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. The 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 is composed by edges and nodes and each edge has an associated polyline. The second one is more suitable for transportation networks and is route oriented, i.e., the network is composed by routes and junctions. A route has also an associated polyline as attribute. We propose the index in terms of the basic algorithms for insertion and querying and test our proposal in an experimental evaluation with generated data sets using as underlying networks the roads and railroads of Germany. The MON-Tree showed good scalability when increasing the number of objects and time units in the index structure, and the query window and time interval in querying. In our tests, the MON-Tree that indexes the route oriented model showed the best results.

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 works in the literature that try to model this problem. The first one in [EGSV99, GBE+00, FGNS00, CFG+03] is a complete framework that serves as a precise and conceptually clean foundation for the representation and querying of spatio-temporal data. This work presents 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 in such a way 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.

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

The other one in [SWCD98, WCD+98, WXCJ98, WSCY99] proposes the Moving Objects Spatio-Temporal (MOST) model and the Future Temporal Logic (FTL) language for querying the current and future locations of moving objects. Their model handles only 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 [TWZC02] 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 ([BKSS90]). Following these ideas, for moving objects databases, which is a spatio-temporal application, and consequently more complex, the need of good indexing techniques is even more important.

As stated in [PJ99], 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, while the second one is the indexing of current and anticipated future positions of moving objects. This can be seen as indexing the two main models presented before. Most of the papers in the literature proposing index structures for moving objects, inclusive [PJ99], 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., [PJT00, SR01, SR03, CAA01, CAA02]. For querying current and future positions of the moving objects, we can cite [TUW98, SJLL00, SJ02, PAH02, XP03, KGT99, PKGT02].

Most of these works for indexing moving objects assume free movement of the objects in space. As stated in [PJ01], 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, typically, 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 effectively occurs in a different space than for the first two scenarios. In [KGT99], this kind of movement is called 1.5 dimensional.

For the constrained movement scenario, a two-step query processing is proposed in [PJ01]. 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 passed then to the second step where a modified version of the R-Tree and the TB-Tree are queried using an approach proposed in [PM98].

In the context of Spatial Network Databases, the work in [PZMT03] 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 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 sub-problem is to index the network data and

[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.

the second one is to index the moving objects.

The first one, the Fixed Network R-Tree (FNR-Tree), proposed in [Fre03], has an index structure that consists of a top level 2D R-Tree whose leaf entries 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 edge of the network, i.e., a leaf entry in the 2D R-Tree, there is an 1D R-Tree indexing the time interval of the objects traversing it. The main disadvantage of this approach is the model of network 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, because distinct entries are needed for every line segment the object traverses. Another problem of the FNR-Tree is that since in the 1D R-Tree only time intervals are stored, it is assumed that the objects' movements always begin and end in nodes. An object cannot end its movement in the middle of an edge, for example. It also cannot change its speed or direction in the middle of an edge, only in the nodes.

The second one proposed in [JP03] stores the network edges also as line segments in a 2D R-Tree and the moving objects into another 2D R-Tree. The difference between both index structures is in the moving objects indexing. 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 work shows the same disadvantage of the FNR-Tree that the model of network indexed leads to a high number of entries in the index structure. An additional disadvantage occurs in the query processing. While in the FNR-Tree, there is a bottom 1D R-Tree for each edge to index the objects' movements on it, here there is a global 2D R-Tree for the movements, which makes the second phase of the query longer. 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 an edge. In this way, 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 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 is composed by edges and nodes and each edge has an associated polyline. This model has been extensively used, e.g. in [PZMT03], 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 [GAD04], where we extended the ADT approach in [GBE+00] for network constrained movements. In this route oriented model the network is composed by routes and a set of junctions between these routes.

The MON-Tree is capable of answering two kinds of query on past states of the database, namely *range query* and *window query*. Both queries 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 returns only the pieces of the objects' trajectory that overlap the query window.

The MON-Tree keeps the good properties of both index structures in [Fre03] and [JP03] and solves their main disadvantages. We test our proposal in an experimental evaluation with generated data sets using as underlying networks the roads and railroads of Germany.

The structure of this paper is as follows: Section 2 reviews in more detail the index structure and query processing of the FNR-Tree proposed in [Fre03]. Section 4 proposes the MON-Tree index structure and the insert and search algorithms. Section 5 experimentally evaluates our

proposed index structure for the two network models and compares them to the corresponding FNR-Tree. Finally, Section 6 concludes the paper and proposes some additional future work.

2 The FNR-Tree

The Fixed Network R-Tree was recently proposed in [Fre03] to solve the problem of indexing objects moving in fixed networks. The structure consists of a top level 2D R-Tree whose leaf entries 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 edge of the network, i.e., a leaf entry in the 2D R-Tree, there is an 1D R-Tree indexing the time interval of the objects traversing it. The main disadvantage of this approach is the model of network 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, because distinct entries are needed for every line segment the object traverses.

Since the spatial objects stored in the 2D R-Tree are line segments, the leaf node entries are in the form $\langle mbb,\ orientation,\ 1drtreepointer\rangle$, where *mbb* is the minimum bounding box (MBB) of the line segment, *orientation* is a flag that describes the two possible orientations that a line segment can assume inside an *mbb*, and $1drtreepointer$ is a pointer to the root node of the associated 1D R-Tree.

Each leaf entry of the 1D R-Tree is in 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 where 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 where the object was moving, its direction, and the time interval when the object traversed the line segment. The insertion 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 1D R-Tree. Instead of using, in the 1D R-Tree the algorithms of the 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 increasing order.

The search algorithm consists of three steps. It receives a spatiotemporal 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 the ones that are fully outside the spatio-temporal query window w are rejected.

Some simple improvements to this algorithm are possible. First, it is not necessary to store the edges in main memory. Since the steps are done in sequence, the edge found in the first step can be directly used in the third step. With this, we can avoid the *edgeid* information in the 1D R-Tree leaf nodes. Second, as the 1D R-Tree stores only time intervals, the direction of the movement is also not necessary. We can re-construct the 3D movement of the object by using only the line segment of the edge and the moving object time interval. The direction of the object does not matter, in this model. With this second modification then, we can avoid the *direction* information in the 1D R-Tree leaf nodes.

The main disadvantage of this indexing approach is the model of network used, where each

edge corresponds to a line segment in the network. This model leads to a high number of entries and lots of updates in the index structure, because distinct entries are needed for every line segment the object traverses. The query performance is also affected, since more 1D R-Trees are queried.

Another problem of the FNR-Tree is that since in the 1D R-Tree the time intervals are stored, it is assumed that the objects' movements always begin and end in nodes. An object cannot, in the FNR-Tree, end its movement in the middle of an edge, for example. It also cannot change its speed or direction in the middle of an edge, only in the nodes. This is solved, in our proposed indexing approach, storing in the bottom trees the objects' positions, instead of only the time interval.

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.

In the first and straightforward model, a network is a graph $G = (N, E)$ where N is a set of nodes and $E = N \times N$ 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_{1e} and n_{2e} 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, $p_1 = p_{n_1}$, and $p_k = p_{n_2}$. A position *epos* 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_{1e} and 1 means that the position lies on the node n_{2e} of the edge. 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 examples of usage of this model, we can cite [PZMT03].

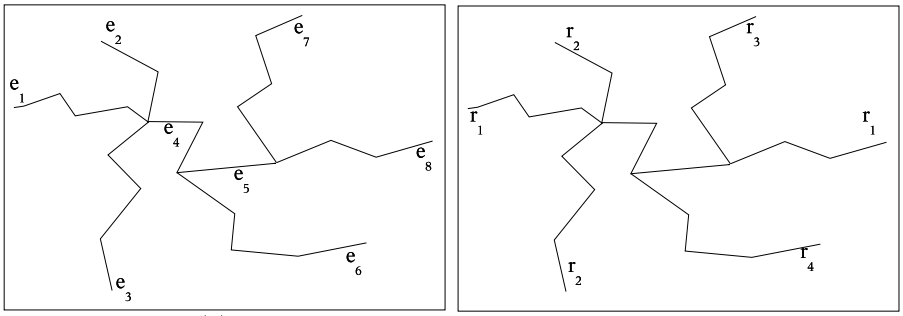


Figure 1: Example of a network (a) in the first model and (b) in the second model.

This first model is simple and straightforward, but not the best one to represent transportation networks. In [GAD04] we extended the framework in [GBE+00] 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 *rpos* 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 $rpos_{r_1}$ and $rpos_{r_2}$. The domain of a moving object position inside a graph G' is $D'(G') = R \times rpos$.

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.

A detailed discussion about why using the route oriented model is given in [GAD04], but the most practical reason to use the route oriented model instead of the edge oriented one, taking indexing techniques into consideration, is 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 going 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 [GAD04]. A similar model is also used in the kilometer-post representation in [HJP+03].

