C. Makris, S. Sioutas, A. Tsakalidis, K. Tsichlas, K. Zaroliagis, ISB-tree: a new indexing scheme with efficient expected behaviour, in: Proceedings of the 13th International Symposium on Algorithms and Computation (ISAAC), Sanya, Hainan, China, 2005, pp. 318–327.
[21] G. Kollios, D. Gunopulos, V. Tsotras, On indexing mobile objects, in: Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS), Philadelphia, PA, 1999, pp. 261–272.
[23] S. Levcopoulos, M.H. Overmars, Balanced search tree with O(1) worst-case update time, Acta Informatica 26 (3) (1988) 269–277.
[24] Y. Manolopoulos, *B*-trees with Lazy parent split, Information Sciences 79 (1–2) (1994) 73–88.
[25] Y. Manolopoulos, Y. Theodoridis, V. Tsotras, Advanced Database Indexing, Kluwer Academic Publishers, 2000.
[26] J. Matousek, Reporting points in halfspaces, in: Proceedings of the 32nd IEEE Symposium on Foundations of Computer Science (FOCS), San Juan, Puerto Rico, 1991, pp. 207–215.
[27] M. Overmars, H. van Leeuwen, Maintenance of configurations in the plane, Journal of Computer Systems and Science 23 (1981) 166–204.
[29] D. Papadopoulos, G. Kollios, D. Gunopulos, V.J. Tsotras, Indexing mobile objects on the plane, in: Proceedings of the 13th International Workshop on Database and Expert Systems Applications (DEXA), Aix-en-Provence, France, 2002, pp. 693–697.
[30] J. Patel, Y. Chen, V. Chakka, STRIPES: an efficient index for predicted trajectories, in: Proceedings of the ACM International Conference on Management of Data (SIGMOD), Paris, France, 2004, pp. 637–646.
[31] R. Raman, Eliminating amortization: on data structures with guaranteed response time, Ph.D. Thesis, Technical Report TR-439, Department of Computer Science, University of Rochester, NY, 1992.
[32] K. Raptopoulou, M. Vassilakopoulos, Y. Manolopoulos, Towards quadtree-based moving objects databases, in: Proceedings of the Eighth East-European Conference on Advanced Databases and Information Systems (ADBIS), Budapest, Hungary, 2004, pp. 230–245.
[33] K. Raptopoulou, M. Vassilakopoulos, Y. Manolopoulos, Efficient processing of past–future spatiotemporal queries, in: Proceedings of the 21st ACM Symposium on Applied Computing (SAC), Minitrack on Advances in Spatial and Image-based Information Systems (ASIIS), Dijon, France, 2006, pp. 68–72.
[34] K. Raptopoulou, M. Vassilakopoulos, Y. Manolopoulos, On past-time indexing of moving objects, Journal of Systems and Software 79 (8) (2006) 1079–1091.
[35] S. Saltenis, C. Jensen, S. Leutenegger, M.A. Lopez, Indexing the positions of continuously moving objects, in: Proceedings of the ACM International Conference on Management of Data (SIGMOD), Dallas, TX, 2000, pp. 331–342.
[36] S. Saltenis, C.S. Jensen, Indexing of moving objects for location-based services, in: Proceedings of the 18th IEEE International Conference on Data Engineering (ICDE), San Jose, CA, 2002, pp. 463–472.
[37] B. Salzberg, V.J. Tsotras, A comparison of access methods for time-evolving data, ACM Computing Surveys 31 (2) (1999) 158–221.
[40] Y. Tao, D. Papadias, J. Sun, The TPR*-tree: an optimized spatio-temporal access method for predictive queries, in: Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), Berlin, Germany, 2003, pp. 790–801.



S. Sioutas was born in Greece, in 1975. He graduated from the Department of Computer Engineering and Informatics, School of Engineering, University of Patras, in December 1997. He received his Ph.D. degree from the Department of Computer Engineering and Informatics, in 2002.

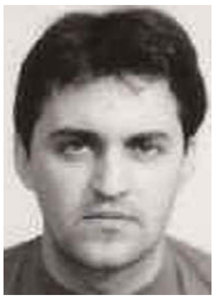
He is now an Assistant Professor in Informatics Department of Ionian University. His research interests include Databases, Computational Geometry, GIS, Data Structures, Advanced Information Systems, P2P Networks and Web Services. He has published over 50 papers in various scientific journals and refereed conferences.

K. Tsakalidis was born in Greece, in 1983. He graduated from the Department of Computer Engineering and Informatics, School of Engineering, University of Patras, in December 2006.

He is now a Ph.D. Student in Computer Science Department, University of Aarhus (Advisor: Gerth S. Brodal). His research interests include Massive Data Algorithmics, I/O complexity, Databases and Data Structures.

K. Tsichlas was born in Greece, in 1976. He graduated from the Department of Computer Engineering and Informatics, School of Engineering, University of Patras, in December 1999. He received his Ph.D. degree from the Department of Computer Engineering and Informatics, in 2004.

He is now a Lecturer in Informatics Department of Aristotle University. His research interests include Data Structures for Main and Secondary Memory, Design and Analysis of Algorithms, Computational Complexity, Computational Geometry, String Algorithmics (Bioinformatics – Music Analysis), Dynamic Graph Algorithms. He has published over 30 papers in various scientific journals and refereed conferences.



C. Makris was born in Greece, in 1971. He graduated from the Department of Computer Engineering and Informatics, School of Engineering, University of Patras, in December 1993. He received his Ph.D. degree from the Department of Computer Engineering and Informatics, in 1997.

He is now an Assistant Professor in the same Department. His research interests include Data Structures, Web Algorithmics, Computational Geometry, Data Bases and Information Retrieval. He has published over 50 papers in various scientific journals and refereed conferences.

Y. Manolopoulos was born in Thessaloniki, Greece in 1957. He received a B.Eng. (1981) in Electrical Engineering and a Ph.D. degree (1986) in Computer Engineering, both from the Aristotle University of Thessaloniki. Currently, he is Professor at the Department of Informatics of the same university. He has been with the Department of Computer Science of the University of Toronto, the Department of Computer Science of the University of Maryland at College Park and the University of Cyprus. He has published over 200 papers in journals and conference proceedings. He is co-author of the books "Advanced Database Indexing", "Advanced Signature Indexing for Multimedia and Web Applications" by Kluwer and of the books "*R*-Trees: Theory and Applications", "Nearest Neighbor Search: a Database Perspective" by Springer. He has co-organized several conferences (among others ADBIS'2002, SSTD'2003, SSDBM'2004, ICEIS'2006, ADBIS'2006, EANN'2007). His research interests include Databases, Data mining, Web Information Systems, Sensor Networks and Informetrics.