Indexing Mobile Objects on the Plane Revisited

S. Sioutas, K. Tsakalidis, K. Tsihlas, C. Makris, and Y. Manolopoulos

Department of Informatics, Ionian University, Corfu, Greece
sioutas@ionio.gr
Computer Engineering and Informatics Department, University of Patras, Greece
{tsakalid,tsihlas,makri}@ceid.upatras.gr
Department of Informatics, Aristotle University of Thessaloniki, Greece
manolopo@csd.auth.gr

Abstract. 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 data structures. Finally, an experimental evaluation is included that shows the performance, scalability and efficiency of our methods.

Keywords: Spatio-Temporal Databases, Indexing.

1 Introduction

This paper focuses on the problem of indexing mobile objects in two dimensions and efficiently answering range queries over the objects locations in the future. 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 when trying to handle this problem; those that deal with discrete and those that deal with continuous movements.

In a discrete environment the problem of dealing with a set of moving objects can be considered to be 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 temporal [30] or/and spatial databases [11]; in [21] it is elegantly exposed how these indexing techniques can be generalized to handle efficiently queries in a discrete spatiotemporal environment. When considering continuous movements there exists a plethora of efficient data structures [2,14,17,22,23,28,29,33].

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 consisted 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

Y. Ioannidis, B. Novikov, and B. Rachev (Eds.): ADBIS 2007, LNCS 4690, pp. 189–204, 2007.

particular, there are approaches based either on R-trees or on Quad-trees as explained in [25,26,27]. 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 [2,17,23] and (b) those that leave the original representation intact by indexing data in their native n-d space [4,22,28,29,33]. The *geometric duality transformation* is a tool heavily used in the Computational Geometry literature, which maps hyper-planes in R^n to points and vice-versa. In this paper we present and experimentally evaluate techniques using the duality transform that are based on previous approaches [17,22] to efficiently index the future locations of moving points on the plane.

In Section 2 we give a formal description of the problem. In Sections 3 and 4 we present our new solutions that outperform the solution presented in [17,22] since they use more efficient indexing schemes. In particular, Section 4 presents two alternative solutions. The first one is very easily implemented and has many practical merits. The second one has only theoretical interest since it uses clever but very complicated data structures, the implementation of which is very difficult and constitutes an open future problem. Section 5 presents an extended experimental evaluation and Section 6 concludes the paper.

2 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. For example, in Figure 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 $t_{now} \leq t_{1_q} \leq t_{2_q}$), given the current motion information of all objects."

3 Indexing Mobile Objects in Two Dimensions

3.1 Indexing Mobile Objects in One Dimension

The Duality Transform. The duality transform, in general, maps a hyperplane h from R^n to a point in R^n 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 1-d case.

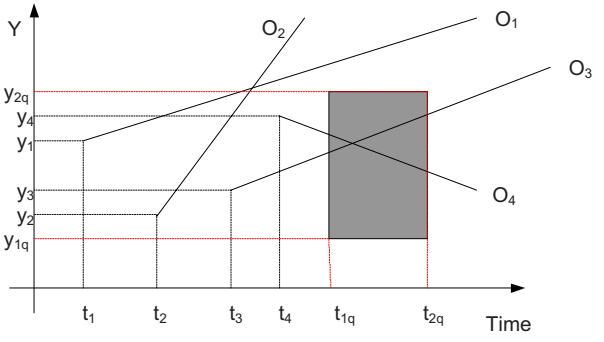

Fig. 1. Trajectories and query in (t, y) plane

Hough-X Transform. 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 objects 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* [17,22]). Accordingly, the 1-d 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 [12], the query in the dual Hough-X plane is expressed as follows (see Figure 2):

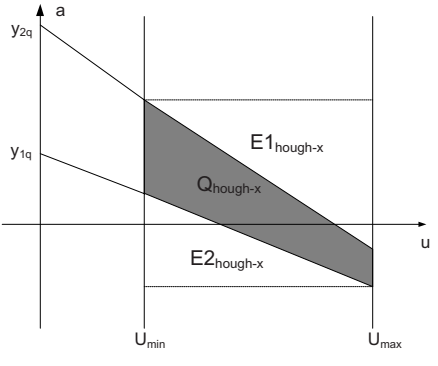

Fig. 2. Query in the Hough-X dual plane

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 following 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.

