CHOROCHRONOS

A Network for Spatiotemporal Database Systems

National Technical University of Athens (NTUA)
Aalborg University (AALBORG)
FernUniversitaet Hagen (HAGEN)
University of L'Aquila (UNIVAQ)
University of Manchester - Institute of Science and Technology (UMIST)
Politecnico di Milano (POLIMI)
Institut National de Recherche en Informatique et en Automatique (INRIA)
Aristotle University of Thessaloniki (AUT)
Technical University of Vienna (TU VIENNA)
Swiss Federal Institute of Technology, Zurich (ETHZ)

TECHNICAL REPORT SERIES

TECHNICAL REPORT CH-99-03

Indexing Trajectories of Moving Point Objects

Dieter Pfoser, Yannis Theodoridis, and Christian S. Jensen

October 1999

CHOROCHRONOS: TMR Research Network Project, No ERBFMRXCT960056

Contact Person: Prof. Timoleon Sellis, Dept. of Electrical and Computer Engineering, National Technical University of Athens, Zografou 15773, GR Tel: +30-1-7721601, FAX: +30-1-7721659, timos@cs.ntua.gr

Abstract

Spatiotemporal applications attract more and more attention, both, from researchers as well as application developers. Especially the peculiarities of spatiotemporal data are the focus of an increasing research effort. In this paper we extend the well-known R-tree method to handle trajectory data stemming from moving point objects. The resulting access method, termed (Spatio-Temporal) STR-tree, differs from the R-tree in that it stores additional information in the entries at the leaf level and, further, has modified insertion and split algorithms. Besides the description of the STR-tree algorithms, we provide an extensive performance study examining the behaviour of the new method as compared to the original R-tree under a varying set of queries and datasets. The collection of queries comprises the typical point and range queries as well as pure spatiotemporal queries based on the semantics of objects' trajectories, the so-called trajectory and navigational queries.

1 Introduction

There are two properties that are inherent to any object in the real world, *space* and *time*. For many applications it is further necessary to model those properties as attributes in a database system. For several years, a lot of effort was devoted to the respective areas of temporal and spatial database research. However, the integration of issues poses further research questions. As we will thoroughly discuss in the paper, it is sometimes not enough to take the "best" of both worlds to find a satisfactory solution to a given spatiotemporal problem. A particular problem in the context of spatiotemporal databases is indexing of spatiotemporal information. That is, to construct a spatiotemporal access method (STAM) to index spatial data changing over time. More specifically, in this work we focus on data stemming from the *movement of spatial point objects*. We deal with point objects, since in many applications the size and shape of an object is of no importance but only its positions matters, e.g., navigational systems.

The data obtained from moving point objects can be seen similar to a "string" arbitrary oriented in three-dimensional 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 *trajectory* (cf. Figure 1).

To define a proper access method, we not only have to be aware of the nature of the data, but also must know about the type of queries the index is used for. Typical queries in spatial and temporal databases are range (window) queries. Queries for spatiotemporal data are more demanding. They are not only related to the spatiotemporal extent of the data but are also connected to objects' trajectories. The spatiotemporal data as described above can be treated as spatial data, where the third dimension is time. Thus, a first step in finding an appropriate index is to study spatial access methods.

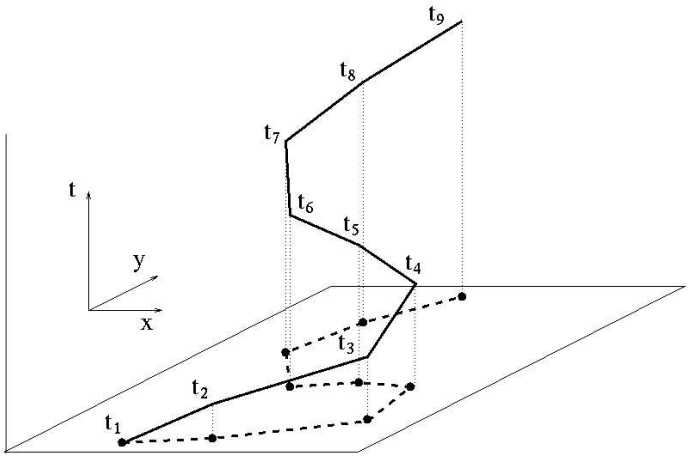


Figure 1: The movement of a spatial object and the corresponding trajectory

In literature there exist several approaches about how, essentially, spatial data can be processed using indexes. Jagadish (1990) suggests a method in which, both, the data as well as the queries are transformed into two simpler geometries to a new space. In the so-called feature space, classical spatial access methods can be used to index the data. The drawbacks of the method are that proximity is not preserved under the transformation (Nievergelt and Hinrichs 1985), and several queries cannot be mapped at all into the feature space (Henrich et al. 1989, Orenstein 1990, and Pagel et al. 1993). Another approach to access spatiotemporal information is to use access methods in the original (also, called native) space. Examples are the R-tree family (Guttman 1984) and the Quad-tree family (Samet 1984 and Tayeb et al. 1998). A drawback of the Quad-tree family is that the space the objects will eventually occupy has to be known in advance. The R-tree was evaluated in its ability to index spatiotemporal information (Nascimento et al. 1999).

A problem not addressed by using any of the above access methods, however, is the preservation of trajectories. All the above access methods treat the data merely as a set of line segments, regardless whether some belong to the same trajectory. Line segments are grouped together merely according to spatial properties such as closeness. This is not enough, since certain type of queries require access to parts of the whole trajectory.

To capture this issue, we propose an access method, namely the Spatiotemporal R-tree (STR-tree), which is based on the popular R-tree method. Through modifications to the original R-tree algorithms we overcome the shortcomings of the original method.

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 dealing with moving point objects. Section 3 derives the requirements to a spatiotemporal access method starting with the description of the R-tree. In Section 4 we present the algorithms comprising the proposed access method, the STR-tree. Section 5 describes the performance studies in which we compare the STR-tree with the R-tree under varying parameter settings. Finally, in Section 6 we give conclusions and directions for future research.

2 Moving Point Objects

In this section we discuss our type of 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.

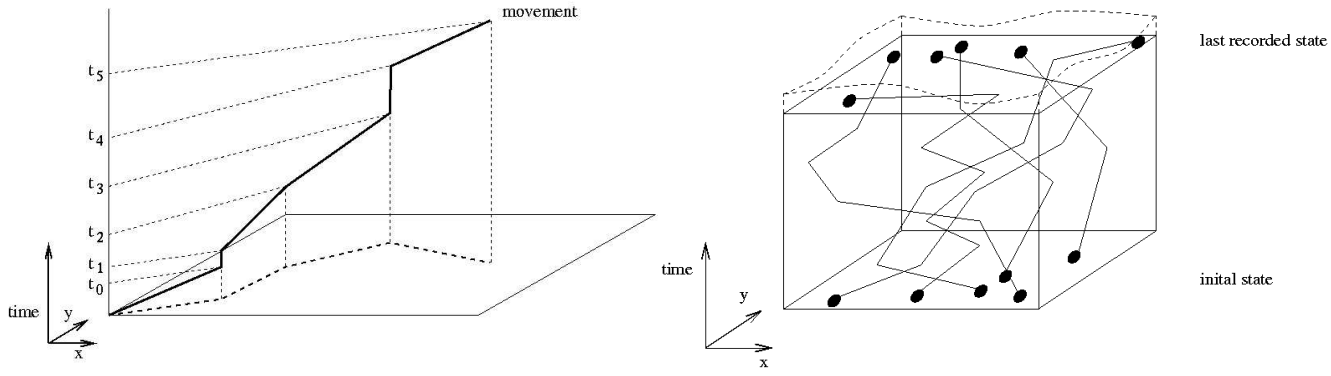
The optimisation of transportation, especially in highly populated areas, is a very challenging task that may be supported by an information system. A core application in this context is fleet management. Vehicles equipped with GPS devices transmit their positions to a central computer using either radio communication links or mobile phones. At the central site, the data is processed and utilized.

