Available online at www.sciencedirect.com

SCIENCE DIRECT®

The Journal of Systems and Software 79 (2006) 1079–1091

The Journal of Systems and Software

www.elsevier.com/locate/jss

On past-time indexing of moving objects ☆

Katerina Raptopoulou [a], Michael Vassilakopoulos [b,*], Yannis Manolopoulos [a]

[a] *Department of Informatics, Aristotle University, GR-54006 Thessaloniki, Greece*
[b] *Department of Informatics, Technological Educational Institute of Thessaloniki, P.O. Box 141, GR-57400 Thessaloniki, Greece*

Received 10 August 2005; received in revised form 15 October 2005; accepted 18 October 2005
Available online 15 December 2005

Abstract

Tracking of mobile objects trajectories is one of many modern applications supported by Spatiotemporal databases. Within the context of this application, queries about the present, future or past positions of the objects need to be answered. Several indexing methods have been proposed to efficiently handle such spatiotemporal queries. In the current paper, we propose a method for indexing the historic (past) positions of moving objects called XBR-tree, a quadtree-like technique that is able to handle both timestamp and window queries. Moreover, we compare experimentally this with other methods proposed in the literature for the same purpose. In particular, we compare XBR-trees with PMR-trees, structures also related to quadtrees and MV3R-trees, R-tree based structures.
© 2005 Elsevier Inc. All rights reserved.

Keywords: Moving objects; Spatiotemporal queries; Spatiotemporal databases; Indexing

1. Introduction

Geographic Information Systems (GIS) are rapidly developing, taking advantage of the advances of the World Wide Web (WWW) and Global Positioning Systems (GPS). Moreover, related technologies that are gradually becoming ubiquitous have emerged. These include mobile computing and wireless technologies, in which devices such as mobile phones and Internet terminals are used.

Within this technological environment, vehicle position tracking and monitoring are regarded as applications of increasing interest. There exist numerous cases where the positions of airplanes, fishing boats and cars need to be observed. For example, consider keeping track of fighter planes in an air force combat, or soldiers in a combat field. Other situations of high importance that take advantage of such applications include traffic control, fleet management, fire or hurricane monitor and weather forecast.

In all the above applications, the need to exactly or approximately locate a "mobile object" arises. Namely, the moving objects have to be effectively represented, indexed and queried. There are several methods that have been proposed within this view. All these methods are classified in two different ways.

- The first categorization is based on whether the movements of the objects refer to the past or to the future. Several techniques that index the past positions of the objects have been proposed (Kumar et al., 1998; Lomet and Salsberg, 1989; Nascimento and Silva, 1998; Pfoser et al., 2000; Tao and Papadias, 2001a,b; Xu et al., 1990). There are also quite as many methods that index the future positions of the objects (Agarwal et al., 2000; Ishikawa et al., 2002; Kalashnikov et al., 2002; Kollios et al., 1999a,b; Lazaridis et al., 2002; Moreira et al., 2000; Procopiuc et al., 2002; Saltenis et al., 2000).

☆ Supported by the ARCHIMEDES project 2.2.14, Management of Moving Objects and the WWW, of the Technological Educational Institute of Thessaloniki (EPEAEK II—funding by the Greek Ministry of Education and Religious Affairs and the European Union) and the Bilateral Greek–Serbian Scientific Protocol 2005–2006 (funding by the General Secretariat of Research and Technology).

* Corresponding author. Tel./fax: +30 2310 791595.
E-mail addresses: katerina@delab.csd.auth.gr (K. Raptopoulou), vasilako@it.teithe.gr (M. Vassilakopoulos), manolopo@delab.csd.auth.gr (Y. Manolopoulos).

0164-1212/$ - see front matter © 2005 Elsevier Inc. All rights reserved.
doi:10.1016/j.jss.2005.10.020

- The second categorization of the indexing methods is based on the queries that each of them can handle. The most often referenced types of queries are the window, the nearest neighbor and the join query. In a window query, a rectangle R is given, and we determine the objects that intersect it from time points t_s to t_e. In a nearest neighbor query, a moving object O is given and we calculate its k nearest neighbors from time points t_s to t_e. Finally, in a join query, two datasets are given, S_1 and S_2, and we specify the pairs of the objects (s_1, s_2), with $s_1 \in S_1$ and $s_2 \in S_2$, such that s_1 and s_2 fulfill a predicate (e.g. overlap) at some time in the interval $[t_s, t_e]$.

In this paper we focus on methods that refer to the past movement of the objects and deal with window and timestamp queries. Timestamp queries constitute a special case of the window queries. In a timestamp query, just as in a window query, a rectangle R is given. The main difference from the window query is that we determine the objects that intersect it only at a specific time point t_s. More specifically, we propose a method for indexing the historic (past) positions of moving objects called XBR-tree. This is a dynamic quadtree-like technique, suitable for very large amounts of data, that is able to handle both timestamp and window queries. Moreover, we compare experimentally this with other methods proposed in the literature for the same purpose. In particular, we compare XBR-trees with PMR-trees (Hoel and Samet, 1991; Nelson and Samet, 1986; Tayeb et al., 1998), structures also related to quadtrees and MV3R-trees (Tao and Papadias, 2001b), R-tree based structures. The comparison is based on artificial data and studies the efficiency of the structures during query processing (it considers I/O cost, as well as, total execution time for window and timestamp queries) and the disk space occupied by the structures. The experimental results show the XBR-trees outperform the other tree structures both in terms of query processing efficiency and space utilization.

The rest of the paper is organized as follows: Related work is presented in the subsequent section, Section 2. Section 3 sets the framework for the monitoring of the movement of the objects. Section 4 presents PMR quadtrees and Section 5 MV3R-trees. The new method proposed, the XBR-tree, is described in Section 6, whereas the experimentation in Section 7. Finally, the conclusion and future research appear in Section 8.

2. Related work

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. (1997) proposed a data model and a query language. The data model was called Moving Objects Spatiotemporal (MOST) model and was used for representing moving objects, whereas the query language was called Future Temporal Logic. Wolfson et al. (1998) addressed uncertainty issues. In this paper, they determined 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 which belong to the R-tree family (Guttman, 1984). To begin with, Saltenis et al. (2000) proposed an R*-tree like access method (the TPR-tree) to index the current and future locations of moving objects. This method is also capable of handling range queries. Pfoser et al. (2000) proposed the STR-tree for trajectory-based queries. This structure is based on the R-tree as well and it is suitable for storing the history of moving objects. Furthermore, the Historical R-tree was proposed by Nascimento and Silva (1998), as an indexing method for spatiotemporal data and range queries. Tao and Papadias (2001b) presented the MV3R-tree. This consists of a Multiversion R-tree to process timestamp queries, and a 3D R-tree to process long interval queries. Finally, Zhu et al. (2002) proposed the octagon trees (OT-tree, O-tree) an extension to the R*-tree, to index moving objects and handle range queries.

All of these methods exploit the concept of Object Space Hierarchy (the partitioning of the regions depends on the data), which is followed by structures of the R-tree family. The method proposed in this paper, the XBR-tree (Vassilakopoulos and Manolopoulos, 1999; Raptopoulou et al., 2004), and a previously proposed method, the PMR-tree (Tayeb et al., 1998), are both based on the concept of Embedding Space Hierarchy (the partitioning of the regions follows a regular fashion) that is obeyed by quadtree like structures. To the authors knowledge, the only paper, apart from (Raptopoulou et al., 2004), which handles the problem of indexing moving points by such a method is presented in Tayeb et al. (1998).

The above structures allow processing of range queries, which extend to three-dimensions, namely, the time (1D) and the space (2Ds) (e.g. which objects will appear in a specific area, within a given time interval). Such a structure can also be used, 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. Kollios et al. (1999b) used the dual transformation in order to improve the performance, during range queries. Similarly, Chon et al. (2001) proposed the SV-model as an alternative method of using a transformation.

