Proceedings of the 26th VLDB Conference, Cairo, Egypt, pp. 395-406, September 10-14, 2000.

Novel Approaches to the Indexing of Moving Object Trajectories*

Dieter Pfoser
Dept. of Computer Science
Aalborg University
Denmark
pfoser@cs.auc.dk

Christian S. Jensen
Dept. of Computer Science
Aalborg University
Denmark
csj@cs.auc.dk

Yannis Theodoridis
Computer Technology Institute
Patras, Greece
ytheod@cti.gr

Abstract

The domain of spatiotemporal applications is a treasure trove of new types of data and queries. However, work in this area is guided by related research from the spatial and temporal domains, so far, with little attention towards the true nature of spatiotemporal phenomena. In this work, the focus is on a spatiotemporal sub-domain, namely the trajectories of moving point objects. We present new types of spatiotemporal queries, as well as algorithms to process those. Further, we introduce two access methods this kind of data, namely the Spatio-Temporal R-tree (STR-tree) and the Trajectory-Bundle tree (TB-tree). The former is an R-tree based access method that considers the trajectory identity in the index as well, while the latter is a hybrid structure, which preserves trajectories as well as allows for R-tree typical range search in the data. We present performance studies that compare the two indices with the R-tree (appropriately modified, for a fair comparison) under a varying set of spatiotemporal queries, and we provide guidelines for a successful choice among them.

1 Introduction

Space and time are two properties inherent to any object in the real world. If modeled in a database (Spaccapietra et al. 1998, Tryfona and Jensen, 1999), efficient ways to query these kinds of data have to be provided. Research efforts in the fields of spatial and temporal databases to index the respective data are numerous, and as we shall see later on in this work, serve as the basis for a more far-reaching effort into spatiotemporal data. It is sometimes not enough to take the "best" of both worlds to obtain a satisfying solution to a given spatiotemporal problem. In our context the problem is the indexing and querying of spatiotemporal information. More specifically, in this work we focus on data stemming from the *movement of spatial point objects*. We consider point objects, since in many applications, the size and shape of an object is of no importance—only its position matters. Examples include navigational systems, but also the thriving developments in mobile computing (Barbará 1999).

The data obtained from moving point objects is similar to a "string," arbitrary oriented in 3D space, where two dimensions correspond to space and one dimension corresponds to time. By sampling the movement of a point object, we obtain a polyline, instead of a "string," representing the trajectory of the moving point object. In pure geometrical terms, this object movement is termed a *trajectory* (cf. Figure 1). In the sequel, we will use "movement" and "trajectory" interchangeably.

When designing an access method, we not only have to be aware of the nature of the data, but must also know the types of queries, the method is to be used for. Typical queries in spatial and temporal databases are range (window/interval) queries. Queries for spatiotemporal data are often more demanding due to the extra semantics involved. An object's trajectory can be treated as spatial (3D) data itself, and thus may besupported by a spatial access method.

In the literature, the following taxonomy exists: (a) work on indexing the present positions of objects and asking future queries (Kollios et al. 1999, Saltenis et al. 2000) and (b) work on indexing the past positions of objects and asking historical queries. Within the latter category, into which the present work also belongs, most approaches deal with spatial data changing discretely over time and do not take continuous changes into account. Examples include R-trees for multimedia data (Theodoridis et al. 1996), overlapping Quadtrees (Tzouramanis et al. 1998) and R-tree variations for spatial data (Nascimento et al. 1999).

A problem not addressed by using any of the above access methods is the *preservation* of trajectories. Related

* Research partially supported by the CHOROCHRONOS project, funded by the European Commission DG XII, contract no. ERBFMRX-CT96-0056. The first and the second author were additionally supported by the Nykredit Corporation.

Proceedings of the 26th International Conference on Very Large Databases, Cairo, Egypt, 2000

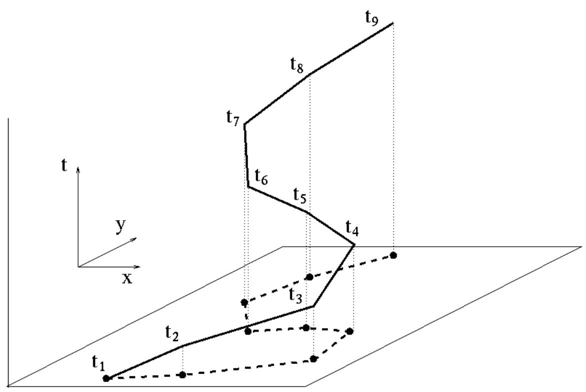


Figure 1: The movement of a spatial object and the corresponding trajectory

work treats data merely as a set of line segments, regardless of whether some belong to the same trajectory. Line segments are grouped together merely according to spatial properties such as proximity. This is not optimal, since certain types of queries require access to parts of the whole trajectory. Further, the presented spatiotemporal data has another particularity; it is considered to be append-only with respect to time, i.e., data grows mainly in the temporal dimension (Theodoridis et al. 1998).

To capture the particularities of spatiotemporal data and queries, we propose two access methods. The first, the Spatio-Temporal R-tree (hereafter called STR-tree), organizes line segments not only according to spatial properties, but also by attempting to group the segments according to the trajectories they belong to. We term this property *trajectory preservation*. The second, the Trajectory-Bundle tree (hereafter called TB-tree), aims only for trajectory preservation and leaves other spatial properties aside.

The outline of the paper is as follows. Section 2 describes the nature of the data as well as the type of queries encountered in applications with moving point objects. Section 3 presents the algorithms comprising the proposed access methods. Section 4 presents query-processing algorithms. Section 5 gives performance studies that compare both methods with the "classic" R-tree, appropriately modified to take gain of the knowledge that the entries to be indexed are line segments. Finally, Section 6 gives conclusions and directions for future research[1].

2 Moving Objects: Data and Queries

In this section, we discuss spatiotemporal data by giving a motivating example. We further introduce sampling as a method to measure positions over time. Also, we introduce a set of queries that are of importance in the given application context.

2.1 Trajectories

The optimization of transportation, especially in highly populated areas, is a very challenging task that may be supported by an information system. A core application in this context is fleet management. Vehicles equipped with GPS devices transmit their positions to a central computer using either radio communication links or mobile phones. At the central site, the data is processed and utilized. In order to record the movement of an object, we would have to know the position at all times, i.e., on a continuous basis. However, GPS and telecommunications technologies only allow us to sample an object's position, i.e., to obtain the position at discrete instances of time, such as every few seconds.

A first approach to represent the movements of objects would be to simply store the position samples. This would mean that we could not answer queries about the objects' movements at times in-between those of the sampled positions. Rather, to obtain the entire movement, we have to interpolate. The simplest approach is to use linear interpolation, as opposed to other methods such as polynomial splines (Bartels et al. 1987). The sampled positions then become the endpoints of line segments of polylines, and the movement of an object is represented by an entire polyline in 3D space. The solid line in Figure 1 represents the movement of a point object. Space and time coordinates are combined to form a single coordinate system. The dashed line shows the projection of the movement on the 2D plane (Pfoser and Jensen 1999). Figure 2 illustrates the spatiotemporal workspace (the cube in solid lines) and several trajectories (the solid polylines). Time moves in the upward direction, and the top of the cube is the time of the most recent position sample. The wavy-dotted lines at the top symbolize the growth of the cube with time.