Hough-Y Transform. By rewriting the equation $y = ut + a$ as $t = \frac{1}{u}y - \frac{a}{u}$, we can arrive to a different dual representation (the so called *Hough-Y transform* in [17,22]). The point in the dual plane has coordinates (b, n), where $b = -\frac{a}{u}$ and $n = \frac{1}{u}$. Coordinate b is the point where the line intersects the line $y = 0$ in the primal space. By using this transform horizontal lines cannot be represented.

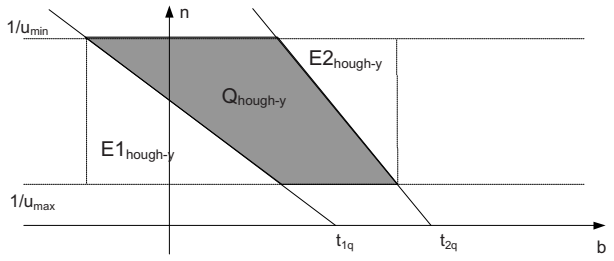

Fig. 3. Query on the Hough-Y dual plane

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.

Similarly, the initial query $[(t_{1_q}, t_{2_q}), (y_{1_q}, y_{2_q})]$ in the (t, y) plane (see Figure 3) can be transformed to the following rectangular query in the (b, n) plane: $[(t_{1_q} - \frac{y_{2_q}}{u_{min}}, t_{2_q} - \frac{y_{1_q}}{u_{max}}), (\frac{1}{u_{max}}, \frac{1}{u_{min}})]$.

3.2 The Proposed Algorithm for Indexing Mobile Objects in Two Dimensions

In [17,22], motions with small velocities in the Hough-Y approach are mapped into dual points (b, n) having large n coordinates ($n = 1/u$). Thus, since few objects can have small velocities, by storing the Hough-Y dual points in an index structure such as an R*-tree, MBR's 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 [17,22] proposes the choice of that transformation which minimizes the following criterion: $c = \frac{E_{Hough-X}}{Q_{Hough-X}} + \frac{E_{Hough-Y}}{Q_{Hough-Y}}$.

The procedure for building the index follows:

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

Partition the objects according to their velocity:

- Objects with small velocity are stored using the Hough-X dual transform, while the rest 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 2-d query follows:

1. Decompose the query into two 1-d 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 that satisfy the query.

In [17,22], $Q_{Hough-X}$ is computed by querying a 2-d partition tree, whereas $Q_{Hough-Y}$ is computed by querying a B$^+$-tree that indexes the b parameters of Figure 3. Our construction instead is based: (a) on the use of the Lazy B-tree [15] instead of the B$^+$-tree when handling queries with the Hough-Y transform and (b) on the employment of a new index that outperforms partition trees in handling polygon queries with the Hough-X transform. In the next section we present the main characteristics of our proposed structures.

4 The Access Methods

4.1 Handling Polygon Queries When Using the Hough-Y Transform

As described in [17,22], polygon queries when using the Hough-Y transform can be approximated by a constant number of 1-d range queries that can be handled by a classical B-tree [9]. 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 [15]. An orthogonal effort towards developing another yet B-tree variant under the same name has been proposed in [20]. The Lazy B-tree of [15] is a simple but non-trivial externalization of the techniques introduced in [24]. In simple words, it is a typical case of a two-level access method as depicted in Figure 4.

The Lazy B-tree operates on the external memory model of computation. The first level consists of an ordinary B-tree, while 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 $1 \leq i \leq O(\log n)$, each of which has $O(\log n)$ size. The rebalancing operations are guided by the *global rebalancing lemma* given in [24] (see also [10,19]). 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 size of the buckets will never be larger than $O(\log^2 n)$.