The evolution of spatial data also finds applications in multimedia environments. For example, Tzouramanis et al. (1998, 2000, 2003) presented several spatiotemporal access methods (like the OLQ-trees, Overlapping Linear Quadtrees and the MVLQ-trees, Multiversion Linear Quadtrees) for storing and retrieving evolving raster images.

Hadjieleftheriou et al. (2002) suggested the Partially Persistent (PPR-tree) as a method for indexing and querying the history of moving objects. These moving objects, in their method, were regarded as having a changing extend (e.g. shrinking). Furthermore, the object movement was described by polynomial and not by linear functions and the queries handled were range ones. Finally, other researchers proposed the use of techniques rooted in computational geometry (for example, in Agarwal et al., 2000 external Range Trees are presented and used for indexing moving points).

The indexing scheme that we propose here, the XBR-tree, is based on the quadtree, and more specifically on the hierarchical and regular subdivision of space. The key ideas behind its design were originally presented in Vassilakopoulos and Manolopoulos (1999), for managing spatial objects, in general. The XBR-trees constitute a family of secondary memory structures, which are suitable for storing and indexing spatial objects for various dimensions. In two-dimensions, the resulting structure is an XBR Quadtree, in three-dimensions an XBR Quadtree, and in higher dimensions an XBR Hyper Quadtree. The main characteristic of all of 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. Moreover, the XBR-trees are variable resolution structures. That is, the number of the space subdivisions is not predefined, making these structures suitable for very large amounts of data. Due to the disjointness of the resulting regions, searches and other queries in these trees, are processed very efficiently.

In this paper, extending the work presented in Raptopoulou et al. (2004), we use the XBR-tree in the context of spatiotemporal databases. More specifically, we adapt the XBR-tree to indexing of the trajectories of the moving objects, in order to answer spatiotemporal queries related to them. Moreover, we experimentally compare the resulting method (that could be used as the physical layer of a Moving Object Database) with the only analogous (quadtree like) method that is based on the PMR quadtree, and was presented by Tayeb et al. (1998) with MV3R-trees, which are R-tree based structures. An important difference between the two quadtree techniques is, that the indexing part of the PMR resides in main memory, whereas the indexing part of the XBR-tree is a multiway, disk-stored, tree. However, the experiments conducted in the present paper cannot be directly compared with the ones presented in Tayeb et al. (1998), since they are performed under completely different conditions and assumptions (in Tayeb et al. (1998) only the present status of the moving objects is maintained, while in this paper, the trajectory of each object, throughout time, is kept).

3. Monitoring of moving objects

We assume that time is discrete and that the location and the velocity vector (direction of movement and speed) 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 each object during this interval is maintained. For the interval starting at the current time point, a line segment that expresses the initial location and velocity vector of each object is maintained.

All these line segments make up a polyline that expresses the trajectory of each object from the starting time point to the point that follows the current time point. In particular, 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 vector. 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 or MV3R) tree leaves and the information guiding the search to the leaves is stored in the internal nodes.

This scheme aims at efficiently supporting range queries, regarding the history of the object 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 with the coordinates specifying each node.

Note that in Tao and Papadias (2001b) MV3R-trees were used for answering the same type of queries, treating objects as discretely moving: the discrete position of an object was stored only when it changed. This approach makes better use of the storage space, however queries for time intervals with a left (right) end different to the predetermined time points return the positions of the moving objects for the time point that follows (precedes) this interval end. In contrast, in the approach followed in this paper, the position of a moving object is linearly interpolated for all times. Moreover, the use of the last line segment of the trajectory of each object that corresponds to the future, allows to answer predictive queries about the positions of the moving objects in the near future.

Although, it is possible to handle both the x- and the y-coordinate of each object (along with time) at the same structure (with tree versions that can handle three-dimensional data), following the approach of Tayeb et al. (1998), we handle x- and y-coordinates independently with XBR-trees and PMR-trees (unlike MV3R-trees). This means that we keep one two-dimensional tree for the x-coordinate, along time and another two-dimensional tree for the y-coordinate, along time. We answer a query using each of the trees and then combine the subanswers. Accordingly, at each time point, we update both trees.

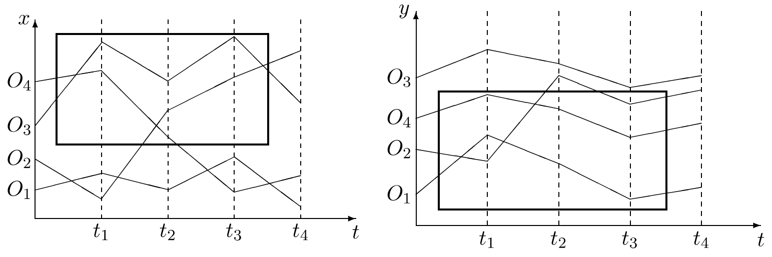

Fig. 1. Monitoring of moving objects.

Fig. 1 presents the polylines of four moving objects. The $x(y)$-coordinate of the objects appears on the left (right) part of the figure, as a function of time, from time point t_1 to time point t_4. The rectangles on the left and right parts of this figure, represent a range query for different x and y ranges, respectively, but for the same t range. The result of the range query as far as the $x(y)$-coordinate is concerned is $\{O_2, O_3, O_4\}$ ($\{O_1, O_2, O_4\}$). The intersection of the two subresults is $\{O_2, O_4\}$ (the objects that reside within both the x and y ranges, during the query time range).

4. The PMR-tree

The PMR-tree (Hoel and Samet, 1991; Nelson and Samet, 1986; Tayeb et al., 1998) 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. Each line segment is stored in every bucket whose quadrant (region) it crosses. A line segment can cross the region of a bucket either fully or partially.

4.1. 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.

In Fig. 2 an example of the creation of PMR quadtree by the successive insertion of line segments is presented. The bucket capacity is two (just for demonstration purposes). Initially, a structure consisting of a leaf node pointing to an empty bucket exists. Fig. 2a–c shows the subdivision of space and the buckets created as line segments are inserted, while the corresponding Fig. 2d–f shows the quadtree part of the PRM tree residing in main memory. The leaf nodes of the quadtree contain pointers to the corresponding buckets (note that one leaf in Fig. 2f does not contain any pointer, since no segments fall within its area). Overflow buckets do not result from the insertions of Fig. 2. Note that the shape of the PRM quadtree depends on the order of insertion of the line segments.

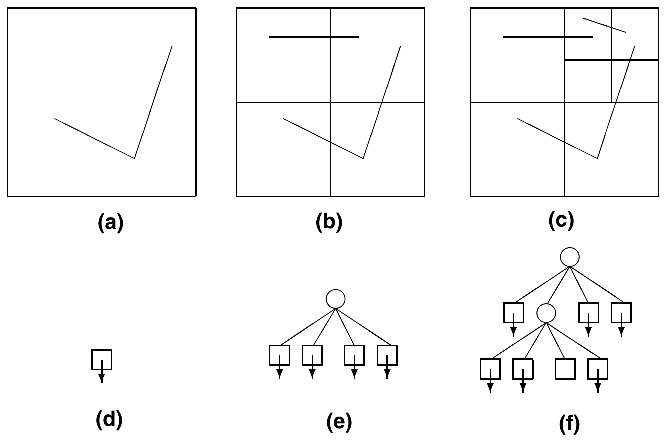

Fig. 2. The creation of a PMR quadtree by the successive insertion of line segments.

4.2. 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.

5. The multi version 3D R-tree (MV3R-tree)

An MV3R-tree (Tao and Papadias, 2001b) was designed as an access method for retrieving the locations of

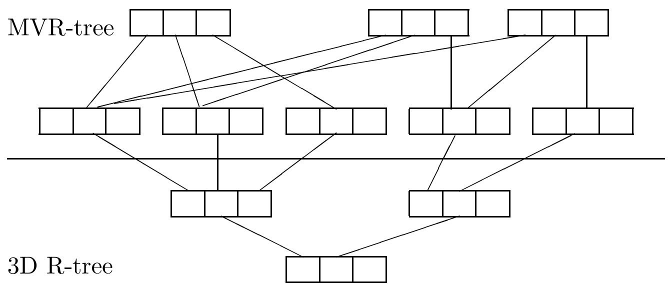

Fig. 3. An MV3R-tree consisting of an MVR-tree (above) and an auxiliary 3D R-tree (below).

discretely moving objects. It consists of two different structures:

- an MVR-tree (Multiversion R-tree) and
- a small auxiliary 3D R-tree built on the leaves of the MVR-tree.

