# Towards Quadtree-based Moving Objects Databases*

Katerina Raptopoulou[1], Michael Vassilakopoulos[2], and Yannis Manolopoulos[1]

[1] Department of Informatics, Aristotle University,
GR-54006 Thessaloniki, Greece. E-mail: katerina,manolopo@delab.csd.auth.gr

[2] Department of Informatics, Technological Educational Institute of Thessaloniki,
GR-54101 Thessaloniki, Greece. E-mail: vasilako@it.teithe.gr

**Abstract.** Nowadays, one of the main research issues of great interest is the efficient tracking of mobile objects that enables the effective answering of spatiotemporal queries. This line of research is relevant to a number of modern applications spanning many contexts. In this paper, we consider the organization of a moving object database by quadtree based structures (structures obeying the Embedding Space Hierarchy). In this context, we adapt an indexing method, called XBR trees, to support range queries about the history of trajectories of moving objects. The XBR tree is a quadtree like external memory, balanced and compact structure that follows regular decomposition. Apart from the presentation of the new method, we experimentally show that it outperforms the only previous Embedding Space Hierarchy approach (based on PRM quadtrees) for indexing moving objects.

*Keywords:* Quadtrees, Moving Objects, Spatiotemporal Queries, Spatiotemporal Databases

## 1 Introduction

In the past few years, the focus of the research in Geographic Information Systems (GISs) has drastically evolved from traditional data management issues (such as modeling, indexing, querying) to new and exciting challenges raised by the emergence of new technologies. Two of the major recent achievements of these technologies, namely the World Wide Web and the development of accurate positioning systems, have a strong impact on GISs. In particular, positioning systems constitute a very challenging area. The Global Positioning System (GPS) and the new European Galileo satellite project (its launching has been decided very recently, at the end of March 2002), are able to determine the position of a moving object with a very high precision (e.g. a few centimeters).

* Research supported by the ARCHIMEDES project 2.2.14 "Management of Moving Objects and the WWW" of the Technological Educational Institute of Thessaloniki (EPEAEK II).

On the other hand, there undoubtedly exists a necessity for numerous applications related to moving objects. Technologies involving mobile computing have appeared to show a great evolution, particularly in the last few years. Devices such as mobile phones and Internet terminals have become ubiquitous.

There are also applications, which include vehicle navigation, tracking and monitoring, where the positions of air, sea or land-based equipment, such as airplanes, fishing boats and cars (e.g. taxis or ambulances) are of interest. An example of such applications is the tracking of fighter planes in air-force combat situations. Being able to correctly locate the planes (that move very fast) at a present time and in the near future can be used to avoid enemy targets and also guide the of fighter planes towards proper targets. Other real life examples that involve objects with positions changing over time, are traffic control, fleet management, fire or hurricane front monitor and weather forecast.

The topic of querying and indexing moving objects has been addressed by several researchers. As far as the theoretical background is concerned, Sistla et al. in [11] proposed a data model, called Moving Objects Spatio-Temporal (MOST) model, for representing moving objects and a query language, called Future Temporal Logic. Wolfson et al. [18] addressed the uncertainty issues, which determine the frequency with which the database has to update the locations of the moving objects, in order to provide an error bound.

Several papers have appeared that base the indexing of moving objects on structures that belong in the family of R-trees [4]. For example, in [10] Saltenis et al. proposed an R*-tree based access method (the TPR-tree) to index the current and future locations of moving objects aiming at handling range queries. Pfoser et al. [9] proposed the STR-tree as an R-tree based indexing scheme suitable for storing the history of moving objects and for trajectory-based queries. Furthermore, the Historical R-tree was proposed by Nascimento et al. [8] as an indexing method for spatiotemporal data and range queries. Finally, in [19], Zhu et al. proposed octagon trees (OT-tree, O-tree) as extensions to the R*-tree to index moving objects and handle range queries.

All these methods are based on the concept of Object Space Hierarchy (the partitioning of regions depends on the data) that is followed by structures of the R-tree family. In this paper, we focus on methods based on the concept of Embedding Space Hierarchy (the partitioning of regions follows a regular fashion) that is followed by structures of the quadtree family. To the authors knowledge, the only paper that addresses the problem of indexing moving points by such a method is presented in [12]. In the present paper, a new such technique is presented and compared to the method of [12].

These structures allow processing of range time and space queries (e.g. which objects will appear in a specific area within a given time interval), or to predict the future position of an object, or to follow the history of the movement of an object.

An alternative perspective to tackle the issue of moving objects is the use of transformations to index their trajectories. In [6] Kollios et al. used the dual transformation with a view to improve the performance during range queries.

Similarly, Chon et al [3], proposed the SV-model as an alternative method of transformation. The use of moving objects can also be applied in multimedia environments. For example, Tzouramanis et al. in [14–16] presented several spatiotemporal access methods (i.e. the OLQ-trees Overlapping Linear Quadtrees and the MVLQ-tress Multiversion Linear Quadtrees) for storing and retrieving evolving raster images.

In [5], Hadjieleftheriou et al., suggested the Partially Persistent (PPR-tree) as a method for indexing and querying the history of moving objects with changing extend (e.g. shrinking). Furthermore, the object movement was described by polynomial and not by linear functions and the queries examined were range ones. Finally, other researchers proposed the use of techniques rooted in computational geometry (for example, in [1] external Range Trees are presented and use for indexing moving points).