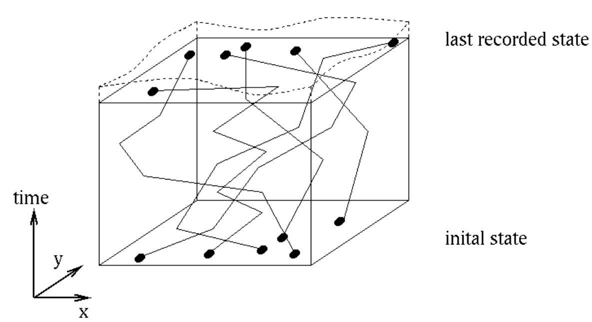


Figure 2: Trajectories of moving point objects in spatiotemporal workspace

Semantically, the temporal dimension is different from the two spatial dimensions. In classical spatial databases, only positional information is available. In our case, however, we have also *derived information*, e.g., speed, acceleration, traveled distance, etc. Consequently, information is derived from the combination of spatial and temporal data. Further, we do not only store a number of

[1] Although in the sequel we consider objects moving on a 2D plane, extending to 3D space (e.g. movement of planes) is straightforward.

spatial objects in the index, i.e., line segments, but rather have entries that are parts of larger objects, the trajectories. As we will see in the next section, these differences create interesting new and inherently spatiotemporal types of queries.

2.2 Queries

A typical search on sets of objects' trajectories includes a selection with respect to a given range, a search inherited from spatial and temporal databases. Queries of the form "*find all objects within a given area* (*or at a given point*) *some time during a given time interval* (*or at a given time instant*)" or "*find the k-closest objects with respect to a given point at a given time instant*" (Theodoridis et al. 1998) remain very important. A query type important in temporal databases is the time-slice query, i.e., in the spatiotemporal context, to determine the positions of (all) moving objects at a given time point in the past (Theodoridis et al. 1996). Using the 3D representation presented in Section 2.1, the time-slice query constitutes a special case of a range query with a query window of *zero extent in the temporal dimension*.

In addition, novel queries become important due to the specific nature of spatiotemporal data. The so-called trajectory-based queries are classified in "*topological*" queries, which involve the whole information of the movement of an object, and "*navigational*" queries, which involve derived information, such as speed and heading.

As such, we distinguish between two types of spatiotemporal queries:

- *coordinate-based queries*, such as point, range, and nearest-neighbor queries in the resulting 3D space, and
- *trajectory-based queries*, involving the topology of trajectories (topological queries) and derived information, such as speed and heading of objects (navigational queries).

Both types of queries will be involved in our performance study in Section 5 while in the sequel we discuss the latter ones in more detail.

2.2.1 Topological Queries

Topological queries involve the whole or a part of the trajectory of an object. They are deemed very important, but also rather expensive. Unfortunately, a definition of a well established set of predicates, such as the 9-intersection model (Egenhofer and Franzosa 1991) for spatial data and the 13 relations between intervals (Allen 1983) for temporal data is not yet available for spatiotemporal data. In one of the first approaches, Erwig and Schneider (1999) discuss extending SQL with the spatiotemporal versions of the basic spatial predicates, *disjoint*, *meet*, *overlap*, *equal*, *covers*, *contains*, *covered-by*, and *inside*, defined by the 9-intersection model as well as composite predicates based on the basic ones, namely *enter* (and its reverse, *leave*), *cross*, and *bypass*.

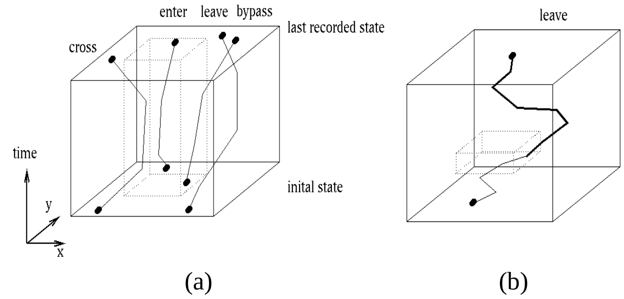


Figure 3: (a) Topological and (b) combined queries

Whether an object *enters*, *crosses*, or *bypasses* a given area can be determined only by examining more than one segment of its trajectory. For instance, an object *entered* into an area with respect to a given time horizon, if the start point of its least recent segment (respectively, the endpoint of its most recent segment) was outside (respectively, inside) the given area. "Recent" here refers to time, e.g., a point is less recent, if its time stamp is older in time. Similar definitions hold for the *leave*, *cross*, and *bypass* predicates, which are also illustrated in Figure 3(a).

2.2.2 Dynamic Information and Navigational Queries

Dynamic information is not explicitly stored, but has to be derived from the trajectory information. The average or top *speed* of an object is determined by the fraction of traveled distance over time. The *heading* of an object is computed by determining a vector between two specified positions. Also, the *area* an object covers is computed by considering the convex hull of its trajectory. From these definitions, it is evident that each property is unique, but depends on the *time interval* considered. For example, the heading of an object in the last ten minutes may have been strictly East, but considering the last hour, it may have been Northeast. The same is true for speed; at the moment, the speed of an object might be 100 mph, but during the last hour, it might average out to 30 mph.

Queries involving speed or heading are expected to be very important in real-life applications. Let us discuss the following examples: "*At what speed does this plane move? What is its top speed?*" (Güting et al. 2000). The former considers the *now* instance as the time horizon, whereas the second one is an aggregation over a longer time period. But again, to compute the result, we have to examine a set of line segments that belong to the same trajectory, as opposed to lie within a spatiotemporal range.

Table 1 summarizes the spatiotemporal query types. We adopt a signature-like notation as presented in (Güting et al. 2000). The "operation" column lists the operations used for several query types and the "signature" column presents the involved types, e.g., a coordinate-based query uses the inside operation to determine the segments within the specified range. The notation {segments} simply refers to a set, it does not capture that this set constitutes one or more trajectories.

Query Type		Operation	Signature
Coordinate-based Queries		overlap, inside, etc.	range × {segments} → {segments}
Trajectory-based Queries	Topological Queries	enter, leave, cross, bypass	range × {segments} → {segments}
	Naviga-tional Queries	traveled distance, covered area (top or average), speed, heading, parked	{segments} → int {segments} → real {segments} → bool

Table 1: Types of spatiotemporal queries

2.2.3 Combined Queries

An important issue in dealing with spatiotemporal queries is to extract information related to (partial) trajectories, i.e., we have to (a) select the trajectories and (b) select the parts of each trajectory we want to return. Selection of trajectories can occur (i) by querying the trajectory identifier, (ii) by selecting a segment of the trajectory using a spatiotemporal range, (iii) by using a topological query, and/or (iv) by using derived information. In the previous examples, we left the identity of the taxi unspecified; it can either be selected by an identifier, e.g., "*taxi no. 120,*" or by spatiotemporal selection, e.g., "*a taxi at the corner of 5th Avenue and Central Park South between 7 a.m. and 7:15 a.m. today*."

In the following we show a more complicated example of combined search: "What were the trajectories of objects after they left Tucson between 7 a.m. and 8 a.m. today, in the next hour?" This query uses the range, "Tucson between 7 a.m. and 8 a.m. today" to identify the trajectories while, "in the next hour" gives a (temporal) range to delimit the parts of the trajectories that we want to retrieve. Figure 3(b) illustrates this principle. The dotted cube represents the spatiotemporal range used when selecting the trajectories, and the polyline stands for a selected trajectory of a moving object. The bold part of the polyline represents the part of the trajectory that is returned (e.g., in the next hour).