The former (latter) structure aims at improved performance during processing of timestamp, or small window queries (large window) queries. An example of an MV3R-tree is depicted in Fig. 3.

5.1. MVR-tree

The MVR-tree makes use of the ideas used in the multiversion B-tree (MVB-tree) (Becker et al., 1996). MVB-trees index the history and the present status of 1-d data (insertions and deletions can only happen at the current time). They represent several B-trees, each corresponding to a different data version. More precisely, in this tree each entry has the form $\langle key, t_{start}, t_{end}, pointer\rangle$. *key* represents the key value of the data item (a node separator value), in case of leaf nodes (internal nodes). *pointer* points to the actual data item record (next level node) for leaf node entries (internal node entries), while t_{start} and t_{end} denote the lifespan interval of the data item, or node related to key.

An MVB-tree consists of a number of root nodes, each of which corresponds to one interval of versions. A query that searches in a specific version, can be answered by following the tree that is rooted by the root whose lifespan contains the version. Each version of the tree is created by either an insertion or a deletion of any data item. For example, the ith update (insertion or deletion) creates the ith version. Any entry is considered to be of version i, or alive at timestamp i, if its lifespan contains i.

In order to group entries which are alive at the same timestamp, each node except the roots is required to contain either none or more than a minimum number of alive entries for each timestamp (version) i. This is called the *weak version condition* and it may be violated (*weak version underflow*) due to deletions, leading to a merge procedure. Insertions and deletions are similar to B-trees, except the fact that overflows and underflows are treated differently. A node overflow leads to a *version split* (a split according to the version values stored in the node that creates a new node). The new node created is required to have a number of entries between a minimum and a maximum. A number of entries above the maximum (below the minimum) leads to a *strong version overflow (underflow)*. A strong version overflow is handled by a *key split* (a split according to the key values stored in the node), while a strong version underflow is handled by a merge.

The MVR-tree extends the ideas of the MVB-tree for spatial data by representing multiple R-trees. The entries are of the form $\langle S, t_{start}, t_{end}, pointer\rangle$, where S is the Minimum Bounding Rectangle (MBR) as defined in the R-tree literature, and t_{start}, t_{end} and *pointer* are defined as in the MVB-tree. The MVR-tree also uses the *weak version condition* that was introduced in the MVB-tree.

In the MVR-tree, the insertion in a leaf and the insertion in an internal node are handled differently, since in the leaves the main aim is to avoid version splits. In such a way, the redundancy and the total space are both reduced. On the contrary, the internal nodes are allowed to retain redundancy. The insertion in an internal node is similar to that of MVB-trees. However, strong version underflows are not considered. Moreover, version splits need to take into account the spatial extends of the nodes. During the insertion in a leaf node a more complicated procedure is followed. In order to avoid version splits, the following alternatives are tried (in order):

- *General Key Split.*
- *Insertion after reinserting one of the entries of the node.*
- *Insertion of the object in another node.*
- *In case all the above fail, a version split occurs.*

On the other hand, in case of a deletion of an object, there are two different situations. If the deletion does not cause an underflow, the procedure is similar to the one of R*-trees. In case of an overflow, different algorithms are followed for internal nodes and leaves. (again, with a view to avoid redundancy in the leaves).

More details, regarding the MVR-tree appear in Tao and Papadias (2001b).

5.2. 3D-tree

In a 3D R-tree, a third-dimension is added in the representation of the space, namely the time. Instead of having static objects represented in a two-dimensional space, these objects are considered to change their position at some timestamps. Considering that an object is not simply a point but has a spatial extend bounded by a 2-d MBR and that this MBR remains static for an interval of the time axis, a 3-d box is formed that represents the objects' position and extend during this static period. Whenever, the object moves to another position, a new 3-d box is created to model its new spatial characteristics. Thus, the modelling of the object's movement consists of a sequence of distinct 3-d boxes. Moreover, within such a 3-d framework, the timestamp (window) query can be modelled as a rectangle vertical to the time axis (as a 3-d box).

The auxiliary 3D R-tree of the MV3R-tree is built on the leaves of the MVR-tree (not the actual objects), in order to be used for the interval queries. Whenever, a leaf node of the MVR-tree is updated, changes are propagated to its entry in the 3D R-tree.

5.3. Query processing in MV3R-trees

The MV3R-tree presents two possibilities for processing queries: to use the auxiliary 3D R-tree for large window queries, or to use the MVR-tree for timestamp, or small window queries. Querying with the auxiliary 3D R-tree and timestamp querying with the MVR-tree is similar to querying an R-tree. During window query processing with the MVR-tree an algorithm to avoid duplicate visits to the same nodes via different parents is employed.

The two structures that constitute the MV3R-tree are R-tree versions and are both characterized by significant space overlap between their nodes. The overlap that was already present in the R-tree in two-dimensions, in three-dimensions is further increased. This is not the case for the XBR-tree, where there exists no overlap between the nodes. This constitutes an advantage of the XBR-tree over the MV3R-tree, as far as the spatiotemporal queries (window and timestamp) are concerned.

6. The XBR-tree

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

There are two types of nodes in an XBR-tree. The first type is the internal nodes that constitute the index. The second type is the leaves containing all the data items, namely the line segments of the trajectories of the moving objects. Both the leaves and the internal nodes reside on disk.

6.1. Internal nodes

In an internal node, a number of pairs of the form $\langle address, pointer\rangle$ are contained. 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.

The variable length coding of addresses can be done in various ways. In the following we present a simple, but quite effective encoding method (for more complicated methods, see (Elias, 1975; Zobel et al., 1992)). For one binary integer x initially we form code γ that consists of two parts. The first has $\lfloor \log_2 x \rfloor$ 0 s 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. In Table 1, in the second column the γ encoding of the numbers of the first column are depicted. The code we finally use is δ that encodes the number $\lfloor \log_2 x \rfloor + 1$ with first part code γ (with the two parts of γ concatenated) and with second part the same to that of code γ (in binary form the number $x - 2^{\lfloor \log_2 x \rfloor}$). In Table 1, in the third column the δ encoding of the numbers of the first column are depicted. Code δ is larger than γ for most values $x < 15$, but beyond that, it is never worse.

The addresses in these pairs are used to represent certain subquadrants that result from the repetitive subdivision of the initial space. This is done by assigning the numbers 0, 1, 2 and 3 to NW, NE, SW and SW quadrants, respectively. For example the address 1 is used to represent the NE quadrant of the initial space, while the address 10 to represent the NW subquadrant of the NE quadrant of the initial space.

This new indexing scheme, the XBR-tree, introduces a new idea. That is, the region of a child is the subquadrant specified by the address in its pair, minus the subquadrants corresponding to all the previous pairs of the internal node to which it belongs.

Fig. 4 presents an XBR-tree of two levels that consists of only an internal node and two leaves. The *, whenever present in the figures, is used to define the end of each address. The address 2* of the left child in the internal node denotes the SW quadrant of the initial space. On the contrary, the address * of the right child specifies the initial space minus the SW quadrant.

For the sake of presentation, in the following we first discuss the insertion (search) of a point datum in the XBR-tree. When a search or an insertion of a point is performed, descending the tree from the root specifies the

Table 1
Examples of encoding

x	Code γ	Code δ
1	1	1
2	01,0	010,0
3	01,1	010,1
4	001,00	011,00
5	001,01	011,01
6	001,10	011,10
7	001,11	011,11
8	0001,000	00100,000

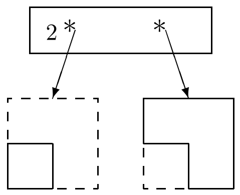

Fig. 4. An XBR-tree with one internal node and two leaves.