The indexing scheme that we propose here, the External Balanced Regular trees (XBR trees), is based on quadtrees, and more specifically on hierarchical and regular subdivision of space. The key ideas behind its design were originally presented in [17] for managing spatial objects, in general. In this paper, we use XBR-trees in the context of spatiotemporal databases. More specifically, we use XBR-trees to index the trajectories of moving objetcs and to answer spatiotemporal queries about these objects. In addition to the material appearing in [17], in this paper, we present a modified algorithm for splitting internal nodes of XBR-trees to deal with extreme conditions and we also describe the steps of the deletion process in XBR-trees.

We experimentally compare the resulting method (that could be used as the physical layer of a Moving Objects Database) with the only analogous (quadtree based) method that is based on the PMR quadtree and was presented in [12] by Tayeb et al. An important difference between two techniques is that the indexing part of the PMR resides in main memory use, whereas the indexing part of XBR trees is a multiway disk-based tree. However, the experiments conducted in the present paper cannot be compared directly with the ones presented in [12], since they are performed under completely different conditions and assumptions (in [12] only the present status of moving objects is maintained, while in this paper the trajectory of each object through time is kept).

The XBR trees constitute a family of new secondary memory structures, which are suitable for storing and indexing multi-dimensional points and line segments. In 2 dimensions, the resulting structure is an External Balanced Quadtree, in 3 dimensions an External Balances Octtree, and in higher dimensions an External Balanced Hyper Quadtree. The main characteristic of all these structures is that they subdivide space (in an hierarchical and regular fashion) into disjoint regions. These spatial access methods are fully dynamic, while insertions are not complicated to program as they affect a single tree path only. Moreover, XBR trees are variable resolution structures. That is, the number of space subdivisions is not predefined, making these structures suitable for very large amounts of data. Due to the balanced nature of these structures and the disjointness of the resulting regions, searches and other queries in these trees

are processed very efficiently. The interested reader will find a short qualitative comparison of XBR-trees with other well known structures, such as R-trees, R+trees, hB-trees and GBD trees, in [17]

The rest of the paper is organized as follows. Section 2 describes the assumptions made with respect to the movement of the objects, Section 3 gives a detailed description of the new structure and a short description of PMR quadtrees. Section 4 exposes the experimental results as far as query performance of the two trees is concerned. Finally, Section 5 presents briefly the conclusions and further research directions.

## 2 Monitoring of Moving Objects

We assume that time is discrete and that the location and velocity of each object is updated only at predefined time points that divide time in a number of time intervals. For each time interval of the past (up to the current time point), a line segment that expresses the movement of the object during this interval is maintained. For the interval starting at the current time point, a line segment that express the initial location and velocity of the object is maintained. All these line segments make up a polyline that expresses the trajectory of the object from the starting time point to the point that follows the current time point. Especially, the last line segment expresses not the actual trajectory, but the expected trajectory from the current time point to the next one.

When time advances to the next time point, each object notifies the system of its actual location and velocity. With this data, the last line segment of the polyline is updated (meaning that, in general, the last line segment must be deleted and reinserted to reflect the actual data) and a new segment that expresses the expected trajectory from the new current time point to the next one is inserted. The resulting line segments are stored in the (XBR, or PMR) tree leaves and information guiding the search to the leaves is stored in internal nodes.

This scheme aims at efficiently supporting range queries regarding the history of the objects movement. For example, to answer the query 'Find all the objects that were positioned inside a particular area during a specific time interval', we traverse the tree from the root, visiting only the nodes which may contain object trajectories satisfying the query. This is done by comparing the area coordinates specified by the query to the coordinates specifying each node.

Although, it is possible to handle X and Y coordinates of each object (along with time) at the same structure (with tree versions that can handle 3-dimensional data), following the approach of [12] we handle X and Y coordinates independently. This means that we keep one 2-dimensional tree for X coordinate along with time and another 2-dimensional tree for Y coordinate along with time. We answer a query using each of the trees and then combine the subanswers. Accordingly, at each time point we update both trees.

## 3 XBR and PMR

### 3.1 The XBR tree

The XBR tree consists of two kinds of nodes: the leaves that occupy disk pages containing the actual data, namely the line segments, and the internal nodes, also occupying disk pages, which provide a multiway indexing method.

Despite the fact that XBR tree is an indexing method capable of being defined for various dimensions, for the sake of brevity, in the sequel we assume two dimensions. For 2 dimensions the hierarchical decomposition of space is the same as the quadtrees. More specifically, the space is subdivided in 4 equal subquadrants, any of which may be further subdivided recursively in 4 subquadrants.

**Internal Nodes** Each internal node in the XBR tree consists of a non-predefined number of pairs of the form <address, pointer>. The number of these pairs is non-predefined because the addresses being used are of variable size. An address expresses a child node region and is paired with the pointer to this child node. Apparently, both the size of an address and the total space occupied by all pairs within a node must not exceed the node size.