2.1 Sampling

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 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 end points of line segments of polylines, and the movement of an object is represented by an entire polyline in three-dimensional space. In geometrical terms, the movement of an object is termed a *trajectory* (we will use ''movement" and ''trajectory" interchangeably). The solid line in Figure 2(a) represents the movement of a point object. Space (x and y-coordinates) and time (t-coordinate) are combined to form one coordinate system. The dashed line shows the projection of the movement in two-dimensional space (x and y coordinates) (Pfoser and Jensen 1999).

Figure 2(b) shows the spatiotemporal space (the cube in solid lines) and several trajectories (the solid lines). 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 symbolise the growth of the cube with time.



(a) (b)

Figure 2: Moving point objects: (a) a trajectory and (b) a spatiotemporal space containing several trajectories

2.2 Queries

Through the combination of space and time further query types emerge. Current research in spatiotemporal databases deals mostly with the handling of now-relevant data (Sistla et al. 1997, Kollios et al. 1999), i.e., one is only interested in the current position, speed, and heading of a moving object (dynamic information). Storing the trajectories of moving objects would not only allow us to determine dynamic information for the current position of the object but *for all historic positions* stored in the database. We distinguish two types of queries involving trajectories of moving objects:

- *Coordinate-based queries*, such as point, range, and nearest-neighbor queries in the resulting three-dimensional space, and
- *Semantics-based queries*, usually involving *trajectory metadata*, such as speed and heading of objects.

The former ones comes as an inheritance from spatial and temporal databases. Queries of the form "*find all objects that lied within a given area (or at a given point) 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) are still very important for STDBMS users. On the other hand, novel queries are also introduced due to the specific nature of data. The so-called semantics-based queries are classified in '*trajectory*' queries, which involve the whole information of the movement of an object, and '*navigational*' queries. We discuss semantics based queries in more detail in the sequel. Both coordinate- and semantics- based queries will be involved in our performance study later in Section 5.

2.2.1 Trajectory queries

Queries involving the whole trajectory of an object are very important (and rather expensive). A definition of a well-established set of predicates, such as the 9-intersection model (Egenhofer and Franzosa 1991) for spatial data and the thirteen 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 by the spatiotemporal versions of the eight 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*.

Whether an object *enters*, *crosses*, or *bypasses* a given area can be replied only after examining more than one parts of its trajectory (i.e., the line segments stored in the database) under a respective set of constraints. For instance, an object *entered* into an area with respect to a given time horizon, iff the start- point of its least recent segment (respectively, the end- point of its most recent segment) was outside (respectively, inside) the given area. Similar definitions hold for the *leave*, *cross*, and *bypass* predicates as can be extracted from the illustration in Figure 3.

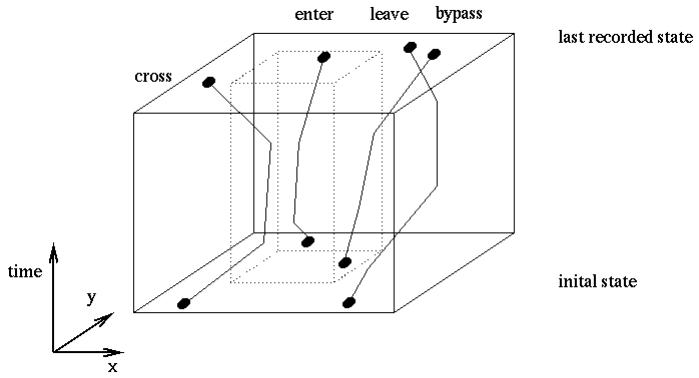


Figure 3: Trajectory-based spatiotemporal operations: ‘enter’, ‘leave’, ‘cross’, and ‘bypass’

2.2.2 Navigational Queries

In our data model dynamic information is not explicitly stored but has to be derived from the trajectory information. The speed of an object is determined by the fraction of travelled distance over time. The heading of an object is computed by determining a vector between two specified positions. From these definitions one can see that neither speed nor heading are unique but depend on the *time horizon* we consider. The heading of an object in the last ten minutes was strictly east, but considering the last hour it was 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 averages out to 30 mph.

Queries involving speed or heading are expected to be very important in real-life applications. Consider for example the following (Erwig et al. 1999): “*At what speed does this plane move? What is its top speed?*” or “*Are two planes heading towards each other (going to crash)?*” The former one is a *spatiotemporal selection* and considers the *now* instance as the time horizon for the first part while an aggregation on a longer time horizon is necessary for the second part. The latter one is a *spatiotemporal join*, if both planes are left unspecified. Obviously, ‘approach’ is different from ‘lie close’ and in order to support queries involving ‘approach’ using R-tree-based methods the information about line segment orientation needs to be maintained. What is common in both cases is that we want to examine a set of line segments selected by belonging to the same trajectory as opposed selected by a spatiotemporal range.

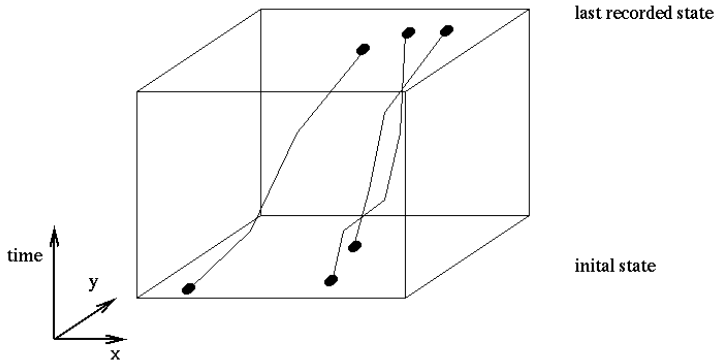


Figure 4: ‘Approach’ is different from ‘lie close’

Further queries could involve the combination of range and navigational subqueries, which are also very interesting from the query optimisation view. For example, the *query "What is the speed and heading of all the objects that crossed New Mexico from July 2 to 5 in 1947"* pipelines the results of a range query involving spatial (New Mexico) and temporal (July 2 to 5, 1947) ranges with a navigational query about speed and heading.

Along this line one can construct various combinations of query combinations, which are all plausible in the spatiotemporal application context.

3 Spatial Access Methods and the R-tree

In this section, we first give an overview of what types of spatial access methods do exist. Subsequently, we give a brief introduction to the R-tree.

3.1 Spatial Access Methods

The main idea behind spatial indexing is to support spatial selection, that is, to retrieve from a large set of spatial objects those in some particular relationship with the query region. A technique generally adopted in indexing spatial objects is approximation. A *Minimum Bounding Box* (MBB), or Minimum Bounding Rectangle (MBR) in 2-D, is used to approximate the spatial object to construct the spatial index. This use leads to a filter-and-refine strategy for query processing. First, based on the approximations, a filtering step is executed that returns a superset of the objects fulfilling the query predicate. Second, in the refinement step, the exact extents are checked against the query predicate (Gueting 1994).

Spatial access methods (SAM) using MBB approximations can be grouped into four categories (Stefanakis et al. 1998). First, the *ordering technique* introduces a one-dimensional ordering among the set of MBBs based on both their location and extent using a space-filling curve (Abel and Smith 1983, Orenstein 1986, and Faloutsos 1988). Second, with the *transformation technique* MBBs are transformed into higher dimensional space, where they can be represented as points (Nievergelt et al. 1984). Third, with the *clipping technique* the space occupied by the set of spatial objects is partitioned into a set of disjoint regions. Thus, a spatial object is associated with all regions which intersect its MBB. Prominent examples of this technique are the R^+-tree (Sellis et al. 1987) and the Cell-tree (Guenther 1989).