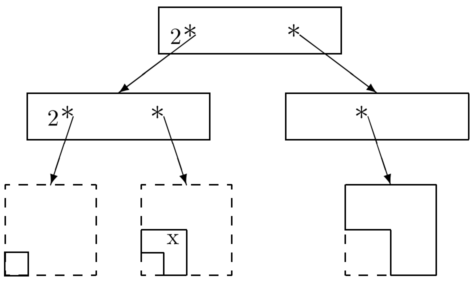

Fig. 5. An XBR-tree with two levels of internal nodes.

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 region under consideration is refined, since we intersect it with the subquadrant of the chosen pair and subtract the subquadrants of the pairs appearing to the left of this pair.

In Fig. 5 a situation where insertions caused the splitting of the left child of Fig. 4 is depicted. This splitting caused a splitting of the internal node too and the creation of a new root. Let us assume that we want to find the data element in the tree, which is marked by x. In that case, we first visit the root and check one by one its pairs. The first pair, with the address 2*, has coordinates that contain the x and therefore we follow its pointer. As we move to the next level, the address 2* of the first pair of the internal node denotes the SW subquadrant of the SW subquadrant of the initial space. This is a region that does not contain x and we choose to follow the pointer of the second pair. Finally, we get to the leaf level and more specifically to the leaf that contains the data element x.

The insertion, or search procedure of a line segment in the XBR-tree is similar to the ones described above, with one key difference: a line segment is stored in the XBR-tree to all the leaves that it crosses. This means that during our descend from the root to the leaf nodes level, in each internal node, we sequentially examine the ⟨address, pointer⟩ pairs and recursively visit every (and not simply the first) child node with a region that is crossed by the specific line segment. During this sequential examination of the pairs of an internal node, we exclude from consideration the part of the line segment that has already fallen within the region of a child node (determined by a previous pair).

6.2. Leaf nodes

The leaves of the XBR-tree contain all the line segments inserted in the tree. The total number of line segments in each leaf node are restricted by a predefined capacity C which cannot be exceeded. When after an insertion of a line segment a specific leaf node overflows then it is split into four equal subquadrants.

All the resulting subquadrants that contain any of the lines segments of the old leaf node are inserted in the internal node. The subquadrants that contain more line segments than the predefined capacity, store these segments in overflow pages. This means that unlike the use of XBR-trees for point data where the tree is completely balanced, in the case of line segments, the XBR-tree is completely height balanced only above the leaf nodes. In practice, since the line segments are short (small in relation to the spatial extent of a leaf node) and the capacity of a leaf node is quite large, the possibility of overflow pages is small (an overflowed leaf node containing many small line segments has very high probability to be split in four non-overflowed subquadrants) and overflow pages do not have a significant effect on the performance of XBR-trees.

Let us assume that after repetitive insertions of line segments in the NW subquadrant of the right leaf of the tree in Fig. 5, this leaf is split into four. Every subquadrant that contains at least one line segment, is inserted in the internal node. Since, in this figure all the line segments are inserted in only one region, the address 00* which specifies this region is inserted in the internal node, before the address *. The modifications of the tree after the split are depicted in Fig. 6.

6.3. Splitting of internal nodes

Whenever an internal node overflows, then a split into two occurs. This split is done in such a way, so that a good balance between the regions of the two resulting nodes is achieved. In order to decide, how this split will be performed, we first construct a quadtree. The nodes of this quadtree contain all the quadrants which exist in the internal node of the XBR-tree to be split. Such an internal node is depicted in Fig. 7a and its regions in Fig. 7b.

The addresses comprising this internal node are all subdividing its region. Each of these addresses appears in the quadtree as a square node. If we follow the path to any such node, then all the intermediate nodes are marked as circles. It is possible, the square of an address to be the ancestor of other squares. Moreover, the address * is used to define the root.

The number assigned to each node represents the number of the squares that will be freed, if we remove the subtree rooted by this node. This number can be easily

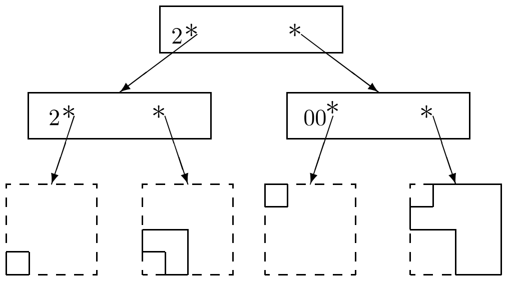

Fig. 6. The XBR-tree after splitting the rightmost leaf of the tree in Fig. 2.

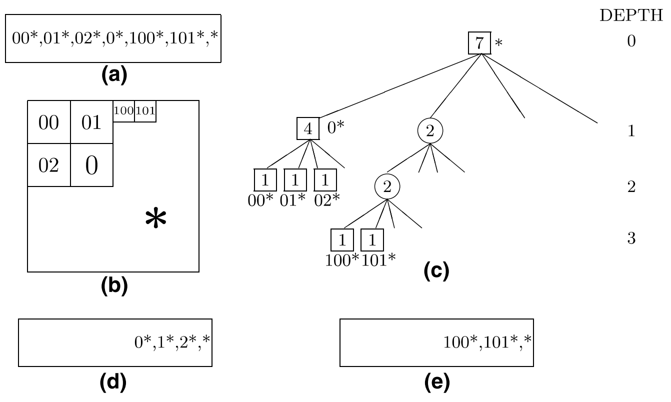

Fig. 7. Splitting of an internal XBR tree node.

calculated by a bottom-up procedure. In Fig. 7c, the number 1 is assigned to all the external squares. These squares are 100*, 101*, 00*, 01* and 02*. Furthermore, the sum of the values of its children plus 1 is assigned to each internal square. For instance, the square of address 0* is assigned the number 4 $(1 + 1 + 1 + 1 = 4)$.

To each circle we only assign the sum of the values of its children. For example, to the second child of the root, which is a circle, 2 is assigned, since it has only one child which is a circle as well with value 2. After the construction of the quadtree, we traverse it with a view to find a node that is not the root, is a square and is assigned the largest number of all squares in the tree.

If we observe the tree in Fig. 7c, we will see that the largest number of squares is 4 and it corresponds to the leftmost root child, with address 0. The two nodes that result from this procedure are presented in Fig. 7d and e. The node of the father node will have to be changed to include the address 0*.

6.3.1. Deletion

Deletion is used while updating the location and the velocity vector 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). Instead of the old line segment, a new one is inserted, to express the expected trajectory from the new current to the next time point.

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, we start from the root and visit each internal node. In each internal node, we sequentially examine the ⟨address, pointer⟩ pairs and recursively visit child nodes with regions that are crossed by the specific 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).

Each leaf node cannot contain less than $x \times C$ line segments, where x is chosen to define the fewer line segments allowed in a leaf node ($x < 0.5$). If a leaf node, from which we remove the line segment, underflows (if it contains less than $x \times C$ 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 rightmost brother of the leaf node).

If this child overflows, then it is split (as described in the "Leaf Nodes" subsection) and the split may propagate to higher levels (hosting internal nodes). Since, the internal nodes do not have a minimum occupancy threshold, the merge process is not applied to the internal nodes. A more sophisticated deletion process that considers alternative merging of an underflowed leaf node with other brother leaves, is currently under development. In this merging, any of the sibling leaves can participate, apart from its rightmost brother. We are also considering the implementation of merging, of the internal nodes.

7. Experimentation

The trees were implemented and the experiments were executed on a Pentium PC of 1600 MHz CPU with 1024 K of main memory. The page size was set to 4 K and the resulting leaf node size was 204 line segments. After experimentation, we came to the conclusion that the use of a buffer of 100 K with least-recently-used page replacement has shown better performance in comparison to other choices.

