Predictive Join Processing between Regions and Moving Objects*

Antonio Corral[1], Manuel Torres[1], Michael Vassilakopoulos[2], and Yannis Manolopoulos[3]

[1] Dept. of Languages and Computing, University of Almeria, 04120 Almeria, Spain.
E-mail: acorral,mtorres@ual.es
[2] Dept. of Informatics with Applications in Biomedicine,
University of Central Greece, Papasiopoulou 2-4, 35100, Lamia, Greece, and
Dept. of Informatics, Alexander TEI of Thessaloniki, 57400 Greece.
E-mail: mvasilako@ucg.gr
[3] Dept. of Informatics, Aristotle University, 54124 Thessaloniki, Greece.
Email: manolopo@csd.auth.gr

Abstract. The family of R-trees is suitable for indexing various kinds of multidimensional objects. TPR*-trees are R-tree based structures that have been proposed for indexing a moving object database, e.g. a database of moving boats. Region Quadtrees are suitable for indexing 2-dimensional regional data and their linear variant (Linear Region Quadtrees) is used in many Geographical Information Systems (GIS) for this purpose, e.g. for the representation of stormy, or sunny regions. Although, both are tree structures, the organization of data space, the types of spatial data stored and the search algorithms applied on them are different in R-trees and Region Quadtrees. In this paper, we examine a spatio-temporal problem that appears in many practical applications: processing of predictive joins between moving objects and regions (e.g. discovering the boats that will enter a storm), using these two families of data structures as storage and indexing mechanisms, and taking into account their similarities and differences. With a thorough experimental study, we show that the use of a synchronous Depth-First traversal order has the best performance balance (on average), taking into account the I/O activity and response time as performance measurements.

Keywords: Moving objects, TPR-trees, R-trees, linear quad-trees, query processing, joins.

1 Introduction

The recent advances of technologies in mobile communications and global positioning systems have increased users attention to an effective management of

* Supported by the Almacenes de Datos Espacio-Temporales basados en Ontologias project (TIN2005-09098-C05-03), funded by the Spanish Ministry of Science and Technology.

information on the objects that move in 2-dimensional space. Those moving objects send their current positions, which can be forwarded periodically to the users for different purposes (e.g. making decision, etc.). This position information is spatio-temporal, since spatial locations of objects change with time. A database that stores information for a large number of objects locations changing with time is called a *moving object database*.

Users queries issued on moving object databases can be categorized into two types: *past-time queries* and *future-time queries* [17]. The past-time query retrieves the history of dynamic objects movements in the past, while the future-time query predicts movements of dynamic objects in the future [17]. In this paper, we discuss join processing of future-time queries between regions and moving objects.

Spatial data are collected and stored in two main generic formats, called *vector* and *raster*. The basic unit of spatial data in the *vector* format corresponds to discrete real world features represented by points, lines or polygons. In the *raster* alternative, the basic unit of spatial data takes the form of a square grid cell, embedded within a grid of equally sized *pixels* (picture elements). On the other hand, the spatial access methods can be classified according to the two following approaches. First, *data-driven spatial access methods* are organized by partitioning the set of spatial objects, and the partitioning adapts to the distribution of the objects in the embedding space. An example of this approach is the R-tree. Second, *space-driven spatial access methods* are based on partitioning of the embedding two-dimensional space into cells of specific shapes and sizes, independently of the distribution of the spatial objects (objects are mapped to the cells according to some geometric criterion). An example of this approach is the Quadtree. The books [16] and [12] provide excellent information sources for the interested reader about Quadtrees and R-trees, respectively.

There are a number of variations of the R-tree all of which organize multi-dimensional data objects by making use of the Minimum Bounding Rectangles (MBRs) of the objects. We will concentrate on access methods that have the capability of dealing with anticipated future-time queries of moving objects or points (dynamic point of view). Generally, to support the future-time queries, databases store the current positions and velocities of moving objects as linear functions. Up to now, for processing current and future-time queries, several indexing methods have been proposed belonging to the R-tree family and the TPR*-tree [19] is the most widely-used index structure for predicting the future positions of moving points, which can be used for future-time queries.

The Region Quadtree is a space-driven spatial access method, which is suitable of storing and manipulating 2-dimensional regional data (or binary images). Moreover, many algorithms have been developed based on Quadtrees. The most widely known secondary memory alternative of this structure is the Linear Region Quadtree [16]. Linear Quadtrees have been used for organizing regional data in GIS [16].

The contributions of this paper consist in the following:

1. We present predictive join processing techniques between two different access methods, TPR*-trees for moving objects (vector data) and Linear Region Quadtrees for regions (raster data), in order to answer future-time (predictive) queries appearing in practical applications, like "Retrieve all the boats covering by a storm within 1 hour". To the best of our knowledge, this is the first study of spatio-temporal joins between different data formats (vector and raster).
2. We distinguish between two types of such future-time queries, depending on the required result: future-time-interval and future-time-parameterized join queries, between vector and raster data.
3. We present a detailed experimental comparison of the alternative methods and highlight the performance winner, for each experimental setting.

The paper is organized as follows. In Section 2, we review the related literature and motivate the research reported here. In Section 3, a brief description of the TPR*-tree and the Linear Region Quadtree are presented. In Section 4, we present the algorithms that perform the predictive join processing between regions and moving objects. In Section 5, a comparative performance study of proposed algorithms is reported. Finally, in Section 6, conclusions on the contribution of this paper and future work are summarized.