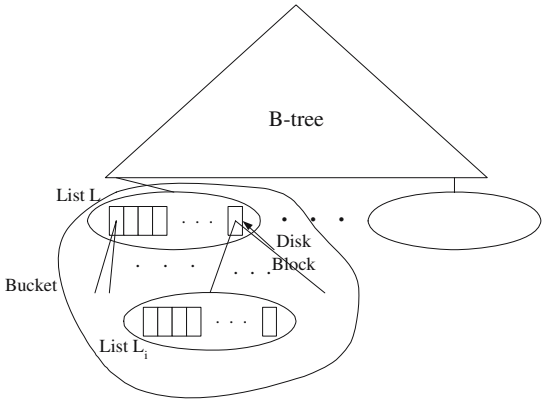

Fig. 4. The Lazy B-tree

Let n be approximately equal to the number of elements stored in the access method, and B be the size of blocks in the external memory. Then:

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.*

4.2 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 justify the validity of our observation.

Case I. Figure 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 $(P_5P_6P_7P_8)$.

Case II. The second case is depicted in the Figure 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 $(P_2P_5P_4P_6)$.

Case III. The third case is depicted in the Figure 7. In this case the polygon query has been transformed to two range queries employing the orthogonal

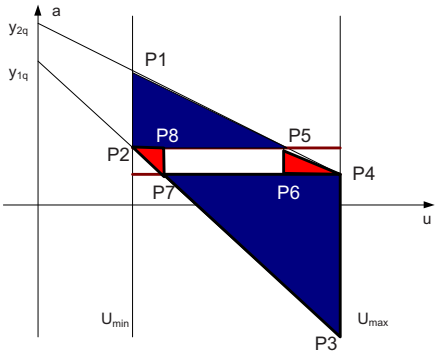

Fig. 5. Orthogonal triangulations: Case I

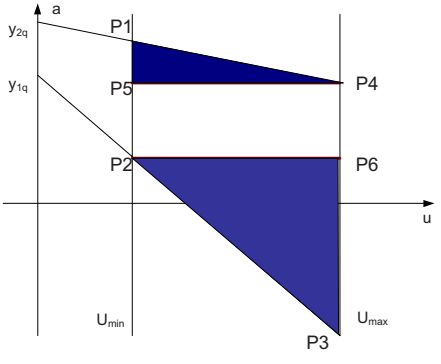

Fig. 6. Orthogonal triangulations: Case II

triangles $(P_1P_4P_5)$ and $(P_2P_3P_6)$ and one range query for querying the rectangle $(P_2P_1P_5P_6)$.

The problem of handling orthogonal range search queries has been handled in [3], 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 $O(\log_B N + T/B)$ I/O's ; the structure also supports updates in $O((\log_B N)(\log(N/B))/\log\log_B N)$ I/O's.

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 solution presented in [6] and achieves optimal $O(\log n + A)$ query time and linear space using the notion of duality. The above main memory

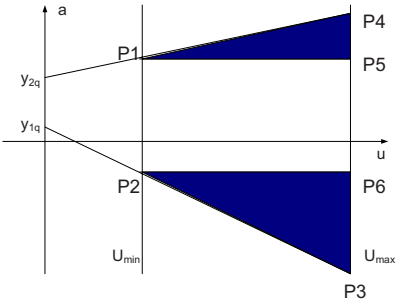

Fig. 7. Orthogonal triangulations: Case III

construction was extended to external memory in [1], where an access method was presented that was the first optimal one for answering 2-d halfpspace range queries in the worst case, based on the geometric technique called *filtering search* [7]. It uses $O(n)$ blocks of space and answers a query using $O(\log_B n + A)$ I/Os, where A is the answer size. We will use these methods 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 Figure 8). A retrieval query for this problem can be supported efficiently by the following 3-layered access method.