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 belong to edges, for the first network model, or to routes, for the second network model. The index structure is composed by 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 A where a complementary index structure is proposed.

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. The need for these two top level index structures is as follows: on the one hand, the insertion algorithm of 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. On the other hand, 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 $0 \leq p_1, p_2 \leq 1$.

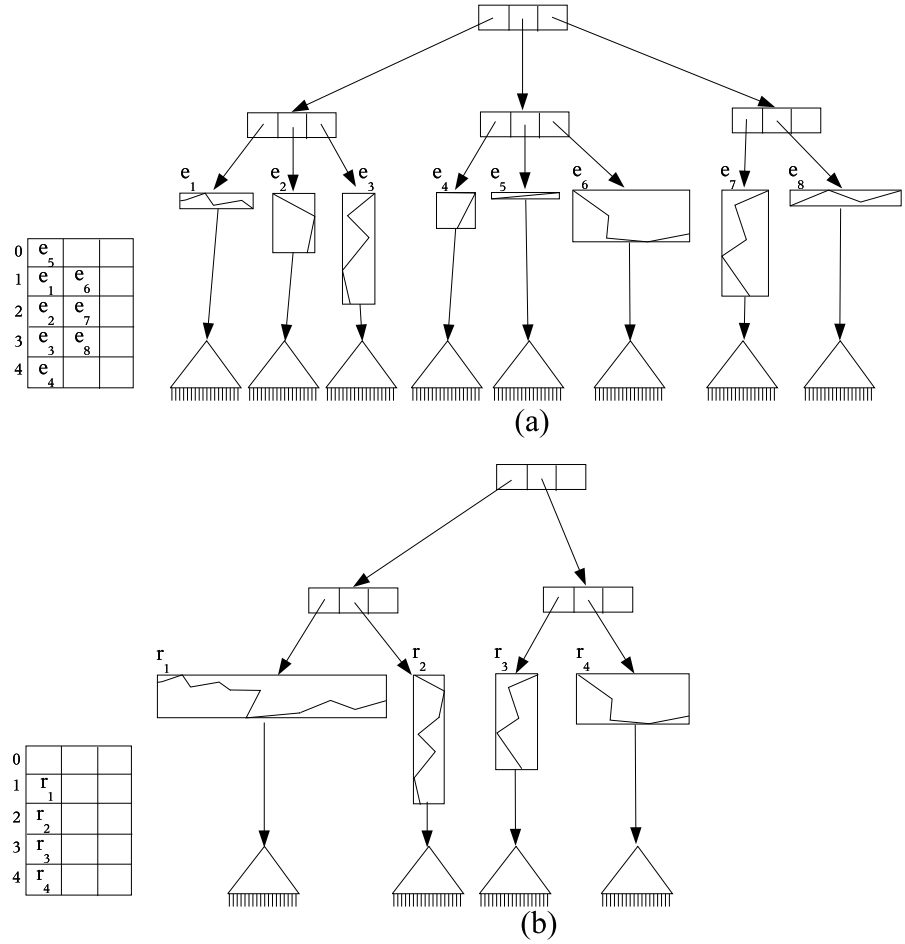


Figure 2: Example of the index structure of the networks in Figure 1.

These two values p_1 and p_2 store the relative position of the objects inside the polyline at times t_1 and t_2 respectively.

4.2 Insertion

In this index structure we allow two different kinds of insertion: polyline insertion and movement insertion. A polyline insertion is needed to construct the basis network. The moving object insertion is necessary 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, while 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 looking in the top hash structure to find 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 insert 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 ([TSPM98]). 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 query spatial window r. Then, in the second step, the intervals where the polyline intersects the spatial query window r are found 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 [PM98] because our problem is much simpler than the one solved there. In [PM98], if the queries are combined (they propose a threshold to decide whether to combine 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 area waste, 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 that we have is as follows: there is a set of query windows w' and a set of entry bounding

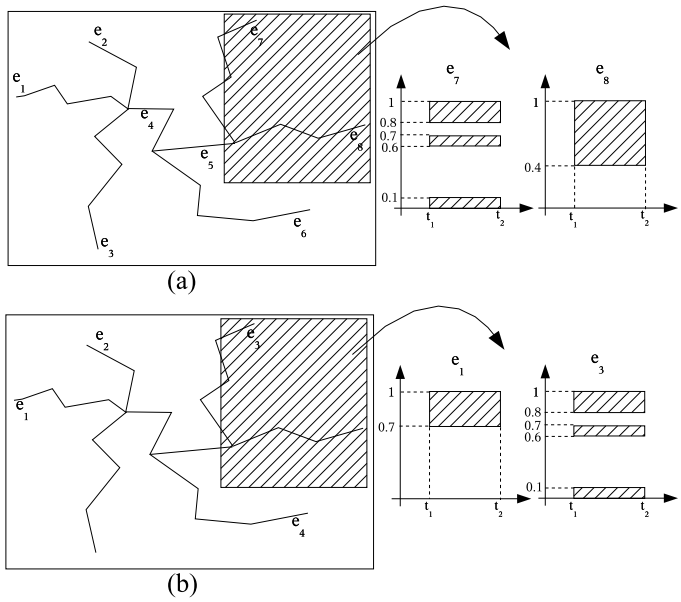


Figure 3: Example of the interval set determination in the search algorithm (a) in the partition into edges without connection; and (b) in the partition into edges containing connections.

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 sweep algorithm in $O(nlogn + k)$ time, where n is the total number of rectangles and k is the number of overlapping pairs. But our real problem is simpler because, 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 seen in the example of the Figure 3. In this way, the idea is to ask for every entry in the node if it intersects the query window w'. This intersection computation can be done in logarithmic time, 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 aditional 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 a main memory structure (an array for example) and after its completion, a duplicate removal is done.

5 Experimental Evaluation

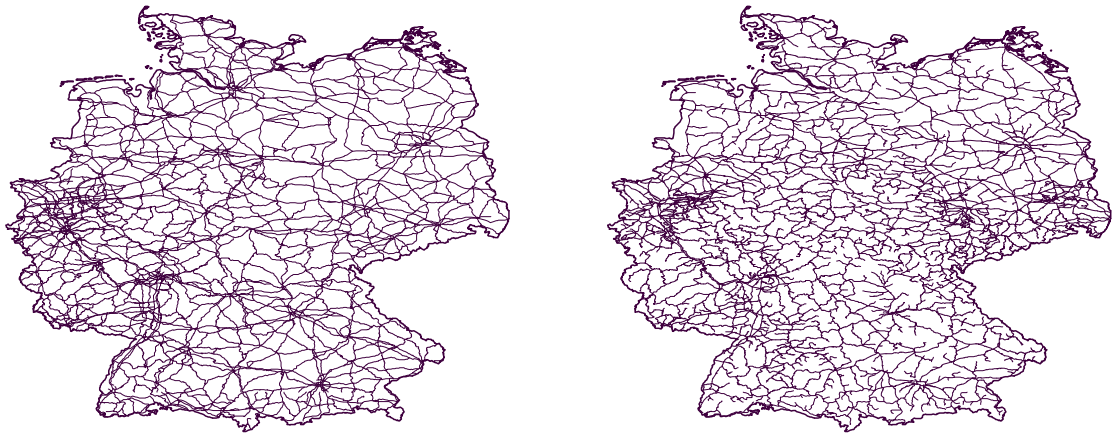
In order to examine the performance of our proposed index structure, the MON-Tree, we did an experimental evaluation. We also implemented the FNR-Tree and compared to our results. We only became aware of the index proposed in [JP03] some time after our implementation and experimental evaluation was done. 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+^{TM}, 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

In all our experiments, we used the network-based moving objects generator proposed in [Bri02] [2]. We used two networks, the roads (*rdline*) and the railroads (*rrline*) of Germany downloaded from the *Geo Community* web site[3] (Figure 4).



(a) **(b)**

Figure 4: The (a) roads and (b) railroads of Germany used as networks in the experiments.

We then adapted the roads (*rdline*) and railroads (*rrline*) data sets to be used by the FNR-Tree and the MON-Tree. For the FNR-Tree, as well as for the input of the data generator, we need to create an edge for each line segment. We call these network data sets *rrline.segment* and *rdline.segment*. For the first model of network described in Section 3 we had to do nothing with the data sets, since they correspond exactly to this model. We call these network data sets *rrline.edge* and *rdline.edge*. For the second model, routes had to be generated combining several connected edges, or paths. There was no additional information on the data sets about routes and so we decided to combine the edges according to their lengths, i.e., edges with big (small) lengths are more likely to be combined with edges with big (small) lengths. These data sets are called *rrline.route* and *rdline.route*. Some statistics about the network data sets can be seen in Table 1.