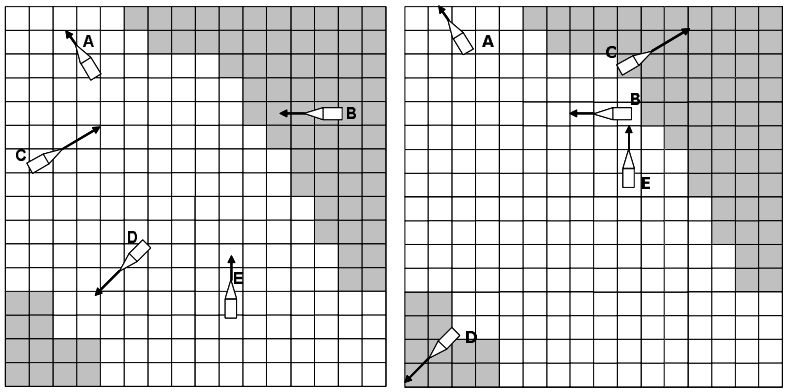
2 Related Work and Motivation

In general, the spatial join combines two sets of spatial objects based on a spatial predicate (usually overlap). Recently, an exhaustive analysis of several techniques used to perform a spatial join taking into account a filter-and-refinement approach has been published in [7]. Regarding spatial joins over static spatial objects, we can classify the spatial join methods in three categories, depending on whether the sets of spatial objects involved in the query are indexed or not. When both sets are indexed, the most influential and known algorithm for joining two datasets indexed by R*-trees was presented in [2], where additionally several techniques to improve both CPU and I/O time have been studied. This algorithm follows a Depth-First synchronized tree traversal order. A breadth-first synchronized tree traversal version to reduce I/O cost was presented in [4]. In the case of just one set being indexed, several spatial join algorithms have been proposed in the literature, and the most representative ones are [9, 13, 11]. In the last category, when both sets are not indexed, the most relevant publications are [14, 10, 6]. We must highlight that, when both sets are indexed, but with incompatible type of indexes, such as by R-trees (hierarchical and non-disjoint indexing for vector data) and by Linear Region Quadtrees based on B^+-trees (hierarchical and disjoint indexing for raster data), the only research work that proposes join algorithms between the different data formats is [3]. The authors proposed several algorithms to perform this spatial join, and the most novel uses a complex buffering system and the FD-order [16] to reduce the I/O cost, while searching in the B^+-tree for the FD-code that overlaps with a point in the R-tree.

From the dynamic point of view, the most representative joins on moving objects have been proposed very recently. In [18], the authors present a set of spatio-temporal queries so-called time-parameterized queries, including the time-parameterized join query, which we will adapt later to our problem setting. In [5], query maintenance algorithms for spatial joins on continuously moving points that support updates were presented. And finally, in [20], the problem of processing continuous intersection join over moving objects, using TPR*-trees, has been addressed.

In practical applications, the need for spatio-temporal predictive joining between different data formats is common. For example, consider Figure 1, where five boats (shapes A-E), along with their moving vectors (arrows) and a storm (gray region) are depicted. At the time instant of Figure 1.a (time I = now), only boat B is in the storm. At the time instant of Figure 1.b (time II = now+10 minutes), boat B has just exited the storm, while boats C and D have just entered the storm. One possible query is: give me all the boats that will be under the storm for the next 9 minutes and 59 seconds (this is an example of a future-time-interval join and the result is: boat B). Note that we assumed that the resolution of time is one second. Another possible query is: give me all the boats that are under the storm now, the time point when this situation will change and the event that will cause the change of the situation (this is an example of a future-time-parameterized join and the result is: boat B is under the storm now, the situation will change in 10 minutes, because boat B will exit and boats C and D will enter the storm). It should be noted that both queries are predictive, since they refer to the future and they are based on the assumption that the moving vectors of a boats do not change (at least significantly) between subsequent updates of the position of a moving object. However, depending on the application, the frequency of updates of objects positions and the possibility of sudden changes of the movement vectors, the result such queries may be enough accurate. Moreover, it should be noted that the storm data are considered static, at least for time periods quite large in comparison to the update frequency of the objects positions.

Although, the queries described above arise naturally in practical applications, the literature (to the best of our knowledge) does not include any techniques for processing them, perhaps due to the different nature of regional and vector data and the different methods used for storing and indexing them. In this paper, we consider that the regional data (e.g. the storm) are stored and indexed using Linear Region Quadtrees (a common choice in GIS systems). The codes of the quad blocks are stored either in B^+-trees [16], or in R*-trees (Oracle, in general, recommends using R-trees over quadtrees [8]), for comparison purposes between the two different alternatives. At time periods large enough for significant changes of the regional data, the whole storage and indexing structure is rebuilt. In this paper, we study the situation within one such time period, during which the regional data are considered static. Moreover, we consider that the changing vector data (e.g. moving boats) are stored and indexed using TPR*-trees [19] (the most widely-used index structure for predicting the future positions of



(a) Initial instant (b) 10 minutes later

Fig. 1. Five moving boats and a storm at a time instant (a) and 10 minutes later (b).

moving points). Thus, we present and study the first algorithms for processing future-time-interval and future-time-parameterized join queries, between vector and raster data.