To set up the access method, 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

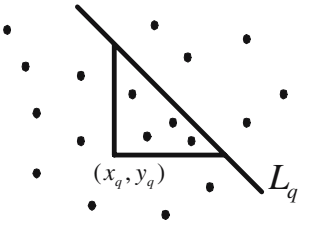

Fig. 8. The query triangle

are organized further to a third level structure as in [1,6] 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 [1,6] 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 main memory solution of [6], then the query time becomes $O(\log^3 n + A)$, whereas the required space is $O(n \log^2 n)$. Otherwise, if we apply the external memory solution of [1], then our method above requires $O(\log^2 n \log_B n + A)$ I/O's and $O(n \log^2 n)$ disk blocks. Although the space becomes superlinear the $O(\log^2 n \log_B n + A)$ worst-case I/O complexity of our method is better than the $O(\sqrt{n/B} + A/B))$ worst-case I/O complexity of a partition tree.

5 Experimental Evaluation

The structure presented in [1] is very complicated and thus it is not easily implemented neither efficient in practice. For this reason, the solution presented in Subsection 4.2 is interesting only from a theoretical point of view. On the other hand, as implied by the following experiments, the solution presented in Subsection 4.1 is very efficient in practice.

This section compares the query/update performance of our solution with those ones that use B$^+$-trees and TPR*-tree [33], respectively. For all experiments, the disk size is set to 1 Kbyte, the key length is 8 bytes, whereas the pointer length is 4 bytes. This means that 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. We use a small page size so that the number of nodes in an index simulates realistic situations. Similar methodology was used in [4]. We deploy spatio-temporal data that contain insertions at a single timestamp 0. In particular, objects' MBRs (Maximum Bounded Rectangles) are taken from the real spatial dataset LA (128971 MBRs) [Tiger], where each axis of the space is normalized to [0,10000]. 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

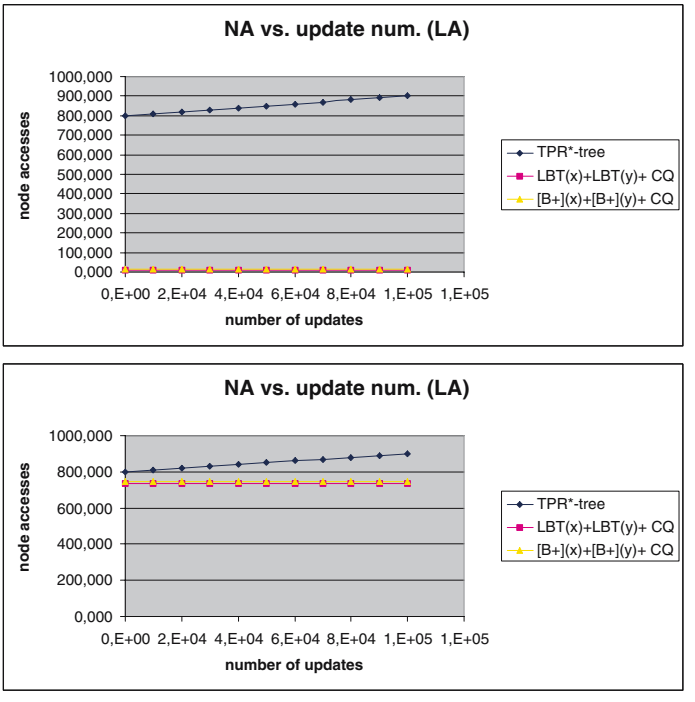

Fig. 9. $q_V len = 5$, $q_T len = 50$ $q_R len = 100$ (top), $q_R len = 2500$ (bottom)