It can be seen in these statistics that the *rrline* and *rdline* data sets have different behavior. They have almost the same number of line segments, but the *rdline.edge* data set has approximately half the number of edges than the *rrline.edge*, because the size of the edges (in average) in the *rdline.edge* is almost the double of that in the *rrline.edge* data set. The *rrline.route* and *rdline.route* data sets do not seem to be so different.

As stated in the Section 2, the FNR-Tree assumes that the objects' movements always begin and end in nodes, because only a time interval is stored in the 1D R-Tree. 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). Since we compare our proposal with the FNR-Tree, we

[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`

Table 1: Statistics about network data sets.

Data set name	# of segments	# of polylines (edges/routes)	Average polyline size (# of segments)	Maximum polyline size (# of segments)
rrline.segment	36,334	36,334	1	1
rrline.edge	36,334	12,707	2.86	34
rrline.route	36,334	2364	15.36	182
rdline.segment	30,674	30,674	1	1
rdline.edge	30,674	5,342	5.74	58
rdline.route	30,674	1966	15.60	136

changed the generator in order to reflect this limitation. In this way, changes of speed inside line segments (edges for the FNR-Tree model) are simply ignored. The end of movements take effect in the last node in the network the moving object has passed. If an object changes its direction inside a line segment, and comes back to the same point, this movement is also ignored, but the time is accumulated to the next movement it does.

We are interested to show the behavior of the indexes according to these three variables:

- Number of objects. We generated data sets with 2k, 4k, 6k, 8k, and 10k moving objects.
- Number of time units. We generated data sets with 25, 50, 75, and 100 time units (a parameter of the generator, see below).
- 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. [LL00] shows the importance of a cache buffering disk pages. We are interested in varying the cache size only in queries. According to that, in the index generation we used the maximum cache size used for queries, which corresponds to 8M bytes of memory.

Another variable that is not tested is the number of buckets for the top level hash structure of the MON-Tree. We fixed this number to 1021, which is the prime number closest to 1k buckets.

We used the network-based moving objects generator proposed in [Bri02]. An important variable that can be set in the generator is the approach for the generation of the moving objects starting nodes. Before an object can move, it is necessary to determine its start position, and these start positions are assumed to be nodes in the network. In the data-space oriented approach (DSO), a position (x, y) in the space is computed using a 2-dimensional uniform distribution function and then the nearest node of the network is computed. We use this behavior for the generation of moving objects over the *rrline* network, because we would like to achieve a more uniform distribution of objects in the 2-dimensional space. In the region-based approach (RB), regions (cells) with more density of nodes have higher probability of having a starting point. We use this approach for objects generated over the *rdline* network, because we think that cars are less uniformly distributed in the 2-dimensional space than trains. Another important concept of the generator is the existence of external objects. This concept was created to simulate weather conditions or similar events with impact on the motion and speed of the moving objects. We do not use external objects for the generation of objects over the *rrline* network, but we do so for the *rdline* network. One can think that trains have a more constant movement than cars,

even stopping in stations, which can be ignored. Another important concept is the existence of classes of objects, where each object belongs to a class and for each class a maximum speed is defined.

As input for the data generation, 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 which determines the maximum speed of the classes of objects, the number of time units, and the report probability to simulate the situations where the moving objects report their positions irregularly.

For varying the number of objects, we set only the variable for the initial number of objects, i.e., number of objects means the number of initially created objects. No objects are created in subsequent time units. We also set the report probability to the maximum value, which means that objects always report their positions.

5.3 Queries

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 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%, 50%, and 100% 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.
- Size of the cache. We set the size of the cache for the queries to 1M, 2M, 4M, and 8M bytes. 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 also for managing the cache, it is desirable that the number of pages in the cache be a prime number. We then used the maximum prime number that multiplied by the page size is less or equal than the cache size. It is also important to state that we kept the index in opened state until all queries were processed to avoid re-creating the cache for every query.

For each combination of the first two variables, we generated randomly 100 queries. We then ran all queries for each cache size.

5.4 Index Generation

In this section we want to show the influence of the three variables in the index construction: the number of objects, the number of time units, and the page size. In this analysis then, to show the influence of one of the variables, we fix the other two variables to their maximum values. We compare the indexes by the number of disk accesses and the time needed for the construction of the index. It is showed in [LL00] that ignoring buffer behavior and using number of nodes accessed as a performance metric can lead to incorrect conclusions, not only quantitatively, but also qualitatively. In this way, we decided to use disk accesses as a metric to compare the structures.

First, a very important measure that differentiates the indexes is the number of insertions. Table 2 shows the average number of insertions divided by the number of objects and the number of time units. This number shows how many entries (in average) each moving object has in the index for a single time unit. This number affects directly on the size of the index. This table shows the good properties of the MON-Tree that avoids a big number of insertions storing

the polylines instead of line segments. The number of insertions for the *route* MON-Tree, for example, is almost half of the number of insertions for the FNR-Tree. This properties lead to an index construction with less insertions, less entries, and consequently a better performance.

Table 2: Average number of insertions divided by the number of moving objects and by the number of time units.

Data set	FNR-Tree	*edge* MON-Tree	*route* MON-Tree
rrline	0.7055	0.4342	0.3719
rdline	0.5266	0.2939	0.2835

Figures 5, 6, and 7 show the influence of the page size, the number of objects, and the number of time units in index construction, respectively. In Figure 5, we can see that the increase of the page size is indirectly proportional to the index generation performance. In parts (a) and (d) we can see that the number of disk accesses is not so affected by the increase of the page size. On the other hand, in the parts (b) and (e), and (c) and (f) we can see that the index creation performance is affected negatively by the increase of the page size, especially for the FNR-Tree, according to the index construction time and the index size, respectively. Using the results in this figure, we will, on the rest of this experimental analysis, to be fair, use the page size of 1k bytes, as it was done in [Fre03].

Figures 6 and 7 show that the MON-Trees clearly outperform the FNR-Tree. An important fact that one should note in these figures is that the behavior of the *route* and *edge* MON-Trees is swapped between the number of disk accessses and time spent, i.e., the *route* MON-Tree generally needs a smaller number of disk accesses but has a worse performance according to the index construction time. A second fact is that, whereas having a big difference in terms of disk accesses, the FNR-Tree has not a very bad performance, according to the index construction time. The explanation of these strange behaviors is only one: cache locality. Generally, the *route* MON-Tree has less and bigger polylines stored in the top R-Tree than the *edge* MON-Tree, and consequently less bottom R-Trees with bigger heights. Because of this, it achieves a better cache locality. The same occurs when comparing the MON-Trees with the corresponding FNR-Trees.

The Figure 8 shows the behavior of the index structures according to the number of objects. This figure plots the number of requested disk accesses instead of the number of real disk accesses, i.e., the number of disk accesses ignoring the cache usage. We can see in this figure that the number of requested disk accesses is bigger for the *route* MON-Tree than the one for the *edge* MON-Tree, and also that the difference between the FNR-Tree and both MON-Trees is not so big. These observations then enforce the hypothesis stated above, that the *route* MON-Tree has a better cache locality than the *edge* MON-Tree and that the MON-Trees have a better cache locality than the FNR-Tree. If an index structure asks for more pages, but has a good cache locality, it still has to process these pages. That is why they consume more time in these cases. Because of this behavior of the index structures, we decided to print the time consumed in the index generation. Similarly, we will also print the query time in the query analysis.

5.5 Queries

In this section, we want to analyze the influence of two variables in the index construction: the number of objects and the number of time units; and the three variables in the query processing: the size of the query window, the size of the time interval, and the cache size. As we did for the index construction time analysis, we fix 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. Note that we