3 The two access methods

3.1 The TPR*-tree

We assume that the reader is already familiar with the R*-tree [1]. The TPR-tree [15] extends the R*-tree, predicts the future locations of moving objects by storing the location and the velocity of each object at a given time point. The locations of moving objects are indexed using CBRs (Conservative Bounding Rectangles) instead of MBRs (Minimum Bounding Rectangles). A CBR is composed of an MBR, representing the region that covers a set of moving objects at a specific time point, and the maximum and minimum moving velocities of the objects within an MBR at each axis (velocity bounding rectangle, VBR). The location of a moving object at any future-time point can be easily predicted with the location and moving velocity stored in a CBR. The predicted region of a node computed by using a current location of an MBR and its maximum and minimum velocities at each axis is defined as a bounding rectangle.

According to [19], a moving object o is represented with (1) an MBR o_R that denotes its extent at reference time 0, and (2) a velocity bounding rectangle (VBR) $o_V = \{o_{V1-}, o_{V1+}, o_{V2-}, o_{V2+}\}$ where o_{Vi-} (o_{Vi+}) describes the velocity of the lower (upper) boundary of o_R along the i-th dimension ($1 \leq i \leq 2$). Figure 2.a shows the MBRs and VBRs of 4 objects a, b, c, d. The arrows (numbers) denote the directions (values) of their velocities. For example, the VBR of c is $c_V = \{-2, 0, 0, 2\}$, where the first two numbers are for the X-axis. A non-leaf entry is also represented with an MBR and a VBR. Specifically, the MBR (VBR) tightly bounds the MBRs (VBRs) of the entries in its child node.

In Figure 2.a, the objects are clustered into two leaf nodes $N1$ and $N2$, which VBRs are $N1_V = \{-2, 1, -2, 1\}$ and $N2_V = \{-2, 0, -1, 2\}$. Figure 2.b shows the MBRs at timestamp 1 (notice that each edge moves according to its velocity). The MBR of a non-leaf entry always encloses those of the objects in its subtree, but it is not necessarily tight.

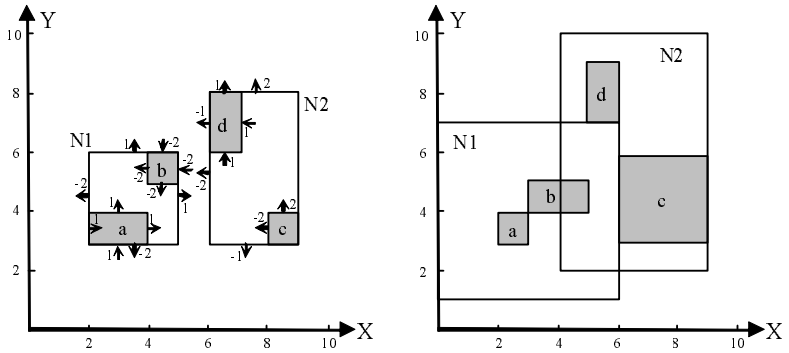


Fig. 2. Entry representation in a TPR*-tree.

The TPR*-tree [19] uses a set of improved algorithms to build the TPR-tree and achieves an almost optimal tree. In general, the TPR*-tree basically uses the same structure as the TPR-tree. During the update operations, however, the TPR-tree employs the insertion and the deletion algorithms of the R*-tree as they are, while the TPR*-tree employs modified versions that reflect objects mobility. This makes it possible to improve the performance of updates and retrievals in the TPR*-tree over the TPR-tree. Since the TPR-tree considers the area, circumference, overlapping, and distance of an MBR only at the time of updates of moving objects, it cannot reflect the property that objects move with time. On the other hand, the TPR*-tree performs updates in such a way that it minimizes the area of a sweeping region, which is an extension of the rectangle that corresponds to a node with time after the updates of moving objects.

Taking into account the insertion strategy, the TPR-tree inserts a moving object into such a node whose MBR extension required is minimum at the time of the insertion. On the other hand, the TPR*-tree inserts a moving object into such a node with a minimum extension of the bounding rectangle after the insertion. Traversing from the root to lower-level nodes, their rectangle extensions required for the insertion are computed, and also are stored into a priority queue. And finally, the optimal node for the insertion is the one having the smallest value. With this strategy, the TPR*-tree requires a cost higher than the TPR-tree for updates. However, its compactness of bounding rectangle, it greatly improves the overall query performance.

3.2 Region Quadtrees

The Region Quadtree is the most popular member in the family of quadtree-based access methods. It is used for the representation of binary images, that is $2^n \times 2^n$ binary arrays (for a positive integer n), where a 1 (0) entry stands for a black (white) picture element. More precisely, it is a degree four tree with height n, at most. Each node corresponds to a square array of pixels (the root corresponds to the whole image). If all of them have the same color (black or white) the node is a leaf of that color. Otherwise, the node is colored gray and has four children. Each of these children corresponds to one of the four square sub-arrays to which the array of that node is partitioned. We assume here, that the first (leftmost) child corresponds to the NW sub-array, the second to the NE sub-array, the third to the SW sub-array and the fourth (rightmost) child to the SE sub-array. For more details regarding Quadtrees see [16].