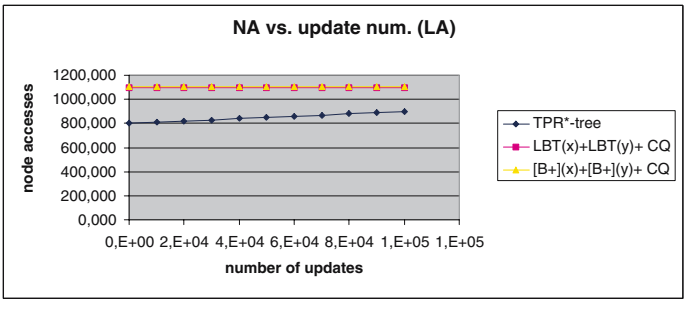

Fig. 10. $q_R len = 3000$, $q_V len = 5$, $q_T len = 50$

in range [0,50], and (c) the velocity can be either positive or negative with equal probability. For each dataset, all indexes have similar sizes. Specifically, for LA, each tree has 4 levels and around 6700 leaves. 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. The main performance metric is measured in number of I/Os.

Query Cost Comparison. We measure the performance of our technique earlier described (two Lazy B-trees, one for each projection, plus the query processing between the two answers), the traditional technique (two B^+-trees, one for each projection, plus the query processing between the two answers) and that one of TPR*-tree, using the same query workload, after every 10000 updates. The following figures 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 ((B+)(x), (B+)(y) plus CQ). This fact is an immediate result of the same time complexity of searching procedures in both structures B^+-tree and Lazy B-trees, respectively. 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 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 experiment.

Figure 9 depicts the efficiency of our solution toward that one of TPR*-tree. The performance of our solution degrades as the length of the query rectangle grows from 100 to 2500. It is almost equally efficient to the solution of B^+-trees.

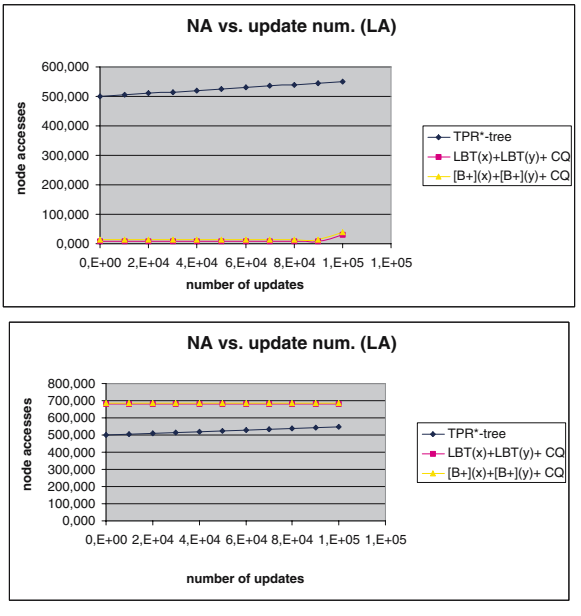

Fig. 11. $q_V len = 10$, $q_T len = 50$, $q_R len = 400$ (top), $q_R len = 2500$ (bottom)

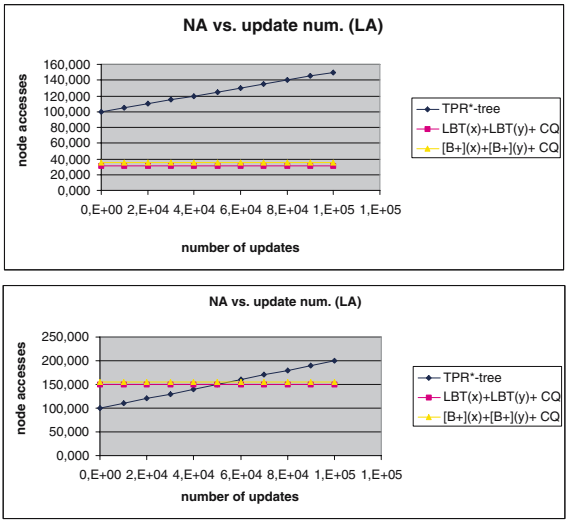

Fig. 12. $q_V len = 5$, $q_T len = 1$, $q_R len = 400$ (top), $q_R len = 1000$ (bottom)