More specifically, the address encoding method that we used works as follows. For a binary integer $x$ initially we form code $\gamma$ that consists of two parts. The first has $\lfloor \log_2 x \rfloor$ 0s and one 1, while the second is the number $x - 2^{\lfloor \log_2 x \rfloor}$ in binary form, expressed with $\lfloor \log_2 x \rfloor$ bits. The code that we finally use is $\delta$ that encodes the number $\lfloor \log_2 x \rfloor + 1$ with the first part of code $\gamma$ (with the two parts of $\gamma$ concatenaded) and with the second part the same to that of code $\gamma$ (in binary form the number $x - 2^{\lfloor \log_2 x \rfloor}$). More details appear in [17].

The addresses being used constitute a representation of a specific subquadrant being produced by quadtree-like hierarchical subdivision of the current space. Each address is formed by a number of directional digits each one representing a particular subquadrant. That is, NW, NE, SW and SE subquadrants of a quadrant are distinguished by the directional digits 0, 1, 2 and 3, respectively. For example, the address 1 represents the NE quadrant of the current space, while the address 10 the NW subquadrant of the NE quadrant of the current space.

One of the main novelties of this particular indexing scheme is the fact that in reality the region of a child is the subquadrant determined by the address in its pair minus the subquadrants corresponding to the previous pairs of the internal node to which it belongs.

For example, Figure 1 depicts an internal node that points to two leaves. While the left child region is the SW quadrant of the original space, the right child region is the whole space minus the region of the first quadrant. Each * symbol denotes the end of a variable size address. In particular, the address of the left child is 2*, where the directional digit 2 corresponds to the SW quadrant of the original space. Moreover, the right child address is * (i.e. no directional digits exist in this address) and the region for this child is the whole space minus the first child region. Each address refers to a minimal quadrant covering the

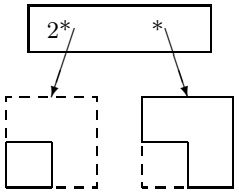


**Fig. 1.** An XBR tree with one internal node and two leaves.

internal node. In this specific example, the minimal subquadrant is the whole space, since the internal node under consideration is the root.

When a search or an insertion of a line segment is performed, descending the tree from the root specifies the appropriate leaves and their regions. At the root, the region that has to be checked is the whole space. When visiting an internal node, we check in turn every contained pair. The first pair with a subquadrant that contains the particular coordinates is chosen and its pointer to the next level is followed. By examining this way the pairs in each node, the path being followed determines the region under consideration by intersecting it with the subquadrant of the chosen pair and subtracting the subquadrants of the pairs appearing to the left of this pair.

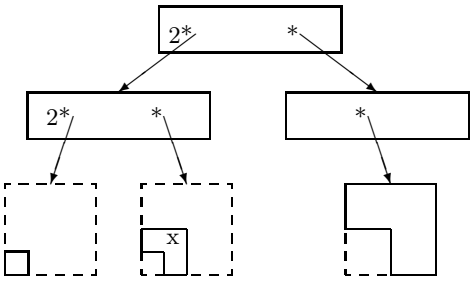


**Fig. 2.** An XBR tree with two levels of internal nodes.

After an insertion in the left child, this child is split and the result is shown in Figure 2. The child split has caused the split of the internal node too, and this has led to the new root creation. If someone wants to detect a data element marked with 'x' , he has to follow the following procedure. At first, he has to visit the root and the pairs that it contains. As we can see, the address 2* that belongs to the first pair, is the one whose respective subquadrant contains the coordinates of 'x'. Therefore, he has to follow the pointer of the first pair. In this node, the address of the first pair, 2*, determines the SW subquadrant of the SW quadrant of the whole space, which does not contain the coordinates of 'x' and the address of the second pair, *, determines the rest of the SW quadrant

of the whole space, which contains the coordinates of 'x'. We follow the pointer of this pair and reach the leaf containing 'x'.

The multi-way nature of the XBR tree is not explicitly depicted by the two examples presented previously. This is done in purpose, only for the sake of brevity.

**Leaf Nodes** Each external node (leaf) in the XBR tree may contain a number of line segments, which is limited by a predefined capacity $C$. When an insertion causes the number of line segments of a particular leaf to exceed $C$, then the leaf is spit following a hierarchical decomposition analogous to the quadtree decomposition, until the resulting regions contain line segments that are less than $x \times C$ and more than $(1 - x) \times C$, where $0.5 < x < 1$.

The constant $x$ is chosen in order to affect the number of necessary subdivisions and the size of addresses that are created after the split. A choice of a value close to 0.5 has proven to cause more subdivisions and larger sizes of addresses. This is due to the fact that hardly ever does a partition split the leaf in subregions that contain almost equal numbers of elements.

With such a value assigned to $x$, we can achieve a better guarantee as far as the space occupancy of leaves is concerned. On the contrary, the PMR quadtree partitions each leaf once and only once. Such a method requires overflow pages. This is not the case for the XBR leaf however, which is guaranteed to contain at most $C$ line segments plus the number of directional digits needed to reach this leaf. Furthermore, in PMR quadtrees there is no minimum occupancy of leaf nodes.

If we consider that $x$ is assigned the value 0.75, then after continuous insertions of line segments in the NW corner of the right leaf region of Figure 2, the region of this leaf splits in four. If from the subregions formed none contains less than $3C/4$ and more than $C/4$ line segments, then the subregion containing the larger number of data elements is split in four. This procedure is repetitively applied until there exists a region with less than $3C/4$ and more than $C/4$ line segments. Then the original leaf will split in two leaves: the subregion created above will represent the region of the left of the two resulting leaves. The rest of the original region is the new right leaf region. Following this policy, both leaves created will be at least 1/4 full. This situation is depicted in Figure 3.