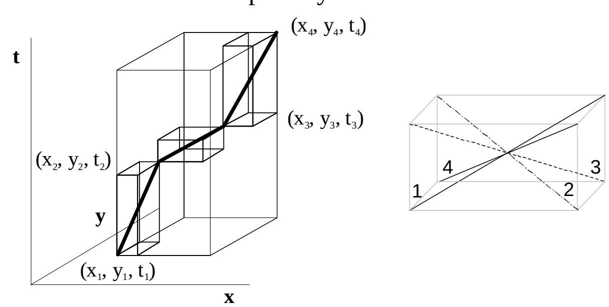
Along these lines, one can construct various query combinations that are plausible in the spatiotemporal application context.

3 The Access Methods

Having described the types of data and queries, the following section defines the two access methods proposed for those types of data and queries. Before that, we will give a short overview of the R-tree (Guttman 1984). The R-tree is a height-balanced tree with the index records in its leaf nodes containing pointers to actual data objects. Leaf node entries are of the form (*id*, *MBB*), where *id* is an identifier that points to the actual object and *MBB* (Minimum Bounding Box) is an *n*-dimensional interval. Non-leaf node entries are of the form (*ptr*, *MBB*), where *ptr* is the pointer to a child node and *MBB* is the covering *n*-dimensional interval. A node in the tree corresponds to a disk page. Every node contains between *m* and *M* entries.

The insertion of a new entry into the R-tree is done by traversing a single path from the root to the leaf level. The path is chosen with respect to the least enlargement criterion (ChooseLeaf algorithm by Guttman (1984)) and covering MBBs are adjusted accordingly. In case an insertion causes splitting of a node, its entries are reassigned to the old node and a newly created one (according to one of the three alternative algorithms, Exhaustive, QuadraticSplit or LinearSplit, proposed by Guttman (1984)). To delete an entry from the R-tree, a reverse insertion procedure applies, i.e., covering MBBs are adjusted accordingly. In case the deletion causes an underflow in a node, i.e., node occupancy falls below *m*, the node is deleted and its entries are re-inserted. When searching an R-tree, we check whether a given node entry overlaps the search window (assuming a range query). If so, we visit the child node and thus recursively traverse the tree. Since overlapping MBBs are permitted, at each level of the index there may be several entries that overlap the search window.

In the context of spatiotemporal data this technique proves to be inefficient. Figure 4(a) shows that in approximating the line segments with MBBs, we introduce large amounts of "dead space." It is evident that the corresponding MBB covers a large portion of the space, whereas the actual space occupied by the trajectory is small. This leads to high overlap and consequently to a small discrimination capability of the index structure.



(a) (b)

Figure 4: (a) approximating trajectories using MBBs, and (b) mapping of line segments in a MBB

Another aspect not captured in R-trees is the knowledge about the specific trajectory a line segment belongs to. To smoothen these inefficiencies (and provide an as fair as possible performance comparison later in Section 5), we modify the R-tree as follows: As can be seen in Figure 4(b), a line segment can only be contained in four different ways in an MBB. This extra information is stored at the leaf level by simply modifying the entry format to (id, MBB, orientation), where the orientation's domain is {1,2,3,4}. Assuming we number the trajectories from 0 to *n*, a leaf node entry is then of the form (id, trajectory#, MBB, orientation).

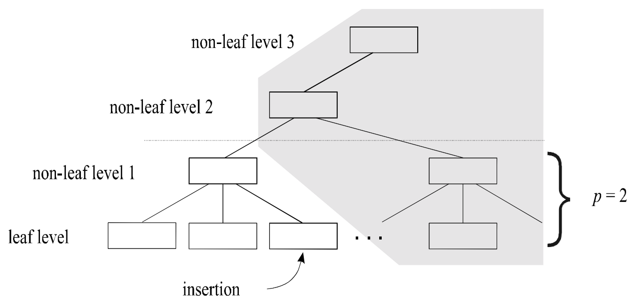


Figure 5: Insertion into the STR-tree

```
Algorithm Insert(N,E)
INS1 Invoke FindNode(N,E)
INS2 IF node N' found,
        IF N' has space,
            insert E
        ELSE
            IF the p-1 parent nodes are full,
                invoke ChooseLeaf(N'',E) on a tree, pointed to by
                N'', which excludes the current branch.
            ELSE invoke Split(N').
     ELSE ChooseLeaf(N,E).

Algorithm FindNode(N,E)
FN1  IF N is NOT a leaf,
        FOR EACH entry E' of N whose MBB intersects with the
        MBB of E,
            invoke FindNode(N',E), where N' is the childnode of
            N pointed to by E'.
     ELSE
        IF N contains an entry that is connected to E,
           RETURN N.
```

Figure 6: STR-tree insert algorithm

Although these suggestions are simple to implement and improve the efficiency of the R-tree to index line segments as parts of trajectories of moving points, we argue that this is not enough and query processing is still problematic. Therefore, we propose two novel approaches in indexing trajectories, the STR-tree and the TB-tree.

3.1 The STR-tree

The STR-tree is an extension of the (appropriately modified, as discussed previously) R-tree to support efficient query processing of trajectories of moving points. The two access methods differ in their insertion/split strategy.

3.1.1 Insertion Algorithm

The insertion process is considerably different from the procedure known from the R-tree. As already mentioned, the insertion strategy of the R-tree is based on the (purely spatial) least enlargement criterion. On the other hand, insertion in the STR-tree not only considers *spatial closeness*, but also partial *trajectory preservation*, i.e., we try to keep line segments belonging to the same trajectory together. As a consequence, when inserting a new line segment, the goal should be to insert it as close as possible to its predecessor in the trajectory. Thus, insertion in the STR-tree involves a new algorithm, FindNode, which returns the node that contains the predecessor. As for the insertion, if there is room in this node, the new segment is inserted there. Otherwise, we have to apply a node split strategy. In Figure 5, we show a sample index in which the node returned by FindNode is marked with an arrow.

The ideal characteristics for an index suitable for object trajectories would be to decompose the overall space according to time, the dominant dimension in which "growth" occurs, while simultaneously preserving trajectories. In the following, we describe the Insert algorithm shown in Figure 6, which includes an additional parameter, called the *preservation parameter*, *p*, that indicates the number of levels we "reserve" for the preservation of trajectories. When a leaf node returned by FindNode is full, the algorithm checks whether the *p*-1 parent nodes are full (in Figure 5, for $p = 2$, we only have to check the node drawn in bold at non-leaf level 1). In case *one of them is not full*, the leaf node is split. In case *all of the p-1 parent nodes are full*, Insert invokes ChooseLeaf on the subtree including all the nodes further to the right of the current insertion path (the gray shaded tree in Figure 5). In the sequel, the so-called ChooseLeaf and QuadraticSplit algorithms will be used without further details, since they are identical to Guttman's original algorithms.

The extended version of this paper (Pfoser et al. 2000) experimentally established that the best choice of a preservation parameter is $p = 2$. A smaller *p* decreases the trajectory preservation and increases the spatial discrimination capabilities of the index. The converse is true for a larger *p*.