At time unit 0, an initial location and velocity vector is assigned to each moving object randomly (based on a uniform distribution). The velocity ranges between 0 and 25 m/s. The movement characteristics for each moving object remain constant during each time interval. Since, we consider that in real life applications all moving objects change their position and their velocity vector really often, this is reflected in the experimentation as well. That is, during the execution of the updates (at each time point), all the moving objects randomly change their movement characteristics (position, velocity vector).

There are two different sets of experiments that have been carried out. In the first set of experiments (Figs. 8–19), we considered 1000 time units being separated into 100 equal time intervals, each one of 10 time units. In the second set of experiments (Figs. 20 and 21), however, the initial 1000 time units are not separated into 100 time intervals but into 10, each one of 100 time units. Several cardinalities of the set of moving objects N are used for the experimentation. Furthermore, the spatiotemporal queries carried out are of two types. In the first type of experiments the query under consideration is a window one, whereas in the second type of experiments it is a timestamp one. The spatial (temporal) range of both the two types of queries takes values 0.1, 0.01 and 0.001 of the total space (of the total time range considered). We also presumed that every 10 time units one window or one timestamp query occurs.

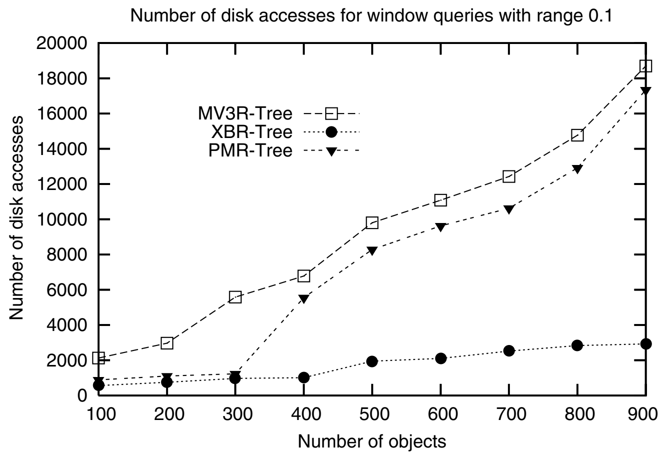

Fig. 8. Disk accesses for window queries with range 0.1.

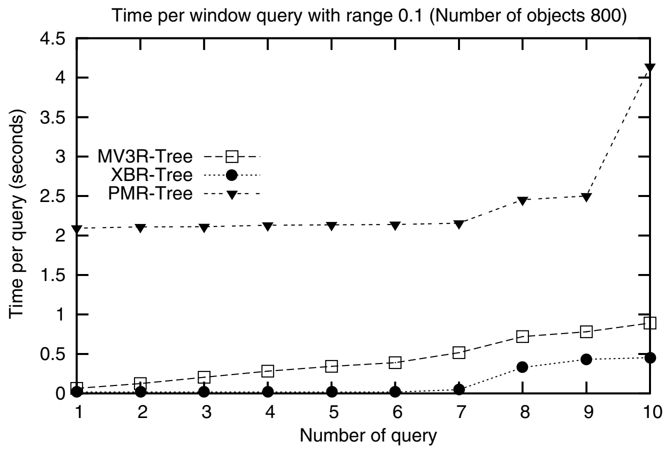

Fig. 11. Elapsed time for window queries with range 0.1.

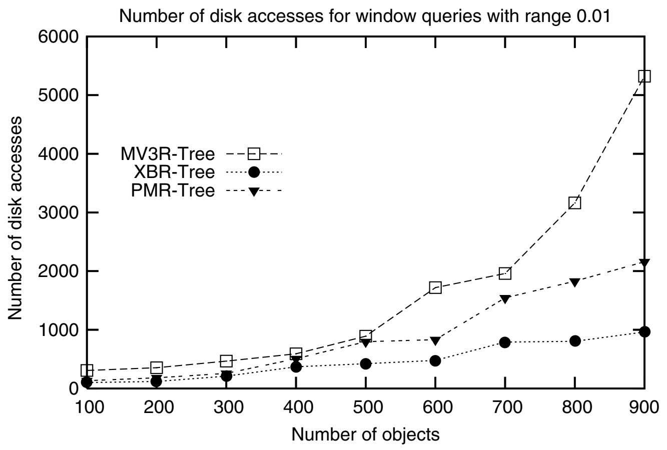

Fig. 9. Disk accesses for window queries with range 0.01.

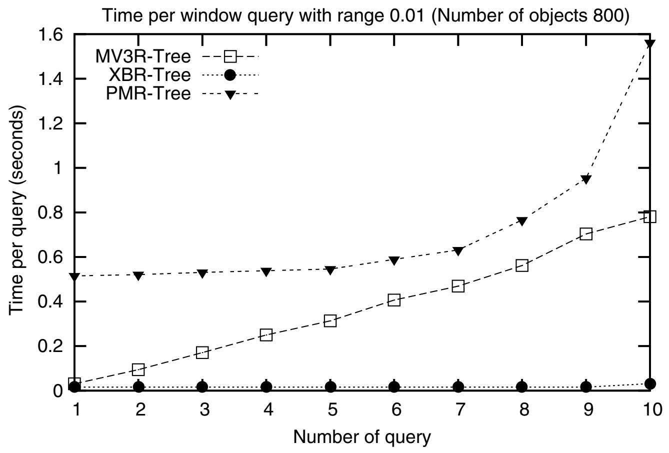

Fig. 12. Elapsed time for window queries with range 0.01.

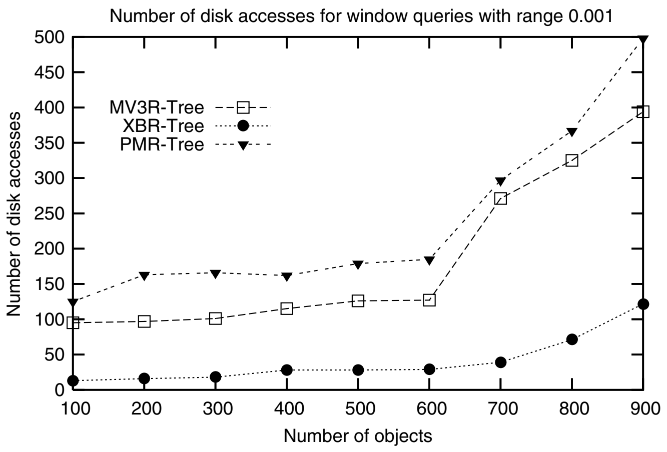

Fig. 10. Disk accesses for window queries with range 0.001.

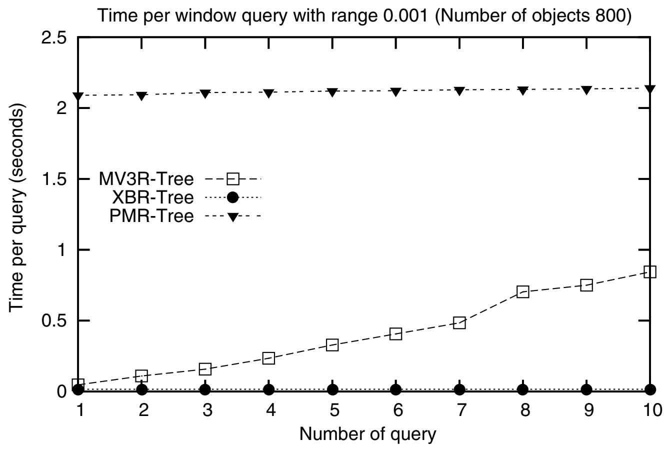

Fig. 13. Elapsed time for window queries with range 0.001.

During the experiments we counted the number of node accesses and the execution time cost during the execution of the window and the timestamp queries.

