Indexing the Trajectories of Moving Objects in Networks *

Victor Teixeira de Almeida
Ralf Hartmut Güting
Praktische Informatik IV
Fernuniversität Hagen, D-58084 Hagen, Germany
{victor.almeida, rhg}@fernuni-hagen.de

Abstract

The management of moving objects has been intensively studied in recent years. A wide and increasing range of database applications has to deal with spatial objects whose position changes continuously over time, called moving objects. The main interest of these applications is to efficiently store and query the positions of these continuously moving objects. To achieve this goal, index structures are required. The main proposals of index structures for moving objects deal with unconstrained 2-dimensional movement. Constrained movement is a special and a very important case of object movement. For example, cars move in roads and trains in railroads. In this paper we propose a new index structure for moving objects on networks, the MON-Tree. We describe two network models that can be indexed by the MON-Tree. The first model is edge oriented, i.e., the network consists of edges and nodes and there is a polyline associated to each edge. The second one is more suitable for transportation networks and is route oriented, i.e., the network consists of routes and junctions. In this model, a polyline also serves as a representation of the routes. We propose the index in terms of the basic algorithms for insertion and querying. We test our proposal in an extense experimental evaluation with generated data sets using as underlying networks the roads of Germany. In our tests, the MON-Tree showes good scalability when increasing the number of objects and time units in the data sets, as well as the query window and time interval in query processing. The MON-Tree that indexes the route oriented model showed the best overall results.

Keywords: spatio-temporal databases, moving objects in networks, index structures

1 Introduction

The management of moving objects has been intensively studied in recent years. A wide and increasing range of database applications has to deal with spatial objects whose position changes over time, such as taxis, air planes, oil tankers, criminals, polar bears, and many more examples. The main interest of these applications is to store and efficiently query the positions of continuously moving objects.

There are two main approaches in the literature that try to model this problem. The first one in [7, 10, 8, 5] is a complete framework that serves as a precise and conceptually clean foundation for the representation and querying of spatio-temporal data. This approach provides a set of data types, such as *moving point* or *moving region*, and suitable operations to support querying. Complete trajectories of the moving objects are stored

*This work was partially supported by a grant Gu 293/8-1 from the Deutsche Forschungsgemeinschaft (DFG), project "Datenbanken für bewegte Objecte" (Databases for Moving Objects)

such that querying past states is possible. Such data types can be embedded as attribute types into object-relational or other data models; they can be implemented and provided as extension packages (e.g. data blades) for suitable extensible DBMS environments.

The other one in [26, 32, 34, 33] proposes the Moving Objects Spatio-Temporal (MOST) model and the Future Temporal Logic (FTL) language for querying current and future locations of moving objects. This model is limited to moving points, which is the most important type of moving objects. Some policies are proposed to decide how often updates of motion vectors are needed to balance the cost of updates against imprecision in the knowledge of positions. The work in [31] addresses the problem of querying moving objects databases capturing the inherent uncertainty associated with the location of moving objects.

Indexing techniques have been used since the advent of relational database management systems with success. Indexing is even more important when the data is more complex and, for spatial databases systems, due to high performance requirements, access methods should be used on every relation for supporting queries efficiently. In spatial applications, the assumption is almost always true that a spatial index exists on a spatial relation ([1]). Following these ideas, for moving objects databases, which is a spatio-temporal application, and consequently more complex, the need of good indexing techniques is obvious.

As stated in [20], two different indexing problems must be solved in order to support applications involving continuous movement. The first one is the indexing of the complete trajectories of the moving objects, whereas the second one is the indexing of current and anticipated future positions of moving objects. These two problem on this statement are exactly the two main models of moving objects presented before. Most of the papers in the literature proposing index structures for moving objects, including [20], try to solve the second problem, and also most of them handle only moving point objects [1].

For querying past trajectories of moving objects, there are some works that propose index structures, e.g., [22, 27, 28, 3, 4]. For querying current and future positions of the moving objects, we can cite [29, 25, 24, 23, 15, 19].

Most of these approaches for indexing moving objects assume free movement of the objects in the 2-dimensional space. As stated in [21], applications dealing with moving objects can be grouped into three movement scenarios, namely unconstrained movement (e.g., vessels at sea), constrained movement (e.g., pedestrians), and movement in transportation networks (e.g., trains and cars). The latter category is an abstraction of constrained movement, i.e., for cars, one might be only interested in their position with respect to the road network, rather than in absolute coordinates. Then, the movement can be viewed as occurring in a different space than for the first two scenarios, which is called 1.5 dimensional space in [15].

For the constrained movement scenario, a two-step query processing is proposed in [21]. A pre-processing step is added where the infrastructure is queried and the query window is divided into a set of smaller query windows, from which the regions covered by the infrastructure have been excluded. This set of smaller query windows is then passed to a second step where a modified version of the R-Tree and the TB-Tree are queried using an approach proposed in [18].

In the context of Spatial Network Databases, the work in [17] proposes an index structure and the counterpart algorithms for spatial queries such as *nearest neighbors*, *range search*, *spatial joins*, and *closest pairs*, using the network distance instead of the Euclidean distance. This work applies for location-based services, but not for moving

[1]Since we, in this paper, are also interested in index structures for moving point objects, we will use the terms *moving objects* and *moving point objects* interchangeably.

object databases, since the objects in the structure are assumed to be static.

Recently, two index structures for indexing the trajectories of moving objects in networks have been proposed. Both use the same idea of converting a 3-dimensional problem into two sub-problems with lower dimensions. The first of these sub-problems is responsible to index the network data and the second one is responsible to index the moving objects.

The first of these index structures, the Fixed Network R-Tree (FNR-Tree), proposed in [9], consists of a top level 2D R-Tree, whose leaf nodes contain pointers to 1D R-Trees. The 2D R-Tree is used to index the edges of the network, indexing their corresponding line segments. For every leaf node in the 2D R-Tree, there is an 1D R-Tree indexing the time interval of the objects traversing the line segments inside of its entries. The main disadvantage of this approach is the network model used, where each edge in the network can represent only a single line segment. This model leads to a high number of entries and lots of updates in the index structure. Another problem of the FNR-Tree is that an object cannot end (or change) its movement in the middle of an edge, but only at nodes.

The second index structure proposed in [13] stores the network edges also as line segments into a 2D R-Tree and the moving objects into another 2D R-Tree. The difference between both index structures lies in the indexing of moving objects. The 2-dimensional movement space (without the temporal extension) is mapped into a 1-dimensional space using a Hilbert curve to linearize the network line segments, and consequently possible objects' positions. This approach has the same disadvantage as the FNR-Tree, with respect to the network model, which leads to a high number of entries in the index structure. An additional disadvantage occurs in the query processing. Since in the FNR-Tree, there is a bottom 1D R-Tree for each node to index the objects' movements on the edges inside it, here a global 2D R-Tree is used to index the movements, which makes the second phase of the query more expensive. A good point (against the FNR-Tree) is that the movements are represented in a way that they can begin and end anywhere along the edges. Is is also possible to change the speed or direction of a moving object inside of an edge. Finally, even being a more representative index structure, it is not expected it to be better than the FNR-Tree in query processing.

In this paper we propose a new index structure, the **M**oving **O**bjects in **N**etworks **Tree** (MON-Tree) to efficiently store and retrieve objects moving in networks. The index structure stores the complete trajectories of the moving objects and is capable to answer queries about the past states of the database.

We use two different network models that can be indexed by the MON-Tree. The first model is edge oriented, i.e., the network consists of edges and nodes and there is a polyline associated to each edge. This model has been extensively used, e.g. in [17], and it is simple and straightforward, but not the best one in the sense of transportation networks. Highways, for example, contain lots of connections (exits) and some junctions. We have names for roads, not for crossings or pieces of roads between crossings. Addresses are given relative to roads. The model should reflect this. We captured these ideas in [11], where we extended the ADT approach in [10] for network constrained movements. In this route oriented model the network consists of routes and a set of junctions between these routes.

The MON-Tree is capable of answering two kinds of queries about past states of the database, namely the *range query* and the *window query*. Both receive a spatio-temporal window as an argument and differ on their results: while the range query returns all objects whose movements overlap the query window, the window query is more precise and only returns the pieces of the objects' trajectory that overlap the query window.

The MON-Tree keeps the good properties of both index structures in [9] and [13] but

eliminates their main disadvantages. Our proposal is tested in an experimental evaluation with generated data sets using the roads of Germany as underlying networks. An extended abstract of this paper without the experimental evaluation can be found in [6].

This paper is structured as follows: Section 2 reviews in more detail the index structure and query processing of the FNR-Tree proposed in [9]. Section 4 proposes the MON-Tree index structure and the inserting and searching algorithms. Section 5 experimentally evaluates the MON-Tree for the two network models and compares them to the corresponding FNR-Tree. Finally, Section 6 concludes the paper and proposes some future work.

2 The FNR-Tree

The Fixed Network R-Tree was recently proposed in [9] to solve the problem of indexing objects moving in fixed networks. The structure consists of a top level 2D R-Tree whose leaf nodes contain pointers to 1D R-Trees. The 2D R-Tree is used to index the edges of the network, indexing their corresponding line segments. For every leaf node in the 2D R-Tree, there is an 1D R-Tree indexing the time interval of the objects traversing the line segments inside of its entries.

Since the spatial objects stored in the 2D R-Tree are line segments, the leaf node entries are of the form $\langle mbb, orientation \rangle$, where mbb is the minimum bounding box (MBB) of the line segment and $orientation$ is a flag that describes the two possible orientations that a line segment can assume inside a MBB.