Region Quadtrees, as presented above, can be implemented as main memory tree structures. Variations of Region Quadtrees have been developed for secondary memory. Linear Region Quadtrees are the ones used most extensively. A Linear Quadtree representation consists of a list of values, where there is one value for each black node of the pointer-based Quadtree. The value a node is an address describing the position and size of the corresponding block in the image. These addresses can be stored in a efficient structure for secondary memory (such as an B-tree or one of its variations). There are also variations of this representations where white nodes are stored too, or variations which are suitable for multicolor images. Evidently, this representation is very space efficient, although it is not suited too many useful algorithms that are designed for pointer-based Quadtrees. The most popular linear implementations are the FL (Fixed Length), the FD (Fixed Length-Depth) and the VL (Variable Length) linear implementation. For more details regarding FL and VL implementations see [16].

In the rest if this paper, like in [3], we assume that Linear Quadtrees are presented with FD-codes stored in a B^+-tree or in an R*-tree. The choice of FD linear representation is not accidental, since it is made of base 4 digits and is thus easily handled using two bits for each digit. Besides, the sorted sequence of FD-codes is a Depth-First traversal of the tree. Since internal and white nodes are omitted, sibling black nodes are stored consecutively in the B^+-tree or, in general, nodes that are close in space are likely to be stored in the same or consecutive B^+-tree leaves. This property helps at reducing the I/O cost of join processing. Since in the same quadtree two black nodes that are ancestor and descendant cannot co-exit, two FD-codes that coincide at all the directional digits cannot exist neither. This means that the directional part of the FD-codes is sufficient for building B^+-tree at all the levels. At the leaf-level, the depth of each black node should also be stored so that images are accurately represented [3]. Since the FD-codes can be transformed in disjoint MBRs we can store the sequence of FD-codes in an R*-tree, and apply of existing query algorithms over this popular spatial index.

4 Future-time Join Algorithms

Before join processing, we must create the indexes that store the static region (linear region quadtree stored in a B$^+$-tree, or an R*-tree) and moving points (TPR*-tree). We have decided to store the sequence of FD-codes in an R*-tree, as an alternative to a B$^+$-tree, because Oracle, in general, recommends using R-trees over quadtrees (due to higher tiling levels in the quadtree that cause very expensive preprocessing and storage costs) [8]. Moreover, the correspondence of the spaces covered by the two structures has been established in [3]. Finally, we have to take into account the two types of future-time join queries that we will study in this paper: future-time-interval join and future-time-parameterized join.

Joining the two structures can be carried out following two join processing techniques: (1) multiple queries and (2) synchronized tree traversal. *Multiple queries* technique performs a window query on the TPR*-tree for each FD-code indexed in the B$^+$-tree, or in the R*-tree. And the *synchronized tree traversal* technique follows a Depth-First or Breadth-First order to traverse both structures during the query processing. More specifically, we have designed and implemented the following five algorithms for future-time-interval join and one algorithm for future-time-parameterized join between regions and moving objects.

4.1 Future-time-interval join

The future-time-interval join receives the time-interval of interest ($[T_{st}, T_{ed}]$) and returns the result, valid only during such as time-interval.

B$^+$ to TPR*-tree join (B-TPR) This algorithm follows the multiple queries technique and it descends the B$^+$-tree from the root to its leftmost leaf. It accesses sequentially the FD-codes present in this leaf, and for each FD-code it performs a predictive window (MBR of FD-code and VBR, which is 0, since the region is static) query in the TPR*-tree (reporting intersections of this FD-code and elements in the TPR*-tree leaves within $[T_{st}, T_{ed}]$. By making use of the horizontal inter-leaf pointers, it accesses the next B$^+$-tree leaf and repeats the previous step. Of course, the reverse alternative (from TPR* to B$^+$, i.e. to scan the entries of the TPR*-tree and perform window queries in the B$^+$-tree) can be easily implemented, and we expect that the results will be very similar.

R* to TPR*-tree join (R-TPR) Assuming that we store the FD-codes in an R*-tree, this algorithm follows the multiple queries technique as well, and it traverses recursively the R*-tree, accessing the MBRs in each node in order of appearance within the node. For the MBR (FD-code) of each leaf accessed, it performs a predictive window (MBR and VBR (it is 0 since the region is static)) query in the TPR*-tree, reporting intersections of this MBR and elements in the TPR*-tree leaves within $[T_{st}, T_{ed}]$. Of course, the reverse alternative (from TPR*

to R*, i.e. to scan the entries of the TPR*-tree and perform window queries in the R*-tree) can be easily implemented, and we expect that the results will be very similar.

Depth-First Traversal join (R-TPR-DFJ) This algorithm follows the synchronized tree traversal technique, using a Depth-First order of both trees for overlap join [2]. It is based on the enclosure property of R-tree nodes: if the MBRs of two internal nodes do not overlap, then there can not be any MBRs below them that overlap. In this case, to apply this technique on TPR*-tree we need an additional function (*compute_intersection_period*) to check if two dynamic entries (MBR and VBR) overlap in the required time-interval.