3.1.2 Split Algorithm

Since the goal is to preserve trajectories in the index, splitting a leaf node requires an analysis of what kinds of segments are contained in a node. Any two segments in a leaf node may belong to the same trajectory or not, and, suppose they belong to the same trajectory, may have common endpoints or not. Thus a node can contain four different types of segments:

- *disconnected* segments, i.e., segments not connected to any other segment in the node,
- *forward*- (respectively, *backward*-) *connected* segments, i.e., the top (respectively, bottom) endpoint, in other words, the more (respectively, less) recent endpoint, of such a segment is connected to the bottom (respectively, top) endpoint of another segment belonging to the same trajectory,
- *bi-connected* segments, i.e., both (top and bottom) endpoints of such a segment are connected to the (bottom and top, respectively) endpoint of two other segments belonging to the same trajectory.

With this, we can distinguish the three split scenarios of Figure 7. In case (a), where all segments are *disconnected*, the QuadraticSplit algorithm is invoked to determine the split. In case (b), where not all but at least

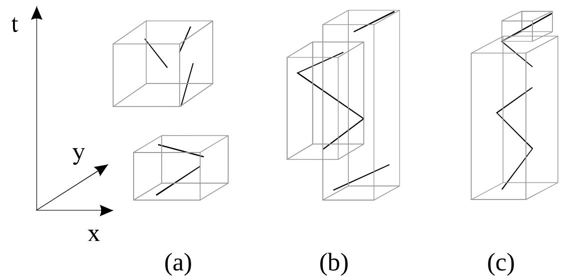


Figure 7: Different split scenarios

Algorithm Split(N)
S1 IF node is a non-leaf node,
invoke **SplitNon-leafNode(N)**.
ELSE invoke **SplitLeafNode(N)**.

Algorithm SplitNon-leafNode(N)
SNN1 Put the new entry into a new node and keep the old one as it is

Algorithm SplitLeafNode(N)
SLN1 IF entries in node are all disconnected segments,
invoke **QuadraticSplit(N)**.
ELSE IF node contains disconnected, and other types of segments,
put all disconnected segments in a new node.
ELSE IF node contains single and disconnected segments,
put the newest single connected segment in new node

Figure 8: STR-tree split algorithm

one segment is *disconnected*, the *disconnected* segments are placed into the newly created node. Finally, in case (c) where no *disconnected* segments exist, the most recent (i.e., with respect to time) *backward-connected* segment is placed in the newly created node.

Figure 8 summarizes the split algorithm. The general idea is to put newer and thus more recent segments into new nodes. Consequently, new segments are much likelier inserted into these nodes, i.e., these nodes have a higher “insertion potential” than the ones containing older nodes. This potential allows us also to relax the constraint of minimum node capacity *m*, known from the R-tree, when splitting a node. Finally, splitting non-leaf nodes is simple, in that we only create a new node for a new entry. Using this insertion and split strategy, we obtain an index that preserves trajectories and considers time as the dominant dimension when decomposing the occupied space.

3.2 The TB-Tree

The TB-tree is fundamentally different from the previously presented access methods. The STR-tree introduces a new insertion/split strategy to achieve trajectory orientation, while not compromising the space discrimination capabilities of the index too much. Apart from this, the STR-tree is an R-tree based access method. The TB-tree takes a more radical step. An underlying assumption when using the R-tree is that all inserted geometries are independent. In our context this translates

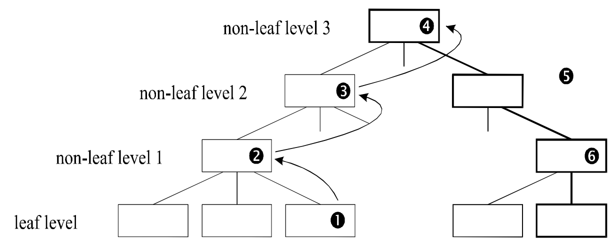


Figure 9: Insertion into the TB-tree

Algorithm Insert(N,E)
INS1 Invoke **FindNode(N,E)**
INS2 IF node N’ is found,
IF N’ has space,
insert new segment.
ELSE
create new leaf node for new segment
ELSE
create new leaf node for new segment

Figure 10: TB-tree insert algorithm

to all line segments being independent. However, line segments are parts of trajectories and this knowledge is only implicitly maintained in the R-tree and the STR-tree structures. With the TB-tree, we aim for an access method that *strictly preserves trajectories* such that a leaf node only contains segments belonging to the same trajectory, thus the index is best understood as a *trajectory bundle*. This approach is only possible when making some concessions to the most important R-tree property, node overlap or spatial discrimination. As a drawback, line segments independent from trajectories that lie spatially close will be stored in different nodes. As the overlap increases, the space discrimination decreases, and, thus, the classical range query cost increases. However, by giving up on space discrimination, we gain on trajectory preservation. As we shall see later, this property is important for answering “pure spatiotemporal” queries[2].

3.2.1 Insertion Algorithm

The goal is to “cut” the whole trajectory of a moving object into pieces, where each piece contains *M* line segments, with *M* being the fanout, i.e., a leaf node contains *M* segments of the trajectory. Figure 9 illustrates the insertion procedure. Important stages throughout the procedure are marked with black, circled numbers 1-6.

The insertion algorithm is formally shown in Figure 10. To insert a new entry, we simply have to find the leaf node that contains its predecessor in the trajectory. We start by traversing the tree from the root and step into every child node that overlaps with the MBB of the new line segment. We choose the leaf node containing a

[2] Both the (modified) R-tree and the STR-tree store entries of the format (*id*, *trajectory#*, *MBB*, *orientation*) at the leaf level. Since the TB-tree does not allow segments from different trajectories to be stored in the same leaf node, the *trajectory#* is assigned to the node rather than to each entry. Thus, the format of a leaf node entry is (*id*, *MBB*, *orientation*) while *trajectory#* can be stored once in the header of the leaf node

segment connected to the new entry (stage 1 in Figure 9). The finding of a segment is summarized in the FindNode algorithm, which is identical to that of the STR-tree. In case the leaf node is full, a *split* strategy is needed. Splitting a leaf node would violate our principle of total trajectory preservation. Thus, we instead create a new leaf node. In our example, we step up the tree until we find a non-full parent node (stages 2 through 4). We choose the right-most path (stage 5) to insert the new node. If there is room in the parent node (stage 6), we insert the new leaf node as shown in Figure 9. In case it is full, we split it by creating a new node at (non-leaf) level 1 that has the new leaf node as its only descendant. If necessary, the split is propagated upwards. Illustratively, the TB-tree is growing from left to right, i.e., the left-most leaf node was the first and the right-most was the last, we inserted.

3.2.2 Trajectory Preservation

At this point one might argue that this strategy leads to an index with a high degree of overlap. This would certainly be the case if it were arbitrary 3D data that was indexed. However, in our case, we only "neglect" two out of three dimensions, the spatial dimensions, with respect to space discrimination. The temporal dimension offers a given space discrimination, in that data is inserted in an append-only fashion (Theodoridis et al. 1998).