Each leaf entry of the 1D R-Tree is of the form $\langle moid, edgeid, t_{entrance}, t_{exit}, direction \rangle$, where $moid$ is the identification of the moving object, $edgeid$ is the identification of the edge, $t_{entrance}$ and t_{exit} represent the time interval when the moving object traversed the edge, and $direction$ is a flag that describes the direction of the moving object, more specifically, the direction 0 means that the movement began at the left-most node of the edge (bottom-most for vertical edge), and 1 otherwise.

The insertion algorithm is executed each time a moving object leaves a given line segment of the network. It takes the line segment on which the object was moving, its direction, and the time interval when the object traversed the line segment. The algorithm first searches in the 2D R-Tree for the line segment and then inserts the time interval (associated with the direction) of the moving object into the associated 1D R-Tree. Instead of using the insertion algorithms of the R-Tree, in the 1D R-Tree, every new entry is simply inserted into the most recent (right-most) leaf of the 1D R-Tree. This is possible because time is monotonic and the insertions are done in time increasing order.

The search algorithm consists of three steps. It receives a spatio-temporal query window $w = (x_1, x_2, y_1, y_2, t_1, t_2)$ and, in a first step finds all edges whose line segments overlap the rectangle $r = (x_1, x_2, y_1, y_2)$. Then, in the second step, for all 1D R-Trees pointed to by the edge found in the first step, it looks for the moving objects traversing it with time interval overlapping $t = (t_1, t_2)$. Finally, in the third step, the corresponding edge among those of the first step (they are stored in main memory) is searched by the $edgeid$ information of the 1D R-Tree leaf node. The object's 3-dimensional movement is then re-composed and those parts that are fully outside of the spatio-temporal query window w are rejected.

The main disadvantage of this approach is the network model used, where each edge in the network can represent only a single line segment. This model leads to a high number of entries and lots of updates in the index structure, since distinct entries are needed for every line segment the object traverses.

Another problem of the FNR-Tree is that in the 1D R-Tree only time intervals are stored. It is assumed that the objects' movements always begin and end in nodes, therefore

an object cannot end its movement in the middle of an edge. Even worse, it also cannot change its speed or direction in the middle of an edge, but only in the nodes.

3 The Network Models

In this section we describe in more detail the two different network models that can be indexed by the MON-Tree.

The first is straightforward. Here, a network is a graph $G = (N, E)$ where N is a set of nodes and $E \subseteq N \times N$ is a set of edges. A node $n \in N$ has an associated point $p_n = (x, y)$ in the 2-dimensional space and an edge $e \in E$ connects two nodes n_{e1} and n_{e2} and has an associated polyline $l_e = p_1, \ldots, p_k$, where p_i are 2-dimensional points, $1 \leq i \leq k$, k is the size of the edge, $p_1 = p_{n_{e_1}}$, and $p_k = p_{n_{e_2}}$. A position pos_e inside an edge e is represented by a real number between 0 and 1, where 0 means that the position lies on the node n_{e_1} and 1 means that the position lies on the node n_{e_2} of the edge e. The domain of a moving object position inside a graph G is $D(G) = E \times pos$. The time is given by a time domain T isomorphic to real numbers. A moving object then, is a partial function $f : T \rightarrow D(G)$. Figure 1 (a) shows an example of a network divided into edges using this model. As an example of usage of this model, we can cite [17].

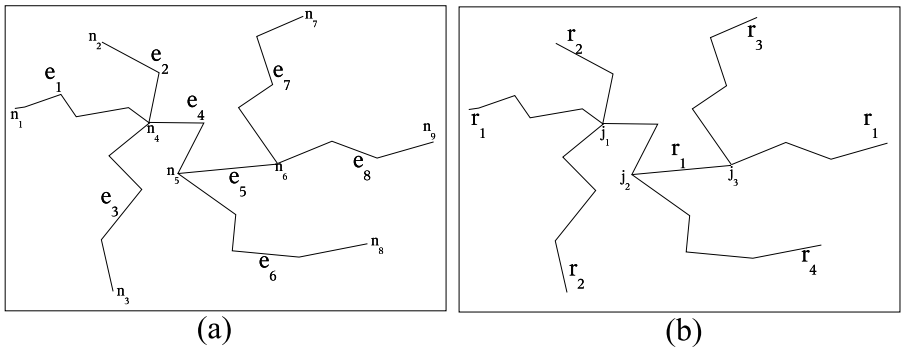

Figure 1: Example of a network (a) in the first model and (b) in the second model.

This first model is simple and straightforward, but not the best one to represent transportation networks. In [11] we extended the framework in [10] to handle network constrained movement, where a route oriented model is used. In this model, the network is represented in terms of routes and junctions between the routes, i.e., a network $G' = (R, J)$, where R is a set of routes and J is a set of junctions. A route $r \in R$ has an associated polyline $l_r = p_1, \ldots, p_k$, where p_i are 2-dimensional points, $1 \leq i \leq k$, and k is the size of the route. A position pos_r inside a route r is represented by a real number between 0 and 1, where 0 means that the position lies on the point p_1 and 1 means that the position lies on the point p_k of the route. A junction $j \in J$ is represented by two routes r_1 and r_2 and two route positions pos_{r_1} and $rpos_{r_2}$. The domain of a moving object position inside a graph G' is $D'(G') = R \times pos$. The time domain T is the same and then, a moving object in this second model is a partial function $f : T \rightarrow D'(G')$. The Figure 1 (b) shows an example of a network divided into routes using this second model. We can see in this figure that the route r_1, for example contains the edges e_1, e_4, e_5, and e_8 of the edge oriented model.

A detailed discussion about why using the route oriented model is given in [11]. The most practical reason for using the route oriented model instead of the edge oriented model is, taking indexing techniques into consideration, that the representation of a moving

object becomes much smaller in this way. If positions are given relative to edges, then for example a car driving along a highway at constant speed needs a change of description at every exit and/or junction, because the edge is changed. If positions are given relative to routes, then the description needs to change only when the car changes the highway. This is directly reflected in the index, where every change leads to another entry in the index structure. Another good reason to use the route oriented model for indexing moving objects in networks is that the index structure can be plugged directly into the framework proposed in [11]. A similar model is also used in the kilometer-post representation in [12].

4 The MON-Tree

In this section we propose a new index structure to efficiently store and retrieve past states of objects moving in networks, the MON-Tree. The Section 4.1 presents the index structure, and the insertion and search algorithms are presented in Section 4.2 and 4.3, respectively.

4.1 Index Structure

The index structure proposed in this paper assumes that objects move along polylines, which can be associated to edges, for the first network model, or to routes, for the second network model. The index structure consists of a 2D R-Tree (the *top R-Tree*) indexing polyline bounding boxes and a set of 2D R-Trees (the *bottom R-Trees*) indexing objects' movements along the polylines. We are aware of the problem of the high dead space in polyline MBBs, which is discussed in Appendix A where a complementary index structure is proposed. It is shown there that there is no gain of using this complementary index structure.

We also use a hash structure in the top level containing entries of the form $\langle$*polyid*, *bottreept*$\rangle$, where *polyid* is the polyline identification and *bottreept* is a pointer to the corresponding bottom R-Tree. The hash structure is organized by *polyid*.

Hence, we have two top level index structures: an R-Tree and a hash structure; pointing to bottom level R-Trees. These two top level index structures are used as follows: first, the insertion algorithm for moving objects takes a polyline identification as an argument, and then uses the top level hash structure to find the bottom level R-Tree into which the movement should be inserted. Second, the search algorithm takes a spatio-temporal window as an argument and starts the search on the top R-Tree, which contains the polylines' bounding boxes.

An example of the MON-Tree index structure can be seen in Figure 2. This figure shows the corresponding index structures for the networks in Figure 1.

In the top R-Tree, the polylines are indexed using a MBB approximation. In this way, the leaves of this tree contain the information $\langle$*mbb*, *polypt*, *treept*$\rangle$, where *mbb* is the MBB of the polyline, *polypt* points to the real representation of the polyline, and *treept* points to the corresponding bottom R-Tree of that polyline. Internal nodes have the following information $\langle$*mbb*, *childpt*$\rangle$, where *mbb* is the MBB that contains all MBBs of the entries in the child node, and *childpt* is a pointer to the child node.

The bottom R-Tree indexes the movement of the objects inside a polyline. The movement is represented by the position interval (p_1, p_2) and a time interval (t_1, t_2), where $0 \leq p_1, p_2 \leq 1$. These two values p_1 and p_2 store the relative position of the objects inside of the polyline at times t_1 and t_2, respectively.