R-TPR-DFJ(nodeR, nodeTPR, Tst, Ted)
If nodeR and nodeTPR are leaves
For each pair of entries (Rentry, TPRentry)
Save Rentry in RenTPRentry
If *compute_intersection_period*(RenTPRentry, TPRentry, Tst, Ted)
Add TPRentry to the result
Else
For each pair of entries (Rentry, TPRentry)
Save Rentry in RenTPRentry
If *compute_intersection_period*(RenTPRentry, TPRentry, Tst, Ted)
nodeRaux = ReadNodeR(Rentry.p)
nodeTPRaux = ReadNodeTPR(TPRentry.p)
R-TPR-DFJ(nodeRaux, nodeTPRaux, Tst, Ted)

An advanced variant of the algorithm applies a local optimization (because it improves the overlap computation with each node-pair join processing) in order to reduce CPU cost. In particular, when joining two nodes, the overlapping of entries is computed using a plane-sweep technique [2] instead of brute-force nested loop algorithm. In general, the MBRs of each node are sorted on the x-axis, and a merge-like algorithm is carried out, reducing significantly the number of intersection tests. We will called to this variant, *R-TPR-DFJ-PS*.

Breadth-First Traversal join (R-TPR-BFJ) This algorithm follows the synchronized tree traversal technique, using a Breadth-First order of both trees [4]. The algorithm traverses down the two trees synchronously level by level. At each level, the algorithm creates an *intermediate join index* (IJI) and deploys global optimization techniques (e.g. ordering) to improve the join computation at the next level. It terminates when the IJI is created by joining the leaf entries in the R*-tree with the dynamic leaf entries in the TPR*-tree. Again, we need the function (*compute_intersection_period*) to check if two dynamic entries (MBR

and VBR) overlap in the required time-interval. According to [4], we have implemented two orderings of intermediate index join as global optimization: ordering by the sum of the centers (OrdSum) and ordering by center point (OrdCen). In this case, the MBR of the TPR*-tree is the bounding rectangle that covers the VBR in the required time-interval.

B-TPR FD-buffer join (B-TPR-FD) This algorithm follows a particular technique for join processing. It uses a complex buffering system and the FD-order in the B^+-tree to reduce the I/O cost for join processing between an R*-tree (TPR*-tree) and a FD-linear quadtree stored in a B^+-tree [3]. In general, the algorithm works as follows: (1) process each entry of the TPR*-tree root in FD-order; (2) read as many FD-codes as possible for the current entry and store them in the FD-buffer, (3) call recursively the join routine for this entry; (4) when the join routine returns, empty the FD-buffer and repeat the previous two steps until the current entry has been completely checked; (5) repeat for the next entry of the TPR*-tree root. On the other hand, The join routine for a TPR*-tree node and the required time-interval works as follows: (1) if the node is a leaf, check intersections at the required time-interval using the *compute_intersection_period* function and return; (2) If not (non-leaf node), for each child of the node that has not been examined in relation to the FD-codes in FD-buffer, call the join routine recursively.

4.2 Future-time-parameterized join (TP-Join)

In general, this type of future-time join does not receive any parameter and it returns (1) the actual result at the time that the query (join) is emitted, (2) the expiry time of the result given in; and (3) the change that causes the invalidation of the result. That is, the answers are in format of triplets (R, T, C) [18].

A static spatial join returns all pairs of objects from two datasets that satisfy some spatial predicate (usually overlap). The join result changes in the future when: (1) a pair of objects, in the current result, ceases to satisfy the join condition, or (2) a pair not in the result starts to satisfy the condition. In general, we denote the *influence time* of a pair of objects (o_1, o_2) as $T_{INF}(o_1, o_2)$, and it the next timestamp that will change the result [18]. The influence time is 1, if a pair will never change the join result, and the expiry time is the minimum influence time. The influence time of two non-leaf entries, $T_{INF}(E_1, E_2)$, should be a lower bound of the $T_{INF}(o_1, o_2)$ of any two objects o_1 and o_2 in the subtrees of the non-leaf entries E_1 and E_2, respectively.

In general, our join algorithm works as follows: it traverses, in Depth-First order, the two trees (R*-tree and TPR*-tree) simultaneously starting from the two roots. Suppose E_1 and E_2 to be two entries in non-leaf nodes, one from the R*-tree and the other from the TPR*-tree. The traversals go down the subtrees pointed by E_1 and E_2 if one of the following conditions holds: (1) the MBRs of E_1 and E_2 overlap, or (2) $T_{INF}(E_1, E_2)$ is less than or equal to the minimum influence time of all object pairs seen so far (in this case their subtrees may

contain object pairs that cause the next result change). Condition (1) finds the current join pairs and condition (2) identifies the next timestamp. The traversals stop when leaf levels are reached for both trees. Notice that to compute T_{INF} we use the *compute_intersection_period* function that returns the time-interval in which the two entries overlap for a given time-interval.