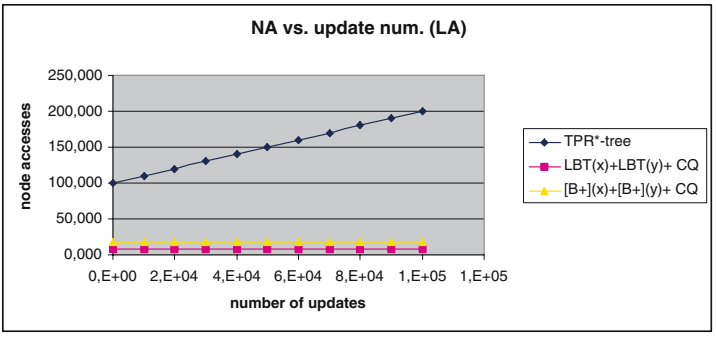

Fig. 13. $q_R len = 400$, $q_V len = 5$, $q_T len = 100$

In Figure 10 the TPR*-tree outperforms the other two solutions since the length of the query rectangle became too large (3000).

Figure 11 depicts the efficiency of our solution towards that one of TPR*-tree in case the velocity vector grows up. The performance of our solution degrades as the length of the query rectangle grows from 400 to 2500. It is almost the same efficient with the solution of B^+-trees.

Figure 12 depicts the efficiency of our solution toward that one of TPR*-tree in case the length of time interval extremely degrades to value 1. The performance of our solution outperforms the TPR*-tree after 50.000 updates have been occurred. It is almost the same efficient as the solution of B^+-trees is.

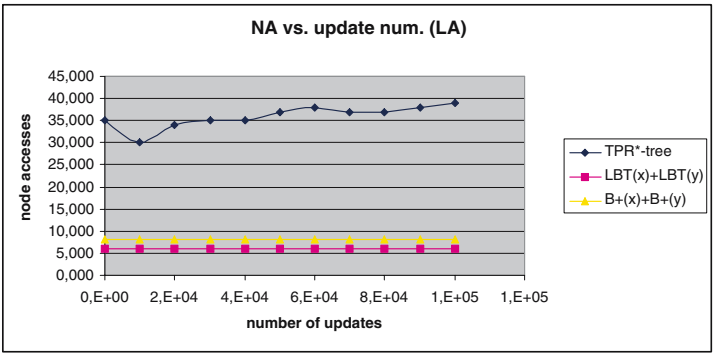

Fig. 14. Update Cost Comparison

Figure 13 depicts the efficiency of our solution toward that one of TPR$*$-tree in case the length of time interval enlarges to value 100. Apparently, the length of the query rectangle remains in sensibly realistic levels. It is almost the same efficient with the solution of B$^+$-trees.

Update Cost Comparison. Figure 14 compares the average cost (amortized over each insertion and deletion) as a function of the number of updates. The Lazy B-trees for the x- and y-projections (LBT(x) and LBT(y) respectively) have nearly optimal update performance and consistently outperform the TPR*-tree by a wide margin. They also outperform the update performance of B$^+$-trees by a logarithmic factor but this is not depicted clearly in Figure 14 due to small datasets.

For this reason we performed another experiment with gigantic synthetic data sets 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 till their number becomes 10^{12}. For each object we considered a synthetic linear function where the velocity value distribution is skewed (zipf) towards 30 in the range [30,50]. The velocity can be either positive or negative with equal probability. For simplicity, all objects are 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 gigantic synthetic dataset we increased the page size from 1024 to 4096 bytes. Since the length of each key is 8 bytes and the length of each pointer is 4 bytes the block size now becomes 341. We have not measured the performance of the initialization bulk-loading procedure. In particular, we have measured the performance of update only operations.

Figure 15 establishes the overall efficiency of our solution. It is also expected that the block transfers for the update operations will remain constant even for gigantic data sets. This fact is an immediate result of the time complexity of update procedures in the Lazy B-tree.