The first three experiments presented in Figs. 8–10 study the number of the disk accesses. Namely, we counted the number of the disk accesses that were required for the three trees (PMR-tree, XBR-tree and MV3R-tree) during the execution of the window queries. In each experiment the varying parameters are the number of the objects and the size of the window query. The

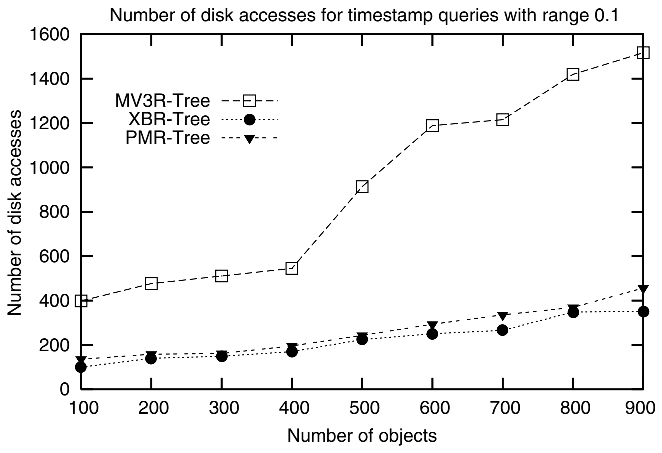

Fig. 14. Disk accesses for timestamp queries with range 0.1.

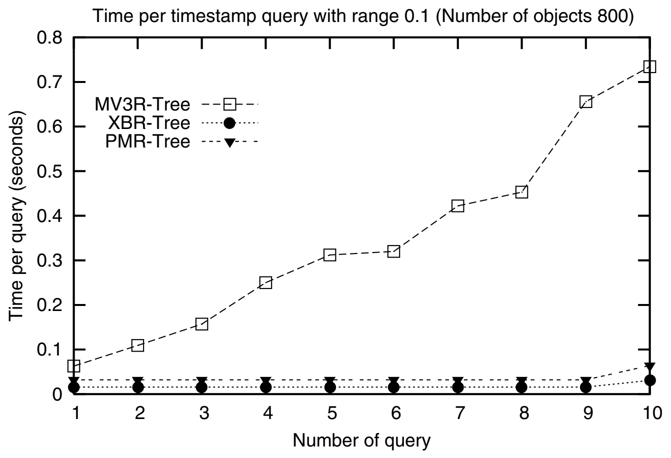

Fig. 17. Elapsed time for timestamp queries with range 0.1.

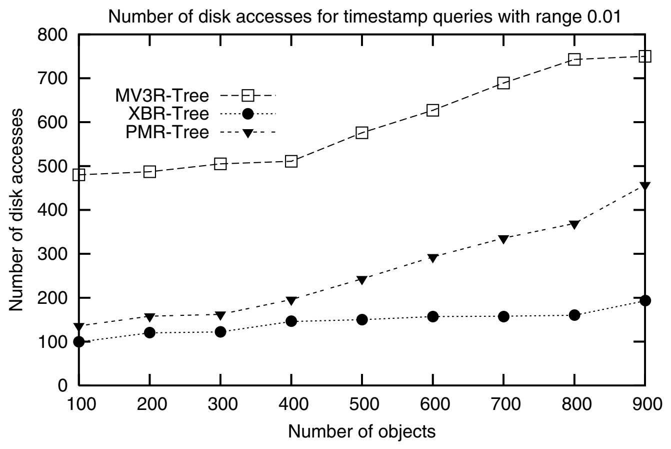

Fig. 15. Disk accesses for timestamp queries with range 0.01.

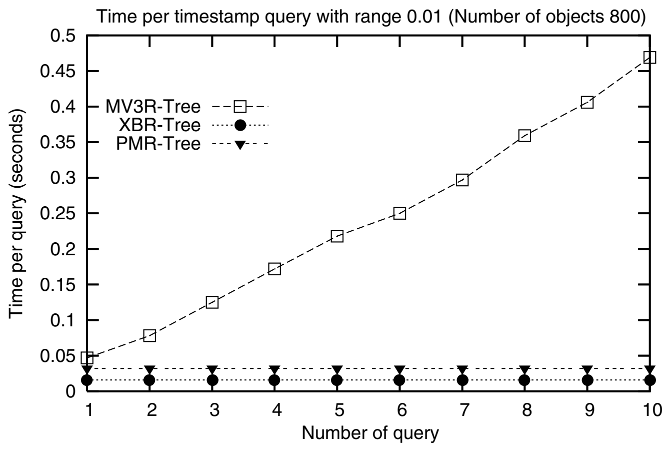

Fig. 18. Elapsed time for timestamp queries with range 0.01.

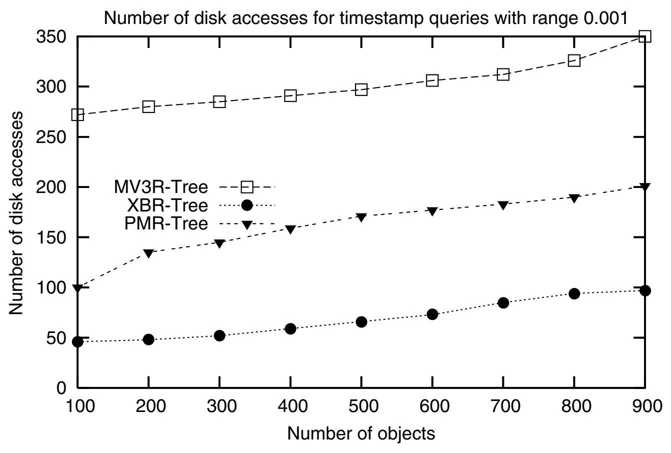

Fig. 16. Disk accesses for timestamp queries with range 0.001.

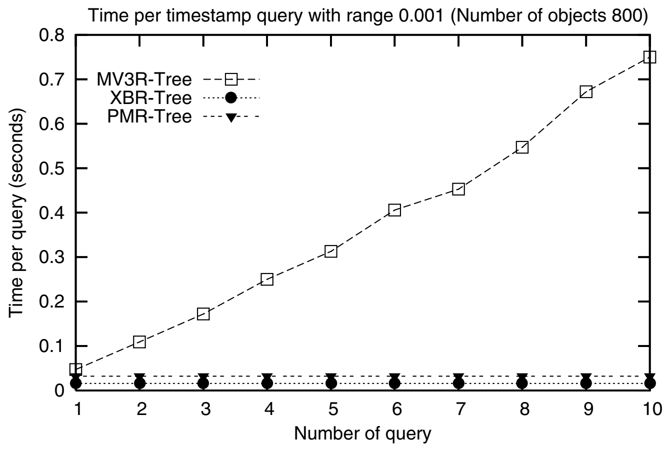

Fig. 19. Elapsed time for timestamp queries with range 0.001.

objects vary from 100 to 900, whereas the query size takes values 0.1, 0.01 and 0.001. In all the three figures, the disk accesses made by the XBR-tree are significantly fewer, than those of both the MV3R-tree and the PMR-tree. Moreover, in Figs. 8 and 9 the PMR-tree appears to have fewer disk accesses than the MV3R-tree, while the situation is the opposite in Fig. 10.

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

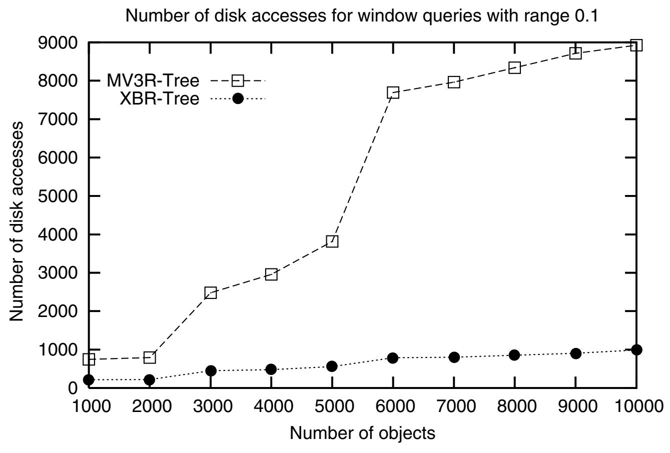

Fig. 20. Disk accesses for window queries.