As such, the structure of the TB-tree is actually a set of leaf nodes, each containing a partial trajectory, organized in a tree hierarchy. In other words, a trajectory is distributed over a set of disconnected leaf nodes. As we shall see later on when discussing about query processing, it is necessary to be able to retrieve segments based on their trajectory identifier. A simple solution we have implemented is to connect leaf nodes by a superimposed data structure. We choose a doubly linked list that connects leaf nodes including parts of the same trajectory in a way that preserves trajectory *evolution*. Figure 11 gives a part of a TB-tree structure and a trajectory illustrating this approach. For clarity, the trajectory is drawn as a band rather than a line. The trajectory symbolized by the gray band is fragmented across six nodes, c1, c3, etc. In the TB-tree these leaf nodes are connected through a linked list.

By visiting an arbitrary leaf node, these links allow us to retrieve the (partial) trajectory at minimal cost: Considering a fanout f at a leaf node, the size of the partial trajectory contained in the leaf node is f. Among the segments stored and assuming that $f \geq 3$, it is by definition that f-2 segments are *bi-connected*, one is *forward-connected* and one is *backward-connected*. To find the remaining segments of the same trajectory, one has just to follow the pointers of the linked list to the next and previous leaf nodes.

4 Query Processing

Section 2 described various types of queries as they occur in spatiotemporal applications. In this section, we present

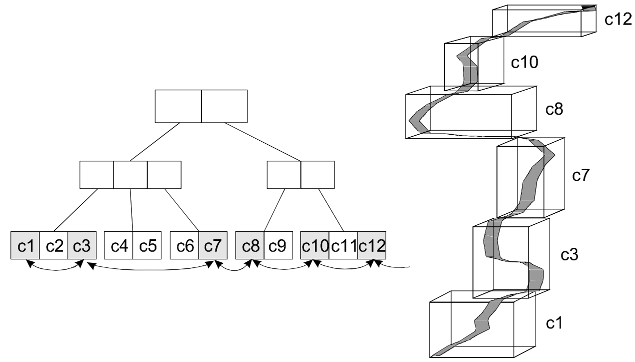


Figure 11: The TB-tree structure

the algorithms for processing those queries using the three access methods. The queries and algorithms can be classified as *coordinate-based*, *trajectory-based*, or *combined* (cf. Section 2).

The processing of coordinate-based queries is a straightforward extension of the classical range query processing using the R-tree; the idea is to descend the tree with respect to coordinate constraints until the entries are found in the leaf nodes. Trajectory-based queries comprise topological and navigational queries. Due to space limitations, we omit the presentation of topological query processing (and the corresponding discussion in the performance section); for details, please refer to Pfoser et al. (2000).

Algorithms for combined queries are different in that not only a spatial, but also a combined search, is performed, i.e., we not only retrieve all entries contained in a given sub-space (range query), but retrieve entries belonging to the same trajectory.

We will devise separate algorithms, on one hand, for the R-tree and the STR-tree and, on the other hand, for the TB-tree. The algorithm for the TB-tree is different because this method provides the data structure of a linked list to retrieve partial trajectories.

4.1 Combined Search in the R-Tree and the STR-Tree

The first step in processing combined queries is to retrieve an initial set of segments based on a spatiotemporal range. We apply the range-search algorithm used in the R-tree. The idea is to descend the tree with respect to intersection properties until the entries are found in the leaf nodes. In Figure 12, we search the tree using the cube c_1 and retrieve two segments of trajectory t_2 (labeled 1 and 2), and four segments of trajectory t_1 (labeled 3 to 6). The six segments are shown in darker gray contained in cube c_1. This completes the first stage of the combined search.

In the second stage, we extract partial trajectories. We now take each of the found segments and try to find its connecting segment, first, in the same leaf node, and, second, in other leaf nodes. Consider segment 1 of trajectory t_2. We find two segments, one connected to the top endpoint (forward connected) and one connected to the bottom endpoint (backward connected).

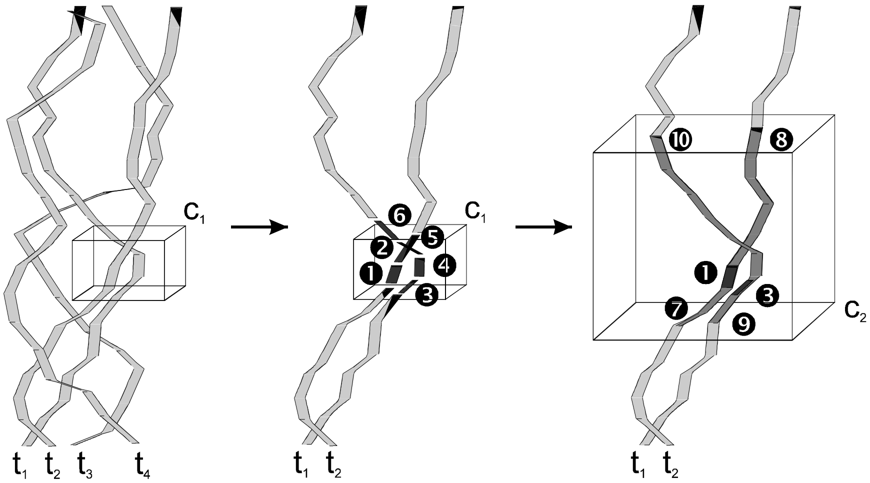


Figure 12: Stages in combined search

We may find those segments in the same leaf node, or we may have to search in other leaf nodes. Searching in other leaf nodes is conducted as a range search, with the endpoint of the segment in question as a predicate. Arriving at the leaf level, the algorithm checks whether a segment is connected to the segment in question in the specified way. Using this recursive approach, we retrieve more and more segments of the trajectory. The algorithm continues until a newly found segment is outside cube c_2. The last segments returned for segment 1 are segments 7 and 8.

Figure 13 outlines the combined search algorithm. One problem remains, namely that of not retrieving the same trajectory twice. The initial range search retrieves two segments, 1 and 2, of trajectory t_2. By using both segments as a starting point, we will retrieve the same trajectory twice. To avoid this, we store the *trajectory#* once it is retrieved and check before querying a new trajectory whether it was retrieved already. In our example, if we use segment 1 first to retrieve a partial trajectory t_1 and store this information, we omit retrieving it again for segment 2.

4.2 Combined Search in the TB -Tree

The combined search algorithm of the TB-tree is similar to the one presented above. The difference lies in how the partial trajectories are retrieved. The R-tree and the STR-tree structures provide little help in retrieving trajectories, i.e., connected segments, but offer only a modified range search algorithm. The linked lists of the TB-tree allow us to retrieve connected segments without searching.

The first stage in combined searching is the same as before. Here, for the seed segments—in our example segments 1 and 2 for t_2 and segments 3 to 6 for t_1—we have to retrieve a partial trajectory contained in the outer range c_2. Again, we have two possibilities: a connected segment can be in the same leaf node or in another node. If it is in the same, finding it is trivial. If it is in another node, we have to follow the next (previous) pointer to the next (previous) leaf node (cf. Section 3.2.2).

Although the approach to retrieve partial trajectories is different, we have to take care not to retrieve the same trajectory more than once (cf. Section 4.2). Once a partial trajectory is retrieved, we store its id, and, before retrieving another trajectory, we check whether it was retrieved already.

Figure 14 contains the updates to the combined search algorithm as presented in Figure 13.