The fourth category comprises access methods employing the *overlapping technique*. Contrary to the clipping technique, regions are derived from MBBs, and each MBB is inserted into a unique region. The most prominent example here is the R-tree (Guttman 1984).

3.2 The R-tree

In the following we give a short overview of R-tree operations (Gaede and Guenther 1998). From the algorithmic point of view, the R-tree is a height-balanced tree with index records in its leaf nodes containing pointers to actual data objects. Leaf node entries are of the form (id, MBB), where id is the pointer to the actual object and MBB represents an n-dimensional interval. A node in the tree (intermediate and leaf) corresponds to a disk page. Every node contains between *m* and *M* entries (except the root). The lower bound *m* prevents the degeneration of the trees. Whenever the number of node entries drops below *m*, the node is

deleted and its entries reinserted. The upper bound *M* is called fanout and is determined by the page size. Whenever the number of node entries would rise above *M*, the node is split. Figure 5 illustrates an exemplary R-tree of fanout *M*=3 for a given set of spatial objects.

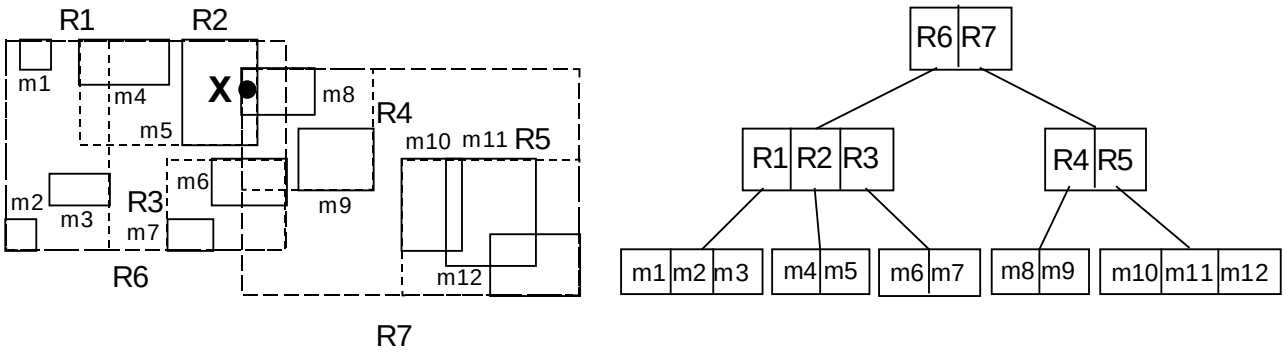


Figure 5: An R-tree index

The most important operation of an access method is to *insert* a new entry into the index. We insert the MBB into the index by traversing a single path starting from the root node to the leaf level. At each level we choose the child node whose MBB needs least enlargement by including the new entry. In case several MBBs satisfy this criterion, we choose the smallest one. This algorithm is called *ChooseLeaf* in Guttman (1984). Upon reaching the leaf level, we insert the new entry, adjust the covering MBB, and propagate the change upwards. If the leaf node does not have space (already *M* entries inside), we split the node. Covering MBBs are adjusted accordingly, and the split is propagated upwards. When *splitting* a node we want to divide its entries between two nodes. To minimise node access during search, the total area of the two covering MBBs after the split should be minimised. Guttman (1984) suggests one *exhaustive* and two heuristic algorithms, one with *quadratic (QuadraticSplit)* and a simpler one with *linear* complexity.

As for *deletion* of an entry in the index, a reversed 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 a given threshold, the respective entries of that particular node are deleted and re-inserted into the index.

In searching an R-tree, we check whether a given node entry satisfies the search range (window query). If so, we visit the child node and thus recursively traverse the tree. Due to the overlapping MBBs, at each level of the index, there may be more entries that satisfy the search range. An example is given in Figure 5, where a point query for point X results into the paths R6-R2-m5 and R7-R4-m8.

In the context of spatiotemporal data this technique proves to be inefficient. Consider the data shown in Figure 4. As shown in Figure 6, approximating the line segments using a MBB introduces large amounts of dead space (Theodoridis et al. 1998). It is evident that the corresponding MBB covers a large portion of the space, whereas the actual space occupied by the trajectory is small[1]. This leads to high overlap and consequently to a small discrimination capability of the index structure.

[1] Dependant on the granularity of the underlying space, the occupied space could be zero. This is true since the volume of a line in three-dimensional space equals zero.

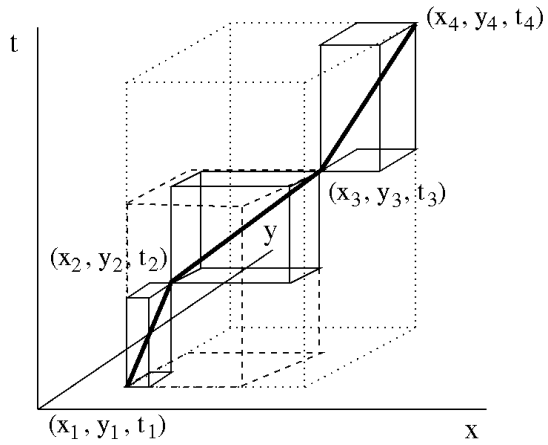


Figure 6: Approximating a trajectory using MBBs

Another drawback of this method is that in storing approximations rather than the actual spatial object, the line segments, in the index, a refinement step is required when processing a query.

Also, in using a regular R-tree, trajectory information is not preserved, since nodes are split purely according to spatial characteristics. Due to this side effect, it is necessary to find alternative representations and/or approximation schemes for these kinds of data.

4 The STR-Tree Structure

The STR-tree is an R-tree based access method supporting efficient query processing of three-dimensional trajectories of spatial data. In the following, we outline its modifications as compared to the R-tree.

4.1 Leaf Node Entries

When using a MBB to approximate a line segment, the exact extent of the object is not known in the index. However, as can be seen in Figure 7, a line segment can only be contained in four different ways in a MBB. We can store this extra information in the leaf node by simply modifying the entry format to (id, MBB, orientation), where orientation is of domain {1,2,3,4}.

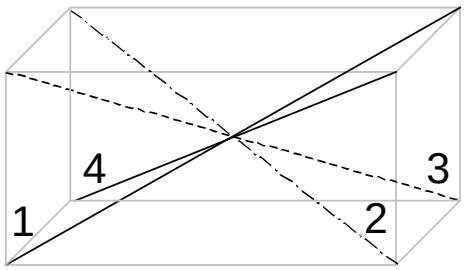


Figure 7: Different line segments are mapped into a single MBB

4.2 Insertion

The insertion process is considerably different from the procedure known from the R-tree. In the R-tree for inserting a new entry two algorithms are involved, *ChooseSubtree* and *Split*. ChooseSubtree chooses the best place for the insertion of the new entry based on minimum enlargement of existing nodes in the tree. If the

chosen leaf node contains the maximum number of entries, the node is split and the split propagated up the tree[2].

The insertion in the STR-tree works differently in that our considerations are not only of pure spatial nature, but we also try to keep line segments belonging to the same trajectory together, i.e., partially preserve trajectories. As a consequence, when inserting a new line segment the target 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. In any other case we have to apply a node split strategy. In Figure 8, we show an exemplary index, in which the node returned by FindNode is marked with an arrow. In the following we will discuss how the above insertion strategy works together with the known R-tree Split algorithm.