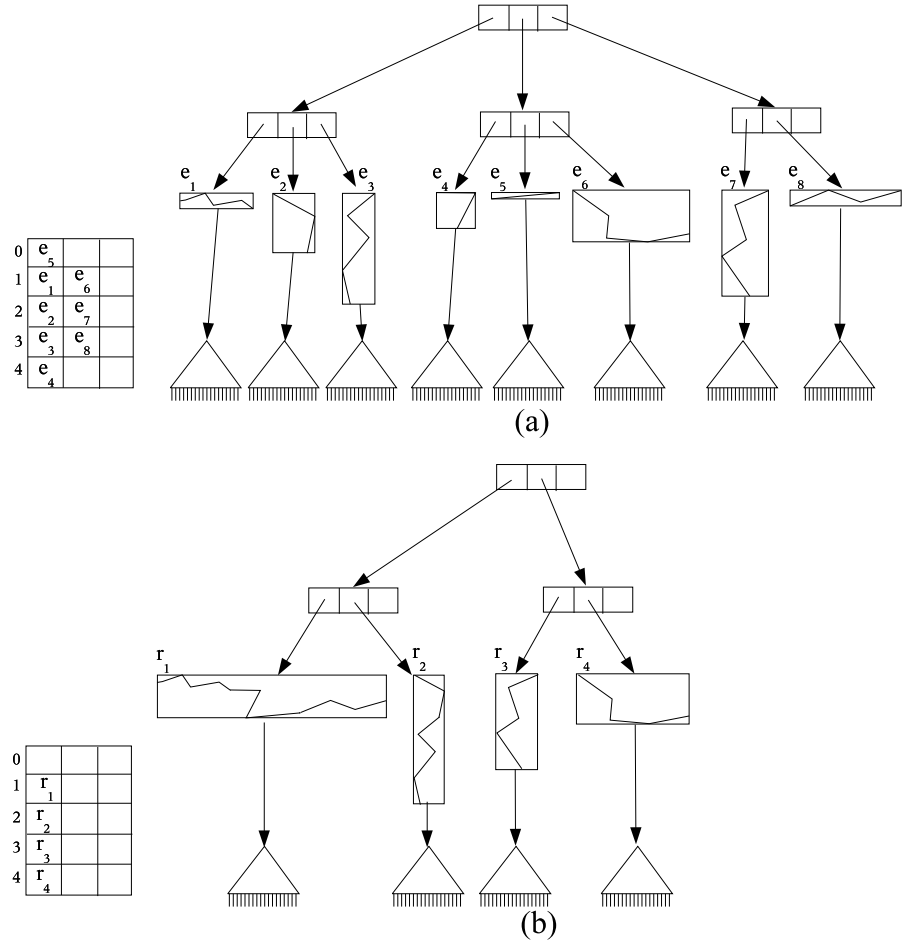

Figure 2: Example of the index structure of the networks in Figure 1.

4.2 Insertion

In this index structure two different kinds of insertion are allowed: polyline insertion and movement insertion. A polyline insertion is needed to construct the basic network. The moving object insertion is needed every time an object is created or it changes its motion vector, i.e., its speed and/or direction. It is also necessary to perform a moving object insertion every time an object changes from one polyline to another.

4.2.1 Polyline Insertion

The algorithm for polyline insertion is very simple: just insert the polyline identification with a *null* pointer in the hash structure. The insertion of the polyline in the top R-Tree is postponed to the insertion of the first moving object traversing it. The reason for this approach is to avoid having polylines without moving objects in the top R-Tree, as long as they do not participate in queries. In this way, we keep the top R-Tree as small as possible.

In our experiments we assume that the network is fixed and has been previously loaded. We also assume that we have a relation containing the polylines (and also information about edges or routes, depending on the model used) of the network stored in a separate file. In this way, when we first create the index, we scan the whole edge/route relation adding entries for them with *null* pointers to bottom R-Trees in the hash structure. It is

important to note that the hash structure does not contain the whole polyline (which can be big), but a pointer to its real representation in the polyline relation.

4.2.2 Movement Insertion

The movement insertion algorithm takes as arguments the moving object identification *moid*, the polyline identification *polyid*, the position interval $p = (p_1, p_2)$ where the object moved along the polyline, and the corresponding movement time interval $t = (t_1, t_2)$. The algorithm starts in the top hash structure searching for the associated polyline, i.e., the polyline which has identification number equal to *polyid*. If the polyline does not have an associated bottom R-Tree yet, then a new one is created and the polyline's MBB is inserted on the top R-Tree. The pointer to this newly created bottom R-Tree is updated in the top hash structure. Now, given the associated bottom R-Tree, the rectangle (p_1, p_2, t_1, t_2) is inserted into it using the insertion algorithm of the R-Tree.

4.3 Search

Given a spatio-temporal query window $w = (x_1, x_2, y_1, y_2, t_1, t_2)$, the query of the form: "find all objects that have lied within the area $r = (x_1, x_2, y_1, y_2)$, during the time interval $t = (t_1, t_2)$" are expected to be the most common ones addressed by spatio-temporal database management system users ([30]). This query is commonly called *range query* in the literature. A variant of this query is to find only the pieces of the objects' movements that intersect the query window w. We call this a *window query*. The main functionality of the MON-Tree index is to answer these two kinds of query.

For the window query, the algorithm receives a spatio-temporal query window w and proceeds in three steps. In the first step, a search in the top R-Tree is performed to find the polylines' MBBs that intersect the spatial query window r. Then, in the second step, the intervals where the polyline intersects r are searched using the real polyline representation. It is important to note that this procedure is done in main memory and the result is a set of windows $w' = \{(p_{11}, p_{12}, t_1, t_2), \ldots, (p_{n1}, p_{n2}, t_1, t_2)\}$, where n is the set size, $n \geq 1$, and the interval (t_1, t_2) is the query time interval t. Moreover, the windows are disjoint and ordered, i.e., $p_{i1} \leq p_{i2} \wedge p_{i2} < p_{(i+1)1}$, $1 \leq i \leq n-1$. An example of the result of this procedure can be seen in Figure 3.

Given this set of windows w', in the third step, the bottom R-Trees are searched using a modified algorithm for searching a set of windows, instead of only one. We decided not to use the algorithm proposed in [18] because our problem is much simpler than the one solved there. In [18], if the queries are combined (a threshold is proposed to decide whether to combine them or not), then a query is performed with a window that contains all window queries in the set, using the R-Tree query algorithm. This approach can lead to a lot of wasted area, and consequently more disk accesses.

To solve our problem, we propose to pass the set of query windows w' as an argument to the search algorithm and change the R-Tree search algorithm to handle multiple query windows. We need to change only the decision to go down in the tree for internal nodes, or to report entries for leaf nodes. The original R-Tree search algorithm goes down in the tree (internal nodes) or reports an entry (leaf nodes) if the entry's bounding box overlaps the query window. Since we have a set of query windows, we go down in the tree (internal nodes) or report an entry (leaf nodes) if the entry's bounding box overlaps at least one of the windows in the set w'. The problem is now: there is a set of query windows w' and a set of entry bounding boxes in the node and we want to go down in the tree or report the entries that have intersection with the set of query windows w'. This amounts to finding overlapping pairs between two sets of rectangles and can be done using a plane

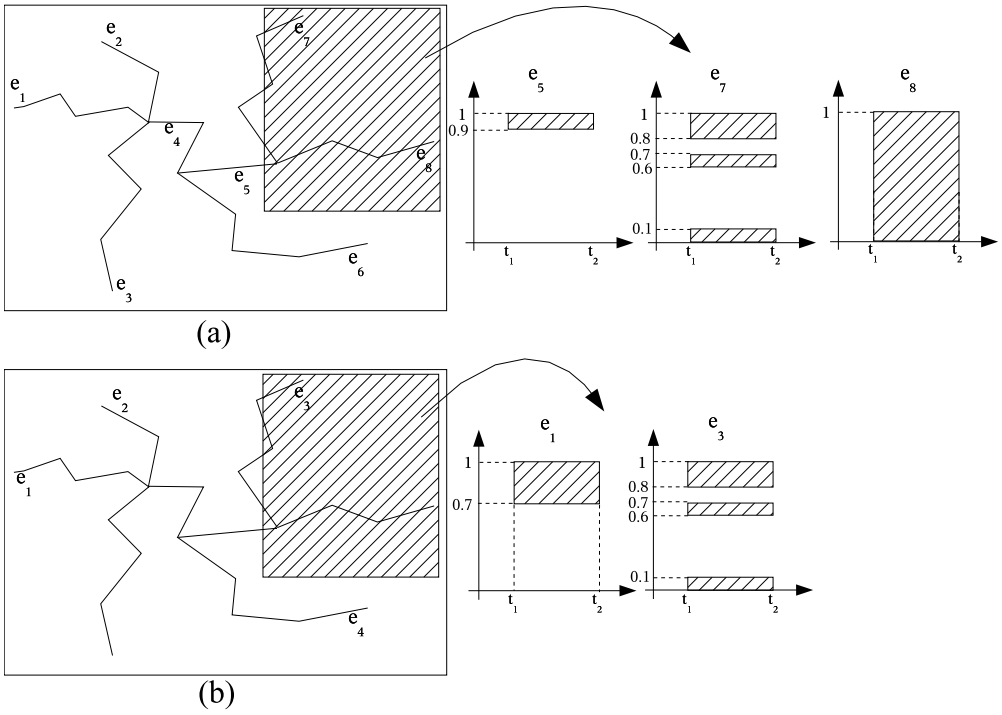

Figure 3: Example of the interval set determination in the search algorithm (a) in the partition into edges without connection; and (b) in the partition into edges containing connections.

sweep algorithm in $O(nlogn + k)$ time, where n is the total number of rectangles and k is the number of overlapping pairs. In our case, a simpler algorithm can be performed, since, as stated before, the set of query windows w' contains the same time interval $t = (t_1, t_2)$, and the position intervals $p' = ((p_{11}, p_{12}), \ldots (p_{n1}, p_{n2}))$ are disjoint and ordered. These properties can be verified in the example of the Figure 3. Thus, the idea is to ask every entry in the node whether it intersects the query window w'. This intersection test can be done in logarithmic time by first checking if the entry's time interval intersects the query time interval t, and if so, checking if it has at least one intersection with the position intervals p' using a binary search.

For the range query processing, we need an additional step after the third step of the window query to remove duplicates and return only the objects' identification. This step can be done in memory, i.e., the objects' identifications found in the third step of the window query can be stored in main memory and finally, a duplicate removal is performed.