5 Performance Comparison

In this section, we aim at comparing the three access methods and establishing conditions, which are optimal for each one. This allows us to delimit the situations in which each access method is useful. Thus, we compare the access methods under varying sets of data and queries. The performance studies were conducted using C implementations of the three access methods. For the parameters in the experiments, we have chosen the page size for the leaf and non-leaf nodes to be 1024 bytes. With this page size, the R-tree and the STR-tree fanout is 28 and 36 for leaf and non-leaf nodes, respectively. Since the leaf node structure of the TB-tree is different, the fanout is 31 and 36 for leaf and non-leaf nodes, respectively.

5.1 Datasets

Unlike spatial data, where there exist several popular real datasets for experimentation purposes (e.g., the TIGER-Line files of geographic features, such as roads, rivers, lakes, boundaries covering the entire United States), well-known and widely accepted spatiotemporal datasets for experimental purposes are missing. Due to the lack of real data, our performance study consists of experiments on synthetic datasets. We utilize the GSTD generator of spatiotemporal datasets (Theodoridis et al. 1999) to create trajectories of moving objects under various distributions. GSTD allows the user to generate a set of line segments stemming from a specified number of moving objects.

```
Algorithm CombinedSearch(N,range1,range2)
CS1 IF N is NOT a leaf,
        FOR EACH entry E' of N whose MBB intersects with
        range1,
            invoke CombinedSearch(N',E), where N' is the
            childnode of N pointed to by E'.
      ELSE
        for all entries E that satisfy range1 AND whose trajectory
        was not yet retrieved,
            invoke DetermineTrajectory(N,E)

Algorithm DetermineTrajectory(N,E,range2)
DT1 Loop through N and find segment E' that is fwd connected to
    E
DT2 WHILE found AND E' is within range2
        Add E' to set of solutions,
        Loop through N and find segment E' that is connected to
        the new E
DT3 IF not found (but within range)
        invoke FindConnSegment(root,E,forward)
        repeat from DT1
DT4 the same as above for bwd connected

Algorithm FindConnSegment(N,E,direction)
FCS1 IF N is NOT a leaf,
         FOR EACH entry E' of N whose MBB intersects with the
         MBB of E,
             invoke FindConnSegment(N',E,direction), where N'
             is the childnode of N pointed to by E'.
       ELSE
         IF N contains an entry that is direction connected to E,
             RETURN N.
```

Figure 13: R-tree and STR-tree: CombinedSearch algorithm for trajectory-based queries

```
Algorithm FindConnSegment(E,N,direction)
FCS1 Set N to be the node pointed to be the direction pointer
```

Figure 14: TB-tree: CombinedSearch algorithm update

Probability functions are used to describe the movement of the objects as a combination of several parameters. More precisely, the user can specify the initial positional distribution of the objects in the unit workspace $[0, 1)^2$ as well as the stepping in time and space for each movement using either uniform, Gaussian, or skewed probability functions.

The parameters of the generator are given the following values: The initial distribution of points is Gaussian, i.e., all points are distributed around the center of the workspace. The movement of points is always ruled by a random distribution of the form `random(-x,x)`, thus achieving an unbiased spread of the points in the workspace. The number of different possible snapshots (i.e., the temporal resolution) is held constant at 100K. Finally, the number of moving objects (i.e., trajectories) varies between 10 and 1000, resulting in datasets consisting of between 15K and 1500K entries (i.e., line segments).

5.2 Space Utilization and Index Size

An aspect often neglected when comparing access methods is the *size* of the created index structures. Table 2 lists the sizes of the three different indices and the corresponding space utilization.

The average space utilization for the R-tree is between 55% and 60%, whereas it approaches 100% in case of the STR-tree and the TB-tree. The reason is that the R-tree construction strategy does not take the unilateral growth of the data in the temporal dimension into account.

The R-tree is roughly twice as big as the other two indices. For example, for datasets of 1000 objects (i.e., consisting of 1500K line segments), the R-tree size is about 95 MB, while the other two indices size about 57 MB. This difference is mainly due to the R-tree's smaller space utilization. The TB-tree is smaller than the STR-tree. The two indices have similar space utilization, but the TB-tree's fanout is larger. For a ten times larger dataset, the index size increases by the same factor for the STR-tree and the TB-tree. The increase is only approximate in the case of the R-tree, since its space utilization can fluctuate.

	R-tree	STR-tree	TB-tree
Index size	~ 95 KB per object	~ 57 KB per object	~ 51 KB per object
Space utilization	55%-60%	~100%	~100%

Table 2: Index sizes and space utilization

5.3 Range Queries

Range queries are important for spatial data as well as spatiotemporal data. In this section, we compare the three access methods for processing range queries. As already mentioned, we use datasets stemming from 10 to 1000 moving objects. We use three sets of query windows with a range of 1%, 10%, and 20% of the total range with respect to each dimension, i.e., 0.0001%, 0.1%, and 0.8% of the total space. Each query set includes 1000 query windows.

Figure 15 shows the number of total node accesses for various range queries and datasets. Do note that both axes are of logarithmic scale, the x-axis is to the base of 2, and the y-axis is to the base of 10. We observe the following trends. For a small number of moving objects, the STR-tree and the TB-tree show superior range query performance over the R-tree. The break-even point at which this trend is reversed depends on the query size. In case of a 1% range per dimension, the break-even point with respect to the R-tree for the STR-tree is at 30 moving objects, and for the TB-tree at 60 moving objects (cf. Figure 15(a)). For a larger, 10% range size per dimension, the break-even point for the STR-tree is at 25 moving objects and for the TB-tree at 200 moving objects (cf. Figure 15(b)). In case of an even larger range, e.g., 20% per dimension, the break-even points increase to 50 and over 1000 moving objects for the STR-tree and the TB-tree, respectively (cf. Figure 15(c)). Both, the TB-tree and the STR-tree, are trajectory oriented. For a smaller number of trajectories the total dataset (line segments) is more oriented along time than it is with respect to space. We term this property the *temporal discrimination*, as the dataset grows only with respect to the temporal dimension. Thus, for such a dataset, the spatial discrimination capabilities of the index are of no importance. However, if the number of trajectories increases, more segments exist at a given point in time. Thus, the spatial discrimination becomes important. Otherwise, the overlap between the nodes increases.

The R-tree does not “know” about the natural discrimination of the data. Its sole purpose is to group objects according to spatial characteristics, i.e., spatial proximity. For a small number of trajectories, this ambition turns out to be a “boomerang.” In this case, the spatial discrimination is of minor importance. The TB-tree puts connected segments in the same node and does not consider spatial discrimination. It thus exploits the temporal discrimination of the data. As the results show, this approach is better up to a certain number of segments. The STR-tree adopts an approach in-between the two extremes. However, although this index performs better than the R-tree for a small number of trajectories, it is always worse than the TB-tree. The STR-tree, too, is heavily dedicated to trajectory preservation. This explains its performance with respect to the R-tree. However, because of its R-tree properties, it is worse than the TB-tree for a small number of trajectories.

5.4 Time Slice Queries

In several applications it is useful to determine the positions of (all) moving objects at a given time point in the past (Theodoridis et al. 1996). This query type constitutes a special case of a range query with a query window of *zero extent at the temporal dimension*. The size of the query window in the spatial dimensions can be arbitrary. In the performance studies we choose 1%, 10%, and 100% of the respective range in each spatial dimension. This corresponds to three sets, each comprising of 1000 individual queries.