Consider the case in which not only the leaf level node but also the parent nodes at the intermediate level 1 and 2 are full (bold boxes in Figure 8). The leaf nodes would be split and the split propagated upwards. In the case of the R-tree and with data arbitrary distributed in space, both resulting leaf nodes would have an equal chance of being the subject of insertion again. However, in our case, entries are inserted with an increasing time horizon. This characterises even the ordering of entries in a single node. Consequently, the split algorithm puts "newer" entries in the new node. As a result, "older" nodes have fewer chances for the

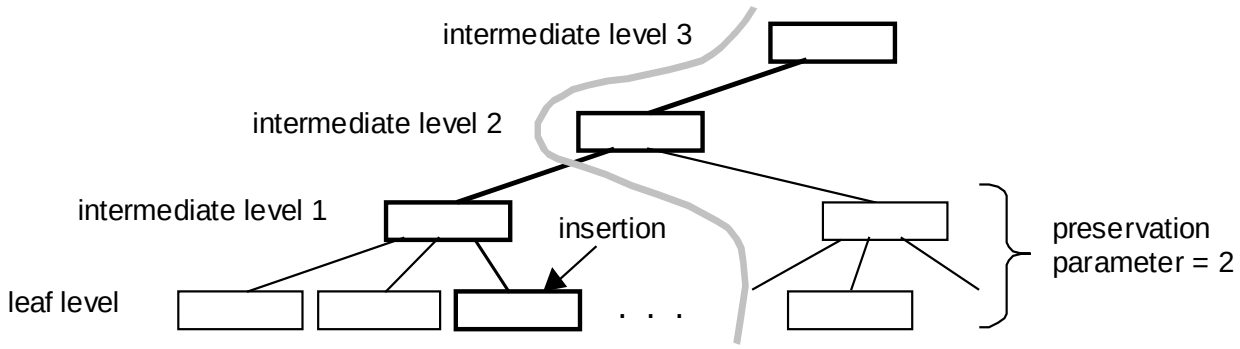


Figure 8: Insertion into the STR-tree

insertion of new segments, i.e., they do not connect to any new segment, and remain empty to a large degree.

Since the nodes of the index are oriented along trajectories, we would further have a high degree of overlap, and the overall space discrimination of the index would be very low. The overall behaviour of the index would be to thinly slice the spatiotemporal cube according to time. The overall preservation of trajectories would be minimal. Since this situation is not satisfactory, we have to rethink our split strategy in the context of this new insertion paradigm of preserving trajectories.

[2] In the sequel, the so-called ChooseSubtree and QuadraticSplit algorithms will be used in discussion of the STR-tree algorithms without further details, since they are identical to the original Guttman algorithms.

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, and at the same time to preserve trajectories. In the following we devise the Insert algorithm shown in Figure 9, which has an additional parameter, called *preservation parameter p*. This parameter indicates the number of levels we "reserve" for the preservation of trajectories. Given the case a leaf node is full, the algorithm checks whether *p*-1 parent nodes are full (in Figure 8, for $p = 2$, we only have to check the node drawn in bold at intermediate 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 ChooseSubtree on a subtree including all the nodes further to the right of the current insertion path (subtree to the right of the bold grey line in Figure 8).

Algorithm Insert
INS1 Invoke **FindNode**
INS2 **IF** node found
    IF node has space,
        insert new segment.
    ELSE
        IF the p-1 (preservation parameter) parent nodes are full
            invoke **ChooseSubtree** but exclude the current branch
        ELSE invoke **Split**.
ELSE ChooseSubtree.

Algorithm FindNode
FN1 Set N to be the root node.
FN2 **IF** N is a leaf,
    IF N contains an entry that is connected to the new segment,
        RETURN N.
ELSE
    Choose N to be the entry which intersects with the new segment
FN3 Set N to be the childnode pointed to by the childpointer of the chosen entry and repeat from FN 2.

Figure 9: Insert algorithm

The split algorithm differs from what is known from the R-tree. Since we try to preserve trajectories in the index, splitting a leaf node requires an analysis of what kinds of segments are contained inside a node. A node can contain four different types of segments,

- *disconnected* segments, i.e., a segment is not connected to any other segment in the node,
- *forward* connected segments, i.e., the recent (in time) end is connected to another segment,
- *backwards* connected segments, i.e., the old (in time) end is connected to another segment, and
- *double-connected* segments, i.e., both ends, recent and old, are connected to other segments.

With this, we can distinguish the three split scenarios of Figure 10. The bold line indicates the split strategy in each case. In the first case, all segments are disconnected (cf. Figure 10(a)) and a QuadraticSplit algorithm is invoked to determine the split. In the second case, we encounter all types of segments, and we place the disconnected ones in a new node. In the third case, all segments are forward, backward, or double connected (no disconnected ones), and we place the most recent forward connected segment in a new node. Figure 11 summarizes the split algorithm.

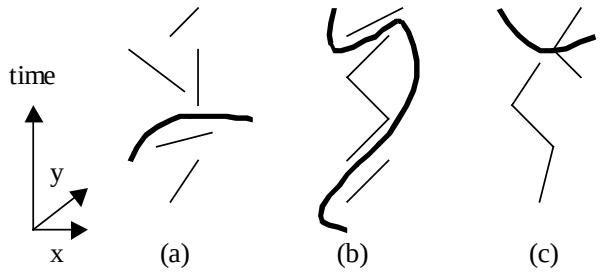


Figure 10: Different split scenarios

The general idea behind the split algorithm is to put newer, and thus, more recent segments, into new nodes. Following that, more recent segments are stored in nodes further to the right of a STR-tree.

Finally, splitting of intermediate 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 to decompose the occupied space.

Algorithm Split
S1 **IF** node is intermediate node, invoke SplitIntermediateNode.
ELSE invoke SplitLeafNode.

Algorithm SplitIntermediateNode
SIN1 Put the new entry into a new node and keep the old one as it is

Algorithm SplitLeafNode
SLN1 **IF** entries in node are all disconnected segments,
invoke **QuadraticSplit**.
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 11: Split algorithm

4.3 Search

The search algorithms for the STR-tree can be grouped according to the query categories *coordinate-based* and *semantics-based* queries. (cf. Section 2.2). Algorithms for the first category are similar to the one used in R-tree. The idea is to descent the tree with respect to intersection properties until the entries are found in the leaf nodes [Guttman 1984].

Algorithms for the second category of queries are different in that not only a spatial but also a semantic 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. Figure 12 presents an outline of such an algorithm. In the algorithm SemanticSearch a query predicate is used to determine a subset of line segments which will be used as input for DetermineTrajectory to extract a partial trajectory. This algorithm retrieves the connected segments necessary for the query specific computation, e.g., speed and heading. The algorithm stops when the newly found segment does not satisfy a condition anymore, e.g., it is outside a given range, or more than X number of segments were retrieved already. This condition can be of any kind and is derived from the

semantics-based query type. DetermineTrajecotry invokes either FindFwdSegment, FindBwdSegment, or both. The choice of algorithm depends, again, on the query and what segments (forward or backward connected) are needed. These algorithm, the actual "trajectory trackers," try to find a connected segment in the same leaf node first, but, if unsuccessful, proceed to a classical search starting at the root.

Algorithm SemanticSearch
SS1 Set N to be the root node
SS2 **IF** N is a leaf
for all entries E that satisfy the query predicate invoke **DetermineTrajectory(E,N)**
SS3 Set N to be the childnode of the current node and repeat from SS2

Algorithm DetermineTrajectory(E,N)
DT1 While FindFwdSegment (FindBwdSegment) returns segments that fulfil a condition
Add found segment E to set of solutions
Invoke **FindFwdSegment(E,N)** (FindBwdSegment)

Algorithm FindFwdSegment(E,N)
FFS1 Loop through N and find forward connected segment to E
FFS2 **IF** not found set N to be the root
FFS3 **IF** N is a leaf
IF N contains an entry E' that is forward connected to E
return E'
ELSE Choose N to be the entry which intersects with E
FFS4 Set N to be the childnode of the current node and repeat from FFS3

Figure 12: Alternative search algorithm for semantics based queries

The algorithm in Figure 12 should only provide a framework for specific algorithms answering semantics-based queries as they are shown in Section 2.2. Further research into this subject is necessary.