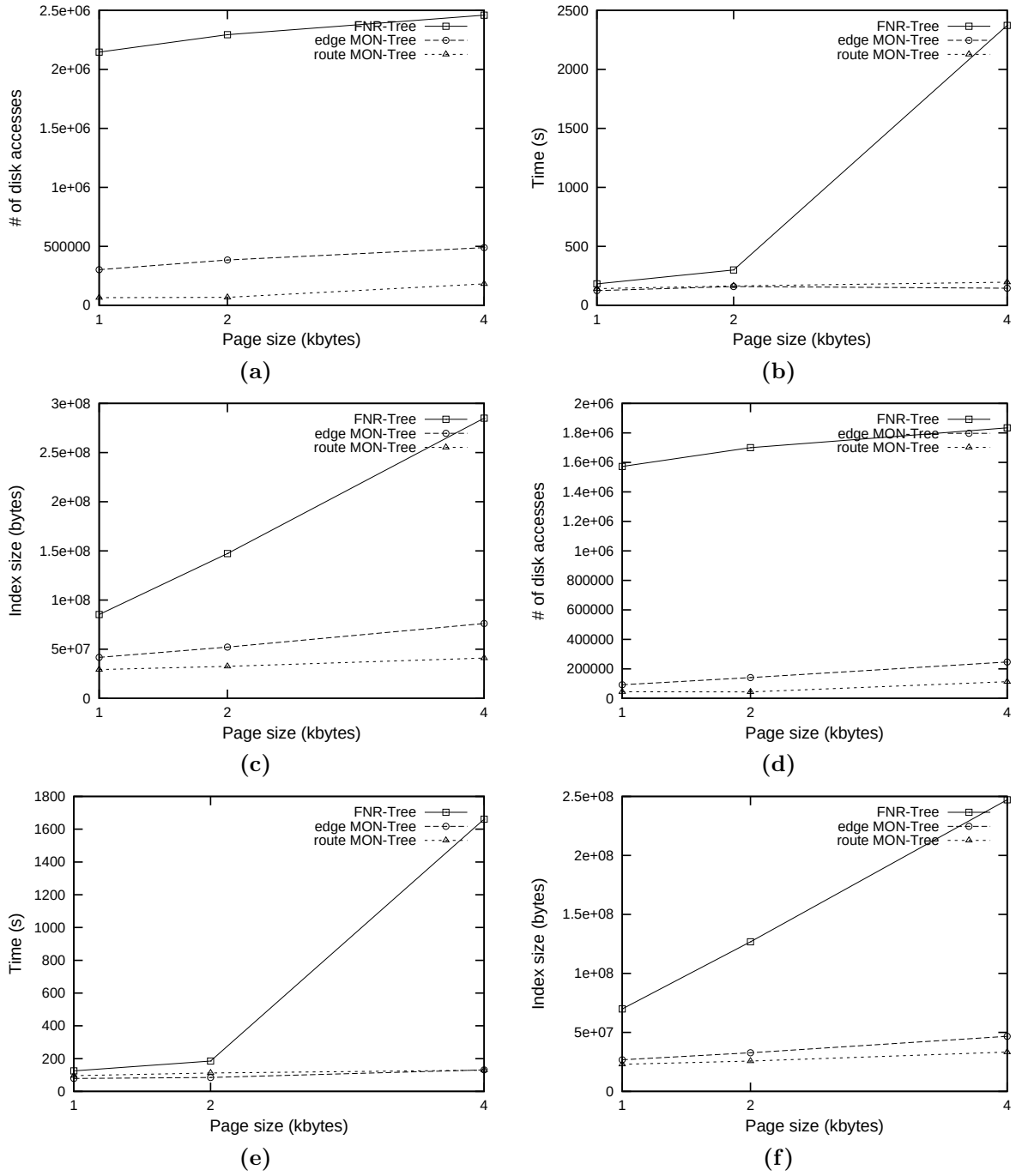


Figure 5: The influence of the page size on index construction considering: (a) and (d) the number of disk accesses; (b) and (e) the index construction time; and (c) and (f) the index size. Parts (a), (b), and (c) represent the *rrline* data set, whereas (d), (e), and (f) represent the *rdline* data set.

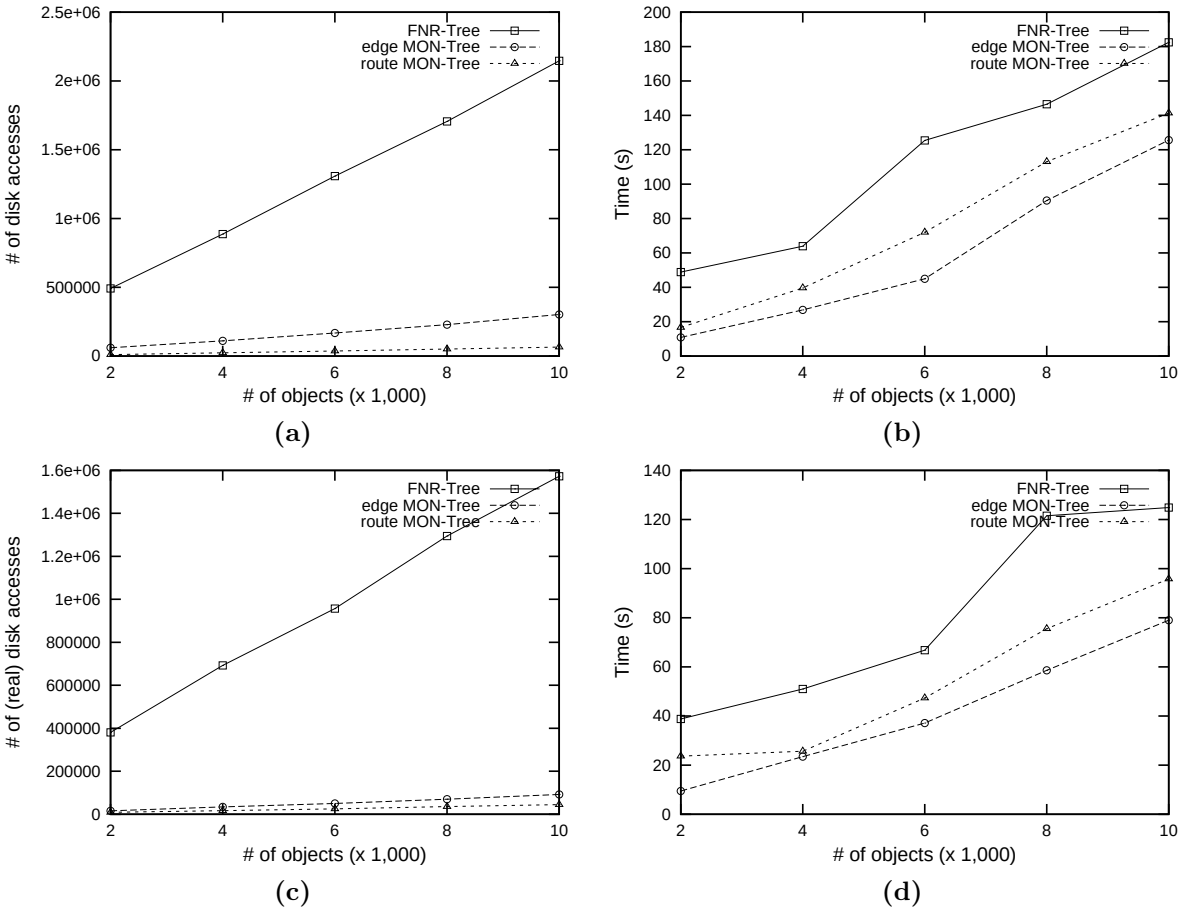


Figure 6: The influence of the number of objects on index construction considering: (a) and (c) the number of disk accesses; (b) and (d) the time. (a) and (b) represents the *rrline* data set, whereas (c) and (d) represents the *rdline* data set.