**Splitting of Internal Nodes** An internal node overflow causes a split in two in a way that achieves a good balance between the space use in the two nodes. In order to perform the split, first of all we construct a quadtree that has as nodes the quadrants existing in the XBR internal node to be spit. This is shown in Figure 4a. This node contains addresses that subdivide the node region. Each address corresponds to a quadtree node represented by a square. By following the path to this node, all intermediate quadtree nodes are marked as circles. There also exists the possibility, that a square may be the ancestor of others squares (for example, the square of address 0* is a parent for the squares of addresses 00*, 01* and 02*). The address * specifies the quadtree root.

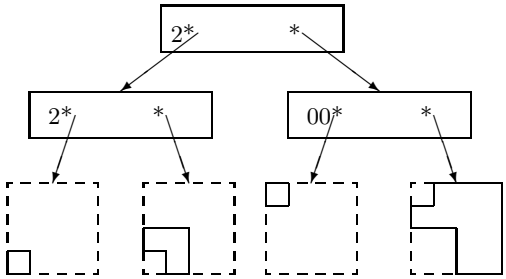


**Fig. 3.** The XBR tree after splitting the rightmost leaf of the tree in Figure 2.

To each node, we assign the number of squares that will be freed, when we eliminate the subtree rooted at this node. A bottom-up procedure rapidly calculates this number. In Figure 4c each external square is assigned the value 1: the squares of 100*, 101*, 00*, 01* and 02* are all assigned the value 1. Each internal square is assigned the sum of values of its children plus 1. For example, the square of 0* is assigned the value 4=1+1+1+1. Finally, a circle is assigned the sum of values of its children only. For example, the second root child is assigned 2, since it has only one child (another circle) with value 2. Next we traverse the tree in order to find a node, apart from the root, which is a square and is assigned the largest number of squares in the tree.

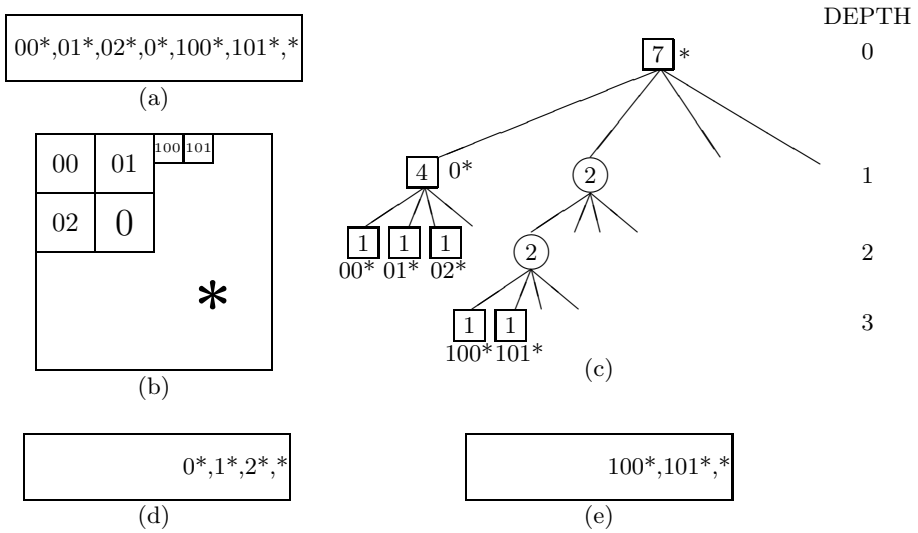


**Fig. 4.** Splitting of an internal XBR tree node.

For example, in the tree of Figure 4c, the node assigned with the largest number of squares, is the leftmost root child with address '0' and number of squares equal to 4. The sought subtree is rooted at this node. The resulting two nodes are depicted in Figure 4d and 4e. Apparently, in the father node of the original internal node, the entry 0*, which corresponds to the minimal quadblock of the left of the resulting nodes, should be inserted.

**Deletion** Deletion is used while updating the location and velocity of each object at each time point. That is, the last line segment of the trajectory of each moving object is updated at the end of each time interval (in general, it must be deleted and reinserted to reflect the actual data) and a new line segment, expressing the expected trajectory from the new current to the next time point, is inserted.

Since a line segment may cross the regions of several XBR-tree leaf nodes, it has to be removed from all these leaf nodes. Following a procedure similar to the insertion of a line segment, at each internal node (starting from the root) we sequentialy examine the <address, pointer> pairs and recursively visit child nodes with regions that are crossed by the line segment. This way, we determine all the leaves that are crossed by the line segment (the line segment must be deleted from each of these leaves).

If a leaf node from which we remove the line segment underflows (if it contains less than $(1 - x) \times C$ line segments), then a merge occurs. First, the <address, pointer> pair corresponding to this leaf that resides in its parent internal node is deleted. Then, the rest line segments of the leaf node are added to the rightmost child of the parent internal node (the righmost brother of the leaf node). If this child overflows, then it is split (as described in the "Leaf Nodes" part of the current subsection) and the split may propage to higher levels (hosting internal nodes).