5 Performance Study

After outlining the principles and algorithms behind the STR-tree, we present results of the performance study comparing it to the R-tree. We first explain the overall design and objectives of the performance study. Subsequently we outline the process of building datasets. The core of this section comprises the results of comparing the STR-tree with the R-tree for a varying set of data and queries. A summary of the results is included at the end of the section.

5.1 General Considerations

Our goal for the performance study is two-fold. First, we experiment in choosing the suitable preservation parameter p for the STR-tree (cf. Section 4). We demonstrate how varying p values affect index creation. Second, we compare the STR-tree with the R-tree under the aspects of different types of data and queries.
The different types of data are distinguished by having

- a *varying time horizon*, i.e., creating indices for data obtained by the same set of moving objects at different instances of time,

- data stemming from a *varying number of moving objects*, i.e., creating indices for data obtained by a varying number of moving objects at the same instance of time, and
- data stemming from objects with *varying movement characteristics*, i.e., objects moving faster or slower with respect to each other.

The different types of queries are *point queries*, i.e., find all line segments which contain the specified position in time, *range queries*, i.e., find all line segments which intersect a given spatiotemporal range, and *semantic queries*, i.e., for a queried set of positions in time, determine the cost for finding and a certain number of connected segments.

The performance studies were conducted using a C implementation of the proposed access methods. As for the parameters of the experiments, we chose the page sizes for data and directory pages to be 1024 bytes. With a size of 28 bytes for a data and 32 bytes for a directory entry, we obtain 31 and 28 entries per page, respectively. To compare the performance of the access methods we chose various data files, containing up to 1.000.000 three-dimensional line segments stemming from up to 100 moving objects.

5.2 Types of 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, etc., covering the entire United States [Census]), 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 utilise the GSTD generator of spatiotemporal datasets (Theodoridis et al. 1999) to create trajectories of moving objects under various distributions[3]. GSTD allows the user to generate a set of line segments stemming from a specified number of moving objects. 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 function.

The parameters of the generator were given the following values:

- The spatial range for the movement of objects was restricted to 10K times 10K points. The temporal extent ranges from 10K to 200K points
- The initial distribution of points was gaussian, i.e., all points were distributed around the centre of the workspace. The movement of points was always ruled by a random distribution of the form `random(-x,x)`, thus achieving unbiased spread of points. The `x` parameter was tuned to different values to simulate slow (`x` = 0.02) versus fast (`x` = 0.06) moving points.
- The number of different possible snapshots or, in other words, the resolution of time, was held constant by 100K.

---

[3] The GSTD generator has been also used in the performance comparison that appears in [Nascimento et al., 1999] between the three-dimensional R-tree and several of its persistent variations for spatiotemporal indexing.

- Finally, the number of moving objects varied between 5 and 100 resulting to datasets consisting of between 5K and 1000K tuples (i.e., line segments)[4].

Table 1 formalises the collection of datasets used in our experiments.

Dataset	Movement function	Time horizon	# objects	# tuples
t1_10	-0.03, 0.03	10K	10	5K
t4_10	-0.03, 0.03	40K	10	20K
t8_10	-0.03, 0.03	80K	10	40K
t12_10	-0.03, 0.03	120K	10	60K
t16_10	-0.03, 0.03	160K	10	80K
t20_10	-0.03, 0.03	200K	10	100K
t20b_10	-0.02, 0.02	200K	10	100K
t20c_10	-0.04, 0.04	200K	10	100K
t20d_10	-0.06, 0.06	200K	10	100K
t20_5	-0.03, 0.03	200K	5	50K
t20_20	-0.03, 0.03	200K	20	200K
t20_50	-0.03, 0.03	200K	50	500K
t20_100	-0.03, 0.03	200K	100	1000K

Table 1: The collection of datasets

5.3 Preservation Parameter

An important parameter of the STR-tree is the number of levels we use in the index to preserve the moving object trajectories. In the first set of experiments, we show the effects of a varying preservation parameter *p* on the quality of the created index. The data file used in the experiments shown in Figure 13 was t20_10, i.e., 100K segments stemming from 10 moving objects with a time horizon of 200K points.

An important characteristic of an access method is the cost for the creation as well as the quality of the resulting index. In the context of this study we measure cost as the *number of node accesses*. The quality of the index is depicted by *space utilisation*.

Figure 13 shows the number of nodes accessed during index creation, where access means reading a page from disk. We distinguish node accesses at intermediate levels and at the leaf level, since varying the preservation parameter has different effects on intermediate and leaf levels during insertion (FindNode), we consider the number of node accesses grouped accordingly. Space utilisation shows the average number of entries per node in percent, i.e., an average space utilisation of 100% means that all nodes are filled. In our experiments, the STR-tree usually exhibits an average space utilisation (without considering the root node for this average) of 96%, whereas the corresponding value for the R-tree is 56%. This figure does not change for STR-tree for varying preservation parameters.

[4] Note that the resulting number of tuples is not equal to the number of objects times the number of snapshots (actually, it is much smaller than that) since GSTD outputs only the necessary ones to reproduce the dataset motion, as defined by the user through the probability functions (Theodoridis et al. 1999).

From Figure 13 one can see that a *preservation parameter p* = 1 or 2 seems to be the best choice. The total number of node accesses is almost the same. For $p = 3$ the number of intermediate node accesses increases drastically. In this case, more levels in the tree are used to preserve trajectories, i.e., the overlap between nodes increases, and finding a segment during insertion requires more node accesses. However, in using $p = 1$, trajectories are hardly preserved (cf. Section 4.2). Thus the obvious choice is a preservation parameter of 2. Compared to the R-tree, the STR-tree has fewer node accesses during index creation. In the next section we will more closely examine the index quality of the STR-tree when compared to the R-tree.

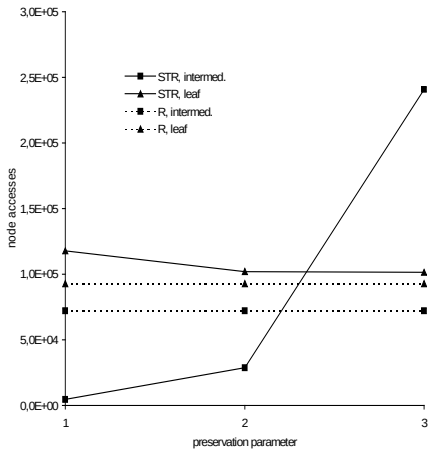


Figure 13: Comparison STR/R-tree number of node accesses
for insertion with varying preservation parameter *p*

5.4 Index Creation

In this section we show the behaviour of the two measures (node accesses and space utilisation) for various indices created for the following types of datasets.

- A fixed number of moving objects, fixed objects' speed, and an increasing time horizon (t1_10 to t20_10)
- A varying number of moving objects, fixed speed, and fixed time horizon (t20_5 to t20_100)
- A fixed number of moving objects, varying speed, and fixed time horizon (t20_10, t20_10b to t20_10d).

In the following figures, we only show the results for node accesses. Space utilisation does not dependant on any of those parameters. It is around 96% for the STR-tree, without considering the root node for this average, and 56% for the R-tree.

Figure 14 shows the results for the number of node accesses during index creation. From Figure 14 (a) and (b) we can see that independent of the time horizon, the STR-tree outperforms the R-tree for a datasets stemming from 30 or less moving objects. This also holds when creating indices using larger page sizes (cf. Figure 15). The reason here is that for a larger number of moving objects, more and more trajectories exist in the index. Thus, more and more trajectories want to be preserved. This preservation decreases the capabilities of the index with respect to spatial subdivision, and consequently the overlap between nodes increases. As a result, the number of node accesses increases during insertion.