5 Experimental Evaluation

In order to examine the performance of the MON-Tree, we did an experimental evaluation. To be able to present comparable results, we also implemented the FNR-Tree. We only became aware of the index proposed in [13] some time after our implementation and experimental evaluation was done. Since we expect that the performance of this index not be better than the FNR-Tree we did not find it necessary to add a further experimental comparison of our work with this index structure.

5.1 Environment

For our experiments we used a personal computer with an AMD Athlon XP 2400+™, 2000 MHz, and 256M bytes of main memory, running SuSe Linux 8.1. The index structures were implemented in C++ and compiled using g++ version 3.2.

5.2 Data Sets Description

In all our experiments, we used the network-based moving objects generator proposed in [2] [2]. We used the roads network of Germany downloaded from the *Geo Community* web site[3]. A screenshot of the generator with the network data can be seen in Figure 4.

Since the downloaded network data set had no information about routes, we have manually connected the lines to create a route network. We tried to create the route network as close as possible to the German highway network. The main highways of Germany, like A1 and A7 for example, are represented in the resulting data set, but the other smaller ones were connected arbitrarily. Later we exported this data set to an edge oriented format and to a set of line segments to be used by the generator and to be indexed by the FNR-Tree.

Some important statistics about the network data sets can be seen in Table 1.

Table 1: Statistics about network data sets.

Data set model	# of units	Minimum unit size	Average unit size	Maximum unit size
segment	30,649	1	1	1
edge	4,273	1	7.17	64
route	995	1	30.80	442

The most interesting information in this table is that we could join approximately 7 segments in each edge and 30 segments in each route (in average). As stated before, if an object moves along the same edge or route with the same speed, it only needs one entry in the index. Then, We expect to have much less index entries when using the edge and route oriented models.

We define (for the generator) three classes of routes based on their lengths. The first class contains the routes with lengths higher than 50% of the maximum length, i.e. the length of the longest route. The second contains the routes with lengths from 10 to 50% of the maximum length, and the third contains the routes with lengths smaller than 10% of the maximum length. The resulting network can be seen in Figure 4.

As stated in the Section 2, the FNR-Tree assumes that the objects' movements always begin and end in nodes, because time intervals are stored in the 1D R-Trees. The consequence of this limitation is that an object cannot end its movement, or change its velocity in the middle of an edge (which is a line segment). In order to be able to compare our proposal with the FNR-Tree, we changed the generator to reflect this limitation[4]. Hence, changes of speed inside of line segments (edges for the FNR-Tree model) are simply ignored. If an object changes its direction inside of a line segment, and goes back to the

[2] The generator is also available in Internet under the URL `http://www.fh-oow.de/institute/iapg/personen/brinkhoff/generator.shtml`

[3] `http://data.geocomm.com/catalog/GM/group103.html`

[4] We did not really change it, but simply used the *NodeReporter* class, that reports the movements only when they reach nodes.

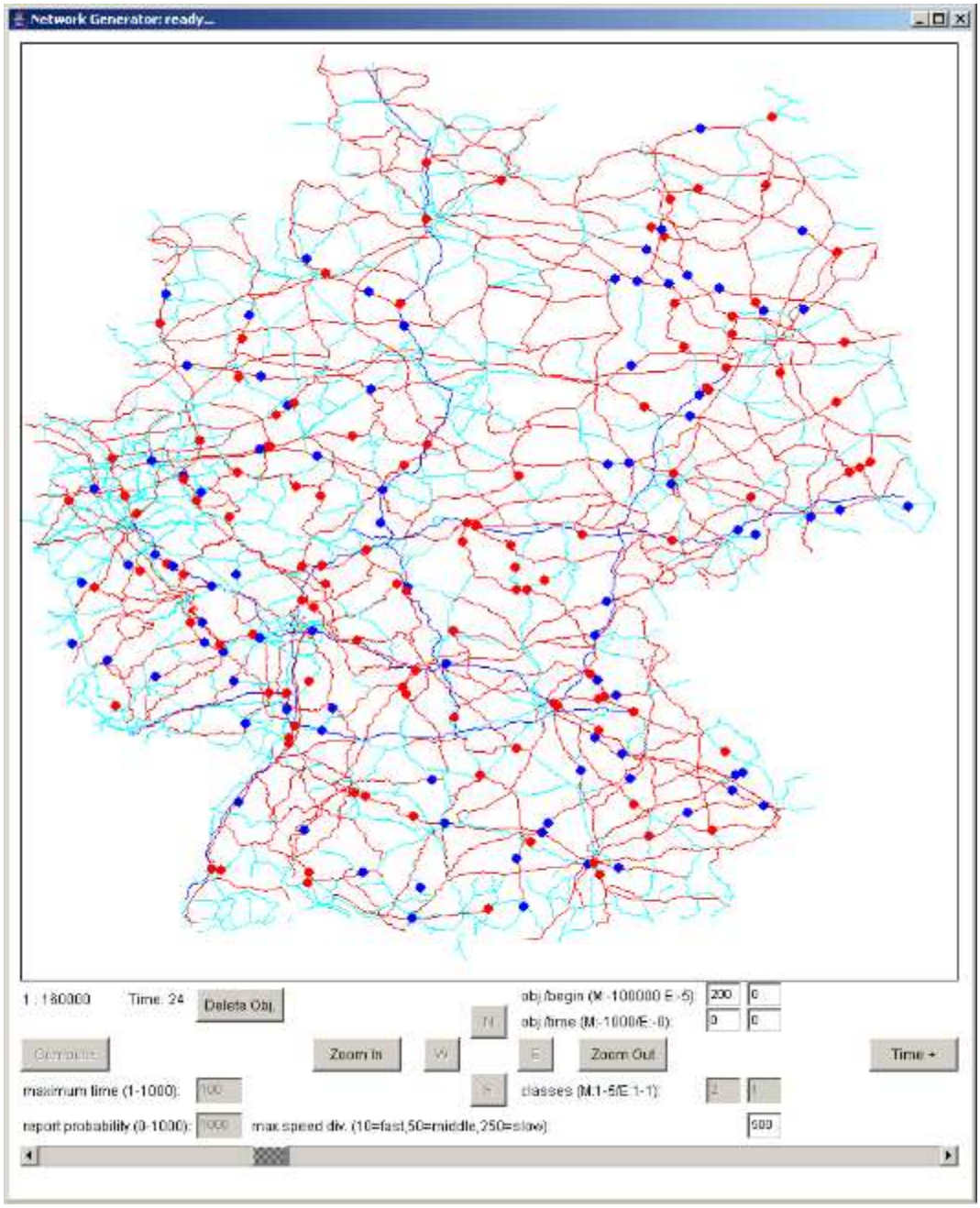

Figure 4: A screenshot of the generator containing the roads of Germany as network.

same point, this movement is also ignored, but the time spent is accumulated to its next movement.

As input for the data generator, which can be seen in Figure 4, one can set the following variables: the number of initial moving and external objects, the number of moving and external objects that are created in each subsequent time unit, the number of classes for moving and external objects, the maximum speed divisor which determines the maximum speed of the classes of objects and edges, the number of time units, and the report probability.

External objects are used to simulate weather conditions or similar events with impact

on the motion and speed of the moving objects, i.e., objects tend to decrease the speed when moving inside them. The report probability is used to simulate situations where the moving objects report their positions irregularly. In our tests we did not make use of external objects and we set the report probability to its maximum value. The number of classes of moving objects were set to two, and the number of time units to 100.

In order to keep the number of objects constant at each time unit, we made the following change in the generator. In the generator, for every object created, a start node and a target node are chosen randomly. When the object reaches the target node, its movement ends and it is logically deleted. We then added a new object creation every time an object reaches its destination. In this way, we keep the number of objects constant and equal to the initial number of objects inserted. The variable of the generator that controls the number of moving and external objects that are created in each subsequent time unit are left empty (with zero value).

The variables in the generator that were varied are the initial number of moving objects and the maximum speed divisor, because we are interested to show the behavior of the indexes according to these three variables:

- Number of objects. We generated data sets with 12500, 25000, 37500, and 50000 of initially inserted moving objects.
- Size of the trajectory. We do not have control over the size of the moving objects' trajectories in the generator, but we discovered that the higher the speed of the objects set in the generator the higher the trajectory sizes are. In this way, we generated objects with values of 250, 5000, 1000, and 2000 for the maximum speed divisor. As this number is a divisor for the maximum speed, the bigger the value, the lower is the maximum speed of the objects.
- Disk page size. We generated indexes using pages of 1k, 2k, and 4k bytes.

For all index structures we use an associated cache, which is a simple LRU cache to avoid some disk page accesses. One should note that every node in the index structures corresponds to a disk page, and consequently we use the terms *node* and *page* interchangeably in the further analysis. [16] shows the importance of a cache buffering disk pages. We used a 1 Mbyte cache size for the generation of the indexes as well as for query processing. One should note that the number of pages in the cache vary according to the page size variable set in the data set definition. Since we use a hash structure for managing the cache, we used prime numbers for the number of pages in the cache. We then used the maximum prime number that multiplied by the page size is less or equal than the cache size. As an example, to achieve a 1Mbyte cache size with a page size of 1kbyte, we can have 1021 pages in the cache.

5.3 Queries Description

In this experimental evaluation we use the so called *window query*, which tries to find all pieces of objects that were moving during a given time interval in a given rectangular area. We are then interested in the behavior of the indexes according to these variables:

- Size of the query time interval. We generated queries with a range of 1, 5, 10, 20 and 50% of the total data set time interval.
- Size of the query window. We generated queries with a range of 1, 5, 10, and 20 of the total data set space.