The results shown in Figure 16 are similar to what could be seen in the previous section. For each set of queries (Figure 16(a)-(c)), there exists a break-even point in terms of number of moving point objects when the number of node accesses for the R-tree is smaller than for the STR-tree and the TB-tree, respectively. The break-even point moves from 60 moving objects (1% range) to 500 moving objects (100% range). This trend can also be observed in the case of range queries. However, there the TB-tree always outperforms the STR-tree. In Figure 16(a)-(c), we observe that the gap between the two indices decreases with an increasing range until the STR-tree outperforms the TB-tree (Figure 16(c)).

The nature of a time slice query is to retrieve all positions of moving objects at a given instance in time. In other words, this query favors particularly an index that organizes its content based on its spatial aspects (R-tree and STR-tree) rather than relying on the temporal discrimination capabilities of the data (TB-tree). For smaller ranges (Figure 16(a)), this phenomenon is not as apparent as for larger ranges.

5.5 Combined Queries

What follows is a performance study related to the algorithms for combined searching as presented in Section 2.2.3. We use datasets stemming from a varying number of moving objects. As for the queries, the size of the inner and the outer range is 1% (0.0001%) and 10% (0.1%), and 1% (0.0001%) and 20% (0.8%) in each dimension (of total space). Each set of queries consists of 1000 individual queries.

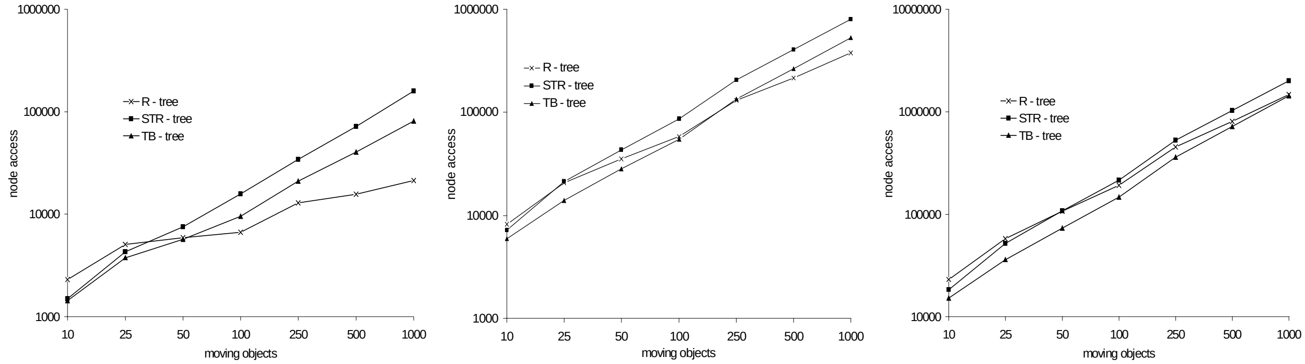
The results in Figure 17 show that the TB-tree is in all cases superior to the STR-tree and the R-tree, up to one order of magnitude with the gap increasing in proportion to the number of objects. Apart from the (partial) trajectory preservation in each node, it is also the additional data structure (a linked list) for retrieving neighbor nodes that contribute to this result. Thus, the numbers of node accesses in case of the TB-tree are only slightly larger than the numbers from the range query experiments in Figure 15(a). Comparing the STR-tree with the R-tree, they only differ in the index structure itself, but have the same combined search algorithms. Just as we have observed a break-even point between those two methods for range queries, it also exists here. For the first experiment, shown in Figure 17(a), the break-even point is at about 300 moving objects. For the second experiment, involving a larger secondary range, the break-even point is at 500 moving objects.

5.6 Summary

The TB-tree supports trajectory-based queries much more efficiently than the R-tree does. At the same time, it is worth to be mentioned that its performance on typical range queries is competitive to the R-tree. As shown in the experiments, for combined queries, the TB-tree’s performance is closely connected to the “number of moving objects” of the dataset. The relative gap between the R-tree and the TB-tree increases with an increasing number of moving objects. As for the STR-tree, although designed to combine the benefits of the TB-tree and the R-tree, it usually performs worse than the TB-tree, with the only exception being time slice queries.

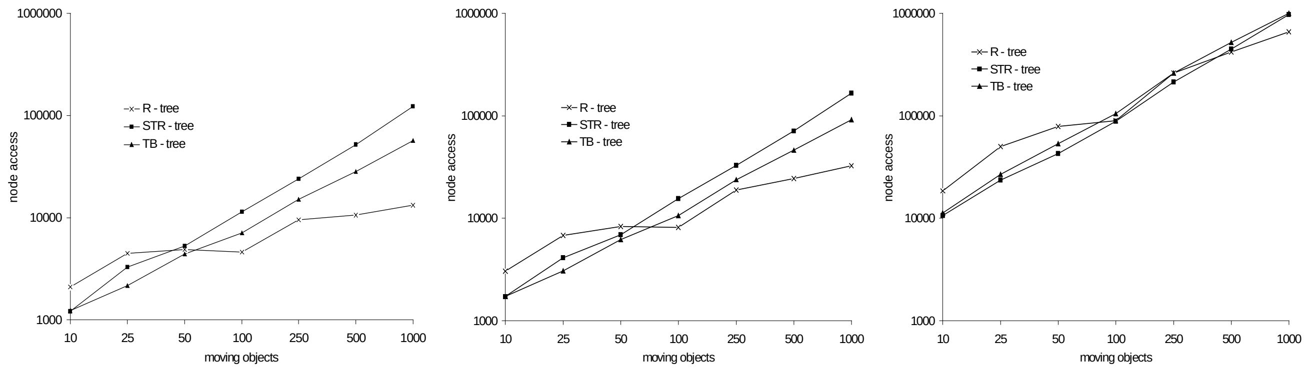
6 Conclusions and Future Work

Work in spatiotemporal query processing has dealt with range queries. However, spatiotemporal data, in the context of trajectories of n-dimensional moving objects, is somewhat different from (n+1)-dimensional spatial data due to the peculiarity of the temporal dimension (Theodoridis et al. 1998). This paper presents a set of pure spatiotemporal queries, the so called trajectory-based (topological and navigational) queries, as well as combined (coordinate- and trajectory- based) queries.



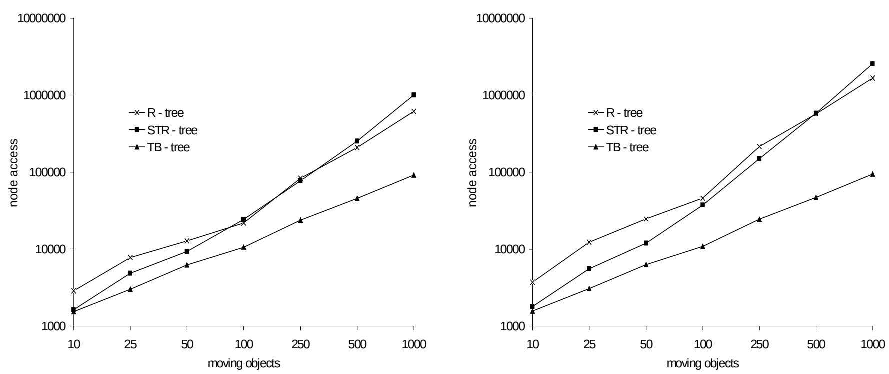
(a) (b) (c)

Figure 15: Range queries: varying range, (a) 1%, (b) 10% and (c) 20% in each dimension



(a) (b) (c)

Figure 16: Time slice queries: varying spatial range, (a) 1%, (b) 10% and (c) 100% in each dimension



(a) (b)

Figure 17: Combined queries: (a) 1% inner- 10% outer range and (b) 1% inner- 20% outer range, in each dimension

Efficient processing of those queries requires indices and access methods for spatiotemporal data; a simple modification to the R-tree as well as two new access methods, namely the STR-tree and the TB-tree, are proposed for indexing the trajectories of moving point objects.