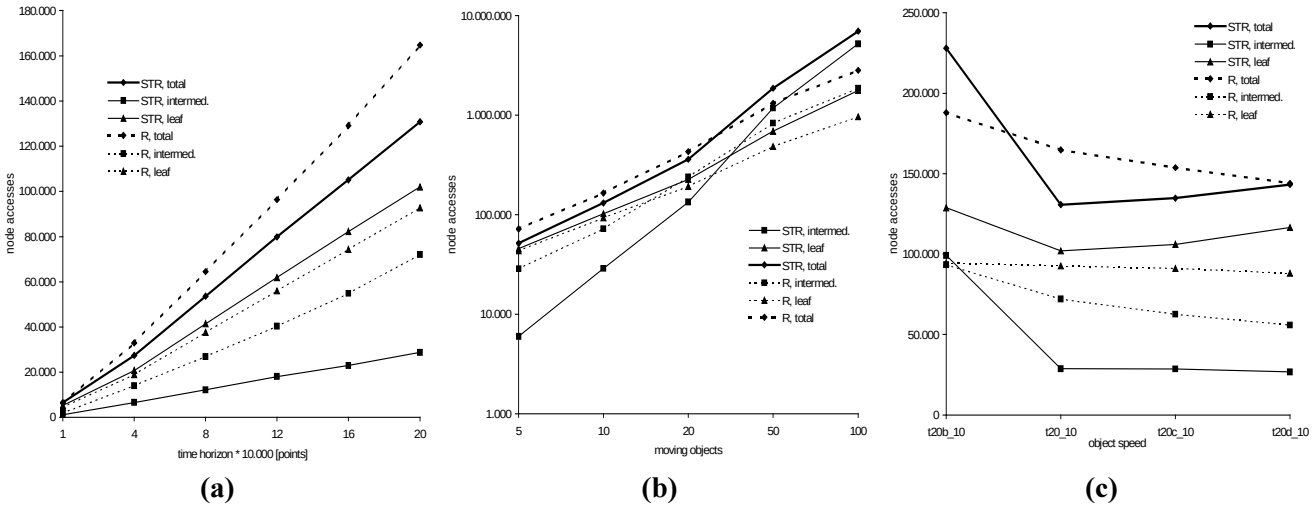


Figure 14: Node access for insertion STR/R-tree: (a)varying time horizon, (b) number of objects, and (c) varying objects' speed

A varying objects' speed affects the geometry of the trajectories (cf. Figure 14(c)). Thus, the alignment in the index and consequently the number of node accesses during insertion changes.

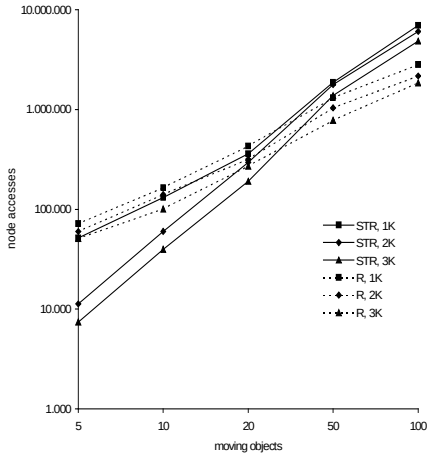


Figure 15: Node accesses for varying page sizes

5.5 Point and Range Queries

Query processing is the foremost important task for the use of indices. In this section we compare the STR and R-tree using two different types of indices stemming from datasets with varying number of moving objects and constant speed (t20_5 to t20_100) and constant number of objects with varying speed (t20_10, t20b_10 to t20d_10). Since the time horizon does not affect the quality of the index, we fixed it to 200K points. We used three different sets of queries (cf. Table 2), one for point and two for range queries. In Table 2, temporal and spatial extent indicate the size of the query window in the respective dimension.

In the following experiments, we measure the cost of a query by the total number of node accesses, but also subdivided into intermediate and leaf node accesses.

Dataset	# Tuples	Temporal Extent	Spatial Extent
qpt20	2000	1pt	1pt
q1boxt20	1000	1000pt(0.5%)	50(0.5%)
q2boxt20	1000	2000pt(2%)	50(0.5%)

Table 2: Query files

Figure 16 shows the results for querying indices stemming from datasets with *varying objects' speed*. It can be clearly seen that the STR-tree outperforms the R-tree in most cases for point queries. However, for both cases of window queries the R-tree remains superior to the STR-tree. Also, observe that the larger the query window, the larger is the gap between the two indices.

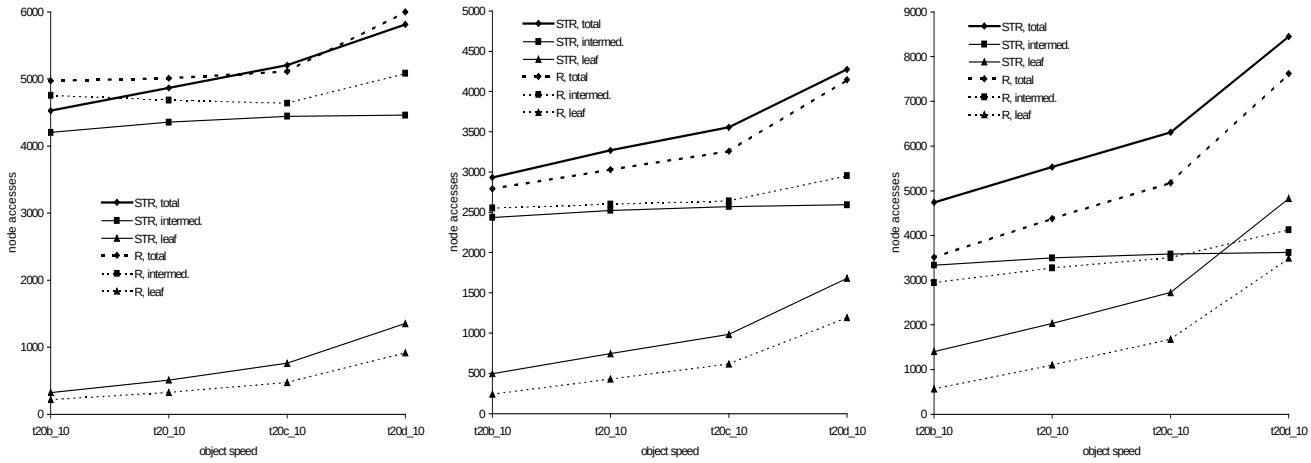


Figure 16: Query processing for indices containing data from slow/fast moving objects: point queries, (a) qpt20, and region queries, (b) q1boxt20 and (c) q2boxt20

Figure 17 shows the results for querying indices stemming from datasets with *a varying number of moving objects*. For point queries (cf. Figure 17 (a)) the STR-tree exhibits a lower cost than the R-tree for up to ca. 50 moving objects. For range queries, the larger the range, this advantage in performance is more and more reduced. For a temporal and spatial range of 2% and 0.5%, respectively (cf. Figure 17(c)), the R-tree has a lower cost in all cases.