For each combination of these variables, we generated randomly 100 queries. Before each query, the cache is refreshed.

5.4 Moving Objects Data Sets

In this section we want to show the benefits of using the edge and the route oriented model. Only properties of the generated data sets of moving objects are shown in this section. In this way, the benefits showed here do not only apply for indexing approaches, but also for efficiently storing the spatio-temporal data in a database system.

What is shown here is that the representation of the moving objects gets smaller from the segment to the edge and finally to the route oriented model, which means less entries in the database and in the index structure. Table 2 shows the number of trajectory entries and the average trajectory sizes of the moving objects for each number of objects initially inserted in the data generation process using the maximum speed achieved with the maximum speed divisor value of 250.

Table 2: Trajectory sizes of the moving objects.

Model	initial # of moving objects	total # of moving objects	# of trajectory entries	Average object trajectory size
segment	12,500	29,196	3,113,483	106.641
edge	12,500	29,196	524,031	17.9487
route	12,500	29,196	207,153	7.09525
segment	25,000	55,602	5,798,500	104.286
edge	25,000	55,602	1,238,141	22.2679
route	25,000	55,602	724,058	13.0222
segment	37,500	78,682	8,015,980	101.878
edge	37,500	78,682	2,008,662	25.5289
route	37,500	78,682	1,375,880	17.4866
segment	50,000	100,049	9,875,964	98.7113
edge	50,000	100,049	2,796,281	27.9491
route	50,000	100,049	2,095,255	20.9423

First, we can observe that the number of insertions needed to fulfill the condition of having a fixed number of objects at each time unit is approximately the double of this number. This means that our data sets have approximately 25000, 50000, 75000, and 100000 moving objects. Second, we can see how big the data sets are, going up to 10 million trajectory entries for the segment oriented model and up to 3 and 2 million entries for the edge and the route oriented model, respectively.

Finally, the most important aspect is the ratio of the average object trajectory sizes between the edge and the route oriented model against the segment oriented model. This ratio goes from approximately 3.5 up to 6 for the edge oriented model and from 5 up to 15 for the route oriented model. This means that, at the maximum values of this ratio, we need 6 and 15 times less entries when using the edge and the route oriented model, respectively.

This ratio is not constant because the generator has an important idea proposed in Statement 4 of [2]: "The number of moving objects will influence the speed of the objects if a threshold is exceeded". This means that if there are lots of objects in the same segment,

than the maximum speed of the segment is decreased, which produces a deceleration of the objects moving on it. This speed changes produces more entries for the edge and route oriented models. Actually, we performed some tests removing this property and the ratio became constant. We can state that the more the objects move with constant speed the better the gain of using the edge and model oriented models. A more detailed study of these ratios and their properties is left to future work.

5.5 Disk Page Size

In this section we want to decide the optimal disk page size that will be use on the rest of the experimental evaluation. For this purpose the index generation time (and necessary disk accesses) is measured and the most expensive set of queries is evaluated.

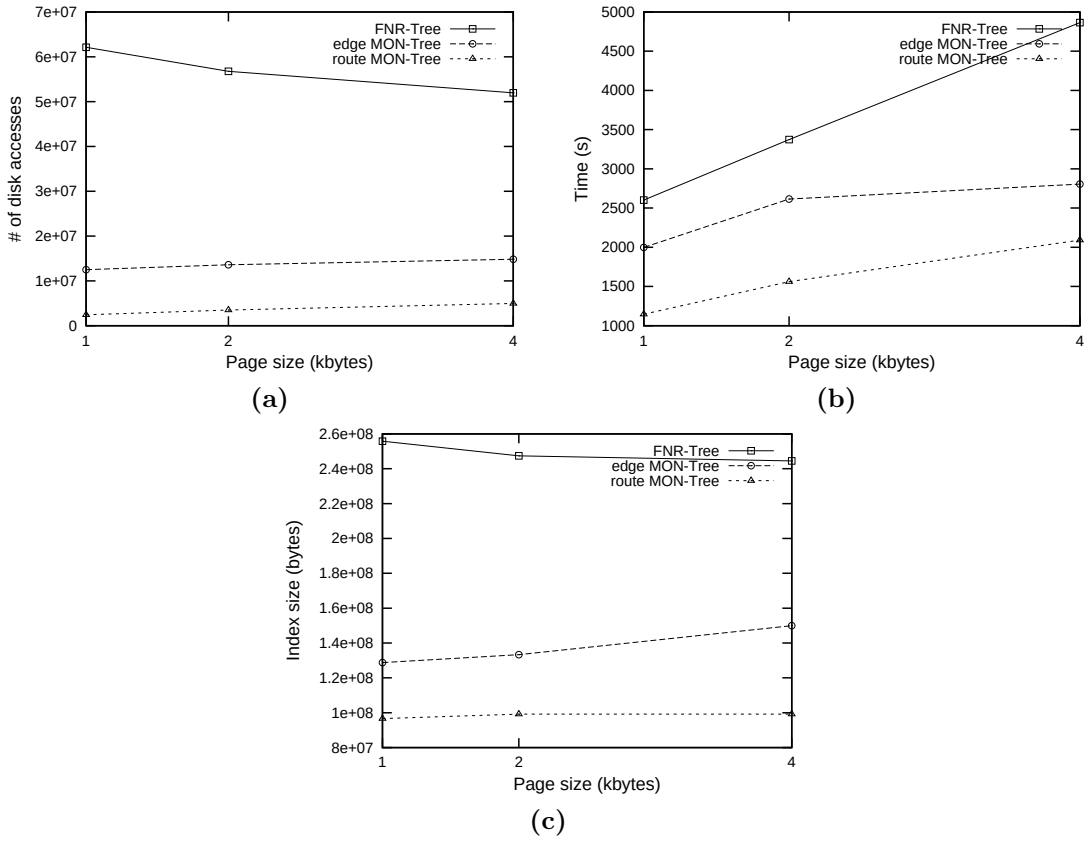

Figure 5: The influence of the page size on index construction considering: (a) the number of disk accesses, (b) the index construction time, and (c) the size of the resulting indexes.

Figure 5 shows the number of disk accesses necessary to create the indexes and the total time spend given the disk page sizes of 1k, 2k, and 4k bytes. It is only showed here the data set generation with the number of objects and speed with their maximum values (50k of objects and 250 as the maximum speed divisor).

First, it is important to note that all index structures have almost constant space utilization (Figure 5(c)). As expected, the FNR-Tree uses much more space than the MON-Trees, in which the route oriented one shows the best results.

Second, we can observe that the number of disk accesses decreases in the FNR-Tree and has a sligt increase in the MON-Trees with the increase of the page size. This behavior can be explained by a good cache locality obtained by the very simple process of insertion on the bottom 1D R-Trees in the FNR-Tree. A fixed cache size of approximately 1 Mbytes for

all index creations is set, which means that the higher the page size, the lower the number of pages in the cache. In fact, all indexes have a lower number of disk page requests, but the number of the real page accesses considering the cache is increased in the MON-Tree.

The time spent in the index construction is always increased with the increase of the page size for all index structures, which is a well known behavior of R-Trees.

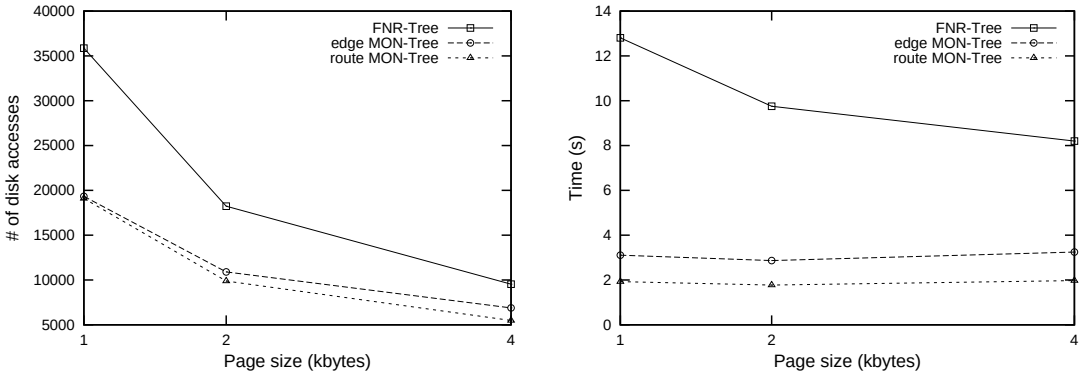

Figure 6: The influence of the page size on query processing considering: (a) and (c) the number of disk accesses and (b) and (d) the index construction time. Parts (a) and (b) corresponds to a query with large window and (c) and (d) with a smaller one.

We should now have a look on query processing for deciding the best disk page size. Figure 6 shows the average number of disk accesses (a) and time spent (b) for the largest set of queries: 50% of the total time and 20% of the total space.

The number of disk accesses decreases for all index structures with the increase of the page size in processing this set of queries (Figure 6(a)). When considering the time spent, the MON-Trees keep almost constant with a small increase for the page size of 4k bytes, but the FNR-Tree still keeps the behavior of decreasing the time spent with the increase of the page size. But, this decrease is not so big from the page size of 2k bytes to 4k bytes, and we expect it to increase if we use page sizes of 8k bytes.

Following these observations, we decide to use the page size of 2k bytes in the rest of this experimental evaluation. With this choice, we will keep a good performance in query processing without sacrificing the index creation (or insertions and updates) for all index structures.