Since internal nodes do not have a minimum occupancy theshold, the merge process is not applied to internal nodes. A more sophisticated deletion process that considers alternative merging of an underflowed leaf node with other brother leaves, except for its rightmost brother and merging of internal nodes is currently under development.

### 3.2 The PMR tree

The PMR tree [12] is an indexing scheme based on quadtrees, capable of indexing line segments. The internal part of the tree consists of an ordinary region quadtree (degree four tree) residing in main memory. The leaf nodes of this quadtree point to the bucket pages that hold the actual line segments and reside on disk (Table 1). Each line segment is stored in every bucket whose quadrant (region) it crosses. A line segment can cross a region of a bucket either fully or partially. The PMR tree was proposed as an access method where the index is kept in main memory, whereas the indexed data, namely the line segments, are stored in secondary storage.

| | XBR-trees | PMR-trees |
|---|---|---|
| **RAM** | Pointer to Root | Internal Nodes |
| **Disk** | Internal Nodes and External Nodes | External Nodes |

**Table 1.** RAM and Disc usage for XBR-trees and PMR-trees.

**Insertion in the PMR tree** A line segment is inserted in a PMR tree by being registered in the buckets that correspond to the quadrants that it crosses. During that procedure the capacity of each bucket that is intersected by the line segment is checked in order to verify whether that insertion causes it to exceed the predefined bucket capacity. If the bucket capacity is exceeded, then the bucket is split once and only once into four equal quadrants (if the bucket has already been split, then a chain of overflow buckets is maintained). Therefore, the bucket capacity is a split threshold. When a bucket is split, four new buckets are created, each one corresponding to a single subquadrant of the quadrant of the original bucket. After this procedure is performed, the old parent bucket is no longer in use. On the contrary, the quadtree pointer (in main memory) that used to point to that bucket now points to a new quadtree node with four pointers that point to the four newly created buckets.

**Deletion in the PMR tree** A line segment is deleted from a PMR quadtree by being removed from all the buckets that correspond to quadrants that it crosses. During this procedure, the capacity of the bucket and its siblings are checked in order to discover if the deletion causes the total number of lines segments in them to be less than a split threshold. If the split threshold is greater than the capacity of the bucket and its siblings, then they merge and the merge procedure is then repeated to the parent quadtree node.

## 4 Experimentation

For the tree implementations and the experiments execution we used a Pentium 1600-MhZ with 1024K memory. The page size used was 4k, which resulted in a leaf node containing 204 lines. After experimentation, we came to the conclusion that the use of a buffer of 100K with least-recently-used page replacement, has shown better performance in comparison to other choices. Therefore, except these 100 disk pages, the entire index comprising both the internal and the external nodes, reside on disk.

For the experiments execution, we considered 1000 time units, being separated into 100 equal time intervals, each one of 10 time units. We conducted several experiments with a varying number of moving objects $N$, and a different size for the range query, which is successively set to 0.1, 0.01, 0.001 of the total

space into consideration. At time unit 0, we randomly generate a velocity and an initial location for each object. We assume that during each time interval the object velocity and location are constant. At the interval end though, these numbers are updated for each object. This procedure is repetitively applied until the end of the time horizon being considered is reached.

The queries performed are range queries and during an experimental execution they are repeated after 10 constant time intervals. During the experiments execution we count the I/O cost for the queries, the cost for the pages that are not found in the buffer and are read from the disk, the execution time cost, the average number of repetitions of each line and the number of nodes that reside on disk for the XBR tree and the number of nodes that reside either in disk or in memory for the PMR tree.

Since both the XBR tree and the PMR tree, belong to the quadtree families, each line segment inserted in them is not kept in a single but in more than one leaves. Therefore counting and comparing the number of the appearances of a line segment in the two trees is considered to be noteworthy.

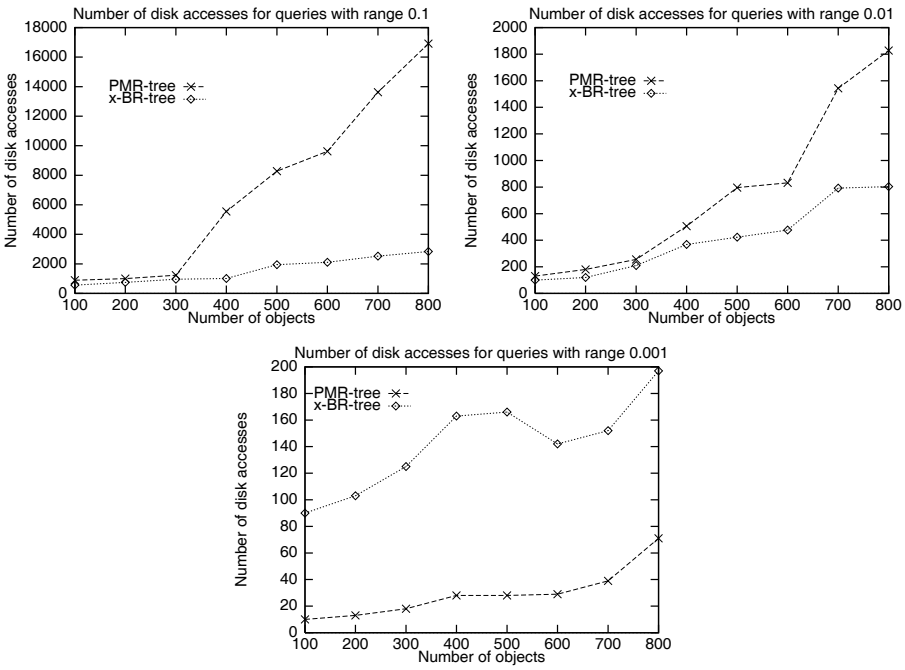


**Fig. 5.** Disk Accesses for queries with range 0.1 (top left), 0.01 (top right) and 0.001 (bottom).

The first three experiments presented in Figure 5 study the number of disk accesses. Namely, we counted the number of disk accesses that were required for both trees during the execution of the range queries. In each experiment the parameters are the number of objects and the size of the range query. The