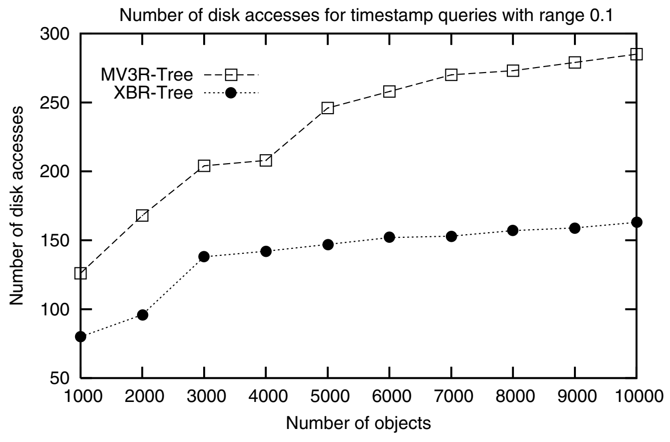

Fig. 21. Disk accesses for timestamp queries.

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. The number of the objects is 800. Since in each experiment there are 10 queries performed, in each figure there are 10 numbers corresponding to the elapsed time. In all the three figures the XBR-tree outperforms the other two, namely the PMR-tree and the MV3R-tree. More precisely, in each query execution, the time spent by it is less than the time spent by the two other trees. Furthermore, unlike the disk accesses, the time spent by the MV3R-tree is less than the PMR-tree, in each query (with the difference growing in 13).

Figs. 14–16 are analogous to Figs. 8–10. That is, they all present the same sorts of results under the same assumptions. The only difference between them is that Figs. 14–16 implement timestamp queries, whereas Figs. 8–10 window queries. In all the figures, the MV3R-tree appears to have made the most disk accesses than the other trees, the PMR-tree and the XBR-tree. On the other hand, the tree with the fewest disk accesses is the XBR-tree.

The next experiment is presented in Figs. 17–19. These figures depict the time elapsed during the execution of ten different timestamp queries. The varying parameter is the number of the query, which takes values from 1 to 10. Moreover, the range of the query is set to 0.1, 0.01 and 0.001 in Figs. 17–19, respectively. The number of the moving objects is 800. By the results presented, we can conclude that the XBR-tree spends less time, than the other two trees (although the PMR quadtree follows very close). The most time, than all the trees, is consumed by the MV3R-tree.

In Figs. 20 and 21 the XBR-tree and the MV3R-tree are used for window and timestamp queries. First, in Fig. 20 the number of the objects varies from 1000 to 10,000 and the range of the window query is set to 0.1. This figure depicts the number of the disk accesses needed by the two trees. Similarly, in Fig. 21, the number of the objects varies from 1000 to 10,000 and the range of the timestamp query is set to 0.1. This figure, also presents the number of the disk accesses, during the execution of the timestamp queries. In this set of experiments, the XBR-tree outperforms the MV3R-tree. The reader may wonder, why in Figs. 20 and 21 we have not presented the results for the PMR quadtree, as well. The answer to this is that for these cardinalities of moving objects the time demanded by the execution of the PMR tree was excessive for gathering the corresponding results.

In an attempt to explain the results of all the above experiments, we can mention that the XBR-tree is a very compact tree, due to the compressed representation of addresses. Therefore, it has a smaller height, and it occupies fewer nodes than the PMR-tree and the MV3R-tree. In Fig. 22 the number of pages occupied by the three structures is presented, when the number of objects ranges from 100 to 900, while in Fig. 23 the number of pages occupied by the XBR and MV3R trees is presented, when the number of objects ranges from 1000 to 10,000 (PMR trees have not been included in this figure, since no query processing results have been presented for these trees for more than 900 objects). It is clear that XBR-trees occupy significantly smaller disk space than the other two structures.

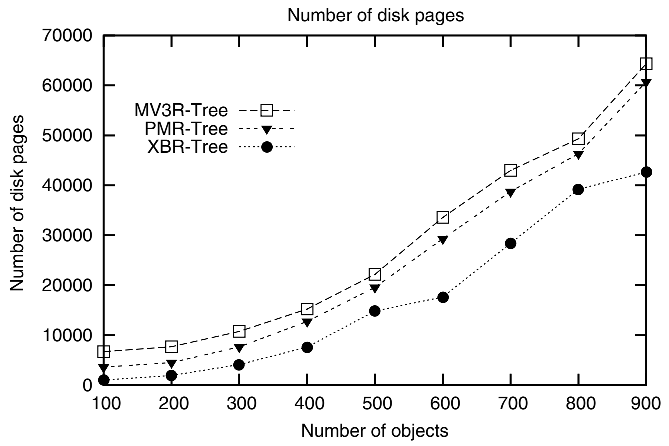

Fig. 22. Number of disk pages occupied by the three structures (number of objects ranges from 100 to 900).

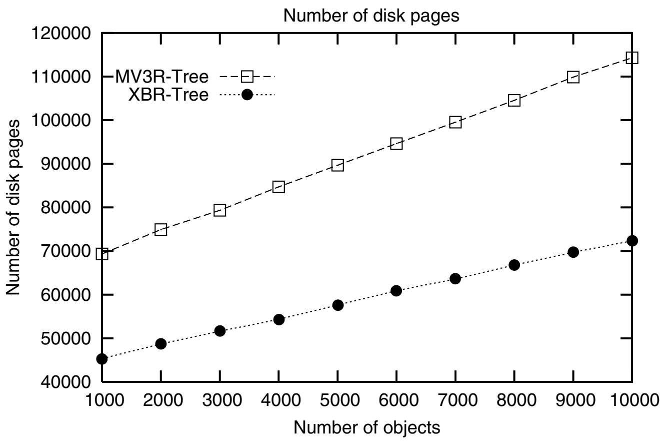

Fig. 23. Number of disk pages occupied by XBR and MV3R trees (number of objects ranges from 1000 to 10,000).

Moreover, (like the PMR quadtree, but unlike the MV3R tree) there is no spatial overlapping between nodes and thus, no revisiting of the same nodes during queries execution. As a result of the above two properties, the disk accesses made during the experiments are significantly fewer, since fewer nodes are accessed, in each query execution (for both the two types of queries, the timestamp and window queries) and the XBR-tree in all the above cases outperforms the other two (the PMR-tree and the MV3R-tree).

8. Conclusion and future work

Since, there exist a great variety of applications that deal with moving objects, their efficient representation, handling and querying are of great interest. Some of the most prominent of these application domains are Mobile Computing and Geographic Information Systems.

In Raptopoulou et al. (2004) the performance of the XBR-tree has been compared to the performance of the PMR-tree for answering window (range) queries. In this paper, we have extended the work of Raptopoulou et al. (2004) by more detailed experimentation and by comparing the XBR-tree with not only the PMR-tree (another quadtree based method), but with the MV3R-tree also (an R-tree based method). The experimentation is based on artificial data (objects that move randomly and change their movement characteristics at each time point) and studies both query processing efficiency, as well as, disk space utilization. Not only window queries, but timestamp queries are considered, as well (unlike Raptopoulou et al., 2004), for several cardinalities of moving objects and sizes of query area. In all the experimentations conducted, it is evident that the XBR-tree outperforms both the PMR-tree and the MV3R-tree: the XBR-tree is clearly more efficient than the other two tree structures in terms of the disk accesses performed and the execution time needed for processing both window and timestamp queries, and at the same time, occupies less disk pages.