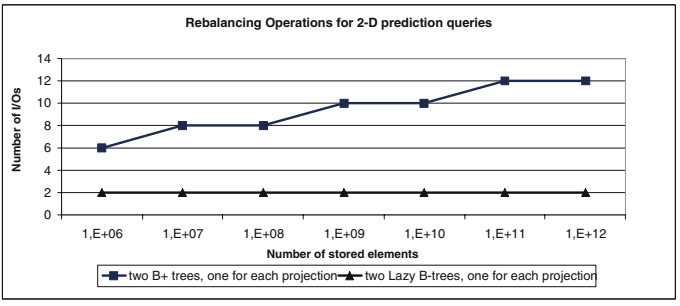

Fig. 15. Rebalancing Operations for the particular problem of 2-D Prediction Queries

6 Conclusions

We presented access methods for indexing mobile objects that move on the plane to efficiently answer range queries about their location in the future. The performance evaluation illustrates the applicability of our first solution since the second solution has only theoretical interest. Our future plan is to simplify the second complicated solution to be more implementable and as a consequence more applicable in practice.

References

1. Agarwal, P.K., Arge, L., Erickson, J., Franciosa, P.G., Vitter, J.S.: Efficient Searching with Linear Constraints. Journal of Computer and System Sciences 61(2), 194–216 (2000)
2. Agarwal, P.K., Arge, L., Erickson, J.: Indexing Moving Points. In: Proceedings 19th ACM Symposium on Principles of Database Systems (PODS), Dallas, TX, pp. 175–186 (2000)
3. Arge, L., Samoladas, V., Vitter, J.S.: On Two-Dimensional Indexability and Optimal Range Search Indexing. In: Proceedings 18th ACM Symposium on Principles of Database Systems (PODS), Philadelphia, PA, pp. 346–357 (1999)
4. Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B.: The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. In: Proceedings ACM International Conference on Management of Data (SIGMOD), Atlantic City, NJ, pp. 322–331 (1990)
5. Chazelle, B.: Optimal Algorithms for Computing Depths and Layers, Brown University, Technical Report CS-83-13 (1983)
6. Chazelle, B., Guibas, L., Lee, D.L.: The Power of Geometric Duality. In: Proceedings 24th IEEE Annual Symposium on Foundations of Computer Science (FOCS), Tucson, AZ, pp. 217–225 (1983)
7. Chazelle, B.: Filtering Search: a New Approach to Query Answering. SIAM Journal on Computing 15(3), 703–724 (1986)
8. Chazelle, B., Cole, R., Preparata, F.P., Yap, C.K.: New Upper Bounds for Neighbor Searching. Information and Control 68(1-3), 105–124 (1986)