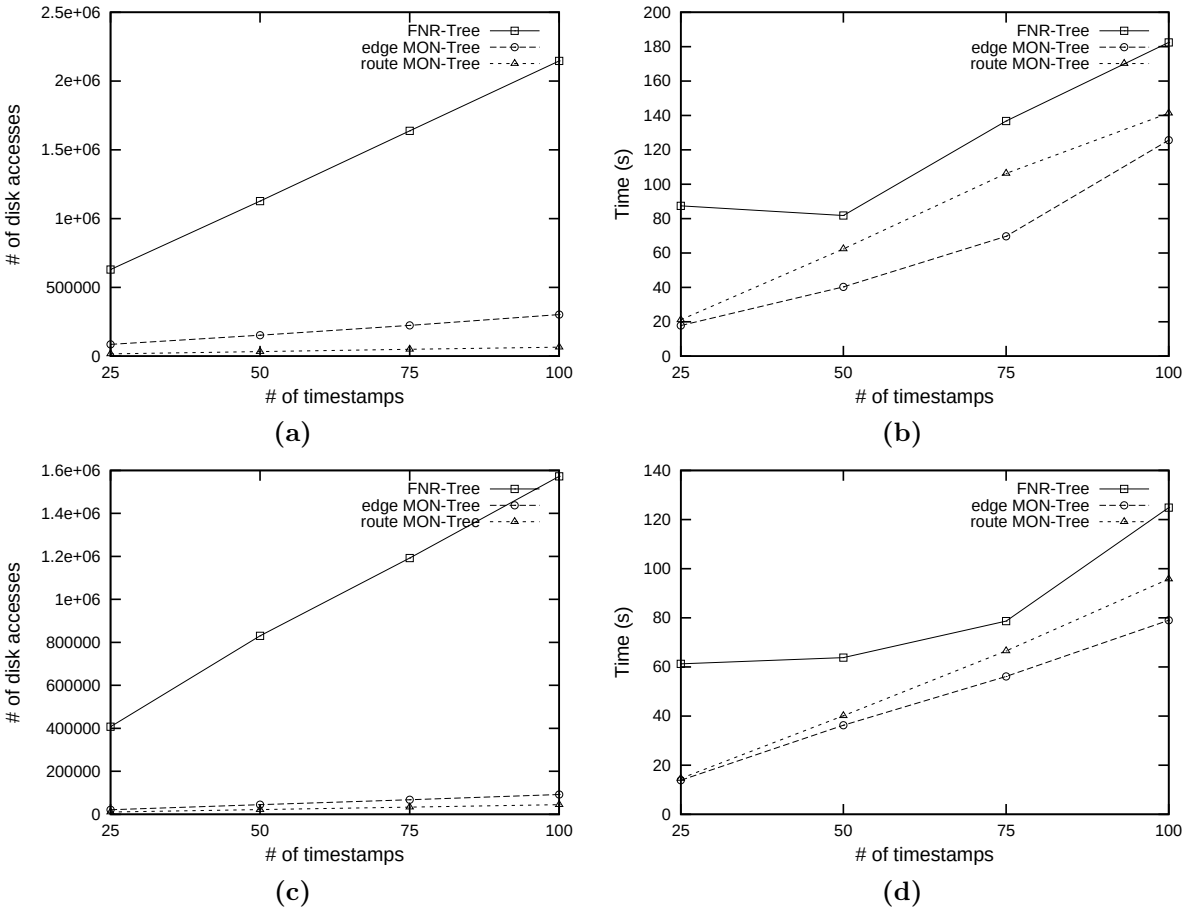


Figure 7: The influence of the number of time units on index construction considering: (a) and (c) the number of disk accesses; (b) and (d) the time. (a) and (b) represents the *rrline* data set, whereas (c) and (d) represents the *rdline* data set.

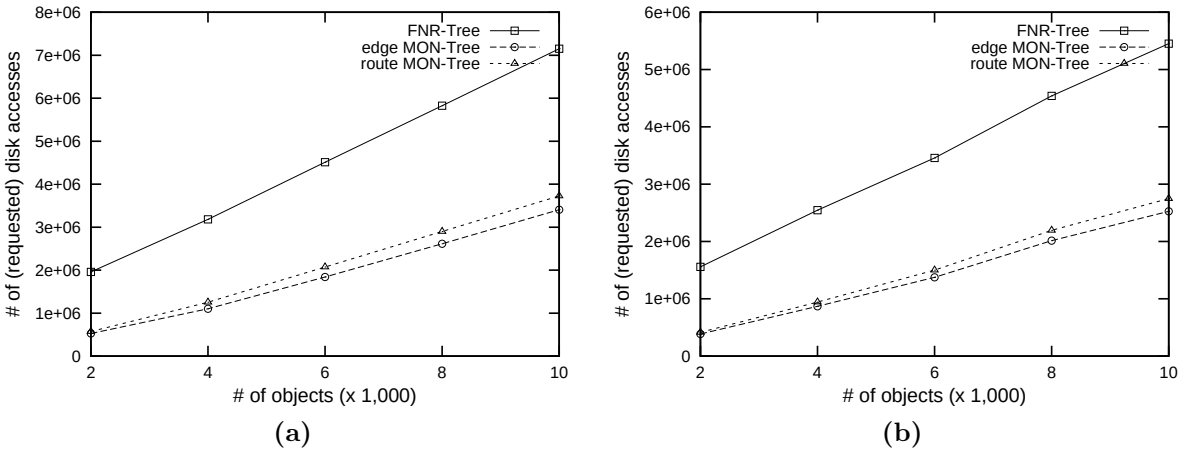


Figure 8: The influence of the number of objects on index construction considering the number of requested disk access for the (a) *rrline* and (b) *rdline* data sets.

executed 100 queries for every combination of the variables, and then, these results are the average number of disk accesses and time spent for one query execution.

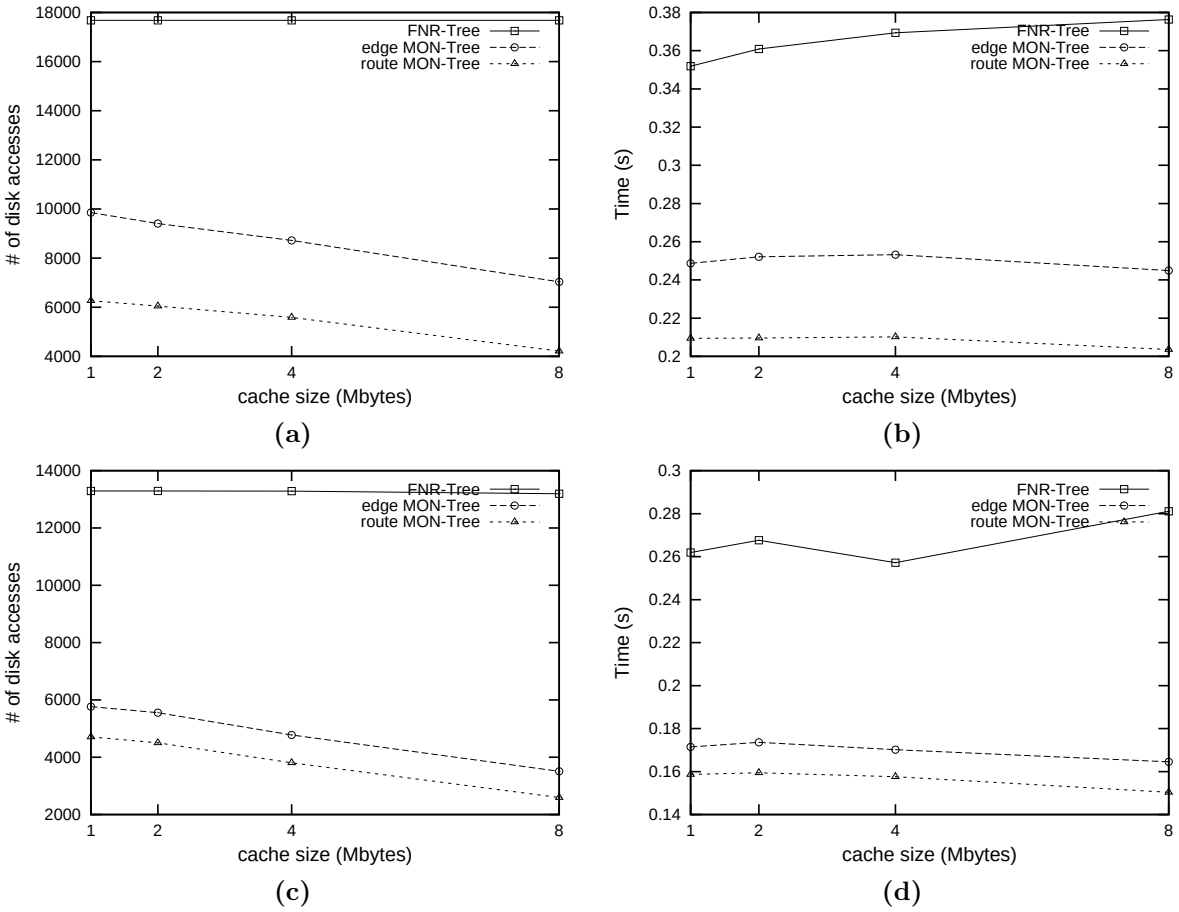


Figure 9: The influence of the cache size in query performance considering: (a) and (c) the number of disk accesses; (b) and (d) the time spent. (a) and (b) represents the *rrline* data set, whereas (c) and (d) represents the *rdline* data set.

Figure 9 shows the influence of the cache size in the performance of the queries. We can see in this figure that the number of disk accesses in the FNR-Tree is insensible to the increase of the cache size, i.e., it has a poor cache locality. Its time for query execution is then increased because more CPU processing is necessary with the increasing of the cache size. For the MON-Trees, the increase of the cache size results in a decrease of the number of disk accesses as well as the time for the query execution. In the next tests, to avoid a big number of plots, our results are presented using a cache size of 4 Mbytes. We decided to use 4 Mbytes to have a fair comparison, since it is a good choice for the MON-Tree and not a bad choice for the FNR-Tree.