5.6 Index Generation

In this section we show the influence of the number of objects (Figure 7) and their trajectory sizes (Figure 8) in the index construction. When analyzing one of the variables, we set the other variables to their maximum values.

First, the figures show that the MON-Trees clearly outperform the FNR-Tree. Second, all index structures have a linear response to the increase of the number of objects. This is not true for the increase of the maximum speed divisor, but one should note that the trajectory sizes do not have a linear growth with the increase of the maximum speed divisor for the route and edge oriented models. Table 3 shows the trajectory sizes for the data sets using the segment, the edge, and the route oriented models with 50k of initial inserted objects. We can see in this table that the trajectory sizes have a linear growth only in the segment model.

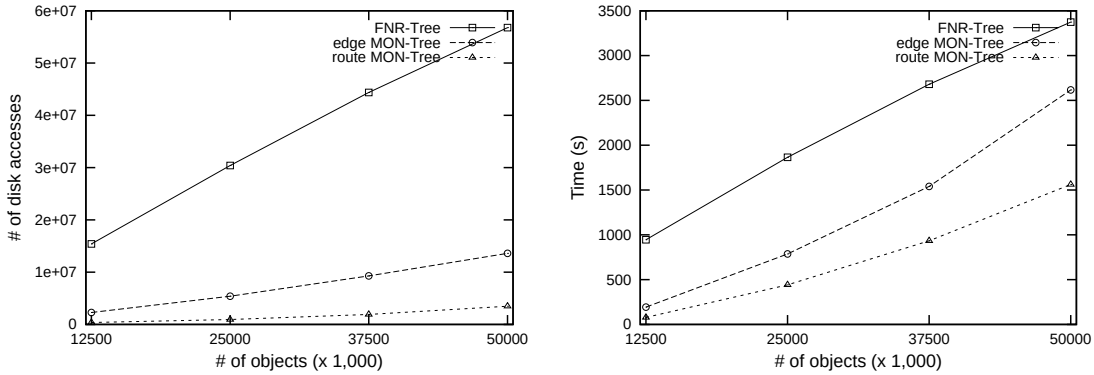

Figure 7: The influence of the number of objects on index construction considering: (a) the number of disk accesses and (b) the time spent.

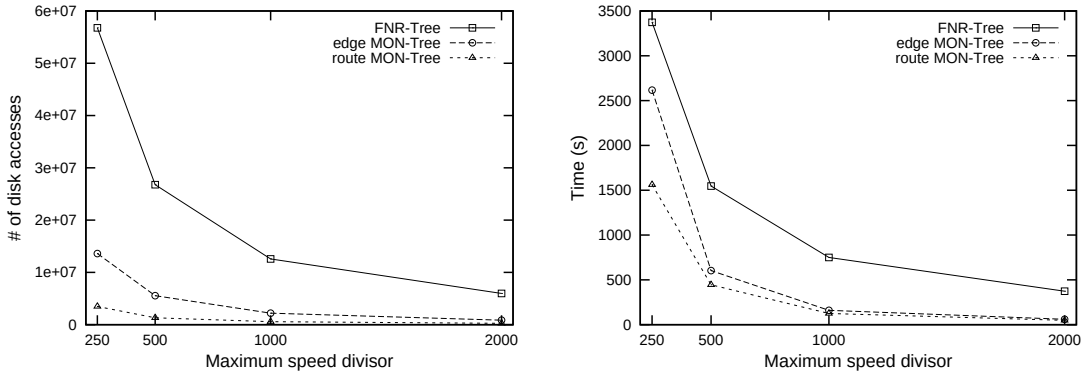

Figure 8: The influence of the trajectory size on index construction considering: (a) the number of disk accesses and (b) the time spent.

Table 3: Trajectory sizes of the moving objects for 50k insertions.

	Average object trajectory size		
Maximum speed divisor	segment	edge	route
250	98.7113	34.4006	29.5907
500	72.6492	20.9353	17.0098
1000	47.1625	11.3796	8.6367
2000	27.124	5.93257	4.35203

5.7 Queries

In this section, we analyze the influence of two variables in the index construction, namely the number of objects and the trajectory size, and the two variables in query processing, which are the size of the query window and the size of the time interval. As we did for the index construction time analysis, we set the other variables to their maximum values, when trying to analyze the behavior of one of them. We also compare the performance of the indexes by the number of disk accesses and the time needed for executing the queries. 100 random queries were executed for every combination of the variables, and then the

results were taken as the average number of disk accesses and time spent for one query execution.

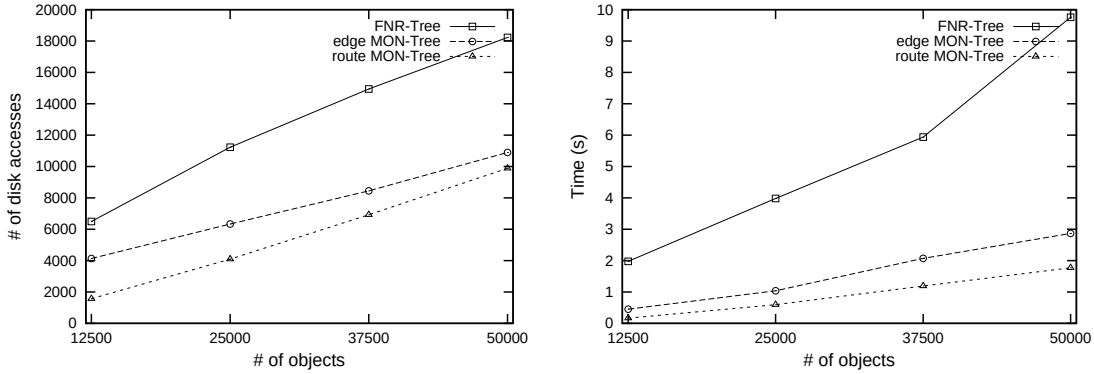

Figure 9: The influence of the number of objects in query performance considering: (a) the query result size, (b) the number of disk accesses, and (c) the time spent.

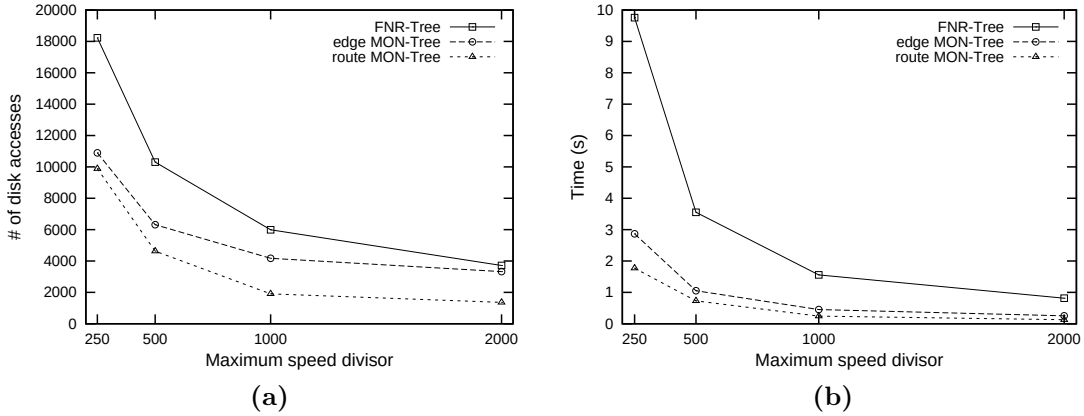

Figure 10: The influence of the trajectory sizes in query performance considering: (a) the query result size, (b) the number of disk accesses, and (c) the time spent.

Figures 9 and 10 show the influence of the number of objects and the trajectory size in query performance, respectively. In all tests, the MON-Trees outperform the FNR-Tree. As seem when examining the index creation time, all structures show a linear behavior with respect to the increase of the number of objects but not with respect to the increase of the maximum speed divisor. We can also see that the *route* MON-Tree query execution performs better than the *edge* MON-Tree in all tests. Another important observation is that when the trajectory sizes gets smaller, the FNR-Tree performance gets closer to the MON-Tree. This is an expected behavior, because the benefits of using the edge and the route oriented models are not so big when the trajectory sizes are small (see Section 5.4).

The most important experiments are when the time interval and the window query sizes are varied. Figures 11 and 12 show the results found according to these variables. Again, we can observe that both index structures have linear behavior with respect to the increase of these two variables. This is very important and desirable for index structures. Then, we can see that again the MON-Trees outperform the FNR-Tree, and are more robust to the increase of these two variables. The only test that the FNR-Tree has a comparable result is for a very small window query size. Finally, we can say that the *route* MON-Tree showed the best overall results.

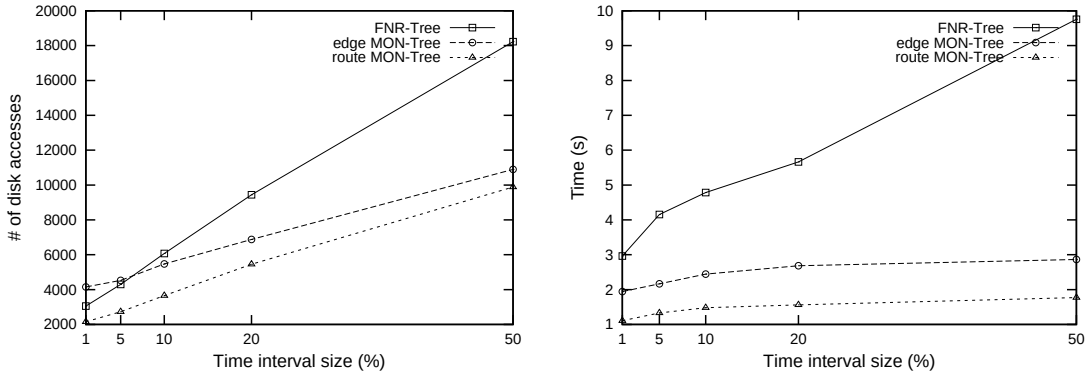

Figure 11: The influence of the query time interval size in query performance considering: (a) the number of disk accesses and (b) the time spent.