objects vary from 100 to 800, whereas the query size takes values 0.1, 0.01 and 0.001.

Figure 5 top left illustrates the number of disk accesses for range queries performed for a query size equal to 0.1 of the whole space under consideration. In this figure, the XBR tree requires significantly fewer disk accesses than the PMR tree. Figure 5 top right depicts the number of disk accesses for queries size equal to 0.01. As in the previous experiment, the number of disk accesses made by the XBR tree are again significantly fewer than those made by the PMR tree. Figure 5 bottom presents the number of disk accesses for the 0.001 query sizes. In this case, the PMR tree requires fewer disk accesses during the execution of the range queries. However, its difference from the XBR tree is very small (notice the scale on the y-axis). This reverse behavior is easily explained. For a very small query size, a small number of leaves is accessed. In the XBR tree (unlike the PMR tree), the internal nodes that are used to reach the leaves, also reside on disk and contribute to the number of disk accesses. This is the penalty the XBR tree has to pay, in order to be capable to handle very large amounts of data (unlike the semi-RAM based PMR quadtree). As it will be shown later in this section, in the experiments that study execution time cost, even under this situation, the XBR tree outperforms the PMR quadtree.

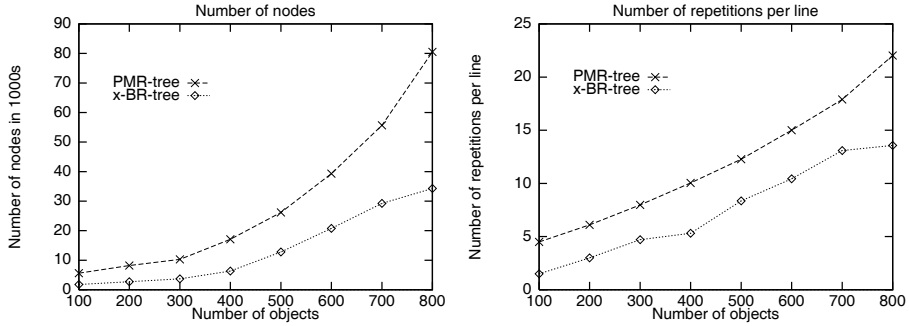


**Fig. 6.** Number of Nodes (left) and Medium Number of Repetitions Per Line (right).

The next two experiments presented in Figure 6, study the number of nodes required for both trees and the average number of appearances for the lines inserted. In each experiment, the parameters are the number of objects under consideration, which varies from 100 to 800.

Figure 6 left shows (in thousands) the number of nodes required by the two trees. During the experiment evolution, the PMR tree grew and was made up of nodes that resided either on disk, or in main memory. By definition, all the XBR tree nodes reside in the disk. The nodes in the former case were by far more than the ones in the latter case. This means that the XBR tree (due to its multiway nature) has a smaller height than the PMR tree, which will help the tree to answer the queries more effectively.

Since both the PMR tree and the XBR tree are quadtrees (and subdivide space in a predefined manner), it follows that the lines inserted in them are not kept in a single leaf. Each line segment may intersect and be inserted in

several leaves, a fact that can delay the query processing. The average number of appearances per line inserted is presented in Figure 6 right. The parameter in this experiment is the number of objects, which again varies from 100 to 800. In this experiment, the XBR tree again stored each line segment in fewer leaves than the PMR tree.

To sum up the results from the experiments in Figure 6, we come out that the XBR tree is a more compact tree, with smaller height, occupying fewer nodes than the PMR tree. Furthermore, the lines inserted in the XBR tree are not repeated as many times as in the PMR case. The second result, namely that the inserted lines are repeated more times in the PMR tree is a logical conclusion drawn from the fact that the PMR has more nodes. This means that the lines inserted in the tree have to be repeated in more nodes.