Figures 10 and 11 show the influence of the number of objects and the number of time units in the performance of the queries, respectively. In these plots, in parts (a) and (d) we added an important information: the query result size in terms of the number of pieces of moving objects found. In all tests, the MON-Trees outperform the FNR-Tree. We can clearly see that all structures have a linear behavior with respect to the increase of the number of objects and time units. We can see also in these figures that the *route* MON-Tree outperforms the *edge* MON-Tree, but not so much in the *rdline* data sets. The ratio between the size of the polygon in terms of number of line segments is approximately 3 and 5 for the *rdline* and *rrline* data sets, respectively (see Table 1). Then, the gain to use bigger polylines is not so much, when this ratio gets smaller, and it gets worse for some ratio value, i.e., there is a threshold for this ratio, but

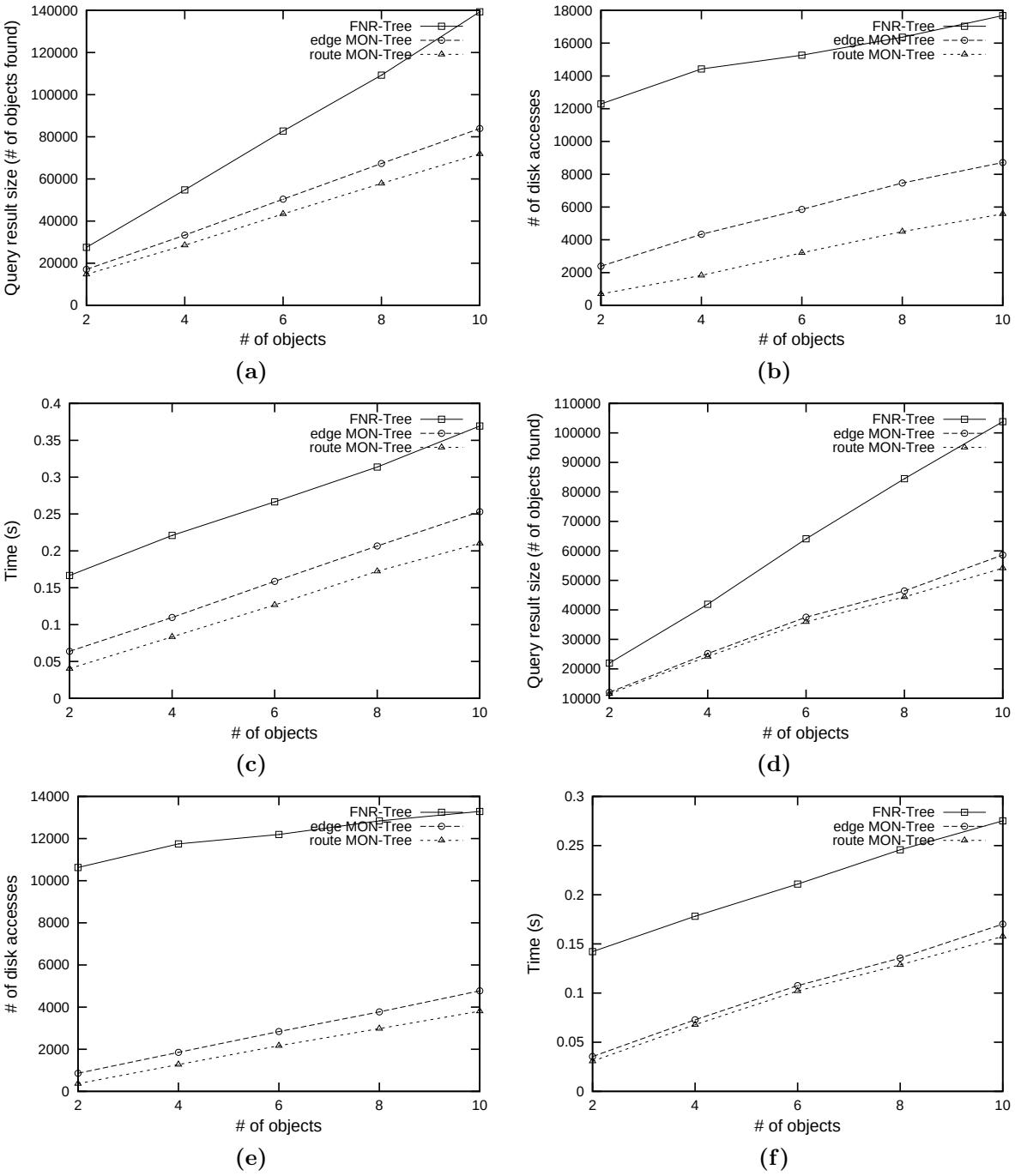


Figure 10: The influence of the number of objects in query performance considering: (a) and (d) the query result size; (b) and (e) the number of disk accesses; and (c) and (f) the time spent. (a), (b), and (c) represents the *rrline* data set, whereas (d), (e), and (f) represents the *rdline* data set.

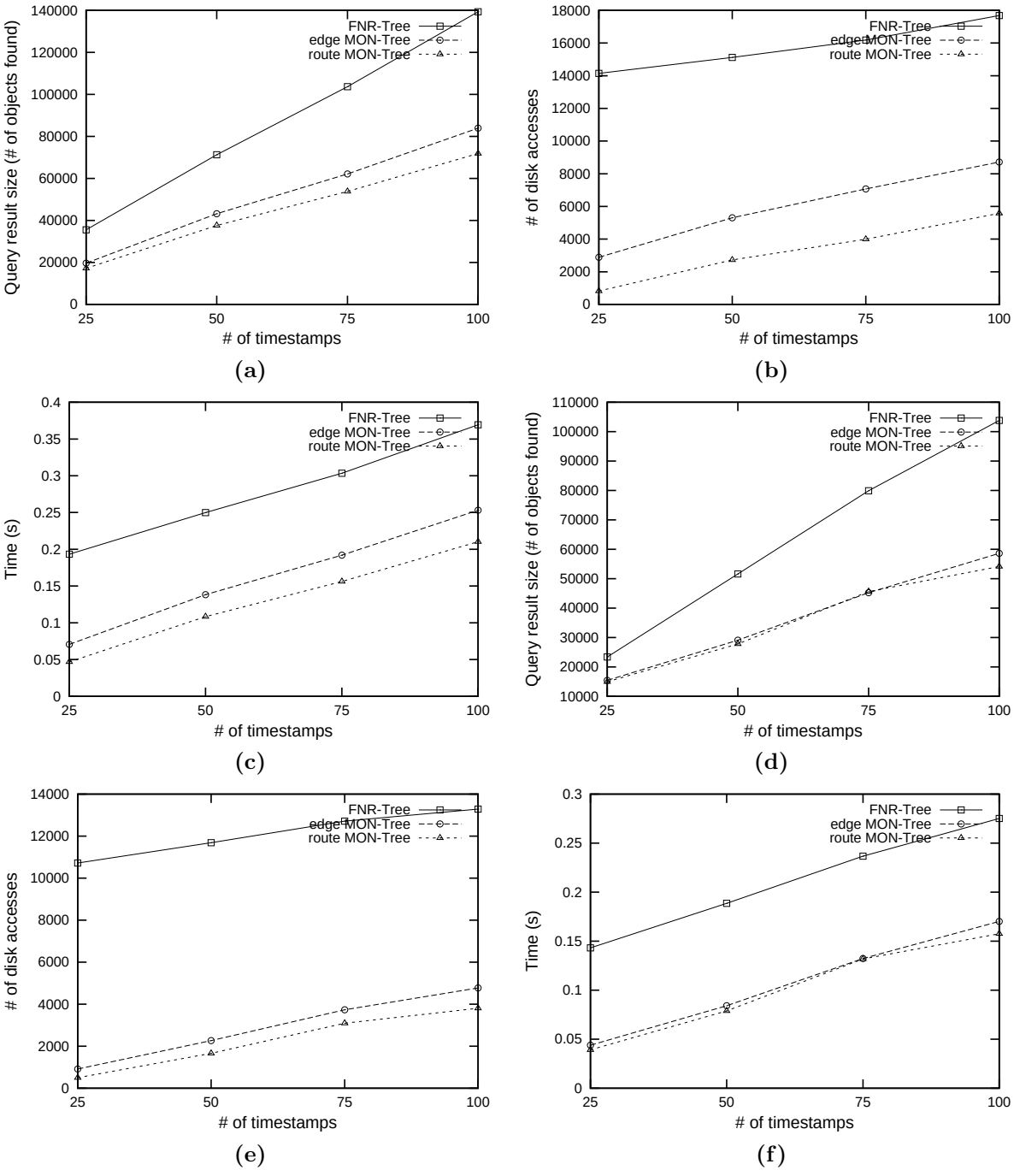


Figure 11: The influence of the number of time units in query performance considering: (a) and (d) the query result size; (b) and (e) the number of disk accesses; and (c) and (f) the time spent. (a), (b), and (c) represents the *rrline* data set, whereas (d), (e), and (f) represents the *rdline* data set.

it is not the intention of this paper to propose it.