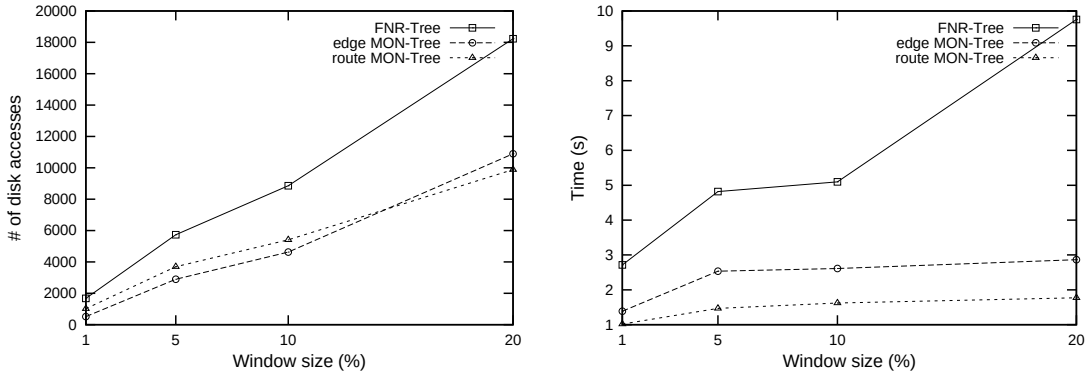

Figure 12: The influence of the query window size in query performance considering: (a) the number of disk accesses and (b) the time spent.

6 Conclusions and Future Work

In this paper we proposed a new index structure for moving objects on networks, the MON-Tree. The MON-Tree stores the complete trajectories of the objects moving in networks. There are two network models that can be indexed by the MON-Tree: the edge oriented and the route oriented models. The MON-Tree is capable of answering two kinds of queries: the range query and the window query, both on past states of the data.

We have experimentally evaluated our proposed index structure against the FNR-Tree, another index structure capable of indexing the trajectories of moving objects in networks. We used generated data sets over a real network: the roads of Germany. In our tests, the MON-Trees outperformed the FNR-Tree, and the MON-Tree indexing the route oriented network model showed the overall best results.

We plan, as a future work, to adapt the structure of the MON-Tree to keep the network connectivity information, using the recent ideas in [17]. In this way, proximity queries can be answered, e.g. the nearest neighbor query. The edge oriented model must be used to achieve this goal.

Acknowledgments

The authors would like to thank Prof. Dr. Thomas Brinkhoff for providing the network-based data generator and especially for providing some direct support. We would also like to thank Dirk Ansorge for his careful review and important comments on the paper.

References

[1] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 322–331, 1990.

[2] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

[3] H. D. Chon, D. Agrawal, and A. E. Abbadi. Using space-time grid for efficient management of moving objects. In *2nd ACM Intl. Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*, pages 59–65, 2001.

[4] H. D. Chon, D. Agrawal, and A. E. Abbadi. Query processing for moving objects with space-time grid storage model. In *Proc. of the 3rd Intl. Conf. on Mobile Data Management (MDM)*, pages 121–, 2002.

[5] J. A. Cotelo Lema, L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. Algorithms for moving objects databases. *The Computer Journal*, 46(6):680–712, 2003.

[6] V. T. de Almeida and R. H. Güting. Indexing the trajectories of moving objects in networks. In *Proc. of the 16th Intl. Conf. on Scientific and Statistical Database Management (SSDBM)*, 2004.

[7] M. Erwig, R. H. Güting, M. Schneider, and M. Vazirgiannis. Spatio-temporal data types: An approach to modeling and querying moving objects in databases. *GeoInformatica*, 3(3):269–296, 1999.

[8] L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. A data model and data structures for moving objects databases. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, volume 29(2), pages 319–330, 2000.

[9] E. Frentzos. Indexing objects moving on fixed networks. In *Proc. of the 8th Intl. Symp. on Spatial and Temporal Databases (SSTD)*, pages 289–305, 2003.

[10] R. H. Güting, M. H. Böhlen, M. Erwig, C. S. Jensen, N. A. Lorentzos, M. Schneider, and M. Vazirgiannis. A foundation for representing and querying moving objects. *ACM Transactions on Database Systems (TODS)*, 25(1):1–42, 2000.

[11] R. H. Güting, V. T. de Almeida, and Z. Ding. Modeling and querying moving objects in networks. Technical Report 308, Fernuniversität Hagen, Fachbereich Informatik, 2004.

[12] C. Hage, C. S. Jensen, T. B. Pedersen, L. Speicys, and I. Timko. Integrated data management for mobile services in the real world. In *Proc. of 21th Intl. Conf. on Very Large Data Bases (VLDB)*, pages 1019–1030, 2003.

[13] C. S. Jensen and D. Pfoser. Indexing of network constrained moving objects. In *Proc. of the 11th Intl. Symp. on Advances in Geographic Information Systems (ACM-GIS)*, 2003.

[14] I. Kamel and C. Faloutsos. On packing R-trees. In *Proc. of the 2nd Intl. Conf. on Information and Knowledge Management (CIKM)*, pages 490–499, 1993.

[15] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proc. of ACM Symp. on Principles of Database Systems (PODS)*, pages 261–272, 1999.

[16] S. T. Leutenegger and Mario A. Lopez. The effect of buffering on the performance of r-trees. *Knowledge and Data Engineering*, 12(1):33–44, 2000.

[17] D. Papadias, J. Zhang, Nikos Mamoulis, and Yufei Tao. Query processing in spatial network databases. In *Proc. of 29th Intl. Conf. on Very Large Data Bases (VLDB)*, pages 802–813, 2003.

[18] A. Papadopoulos and Y. Manolopoulos. Multiple range query optimization in spatial databases. In *Proc. of the 2nd East European Symp. on Advances in Databases and Information Systems (ADBIS)*, pages 71–82, 1998.

[19] D. Papadopoulos, G. Kollios, D. Gunopulos, and V. J. Tsotras. Indexing mobile objects on the plane. In *Proc. of the 13th Intl. Workshop on Database and Expert Systems Applications (DEXA)*, pages 693–697, 2002.

[20] D. Pfoser and C. S. Jensen. Capturing the uncertainty of moving-object representations. In *Proc. of Advances in Spatial Databases, 6th Intl. Symp. (SSD)*, pages 111–132, 1999.

[21] D. Pfoser and C. S. Jensen. Querying the trajectories of on-line mobile objects. In *Proc. of the 2nd ACM Intl. Workshop on Data Engineering for Wireless and Mobile Access (MobiDE)*, pages 66–73, 2001.

[22] D. Pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *Proc. of 26th Intl. Conf. on Very Large Data Bases (VLDB)*, pages 395–406, 2000.

[23] C. M. Procopiuc, P. K. Agarwal, and S. Har-Peled. STAR-tree: An efficient self-adjusting index for moving objects. In *Algorithm Engineering and Experiments, 4th Intl. Workshop (ALENEX)*, pages 178–193, 2002.

[24] S. Saltenis and C. S. Jensen. Indexing of moving objects for location-based services. In *Proc. of the 18th Intl. Conf. on Data Engineering*, pages 463–472, 2002.

[25] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proc. of the SIGMOD Intl. Conf. on Management of Data*, pages 331–342, 2000.

[26] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Querying the uncertain position of moving objects. In *Temporal Databases: Research and Practice*, volume 1399, pages 310–337. LNCS, 1998.

[27] Z. Song and N. Roussopoulos. Hashing moving objects. In *Proc. of the 2nd Intl. Conf. on Mobile Data Management (MDM)*, pages 161–172, 2001.

[28] Z. Song and N. Roussopoulos. SEB-tree: An approach to index continuously moving objects. In *Proc. of the 4th Intl. Conf. on Mobile Data Management (MDM)*, pages 340–344, 2003.

[29] J. Tayeb, Ö. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998.

[30] Y. Theodoridis, T. K. Sellis, A. Papadopoulos, and Y. Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. In *Proc. of the 10th Intl. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 123–132, 1998.

[31] G. Trajcevski, O. Wolfson, F. Zhang, and S. Chamberlain. The geometry of uncertainty in moving objects databases. In *Proc. of the 8th Intl. Conf. on Extending Database Technology (EDBT)*, pages 233–250, 2002.

[32] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and imprecision in modeling the position of moving objects. In *Proceedings of the 14th Intl. Conf. on Data Engineering*, pages 588–596, 1998.

[33] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and querying databases that track mobile units. *Distributed and Parallel Databases*, 7(3):257–387, 1999.

[34] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Proc. of the 10th Intl. Conf. on Scientific and Statistical Database Management (SSDBM)*, pages 111–122, 1998.