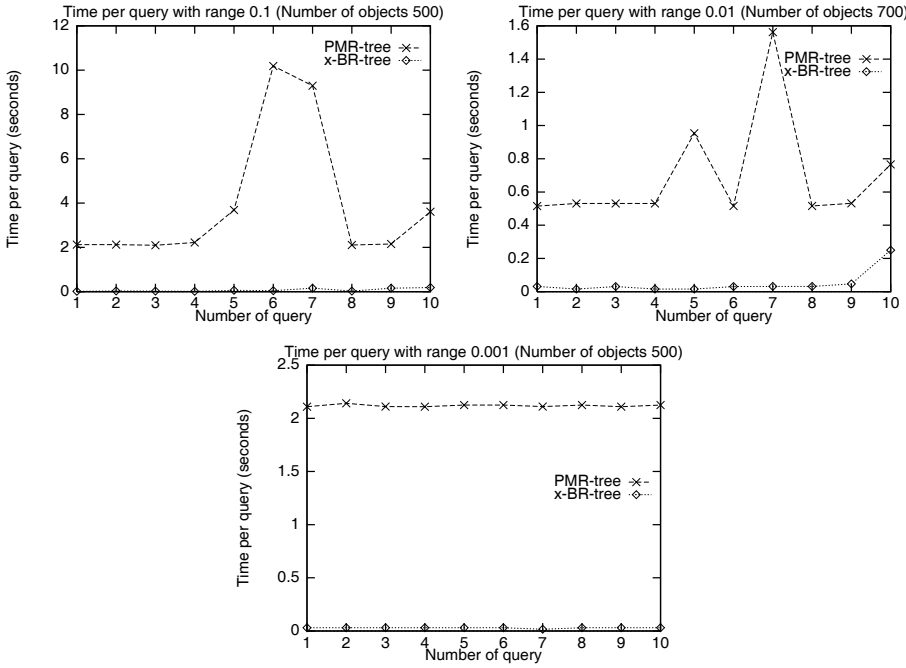


**Fig. 7.** Elapsed time for queries with range 0.1 (top left), range 0.01 (top right) and range 0.001 (bottom).

The next three experiments study the time elapsed during each query execution (execution time cost). For each tree, we performed 10 queries, each one during a constant time interval. The parameters in these experiments are the number of the query from 1 to 10, the query range that takes values 0.1, 0.01 and 0.001 and the number of objects, which takes values 500, 700 and 500. Since in each experiment there are 10 queries performed, in each figure there are 10 numbers corresponding to the elapsed time.

In Figure 7 top left the experiment was conducted with 500 moving objects and query size equal to 0.1. In this figure, the execution time for the XBR tree

is significantly less than the time for the PMR tree. Another observation, as far as the experiment is concerned, is that the time spent for the PMR tree shows an instant great increase during the queries from 5 to 8. This instability in the PMR tree shows that there is no guarantee for the time needed, which can be either small or bigger. In Figure 7 top right 700 moving objects are considered in combination with 0.01 queries. As in the previous experiment, the time needed by the XBR tree in this figure is significantly less. Furthermore, the time spent by the PMR tree still appears to show a great instant increase during the queries from 4 to 8. Finally, in Figure 7 bottom there are 500 moving objects and 0.001 range queries. In this case, again, the XBR tree consumes significantly less time than the PMR tree for all the range queries conducted. Note that this happens contrary to the higher number of disk accesses needed by the XBR tree (Figure 5 bottom). In other words, although for 0.001 range queries the PRM quadtree slightly outperfoms the XBR tree in disk accesses, overall (in execution time) the XBR tree significantly outperforms the PMR quadtree.

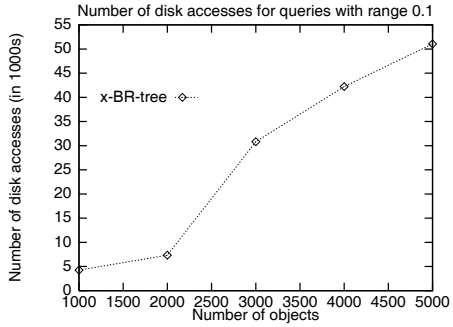


**Fig. 8.** Node Accesses.

As a final experiment, in Figure 8 we counted the number of disk accesses that were required for the XBR tree. The parameter in this experiment is the number of moving objects, which varies from 1000 to 5000, whereas the query range into consideration is 0.1 of the whole space. The reader may ask why we have not presented results for the PMR quadtree also. The answer is that, for these cardinalities of moving objects, the execution needed to gather such results for the PMR quadtree was excessive.

## 5 Conclusions and Future Work

Considering the great application demands for monitoring of mobile objects, to be able to efficiently locate and answer queries related to the position of these objects in time is very important. More specifically, modern applications, such as Mobile Computing and Geographic Information Systems, make the development of research within this field inevitable.

In this paper, we proposed a method, XBR trees, that follows the Embedding Space Hierarchy and can efficiently keep track of the moving objects his-

tory and answer spatiotemporal range queries. According to the experimentation presented, this scheme is indeed efficient. In all experiments conducted, the XBR tree outperforms the PMR quadtree (the only Embedding Space Hierarchy method up to now that has been proposed for monitoring moving objects). Even in the case of very small query ranges, where the PRM quadtree slightly outperformed the XBR tree in disk accesses, overall (in execution time) the XBR tree significantly outperformed the PMR quadtree.

The queries answered during experimentation were statiotemporal range queries that tracked the history of the motion of the moving objects. One possible future extension to this investigation is the implementation and experimentation with other types of spatiotemporal queries. Such types may include

- nearest neighbor queries (e.g. 'indicate the nearest neighbor of an object at each position of its trajectory', a query of great importance), or
- spatiotemporal joins, namely queries that deal with moving objects combined with moving regions (e.g. 'Find all airplanes that intersect clouds while they move').