Our future research plans include the following:

- The implementation of different types of spatiotemporal queries, besides the timestamp and window ones. Such queries include nearest neighbor queries (“Find the k nearest neighbors of a moving object, during each time interval of its movement”) and spatiotemporal joins (e.g. “From all the airplanes moving in the sky, indicate the ones that intersect clouds while they move”).
- Since for our purposes we have used two different two-dimensional XBR-trees, one for the axes (x, t) and one for the axes (y, t), we plan to examine the combination of these two trees in one of three-dimensions.
- In this paper, we have modelled the movement of objects by polylines, unlike (Tao and Papadias, 2001b) where the discrete positions of the objects are stored. We plan to follow the approach of Tao and Papadias (2001b) also, using an XBR tree version for point data, and compare the performance of the different structures under this alternative modeling of movement.
- In this paper, we have only considered the historic movement of the objects. An evident future extension is to extend our techniques and algorithms for handling of the future movement of the objects, as well (taking advantage of the last segment of each object trajectory that corresponds to the near future, along with the use of other techniques).

References

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

Becker, B., Gshwind, S., Ohler, T., Seeger, B., Widmayer, P., 1996. An asymptotically optimal multiversion B-tree. The VLDB Journal 5 (4), 264–275.

Chon, H.D., Agarwal, D., Abbadi, A.E., 2001. Storage and retrieval of moving objects. In: Proceedings of the 2nd International Conference on Mobile Data Management (MDM), Hong-Kong, China, pp. 173–184.

Elias, P., 1975. Universal codeword sets and representations of the integers. IEEE Transactions on Information Theory 21, 194–203.

Guttman, A., 1984. R-trees: a dynamic index structure for spatial searching. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Boston, MA, pp. 47–57.

Hadjieleftheriou, M., Kollios, G., Tsotras, V.J., Gunopoulos, D., 2002. Efficient indexing of spatiotemporal objects. In: Proceedings of the 8th International Conference on Extending Database Technology (EDTB), Prague, Czech Republic, pp. 251–268.

Hoel, E.G., Samet, H., 1991. Efficient processing of spatial queries in line segment databases. In: Proceedings of the 2nd International Symposium on Spatial Databases (SSD), Zurich, Switzerland, pp. 237–256.

Ishikawa, Y., Kitagawa, H., Kawashima, T., 2002. Continual neighborhood tracking for moving objects using adaptive distances. In: Proceedings of the International Database Engineering and Applications Symposium (IDEAS), Edmonton, Alberta, pp. 54–63.

Kalashnikov, D.V., Prabhakar, S., Hambrusch, S.E., Aref, W.G., 2002. Efficient evaluation of continuous range queries on moving objects. In: Proceedings of the 13th International Conference on Database and Expert Systems Applications (DEXA), Aix-en-Provence, France, pp. 731–740.

Kollios, G., Gounopoulos, D., Tsotras, V.J., 1999a. Nearest neighbor queries in a mobile environment. In: Proceedings of the International Workshop on Spatio-temporal Database Management (STDBM), pp. 119–134.

Kollios, G., Gunopoulos, D., Tsotras, V., 1999b. On Indexing mobile objects. In: Proceedings of the 18th ACM Symposium on Principles of Database Systems (PODS), Philadelphia, PA, pp. 261–272.

Kumar, A., Tsotras, V.J., Faloutsos, C., 1998. Designing access methods for bitemporal databases. IEEE Transaction on Knowledge and Data Engineering 10 (1), 1–20.

Lazaridis, I., Porkaew, I., Mehrotra, S., 2002. Dynamic queries over mobile objects. In: Proceedings of the 8th International Conference on Extending Database Technology (EDTB), Prague, Czech Republic, pp. 269–286.

Lomet, D., Salsberg, B., 1989. Access methods for multiversion data. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Portland, OR, pp. 315–324.

Moreira, J., Ribeiro, C., Abdessalem, T., 2000. Query operations for moving objects database systems. In: Proceedings of the 8th ACM International Symposium on Advances in Geographic Information Systems (GIS), Atlanda, GA, pp. 108–114.

Nascimento, M.A., Silva, J.R.O., 1998. Towards historical R-trees. In: Proceedings of the 13th ACM Symposium on Applied Computing (SAC), Atlanda, GA, pp. 235–240.

Nelson, R.C., Samet, H., 1986. A consistent hierarchical representation for vector data. Computer Graphics 20 (4), 197–206.

Pfoser, D., Jensen, C.S., Theodoridis, Y., 2000. Novel approaches to the indexing of moving object trajectories. In: Proceedings of the 26th International Conference on Very Large Data Bases (VLDB), Cairo, Egypt, pp. 395–406.

Procopiuc, C.M., Agarwal, P.K., Har-Peled, S., 2002. STAR-tree: an efficient self-adjusting index for moving objects. In: Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX), San Fransisco, CA, pp. 178–193.

Raptopoulou, K., Vassilakopoulos, M., Manolopoulos, Y., 2004. Towards quadtree-based moving objects databases. In: Proceedings of the 8th East European Conference on Advances in Databases and Information Systems (ADBIS), Budapest, Hungary, pp. 230–245.

Saltenis, S., Jensen, C.S., Leutenegger, S., Lopez, M., 2000. Indexing the positions of continuously moving objects. In: Proceedings of the ACM SIGMOD International Conference on Management of Data, Dallas, TX, pp. 331–342.

Sistla, A.P., Wolfson, O., Chamberlain, S., Dao, S., 1997. Modeling and querying moving objects. In: Proceedings of the 13th IEEE International Conference on Data Engineering (ICDE), Birmingham, UK, pp. 422–432.

Tao, Y., Papadias, D., 2001a. Efficient historical R-trees. In: Proceedings of the 13th International Conference on Scientific and Statistical Database Management (SSDBM), Fairfax, VA, pp. 223–232.

Tao, Y., Papadias, D., 2001b. MV3R-tree—a spatio-temporal access method for timestamp and interval queries. In: Proceedings of the 27th International Conference on Very Large Data Bases (VLDB), Roma, Italy, pp. 431–440.

Tayeb, J., Ulusoy, O., Wolfson, O., 1998. A quadtree based dynamic attribute indexing method. The Computer Journal 41 (3), 185–200.

Tzouramanis, T., Vassilakopoulos, M., Manolopoulos, Y., 1998. Overlapping linear quadtrees: a spatiotemporal indexing method. In: Proceedings of the 6th ACM International Symposium on Advances in Geographic Information Systems (GIS), Bethesda, MD, pp. 1–7.

Tzouramanis, T., Vassilakopoulos, M., Manolopoulos, Y., 2000. Multiversion linear quadtrees for spatiotemporal data. In: Proceedings of the 4th East-European Conference on Advances in Databases and Information Systems (ADBIS-DASFAA), Prague, Czech Republic, pp. 279–292.

Tzouramanis, T., Vassilakopoulos, M., Manolopoulos, Y., 2003. Overlapping linear quadtrees and spatio-temporal query processing. The Computer Journal 43 (4), 325–343.

Vassilakopoulos, M., Manolopoulos, Y., 1999. External balanced regular (x-BR) trees: new structures for very large spatial databases. In: Proceeding 7th Panhellenic Conference on Informatics, Ioannina, Greece, pp. III.61–III.68.

Wolfson, O., Xu, B., Chamberlain, S., Jiang, L., 1998. Moving objects databases: issues and solutions. In: Proceedings of the 10th International Conference on Scientific and Statistical Database Management (SSDBM), Capri, Italy, pp. 111–122.

Xu, X., Han, J., Lu, W., 1990. RT-tree: an improved R-tree index structure for spatio-temporal databases. In: Proceedings of the 4th International Symposium on Spatial Data Handling (SSDH), Zurich, Switzerland, pp. 1040–1049.

Zhu, H., Su, J., Ibarra, O.H., 2002. Trajectory queries and octagons in moving object databases. In: Proceedings of the 11th ACM International Conference on Information and Knowledge Management (CIKM), McLean, VA, pp. 413–421.

Zobel, J., Moffat, A., Sacks-Davis, R., 1992. An efficient indexing technique for full-text database systems. In: Proceedings of the 18th International Conference on Very Large Data Bases (VLDB), Vancouver, British Columbia, pp. 352–362.