TP-Join(nodeR, nodeTPR)
If nodeR and nodeTPR are leaves
For each pair of entries (Rentry, TPRentry)
If T_{INF}(Rentry, TPRentry) < T
C = (Rentry, TPRentry);
T = T_{INF}(Rentry,TPRentry);
Else if T_{INF}(Rentry, TPRentry) == T
C = C ∪ (Rentry, TPRentry)
If Rentry overlaps TPRentry
R = R ∪ (Rentry, TPRentry)
Else
For each pair of entries (Rentry, TPRentry)
If (T_{INF}(Rentry, TPRentry) ≤ T) or (Rentry overlaps TPRentry)
nodeRaux = ReadNodeR(Rentry.p)
nodeTPRaux = ReadNodeTPR(TPRentry.p)
TP-Join(nodeRaux, nodeTPRaux)

Note that all the above algorithms are significantly different from existing R-tree based join algorithms. The special properties of TPR*-trees for query processing have been utilized, as well as, the function *compute_intersection_period()* has been used. Besides, previous algorithms that combine FD-Linear-Quadtrees stored in a B^+-tree and R*-trees have been adapted for use with TPR*-trees (not a trivial task).

5 Experimental Results

In this section, we have evaluated the performance of our predictive join algorithms over regional data (black-white images of $2^{11} \times 2^{11}$ pixels) and moving points using synthetic (uniform distribution) and real data (24493 populated places of north-america). The regional data correspond to visible spectrums of areas of California (Sequoia data). Notice that, since $n = 11$, an FD-code for such an image requires $2 \times 11 + \lceil \log_2(11+1) \rceil = 26$ bits. For the moving objects, each object is associated with a VBR such that on each dimension, the velocity value distribution is uniform in the range [0,5]. All experiments were performed on an Intel/Linux workstation with a Pentium IV 2.5 GHz processor, 1 GByte of main memory, and several GBytes of secondary storage, using the gcc compiler. The node size for the tree structures (B^+-tree, R*-tree and TPR*-tree) is

1 KByte, according to [19]. The performance measurements are: (1) the number of page accesses and (2) the response time (elapsed time) reported in seconds.

Our first experiment seeks the most appropriate LRU buffer size (nodes) for our predictive join algorithms that will be used in the next experiments. We have considered the following configuration: the number of moving objects (synthetic-uniform) is 10000, the query time-interval is [0, 5], and the LRU buffer size is variable (8, 32, 64, 128 and 512 Kbytes, or nodes). In Figure 3, the results of 6 algorithms are shown. We have to highlight that R-TPR-DFJ-PS was very similar to R-TPR-DFJ; and R-TPR-BFJ1 (OrdSum) obtained very similar results to R-TPR-BFJ2 (OrdCen), for this reason we only show the results of R-TPR-DFJ-PS (R-TPR-DFJ enhanced with the *plane-sweep technique*) and R-TPR-BFJ1. B-TPR and R-TPR are the most I/O-consuming when the buffer sizes are small, but when they are large enough ($\geq$32); these algorithms obtain the best behavior (the best is B-TPR, 3112 node accesses for 512 nodes in the LRU buffer). The reason of this excellent I/O behavior is due to the good spatial locality of the TPR*-tree, there is a high probability that in the next predictive window query over the TPR*-tree, a great part of this index remains in the LRU buffer. On the other hand, these algorithms are the most time-consuming, due to the join processing technique, i.e. multiple queries. R-TPR-DFJ-PS shows an excellent behavior with respect to the I/O cost and response time, mainly due to the use of synchronized tree traversal as a join processing technique. R-TPR-BFJ1 does not improve the previous algorithms, due to the high cost of managing the global IJI, although for big LRU buffer size the I/O activity is acceptable. R-TPR-FD is an algorithm designed for reducing the number of node accesses, and for this reason it gets good behavior for this performance measurement, but it consumes a lot of time to return the final result, due to the continuous searches the FD-codes in the B^+-tree and the management of the FD-buffer. Finally, the TP-join query gets a similar behavior to the R-TPR-DFJ-PS for I/O activity, but the response time is slightly larger, since this algorithm has to report three answers (R, T, C) and the influence time [18].

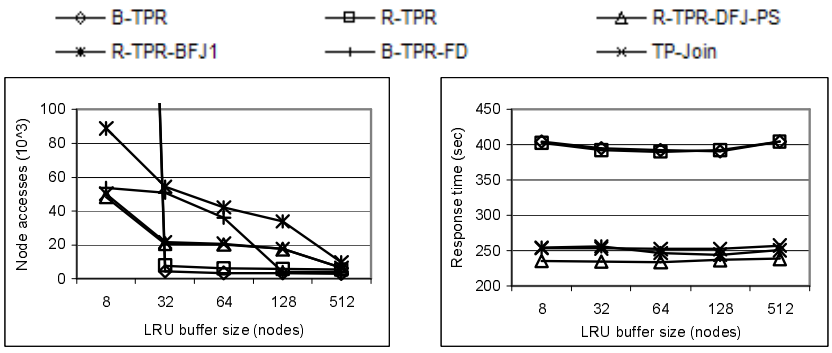


Fig. 3. Performance comparison with respect to the LRU buffer sizes.

In the second experiment, we have studied the behavior of the predictive join algorithms when the cardinality of the moving objects datasets varies. We have the following configuration: LRU buffer size is 128 nodes, the query time-interval [0, 5], and the cardinality of the datasets is variable (1000, 10000, 50000 and 100000). Figure 4 shows B-TPR and R-TPR get the best results for this LRU buffer size, although they are time-consuming. B-TPR-FD obtains also good behavior for I/O activity until the number of moving points is 100000. R-TPR-BFJ1 needs many node accesses, although it needs an acceptable response time. R-TPR-DFJ-PS is the fastest, although it needs more disk accesses than the join algorithms that use multiple queries as the technique for the join processing. TP-join has similar behavior than R-TPR-DFJ-PS, since they follow the same join processing technique, although the response time is slightly larger.