Future research may also include

- experimenting with 3-dimensional versions of the quad-based trees (to combine time and the two coordinates X and Y in a single indexing structure),
- employing alternative buffering methods to improve tree performance, or
- comparing the XBR tree indexing method with other spatiotemporal trees (especially ones following the Object Space Hierarchy, like R-tree based structures), so that we can come up with more general conclusions about the winner structure for moving objects management.

## References


1. P.K. Agarwal, L. Arge, J. Erickson: "Indexing Moving Points", *Proceedings 19th ACM Symposium on Principles of Database Systems (PODS'2000)*, pp.175-186, Madison, WI, 2000.
2. N. Beckmann, H. Kiegel, R. Scheider, B. Seeger: "The R*-tree: an Efficient and Robust Access Method for Points and Rectangles", *Proceedings ACM SIGMOD International Conference on Management of Data*, pp.322-331, Atlantic City, NJ, 1990.
3. H.D. Chon, D. Agarwal, A.E. Abbadi: "Storage and Retrieval of Moving Objects", *Proceedings 2nd International Conference on Mobile Data Management*, pp.173-184, Hong-Kong, China, 2001.
4. A. Guttman: "R-trees: a Dynamic Index Structure for Spatial Searching", *Proceedings ACM SIGMOD International Conference on Management of Data*, pp.47-57, Boston, MA, 1984.
5. M. Hadjieleftheriou, G. Kollios, V.J. Tsotras, D. Gunopoulos: "Efficient Indexing of Spatiotemporal Objects", *Proceedings 8th International Conference on Extending Database Technology (EDTB'2002)*, pp.251-268, Prague, Czech Republic, 2002.
6. G.Kollios, D. Gunopoulos, V.J. Tsotras: "On Indexing Mobile Objects", *Proceedings 18th ACM Symposium on Principles of Database Systems (PODS'99)*, pp.261-272, Philadelphia, PA, 1999.

7. G. Kollios, D. Gunopoulos, V.J. Tsotras, A.Delis, M. Hadjieleftheriou: "Indexing Animated Objects Using Spatiotemporal Access Methods", *IEEE Transactions on Knowledge and Data Engineering*, Vol.13, No.5, pp.758-777, 2001.
8. M.A. Nascimento, J.R.O. Silva: "Towards Historical R-trees", *Proceedings ACM Symposium on Applied Computing (SAC'98)*, pp.235-240, Atlanta, GA, 1998.
9. D. Pfoser, C. Jensen, Y. Theodoridis: "Novel Approaches to the Indexing of Moving Object Trajectories", *Proceedings 26th International Conference on Very Large Databases (VLDB'2000)*, pp.189-200, Cairo, Egypt, 2000.
10. S. Saltenis, C.S. Jensen, S.T. Leutenegger, M.A. Lopez: "Indexing the Positions of Continuously Moving Objects", *Proceedings ACM SIGMOD International Conference on Management of Data*, pp.331-342, Dallas, TX, 2000.
11. A.P. Sistla, O. Wolfson, S. Chamberlain, S. Dao: "Modeling and Querying Moving Objects", *Proceedings 13rd IEEE International Conference on Data Engineering (ICDE'97)*, pp.422-432, Birmingham, UK, 1997.
12. J. Tayeb, O. Ulusoy, O. Wolfson: "A Quadtree based Dynamic Attribute Indexing Method", *The Computer Journal*, Vol.41, No.3, pp.185-200, 1998.
13. Y .Theodoridis, M. Vazirgiannis, T. Sellis: "Spatio-Temporal Indexing for a Large Multimedia Applications" *Proceedings 3rd IEEE International Conference on Multimedia Computing and Systems*, pp.441-448, Hiroshima, Japan, 1996.
14. T. Tzouramanis, M. Vassilakopoulos, Y. Manolopoulos: "Overlapping Linear Quadtrees: a Spatiotemporal Indexing Method", *Proceedings 6th ACM Symposium on Advances in Geographic Information Systems (ACM-GIS'98)*, pp.1-7, Bethesda, MD, 1998.
15. T. Tzouramanis, M. Vassilakopoulos, Y. Manolopoulos: "Multiversion Linear Quadtrees for Spatiotemporal Data", *Proceedings 4th East-European Conference on Advances in Databases and Information Systems (ADBIS-DASFAA'2000)*, pp.279-292, Prague, Czech Republic, 2000.
16. T. Tzouramanis, M. Vassilakopoulos and Y. Manolopoulos: "Overlapping Linear Quadtrees and Spatio-temporal Query Processing", *The Computer Journal*, Vo.43, No.4, pp.325-343, 2003.
17. M. Vassilakopoulos and Y. Manolopoulos: "External Balanced Regular (x-BR) Trees: new Structures for Very Large Spatial Databases", *Proceeding of the 7th Panhellenic Conference on Informatics*, pp.III.61-III.68, Ioannina, Greece, 1999,
18. O. Wolfson, B. Xu, S. Chamberalin, L. Jiang: "Moving Objects Databases: Issues and Solutions", *Proceedings 10th International Conference on Scientific and Statistical Database Management (SSDBM'98)*, pp.111-122, Capri, Italy, 1998.
19. H. Zhu, J. Su, O.H. Ibarra: "Trajectory Queries and Octagons in Moving Object Databases", *Proceedings 11th ACM International Conference on Information and Knowledge Management (CIKM'2002)*, pp.413-421, McLean, VA, 2002.