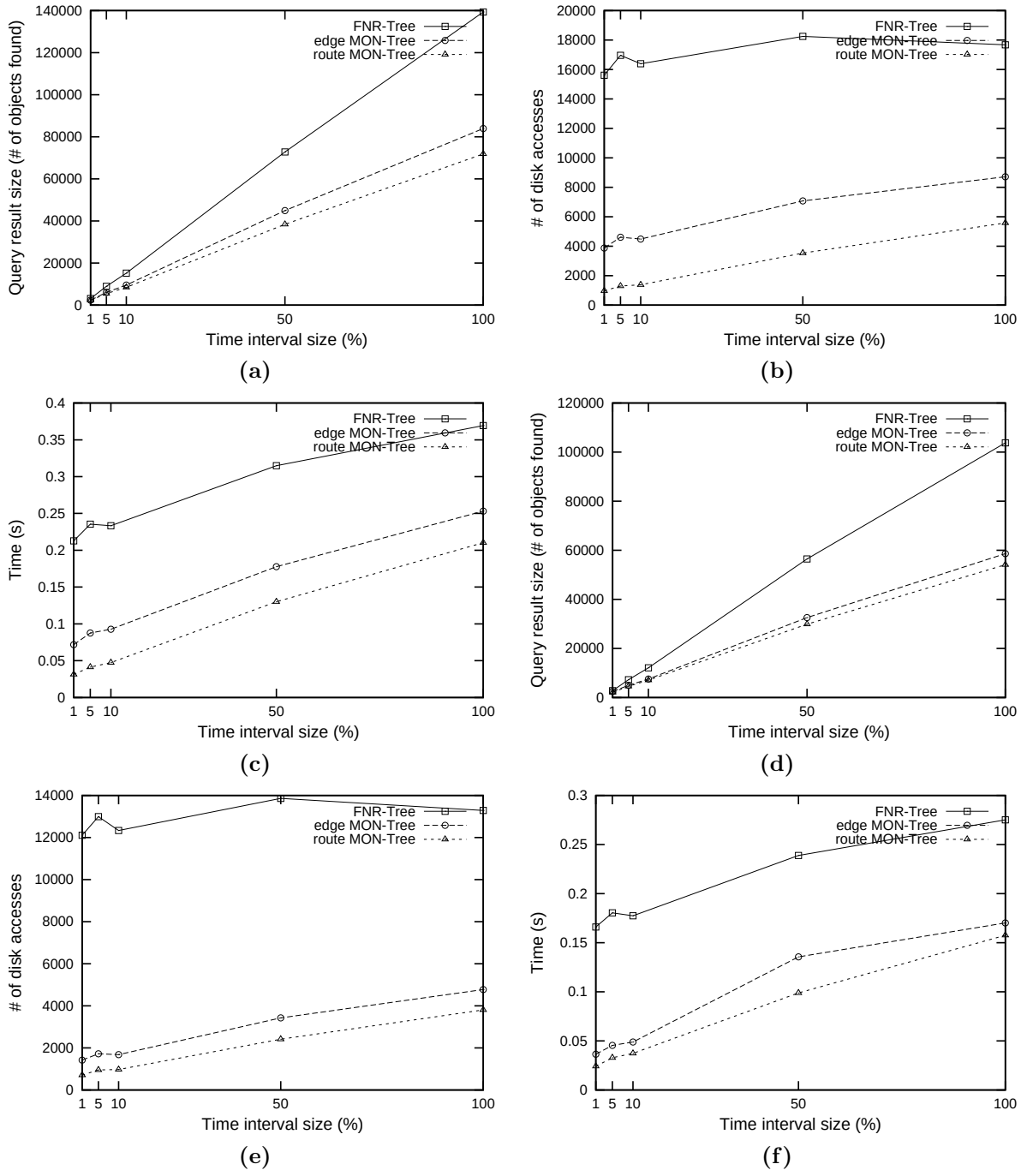


Figure 12: The influence of the query time interval size in query performance considering: (a) and (d) the query result size; (b) and (e) the number of disk accesses; and (c) and (f) the time spent. (a), (b), and (c) represents the *rrline* data set, whereas (d), (e), and (f) represents the *rdline* data set.

The most important experimental tests appear when we vary the time interval query and window query sizes. Figures 12 and 13 show the results found according to these variables. First, we can observe that both index structures have linear behavior in respect to the increase of these two variables. This is very important and desirable for index structures. Second, we can see that the MON-Trees outperform again the FNR-Tree in all tests, and that the *route* MON-Tree shows the best results.

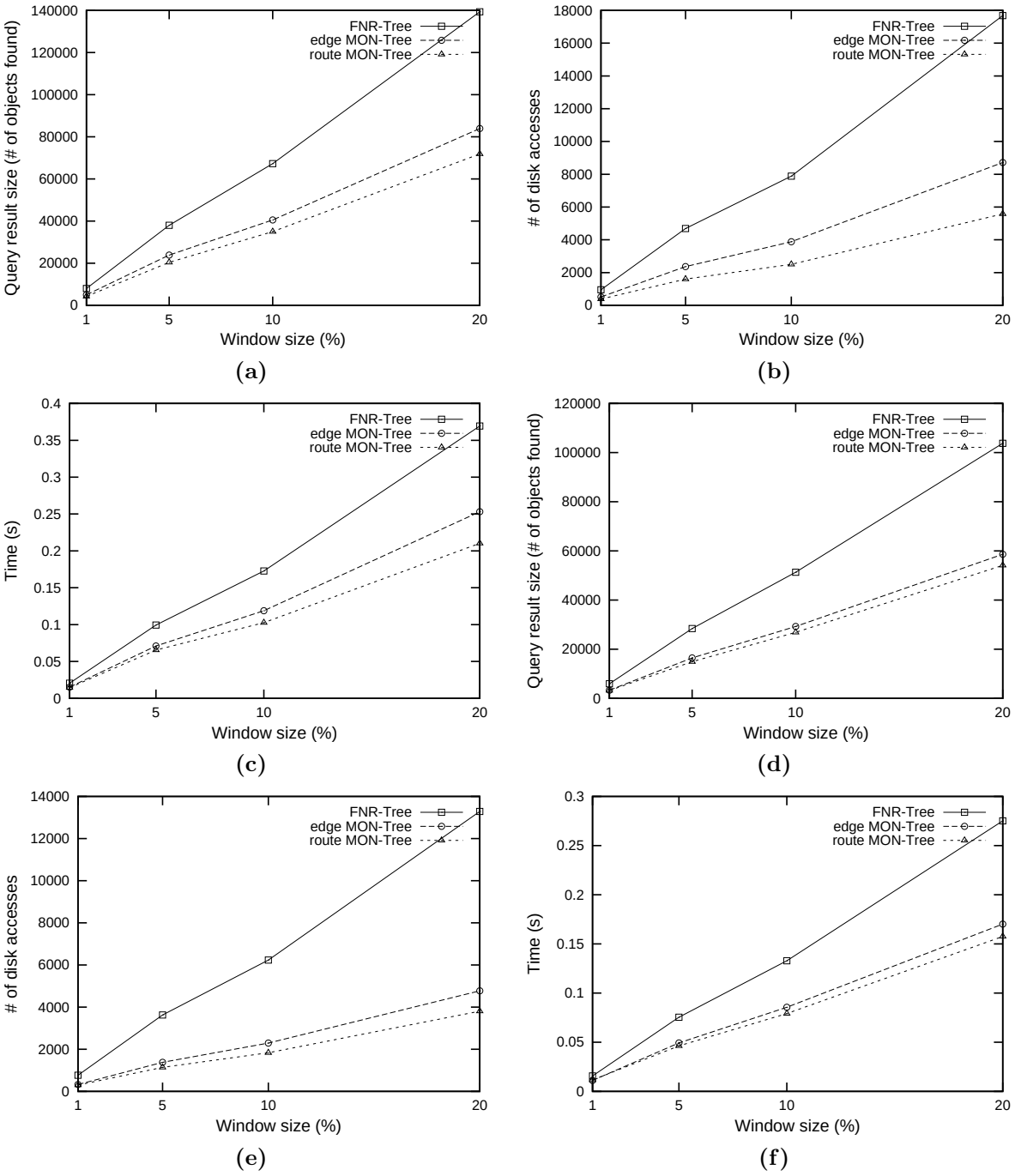


Figure 13: The influence of the query window size in query performance considering: (a) and (d) the query result size; (b) and (e) the number of disk accesses; and (c) and (f) the time spent. (a), (b), and (c) represents the *rrline* data set, whereas (d), (e), and (f) represents the *rdline* data set.

6 Conclusions

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: an edge oriented model and a route oriented one. The MON-Tree is capable of answering two kinds of query: 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 moving objects in networks. We used generated data sets over two real networks, the roads and railroads from Germany. In all our tests, the MON-Trees outperformed the FNR-Tree, and the MON-Tree indexing the route oriented network model showed the 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 [PZMT03]. In this way, queries about proximity can be answered, e.g the nearest neighbour query.

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.

References

[BKSS90] 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.

[Bri02] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

[CAA01] 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.

[CAA02] 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.

[CFG+03] 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.

[EGSV99] 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.

[FGNS00] 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.

[Fre03] 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.

[GAD04] R. H. Güting, V. T. Almeida, and Z. Ding. Modeling and querying moving objects in networks. Technical Report 308, Fernuniversität Hagen, Fachbereich Informatik, 2004.

[GBE+00] 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.

[HJP+03] 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.

[JP03] 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.

[KF93] 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.

[KGT99] 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.