Appendix A Revising the Index Structure

The index structure of the MON-Tree indexes the polylines' MBBs in the top R-Tree for both edge and route oriented models. In the route oriented model, the polylines tend to be bigger and the amount of dead space in their MBB representation tends to grow. This is a known problem of indexing polylines in R-Trees. A solution to this problem would be to index the MBBs of every polylines' line segment. We argue that there is a tradeoff between the two representations. On the one hand, indexing the polylines' MBBs the R-Tree creates less leaf entries in the tree and consequently a smaller tree height, but produces more dead space inside of the MBBs and consequently has more overlapping area. On the other hand, by indexing the line segments of the polylines, one can diminishes the overlapping area, but the height of the tree can get bigger and a duplicate elimination mechanism must be added.

It is not clear for the authors of this paper which strategy is the best one, because it depends on the network data sets. We propose then a small change in the index structure to support indexing the polylines' line segments instead of the whole polyline. The revised index structure of the example in the Figure 1 (b) can be seen in Figure A-1. The top R-Tree indexes line segments' MBBs and all leaf nodes of the same polyline point to the same bottom R-Tree. We call this index structure MON-Tree II.

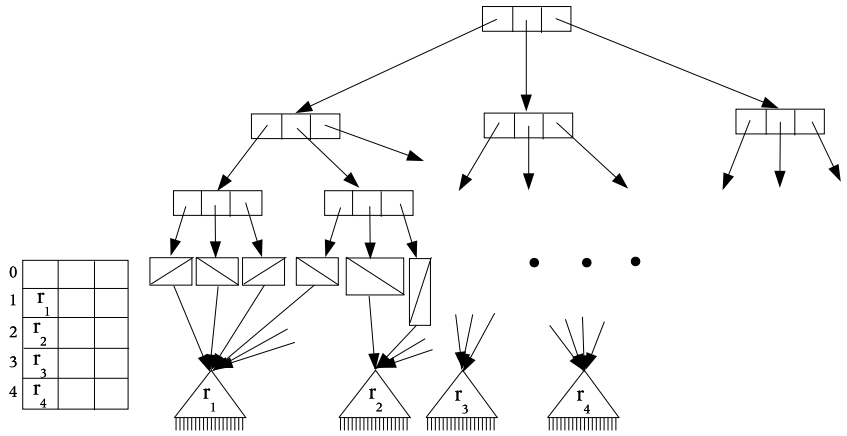

Figure A-1: Example of the revised index structure of the network in Figure 1 (b).

The algorithms must be changed to support this new index structure. The polyline insertion algorithm remains the same, because the real insertion of the polyline in the top R-Tree is postponed to the insertion of the first object moving on it. The moving object insertion is quite the same too, except for the polyline insertion in the top R-Tree. Instead of inserting the whole polyline's MBB, all line segments' MBBs are inserted in the top R-Tree, pointing all to the same bottom R-Tree and to the real representation of the polyline. The movements insertion remains the same, since they do not use the top R-tree.

The search algorithm must remove the duplicates before searching the bottom R-Trees. For this purpose, a main memory hash structure containing the polylines identifications is used to avoid searching the bottom R-Trees more than once. In this way, in the first attempt to open a bottom R-Tree, the polyline identification is stored in this main memory hash structure, and every time a leaf node in the top R-Tree is reached, this main memory hash structure is searched to see if its corresponding bottom R-Tree was already searched.

In the following we make an analysis of these two index representations using the

network data sets of our experimental evaluation. The improvement of this approach can bring is two-fold: first, some disk accesses in the top level R-Tree search can be avoided. Second, it is expected that less false hits are returned in this search, which means that less polylines that do not really intersects the spatial query windows are returned. Some disk accesses (and CPU time) are also avoided to compute the real intersection (and clipping) algorithm for these polylines.

It is very important to remark that, with this new index structure, the representation of the objects' movements according to both edge and route oriented network models are kept unchanged. Only the top R-Tree index structure is changed. Even when storing line segments' MBBs in the top R-Tree, the good properties of the MON-Tree are kept. We can see the queries as having two phases, one for searching the top R-Tree to find the polylines of edges/routes that intersects the query window and the second to find the moving objects whose movement intersects the query window. It is expected, independent of the model used, that the second phase is the most critical in terms of time consumption, because it is also expected that the number of objects trajectory entries in the bottom level is much bigger than the number of network piece entries. In this way, the refinements proposed here in the MON-Tree structure must lead to a very significant gain in the first query phase to represent an important improvement in the overall query processing.

First, to show the tradeoff between the index representations in the top level R-Tree, let us take the equation proposed in [14]:

$$P(q_x, q_y) = TotalArea + q_x \times L_y + q_y \times L_x + N \times q_x \times q_y \quad \text{(A-1)}$$

where q_y and q_y are the sizes of the rectangular query $r = q_x \times q_y$; N is the number of nodes in the tree; $TotalArea$ is the sum of the areas of all nodes in the tree; L_x and L_y are the sums of x and y extents of all nodes in the tree; and $P(q_x, q_y)$ is the number of accesses expected for the range query r.

In this equation we can see that minimizing the total area of the nodes is important, but it is also important to minimize their perimeter and their number. The number of disk accesses also increases with the size of the query window.

Table A-1 shows the values of the parameters of the equation for the *edge* and *route* models using both index structures. As additional information, we added the height of the top R-Tree.

Table A-1: Parameters of the equation A-1

	edge MON-Tree	edge MON-Tree II	route MON-Tree	route MON-Tree II
$Height$	1	2	1	2
N	83	627	22	634
$TotalArea$	1.86368	2.75194	2.95433	2.67016
L_x	9.77845	27.1357	7.37443	25.3895
L_y	8.43794	24.7034	6.88296	24.94

It is expected that, with the network data sets used in this experimental evaluation, this approach of dividing the polylines into line segments in the top R-Tree will not bring any improvement in the top level R-Tree search, since all the values of the equation A-1 are bigger for the line segments division approach. This is true even for the $TotalArea$, except for a very small difference in the route oriented model. Dividing a polyline into line

segments can decrease the total area of the leaf entries, but it increases a lot the number of nodes in the tree.

Unfortunately, we cannot make such analisys for the number of disk accesses (and CPU time) avoided with the decrease of the false hits in the top level R-Tree search.

In order to find out which is the best index structure, we did all the experiments shown in the paper including the MON-Tree II. To save space, we show only part of these results. Figure A-2 is the corresponding plot of Figure 7. It can be seen in this figure that the number of disk accesses of both versions of the MON-Tree are almost the same and there is only a small difference in the time spent for the index creation.

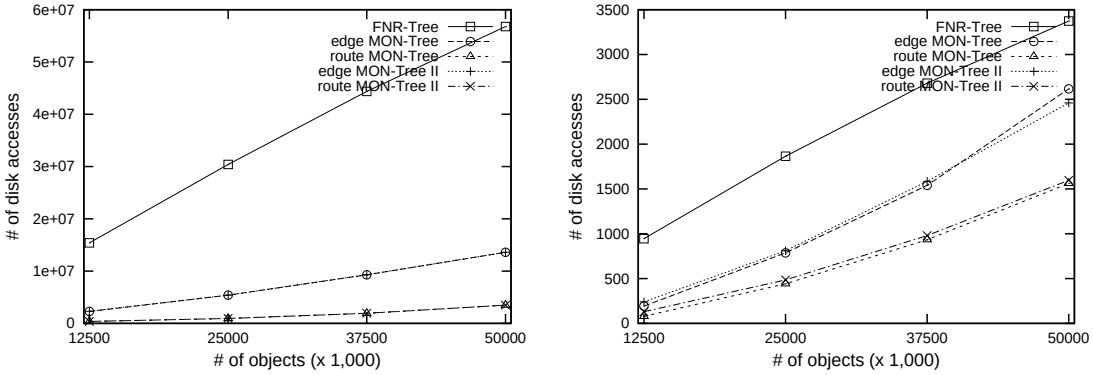

Figure A-2: The influence of the number of objects on index construction considering: (a) the number of disk accesses and (b) the time spent.

Figure A-3 is the corresponding plot of Figure 11. The same behavior can be seen in this figure. In fact, all the other figures from the experimental evaluation of this paper showed the same behavior, which means that there is no gain on choosing the complementary index structure proposed in this appendix.

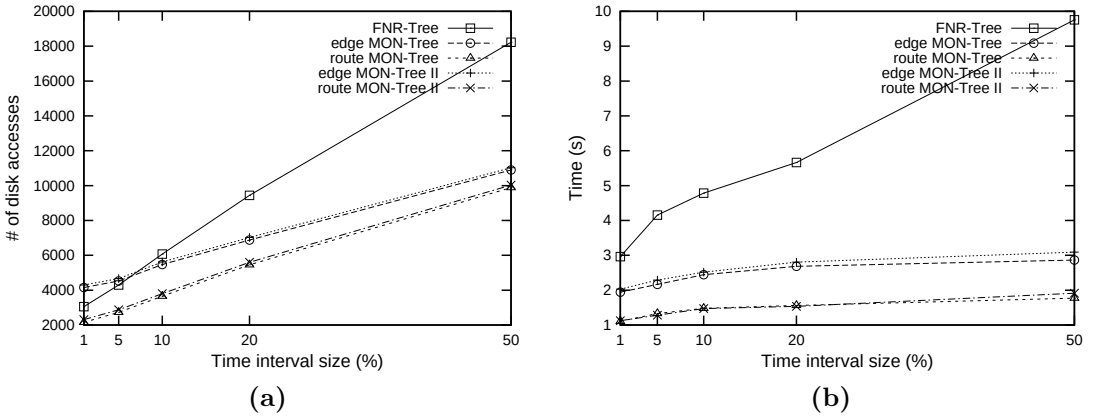

Figure A-3: The influence of the query time interval size in query performance considering: (a) the number of disk accesses and (b) the time spent.