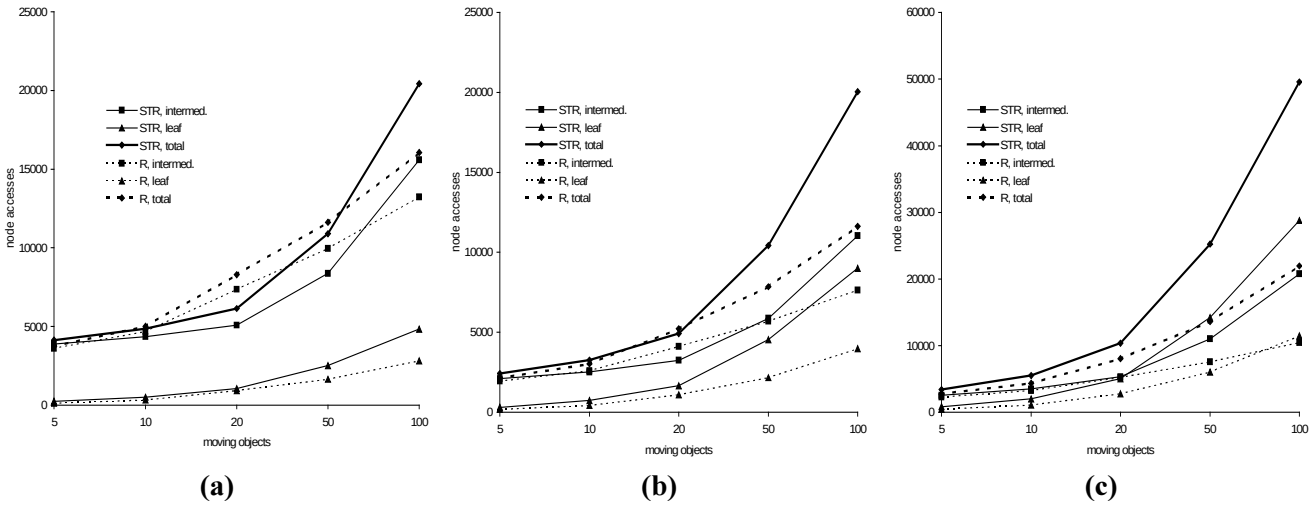


Figure 17: Query processing for indices containing data a varying number of moving objects: point queries, (a) qpt20, and region queries, (b) q1boxt20 and (c) q2boxt20

5.6 Semantics-Based Queries

Spatiotemporal applications are peculiar in that besides the classical spatial queries new types can be derived from the semantics of the data. As we saw in Section 2.2, semantics-based queries require the extraction of a partial trajectory to either derive information (speed and heading), or to simply evaluate a spatiotemporal relationship (enter, cross, etc.).

The STR-tree was designed to store connected segments close to each other by at the same time considering spatial decomposition. In this section we give the result of an experiment that should demonstrate the superiority of the STR-tree in querying a partial trajectory. We compare the STR-tree with a modified version of the R-tree that stores the orientation of the line segments in its leaf nodes. Essentially, we compare the Insertion and Split algorithms of the STR and R-tree.

For the experiments we measure the number of connected segments found depending on how many leaf levels are visited. Consider the example shown in Figure 18. Using an ordinary range query we find the bold

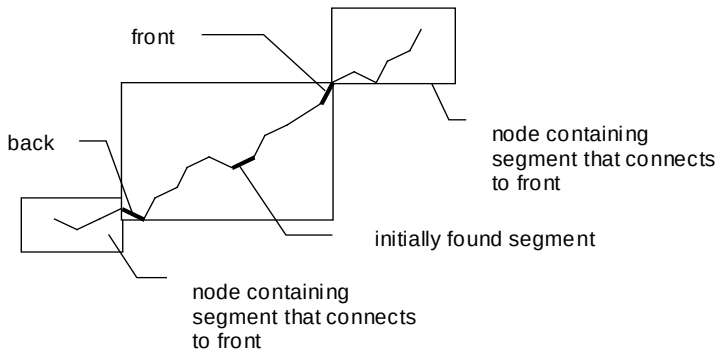


Figure 18: Finding connected line segments

segment in the center. Within the same leaf node (box) we find more segments that are connected to the first. The most extreme ones are labelled “front” and “back”. Depending on the parameter in the experiment we search only in the initial node (number of levels = 1), or two more leaf nodes which contain the segments connected to front and back, respectively (number of levels = 2), thus, visiting a total of three leaf nodes. In case the number of levels = 3, we would visit five leaf nodes, etc.

The cost of this whole operation is determined by the number of node accesses. Here, we distinguish visits to a node and node reads. When visiting a certain leaf node, a part of its path is already in memory, e.g., if the current leaf node has the same parent node as the one visited before, we only have to read the new leaf node from disk since the rest of its path through the tree is already in memory. Visiting in this context means accessing a node, whereas reading means actually reading it from disk.

According to the experiments in the previous sections, the number of moving objects is the most important parameter that alters the performance characteristics of the STR-tree. For these experiments, we chose three different datasets, t20_5, t20_10, and t20_50. For querying the initial line segments we use the range query file q1boxt.

In Figure 19 we measure the cost for searching. For level = 1 the cost is equal to the cost of a range query as shown in Figure 17. The cost for searching *n* levels is generally lower for the STR-tree than for the R-tree. However, the rate of increase in the cost increases in both cases with the number of moving objects.

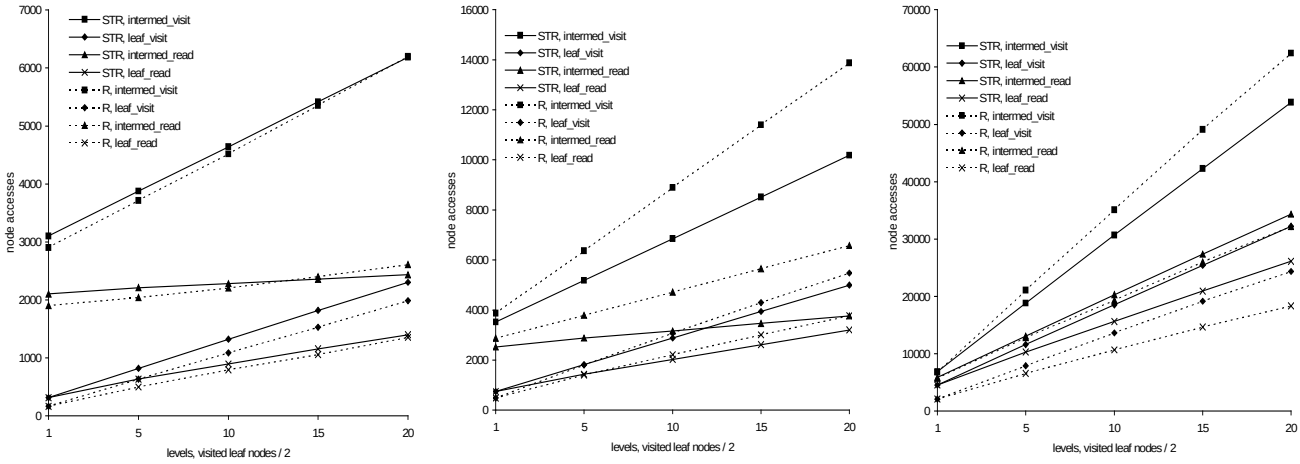


Figure 19: Number of node accesses for different number of levels and datasets, (a) t20_5, (b) t20_10, and (c) t20_50

In Figure 20 we see the actual number of connected line segments which were found for a given number of levels. Here, the STR-tree outperforms the R-tree by a factor of up to four. Interestingly, for indices from a large number of moving objects the STR-tree shows the highest factor. This seems to be plausible, since trajectories are preserved “accidentally” in the R-tree, and for a larger number of moving objects this preservation vanishes. The number of found segments decreases with the number of moving objects, although, the rate of decrease is higher for the R-tree than it is for the STR-tree.