[LL00] 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.

[PAH02] 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.

[PJ99] 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.

[PJ01] 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.

[PJT00] 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.

[PKGT02] 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.

[PM98] 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.

[PZMT03] 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.

[SJ02] 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.

[SJLL00] 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.

[SR01] Z. Song and N. Roussopoulos. Hashing moving objects. In *Proc. of the 2nd Intl. Conf. on Mobile Data Management (MDM)*, pages 161–172, 2001.

[SR03] 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.

[SWCD98] 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.

[TSPM98] 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.

[TUW98] J. Tayeb, Ö. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998.

[TWZC02] 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.

[WCD+98] 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.

[WSCY99] 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.

[WXCJ98] 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.

[XP03] Y. Xia and S. Prabhakar. Q+Rtree: Efficient indexing for moving object database. In *Proc. of the 8th Intl. Conf. on Database Systems for Advanced Applications (DASFAA)*, pages 175–182, 2003.

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 more dead space inside the MBBs and consequently more overlapping area. On the other hand, by indexing the line segments of the polylines, one can diminishes the ovelapping area, but the height of the tree can get bigger and a duplicate elimination mechanism should 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. In this way, we propose 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 1 (b) can be seen in figure 2. The top R-Tree indexes line segments' MBBs and all leaf nodes of the same polyline point to the same bottom R-Tree.

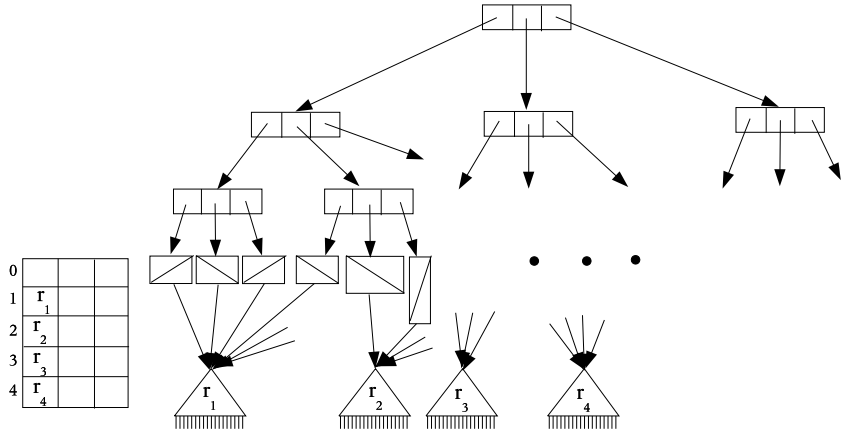


Figure A-1: Example of the revised index structure of the network in Figure 1 (b).

The algorithms must be changed to reflect 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 also, unless 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 equal, since they do not use the top R-tree.

The search algorithm must remove the duplicates before searching the bottom R-Trees. We will proceed in the search as follows. A main memory hash structure containing the polylines identifications will be used to avoid searching more than once the bottom R-Trees. 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 try to make an analysis of these two index representations using the network data sets of our experimental evaluation. We compare only the route oriented model, which is expected to suffer more of the problem of high dead space. To show the tradeoff between

the index representations, let us take the equation proposed in [KF93]

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

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.

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

	rrline (segs.)	rrline (routes)	rdline (segs.)	rdline (routes)
$Height$	3	2	3	2
N	2428	138	2085	125
$TotalArea$	3.57373	3.07067	3.51333	3.11638
L_x	48.8031	14.0031	47.0408	13.9543
L_y	49.7871	16.4717	45.1148	14.3722

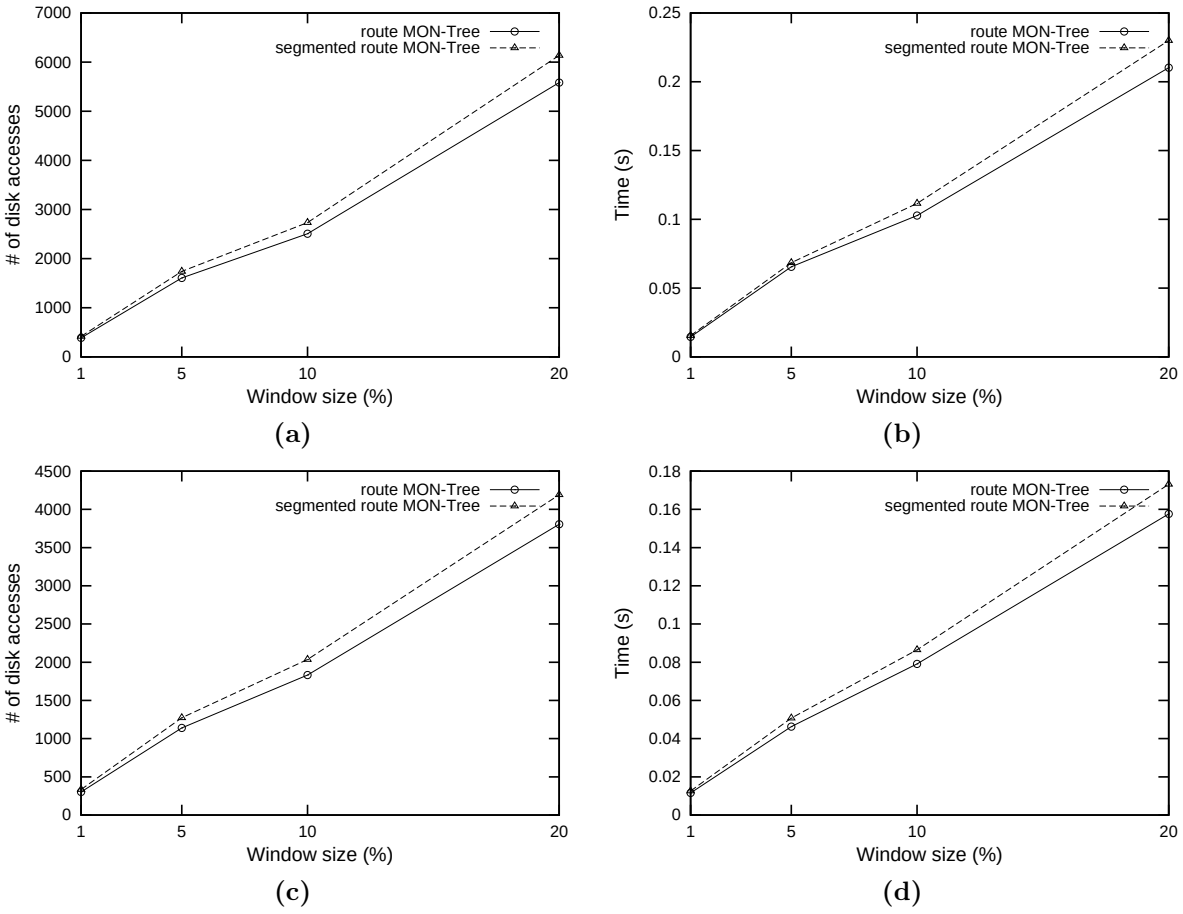


Figure A-2: The influence of the query window size in query performance considering: (a) and (c) the number of disk accesses; and (b) and (d) the time spent. (a) and (b) represents the *rrline* data set, whereas (c) and (d) represents the *rdline* data set.

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. This cost increases with the size of the query window.

Table A-1 shows the values of the parameters of the equation for the *rrline* and *rdline* data sets. As additional information, we added the height of the top R-Tree.

It is expected then that in the network data sets used in the experimental evaluation, this approach of dividing the polylines into line segments in the top R-Tree will not bring any improvement, since all the values of the equation A-1 are bigger for the line segments division approach, even the $TotalArea$. Dividing a polyline into line segments diminishes the total area of the leaf entries, but increases a lot the number of nodes in the tree. It is not true that this approach always diminishes the total area of the nodes in the tree, and for our data sets it is the contrary, i.e., the total area of the nodes is increased dividing the polyline into line segments.

In the MON-Tree, the difference that can determine the query time execution between the original approach and this proposed in this appendix, is the number of disk accesses in the top R-Tree and the duplicate elimination necessary in the second approach.

In order to confirm the results shown in table A-1 we compare the query execution of both approaches applied to the route oriented model. This is shown in figure A-2. The query times and disk accesses for the route MON-Tree are the same as shown in figure 13. It can be seen in this figure that the MON-Tree outperforms in all cases the *segmented* MON-Tree proposed in this appendix, but the difference is very small.

It is very important to remark that, with this new index structure, the representation of the objects' movements according to the both edge and route oriented network models are kept unchanged. Only the top R-Tree index structure is changed. Even 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 trajectories entries in the bottom level is much bigger than the number of network pieces entries. In this way, the change proposed in this appendix in the MON-Tree structure must present a very significative gain (in the first query phase) to represent an important improvement in the overall query processing.