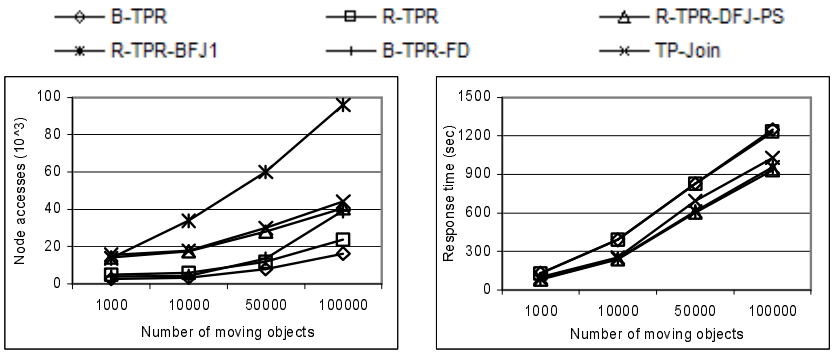


Fig. 4. Performance comparison with respect to the moving objects datasets sizes.

In the third experiment, we have compared the behavior of the predictive join algorithms, varying the query time-interval. We have the following configuration: LRU buffer size is 128 nodes, the cardinality of the moving objects dataset is 10000, and the query time-intervals are [0, 0], [0, 5], [0, 10] and [0, 20]. Since TP-Join does not receive any query time-interval, the result is not reported. We have to highlight that when the time-interval enlarges, the moving objects cover more space along their movement and the MBRs that cover them grow as well. This fact generates more overlaps between MBRs, and it increments the number of operations necessary to solve the join.

Figure 5 shows again that B-TPR and R-TPR get the best results for the number of node accesses (B-TPR needs less node accesses than R-TRP to perform the same query), although they are time-consuming. Moreover, B-TPR-FD gets interesting results for small query time-intervals, but for larger ones it needs more node accesses. The best performance balance corresponds to R-TPR-DFJ-PS, which consumes a reasonable quantity of node accesses, but it is the fastest for small query time-interval ([0, 0] and [0, 5]), but for [0, 10] and [0, 20] sizes it is slightly slower than R-TPR-BFJ1. The join algorithm which uses a Breadth-

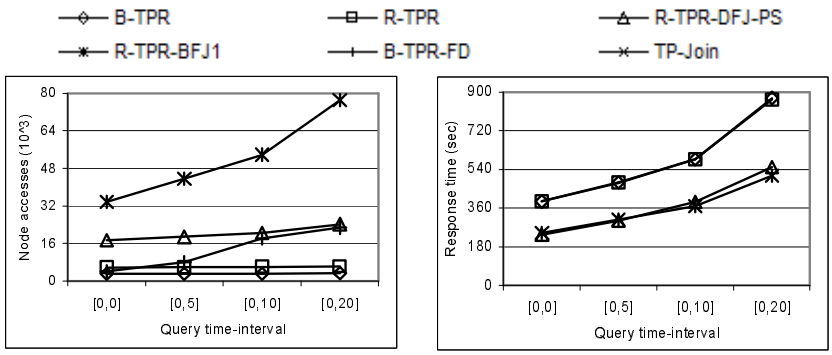


Fig. 5. Performance comparison with respect to the query time-interval sizes.

First traversal order of both trees has a surprising behavior; it is I/O-consuming, but it is the fastest for large query time-intervals. It is mainly due to the applied query processing technique, since it only considers the overlapped elements level by level (there is no backtracking) and when the leaf nodes are reached, the result is reported. Of course, it also needs additional main memory resources to store the IJI.

The last results reported here are a summary (representative set) of experiments with real moving object datasets. In general, the tendencies are very similar to the synthetic uniform data. For example, the left chart of Figure 6 is very similar to the left chart of Figure 3, except for the LRU buffer size for B-TPR and R-TPR, starting from which they become the best ($\geq$128). Moreover, observe that in the right charts of Figures 5 and 6, the trends are quite similar, where R-TPR-DFJ-PS is the fastest for small query time-interval and for larger ones R-TPR-BFJ1 consumes slightly less time to report the final result; and B-TPR and R-TPR are the most expensive alternatives for response time consumed (R-TPR is slightly faster than B-TPR for the same query).

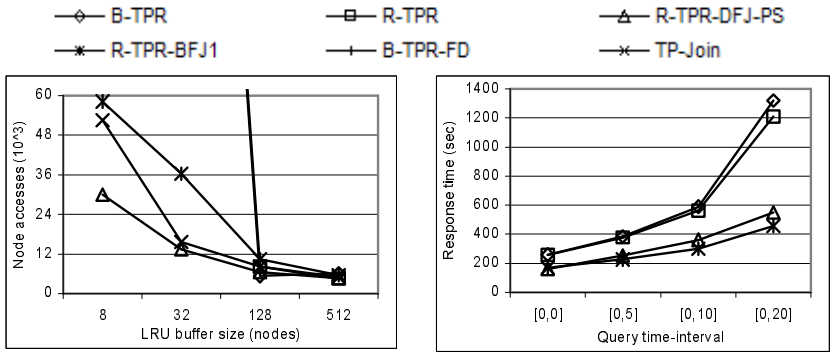


Fig. 6. Performance for real data, for LRU buffer and query time-interval sizes.