First, trajectory data and a set of queries are defined to derive requirements. Trajectory data is obtained by discretely sampling the movement of point objects in time. Linear interpolation is considered in-between the samples. The set of queries is then presented. Subsequently, the paper discusses the R-tree to determine the shortcomings of this method with respect to spatiotemporal data and queries, and introduces modifications to overcome these limitations. Then the STR-tree and the TB-tree, both tailored to the requirements of trajectory data and spatiotemporal queries, are proposed. They can also easily be implemented on top of the R-tree, which is already adopted in commercial extensible database systems.

The performance study presents results from experiments involving spatial range queries, as well as experiments related to navigational and combined queries. The TB-tree proves to be an access method well suited for trajectory-based queries, and also has a good spatial search performance. The STR-tree performance stays behind the

TB-tree. Although designed to combine the "best of both worlds," it seems that the STR-tree is rather a weak compromise. The "pure" concepts of the R-tree and the TB-tree seem to be far superior in their respective domains.

Although recent literature includes related work on indexing trajectories of moving objects by maintaining the complete history of object movement (Theodoridis et al. 1996, Tzouramanis et al. 1998, Nascimento et al. 1999), the work presented in this paper is the first to

- propose an access method (namely, the TB-tree) clearly addressing the requirements and peculiarities of this context by considering *trajectory preservation*,
- propose and implement specific modifications to the "classic" R-tree in order to overcome (some of) its inefficiencies with respect to trajectories, and
- present novel algorithms for "pure" spatiotemporal searching apart from the typical range querying.

This work points to several future research directions. The present work only presents first algorithms to process navigational and topological queries. Derived from the requirements from real spatiotemporal applications, e.g., fleet management, these algorithms can be refined in more detail. Furthermore, not only novel queries, such as the previous ones, but also known though expensive spatial queries deserve more attention in the spatiotemporal domain; examples include neighbor searching (Roussopoulos et al. 1995) and joins (Mamoulis and Papadias 1999). Finally, investigating geometric shapes other than MBBs as approximations for moving objects' trajectories deserves further research; for instance, extending related work on indexing line segments (Bertino et al. 1998).

References

Allen, J.F.: Maintaining Knowledge About Temporal Intervals. *Communications of the ACM*, 26(11), pp. 832-843, 1983.

Barbará, D.: Mobile Computing and Databases – a Survey. *IEEE Transactions of Knowledge and Data Engineering*, 11(1), pp. 108-117, 1999.

Bartels, R., Beatty, J., and Barsky, B.*: An Introduction to Splines for Use in Computer Graphics & Geometric Modeling*. Morgan Kaufmann Publishers, Inc., 1987.

Bertino, E., Catania, B., and Shidlovsky, B.: Towards optimal indexing for segment databases. In *Proc. of Int'l Conference on Extending Database Technology*, pp. 39-53, 1998.

Egenhofer, M. and Franzosa, R.: Point-Set Topological Spatial Relations. *Int'l Journal of Geographic Information Systems*, 5(2), pp. 161-174, 1991.

Erwig, M. and Schneider, M.: Developments in Spatio-Temporal Query Languages, In *Proc. of DEXA Workshop on Spatio-Temporal Data Models and Languages*, 1999

Güting, R., Böhlen, M., Erwig, M., Jensen, C. S., Lorentzos, N., Schneider, M., and Vazirgiannis, M.: A Foundation for Representing and Querying Moving Objects. *ACM Transactions on Database Systems*, to appear, 2000.

Guttman, A.: R-trees: a Dynamic Index Structure for Spatial Searching. *In Proc. of ACM-SIGMOD Conference on the Management of Data*, pp. 47-57, 1984.

Kollios, G., Gunopulos, D., and Tsotras, V.: On Indexing Mobile Objects. In *Proc. of the 18th ACM Symposium on Principles of Database Systems*, pp. 261-272, 1999.

Mamoulis, N. and Papadias, D.: Integration of Spatial Join Algorithms for Processing Multiple Inputs. In *Proc. of ACM-SIGMOD Conference on Management of Data*, pp. 1-12, 1999.

Nascimento, M., Silva, J., and Theodoridis, Y.: Evaluation of Access Structures For Discretely Moving Points. In *Proc. of Int'l Workshop on Spatio-Temporal Database Management*, pp. 171-188, 1999.

Papadias, D., Theodoridis, Y., Sellis, T., and Egenhofer, M.: Topological Relations in the World of Minimum Bounding Rectangles: A Study with R-trees. In *Proc. of ACM-SIGMOD Conference on Management of Data*, pp. 92-103, 1995.

Pfoser, D. and Jensen, C.: Capturing the Uncertainty of Moving-Object Representations, In *Proc. of the 6th Int'l Symposium on Spatial Databases*, pp. 111-132, 1999.

Pfoser, D., Jensen, C. S., and Theodoridis, Y.: Novel Approaches In Query Processing For Moving Objects. CHOROCHRONOS Technical Report, CH-00-3, 2000.

Roussopoulos, N., Kelley, S., and Vincent, F.: Nearest Neighbor Queries. In *Proc. of ACM-SIGMOD Conference on Management of Data*, pp. 71-79, 1995.

Saltenis, S., Jensen, C. S., Leutenegger, S., and Lopez, M.: Indexing the Positions of Continuously Moving Objects. In *Proc. of ACM-SIGMOD Conference on Management of Data*, pp. 331-342, 2000.

Spaccapietra, S., Parent, C., and Zimanyi, E.: Modeling Time from a Conceptual Perspective. In *Proc. of Int'l Conference on Information and Knowledge Management*, pp. 432-440, 1998.

Theodoridis, Y., Sellis, T., Papadopoulos, A., and Manolopoulos, Y.: Specifications for Efficient Indexing in Spatiotemporal Databases, In *Proc. of the 10th Int'l Conference on Scientific and Statistical Database Management*, pp. 123-132, 1998.

Theodoridis, Y., Silva, R., and Nascimento, M.: On the Generation of Spatiotemporal Datasets. In *Proc. of the 6th Int'l Symposium on Spatial Databases*, pp.147-164, 1999.

Theodoridis, Y., Vazirgiannis, M., and Sellis, T.: Spatio-Temporal Indexing for Large Multimedia Applications. In *Proc. of the 3rd IEEE Int'l Conference on Multimedia Computing and Systems*, pp. 441-448, 1996.

Tryfona, N. and Jensen, C. S.: Conceptual Data Modeling for Spatiotemporal Applications, *Geoinformatica*, 3(3), pp. 245-268, 1999.

Tzouramanis, T., Vassilakopoulos, M., and Manolopoulos, Y.: Overlapping Linear Quadtrees: A Spatio-Temporal Access Method. In *Proc. of the 6th Int'l Symposium on Advances in Geographic Information Systems*, pp. 1-7, 1998.