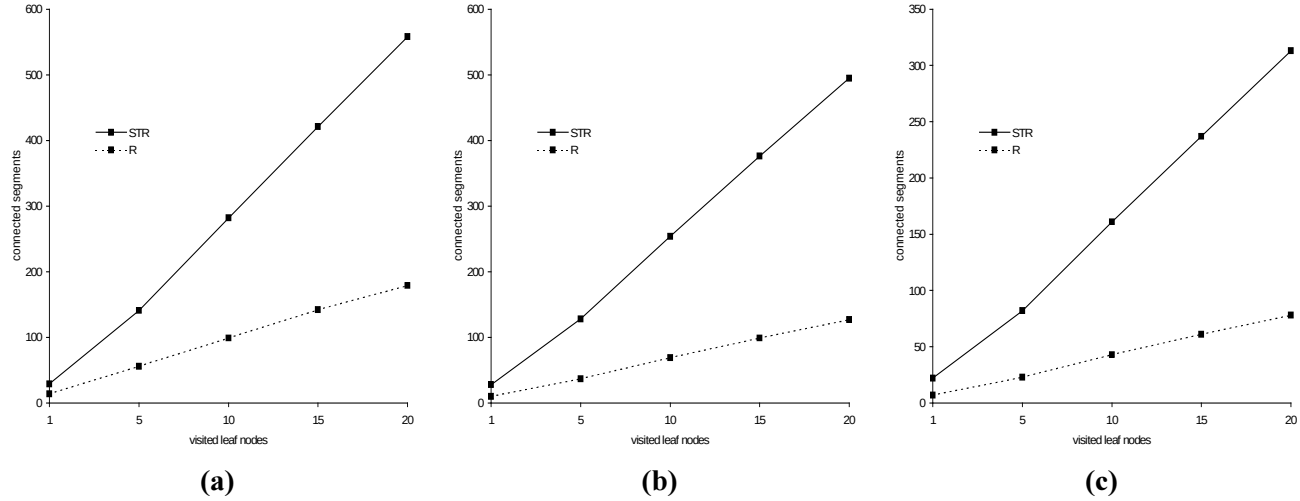


Figure 20: Found connected segments for different number of levels and datasets, (a) t20_5, (b) t20_10, and (c) t20_50

5.7 Summary of Performance Studies

The performance studies can be concluded by saying that the STR-tree is an index supporting trajectory and navigational queries more efficiently than the R-tree. At the same time, its performance on typical point and range queries is within the range of the R-tree. As shown in the experiments, the STR-tree performance is closely connected to the "number of moving objects" parameter of the datasets; the STR-tree performance degrades with an increasing number of moving objects.

6 Conclusions

The paper presents a new access method for indexing the trajectory data of moving point objects. First, trajectory data of moving point objects and a set of queries are defined to derive the requirements to the access method. Trajectory data is obtained by discretely sampling the movement of point objects in time. A linear interpolation is used in between the samples. The set of queries comprises classical spatial queries, e.g., point and range queries, as well as pure spatiotemporal queries, e.g., trajectory and navigational queries.
Subsequently, the R-tree is discussed to determine the shortcomings of this method with respect to spatiotemporal data and queries. The paper proceeds to introducing the STR-tree, an access method tailor-made to the requirements of trajectory data and spatiotemporal queries. The STR-tree can be easily implemented on top of the R-tree, a method already adopted in commercial extensible database systems, since it maintains several properties and construction algorithms of it.
The performance study presents results from experiments involving, index creation, spatial point and range queries, as well as experiments related to trajectory and navigational queries. The STR-tree proves to be an access method well suited for spatiotemporal data and queries, by at the same time having a good spatial search performance.

This work points to several future research directions. One concern with the STR-tree is that its spatial search performance degrades with data stemming from a larger number of moving point objects. Research should go into modifying the STR-tree to overcome this limitation. In the STR-tree we use MBBs as approximations. A suggestion would be to use other geometric bodies that are a more suitable approximation for moving objects' trajectories. The present work only presents preliminary algorithms to process navigational and trajectory queries. Derived from the requirements from real spatiotemporal applications, e.g., fleet management, these algorithms can be refined and defined in more detail. Also, it would be interesting to see, whether those two types of queries are the only ones particular for the present application context.

7 References

Abel, D.J., and Smith, J.L.: A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics and Image Processing*, 24, pp. 1-13, 1983.

Allen, J.F.: Maintaining knowledge about temporal intervals. *Communications of the ACM*, 26(11), pp. 832-843, 1983.

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

Census, Bureau of the: Tiger/Line Census Files: Technical Documentation. U.S. Department of Commerce, 1991.

Egenhofer, M. and Franzosa, R.: Point-set topological spatial relations. *International Journal of Geographic Information Systems*, 5(2), pp. 161-174, 1991.

Erwig, M. and Schneider, M.: Developments in spatio-temporal query languages, Workshop on Spatio-Temporal Data Models and Languages, Florence, Italy, 1999

Erwig, M., Gueting, R.H., Schneider, M., and Vazirgiannis, M.: Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases, *GeoInformatica*, 3(3), 1999. To appear.

Faloutsos, C.: Gray codes for partial match and range queries. *IEEE Transactions on Software Engineering*, 14, pp. 1381-1393, 1988.

Gaede, V. and Guenther, O.: Multidimensional access methods. *ACM Computing Surveys*, pp. 170-231, 1998

Guenther, O.: The design of the cell-tree: an object-oriented index structure for geometric databases. In *Proceedings of the IEEE 5th Conference on Data Engineering*, pp. 598-605, 1989.

Gueting, R.H.: An introduction to spatial database systems. *VLDB Journal – Special Issue on Spatial Database Systems*, 3(4), pp. 357-399, 1994.

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

Henrich, A., Six, H.W., and Widmayer,P.: The LSD-tree: spatial access to multidimensional point and non-point objects. In *Proceedings of the 15th International Conference on Very Large Databases*, pp. 45-53, 1989.

Jagadish, H.V.: On indexing line segments. *In Proceedings of the 16th International Conference on Very Large Databases*, pp. 614-625,1990.

Kollios, G., Gunopulos, D., and Tsotras, V.: On indexing mobile objects. In *Proceedings of the 18th ACM Symposium on Principles of Database Systems,* pp. 261-272, 1999.

Nascimento, M., Silva, J., and Theodoridis, Y.: Evaluation of access structures for discretely moving points. In *Proceedings of the International Workshop on Spatio-Temporal Database Management*, 1999.

Nievergelt, J., Hinterberger, H., and Sevcik, K.: The Grid File: an adaptable, aymmetric multikey file structure. Transactions on Database Systems, 9(1), pp. 38-71, 1984.

Nievergelt, J. and Hinrichs, K.: Storage and access structures for geometric databases. In *Proceedings of the International Conference on Foundations of Data Organisation*, pp. 441-455, 1985.

Orenstein, J.: Spatial query processing in an object-oriented database system. *In Proceedings of the ACM-SIGMOD Conference on the Management of Data*, pp. 326-336, 1986.

Orenstein, J.: A comparison of spatial query processing techniques for native and parameter spaces. In *Proceedings of the ACM-SIGMOD Conference on Management of Data*, pp. 343-352, 1990.

Pagel, P., Six, H., Toben, H., and Widmayer, P.: Towards an analysis of range query performance. In *Proceedings of the 12th ACM Symposium on Principles of Database Systems*, pp. 214-221, 1993.

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

Samet, H.: The Quadtree and related hierarchical data structures. *ACM Computing Surveys* 16(2): pp. 187-260, 1984.

Sellis, T., Roussopoulos, N., and Faloutsos, C.: The R^+-tree: a dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Databases*, pp. 507-518, 1987.

Sistla, A., Wolfson, O., Chamberlain, S., and Dao, S.: Modeling and querying moving objects. In *Proceedings of the 13th International Conference on Data Engineering*, pp. 422-432, 1997.

Stefanakis, E., Theodoridis, Y., Sellis, T., and Lee, Y.C.: Point Representation of Spatial Objects and Query Window Extension: A New Technique for Spatial Access Methods. *International Journal of Geographical Information Science*, 11(6), pp. 529-554, 1998.

Tayeb, J., Ulusoy, O., and Wolfson, O.: A Quadtree Based Dynamic Attribute Indexing Method. *Computer Journal*, 41(3), pp. 185-200, 1998.

Theodoridis, Y., Sellis, T., Papadopoulos, A., and Manolopoulos, Y.: Specifications for efficient indexing in spatiotemporal databases", In *Proceedings of the 10th International 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 *Proceedings of the 6th International Symposium on Spatial Databases*, pp.147-164, 1999.

Theodoridis, Y., Stefanakis, E., Sellis, T.: Efficient cost models for spatial queries using R-trees. In *IEEE Transactions on Knowledge and Data Engineering*, to appear, 2000.