From the previous performance comparison (for synthetic and real data), the most important conclusions are the following: (1) B-TPR and R-TPR are appropriate when we have available enough resources for buffering. (2) The predictive join algorithms which use a Breadth-First traversal order of both trees (R-TPR-BFJ) have a good behavior for large query time-interval and buffer sizes, obtaining the best response time. (3) B-TPR-FD reports interesting results with respect to the I/O activity, but it is time-consuming due to the high computational cost of managing the FD-buffer. (4) Finally, the predictive join algorithms which use a synchronous Depth-First traversal order (R-TPR-DFJ-PS and TP-join) have the best performance balance (on average) in all executed experiments, taking into account the I/O activity and response time.

6 Conclusions and Future Work

The contribution of this paper falls within in the study of a spatio-temporal problem that appears in real-world applications: processing of predictive joins between moving objects and regions. To the best of our knowledge, this is the first study of its kind. For this purpose: (1) We have considered two types of future-time join queries: future-time-interval join and future-time-parameterized join. To solve these queries, we have used the TPR*-tree, which is the most widely-used index structure for predicting the future positions of moving points, and the Linear Region Quadtree (FD Linear Quadtree, as pointerless representation) stored in a B^+-tree, or in R*-tree. (2) We have proposed several join algorithms between these two indexes, following two join processing techniques: multiple queries and synchronized tree traversal, to solve such future-time join queries. (3) By extensive experimental results, we have shown that the use of a synchronous Depth-First traversal order (R-TPR-DFJ-PS and TP-join) has the best performance balance (on average), considering the I/O activity and response time.

Future research may include the extension of our algorithms to perform continuous intersection joins [20], and the use of moving and/or changing, instead of static, regions.

References

1. N. Beckmann, H.P. Kriegel, R. Schneider and B. Seeger. The R*-tree: an Efficient and Robust Access Method for Points and Rectangles. SIGMOD Conference, pp. 322-331, 1990.
2. T. Brinkhoff, H.P. Kriegel and B. Seeger, B. 1993. Efficient Processing of Spatial Joins Using R-trees. SIGMOD Conference, p.p. 237-246, 1993.
3. A. Corral, M. Vassilakopoulos and Y. Manolopoulos. Algorithms for Joining R-trees and Linear Region Quadtrees. SSD Conference. LNCS Vol. 1651, p.p. 251-269, 1999.
4. Y.M. Huang, N. Jing, and E. Rundensteiner. Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations. VLDB Conference, p.p. 395-405, 1997.

5. G.S. Iwerks, H. Samet and K.P. Smith. Maintenance of K-nn and Spatial Join Queries on Continuously Moving Points. TODS 31(2), p.p. 485-536, 2006.
6. E.H. Jacox and H. Samet. Iterative Spatial Join.TODS 28(3), p.p. 230-256, 2003.
7. E.H. Jacox and H. Samet. Spatial Join Techniques. TODS 32(1), article 7, p.p. 1-44, 2007.
8. R.K. Kothuri, S. Ravada and D. Abugov. Quadtree and R-tree Indexes in Oracle Spatial: A Comparison using GIS Data. SIGMOD Conference, p.p. 546-557, 2002.
9. M.L. Lo and C.V. Ravishankar. Spatial Joins Using Seeded Trees. SIGMOD Conference, p.p. 209-220, 1994.
10. M.L. Lo and C.V. Ravishankar. Spatial Hash-Joins. SIGMOD Conference, p.p. 247-258, 1996.
11. N. Mamoulis and D. Papadias. Slot Index Spatial Join. TKDE 15(1), p.p. 211-231, 2003.
12. Y. Manolopoulos, A. Nanopoulos, A.N. Papadopoulos and Y. Theodoridis. R-Trees: Theory and Applications. Springer, 2006.
13. A.N. Papadopoulos, P. Rigaux and M. Scholl. A Performance Evaluation of Spatial Join Processing Strategies. SSD Conference. LNCS, Vol. 1651, p.p. 286-307, 1999.
14. J.M. Patel and D.J. Dewitt. Partition Based Spatial-Merge Join. SIGMOD Conference, p.p. 259-270, 1996.
15. S. Saltenis, C. S. Jensen, S. T. Leutenegger and M. A. Lopez. Indexing the Positions of Continously Moving Objects. SIGMOD Conference, p.p. 331-342, 2000.
16. H. Samet. Applications of Spatial Data Structures. Addison-Wesley, Reading MA, 1990.
17. A.P. Sistla, O. Wolfson, S. Chamberlain and S. Dao. Modeling and Querying Moving Objects. ICDE Conference, pp. 422-432, 1997.
18. Y. Tao and D. Papadias. Time-Parameterized Queries in Spatio-Temporal Databases. SIGMOD Conference, pp. 334-345, 2002.
19. Y. Tao, D. Papadias and J. Sun. The TPR*-tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. VLDB Conference, p.p. 790-801, 2003.
20. R. Zhang, D. Lin, K. Ramamohanarao and E. Bertino. Continuous Intersection Joins Over Moving Objects. ICDE Conference, pp. 863-872, 2008.