9. Comer, D.: The Ubiquitous B-Tree. ACM Computing Surveys 11(2), 121–137 (1979)
10. Dietz, P., Raman, R.: A Constant Update Time Finger Search Tree. Information Processing Letters 52(3), 147–154 (1994)
11. Gaede, V., Gunther, O.: Multidimensional Access Methods. ACM Computing Surveys 30(2), 170–231 (1998)
12. Goldstein, J., Ramakrishnan, R., Shaft, U., Yu, J.B.: Processing Queries by Linear Constraints. In: Proceedings 16th ACM Symposium on Principles of Database Systems (PODS), Tucson, AZ, pp. 257–267 (1997)
13. Guttman, A.: R-trees: a Dynamic Index Structure for Spatial Searching. In: Proceedings ACM International Conference on Management of Data (SIGMOD), Boston, MA, pp. 47–57 (1984)
14. Jensen Christian, S., Lin, D., Ooi, B.C.: Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In: VLDB 2004, pp. 768–779 (2004)
15. Kaporis, A., Makris, C., Sioutas, S., Tsakalidis, A., Tsichlas, K., Zaroliagis, K.: ISB-Tree: a New Indexing Scheme with Efficient Expected Behaviour. In: Proceedings International Symposium on Algorithms and Computation (ISAAC), Sanya, Hainan, China (2005)
16. Kollios, G., Gunopulos, D., Tsotras, V.: Nearest Neighbor Queries in a Mobile Environment. In: Proceedings 1st Workshop on Spatio-Temporal Database Management (STDBM), Edinburgh, Scotland, pp. 119–134 (1999)
17. Kollios, G., Gunopulos, D., Tsotras, V.: On Indexing Mobile Objects. In: Proceedings 18th ACM Symposium on Principles of Database Systems (PODS), Philadelphia, PA, pp. 261–272 (1999)
18. Kollios, G., Tsotras, V.J., Gunopulos, D., Delis, A., Hadjieleftheriou, M.: Indexing Animated Objects Using Spatiotemporal Access Methods. IEEE Transactions on Knowledge and Data Engineering 13(5), 758–777 (2001)
19. Levcopoulos, S., Overmars, M.H.: Balanced Search Tree with O(1) Worst-case Update Time. Acta Informatica 26(3), 269–277 (1988)
20. Manolopoulos, Y.: B-trees with Lazy Parent split. Information Sciences 79(1-2), 73–88 (1994)
21. Manolopoulos, Y., Theodoridis, Y., Tsotras, V.: Advanced Database Indexing. Kluwer Academic Publishers, Dordrecht (2000)
22. Papadopoulos, D., Kollios, G., Gunopulos, D., Tsotras, V.J.: Indexing Mobile Objects on the Plane. In: Hameurlain, A., Cicchetti, R., Traunmüller, R. (eds.) DEXA 2002. LNCS, vol. 2453, pp. 693–697. Springer, Heidelberg (2002)
23. Patel, J., Chen, Y., Chakka, V.: STRIPES: an Efficient Index for Predicted Trajectories. In: Proceedings ACM International Conference on Management of Data (SIGMOD), Paris, France, pp. 637–646 (2004)
24. Raman, R.: 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)
25. Raptopoulou, K., Vassilakopoulos, M., Manolopoulos, Y.: Towards Quadtree-based Moving Objects Databases. In: Benczúr, A.A., Demetrovics, J., Gottlob, G. (eds.) ADBIS 2004. LNCS, vol. 3255, pp. 230–245. Springer, Heidelberg (2004)
26. Raptopoulou, K., Vassilakopoulos, M., Manolopoulos, Y.: Efficient Processing of Past-future Spatiotemporal Queries. In: Proceedings 21st ACM Symposium on Applied Computing (SAC), Minitrack on Advances in Spatial and Image-based Information Systems (ASIIS), Dijon, France, pp. 68–72 (2006)
27. Raptopoulou, K., Vassilakopoulos, M., Manolopoulos, Y.: On Past-time Indexing of Moving Objects. Journal of Systems and Software 79(8), 1079–1091 (2006)

28. Saltenis, S., Jensen, C., Leutenegger, S., Lopez, M.A.: Indexing the Positions of Continuously Moving Objects. In: Proceedings ACM International Conference on Management of Data (SIGMOD), Dallas, TX, pp. 331–342 (2000)
29. Saltenis, S., Jensen, C.S.: Indexing of Moving Objects for Location-Based Services. In: Proceedings 18th IEEE International Conference on Data Engineering (ICDE), San Jose, CA, pp. 463–472 (2002)
30. Salzberg, B., Tsotras, V.J.: A Comparison of Access Methods for Time-Evolving Data. ACM Computing Surveys 31(2), 158–221 (1999)
31. Samet, H.: The Design and Analysis of Spatial Data Structures. Addison Wesley, Reading (1990)
32. Sellis, T., Roussopoulos, N., Faloutsos, C.: The R^+-tree: a Dynamic Index for Multi- Dimensional Objects. In: Proceedings 13th International Conference on Very Large Data Bases (VLDB), Brighton, England, pp. 507–518 (1987)
33. Tao, Y., Papadias, D., Sun, J.: The TPR*-Tree: an Optimized Spatio-Temporal Access Method for Predictive Queries. In: Proceedings 29th. International Conference on Very Large Data Bases (VLDB), Berlin, Germany, pp. 790–801 (2003)