

# Querying the Trajectories of On-Line Mobile Objects

Dieter Pfoser and Christian S. Jensen

June 6, 2001

TR-57

A TIMECENTER Technical Report



## Abstract

Position data is expected to play a central role in a wide range of mobile computing applications, including advertising, leisure, safety, security, tourist, and traffic applications. Applications such as these are characterized by large quantities of wirelessly Internet-worked, position-aware mobile objects that receive services where the objects' position is essential. The movement of an object is captured via sampling, resulting in a trajectory consisting of a sequence of connected line segments for each moving object. This paper presents a technique for querying these trajectories. The technique uses indices for the processing of spatiotemporal range queries on trajectories. If object movement is constrained by the presence of infrastructure, e.g., lakes, park areas, etc., the technique is capable of exploiting this to reduce the range query, the purpose being to obtain better query performance. Specifically, an algorithm is proposed that segments the original range query based on the infrastructure contained in its range. The applicability and limitations of the proposal are assessed via empirical performance studies with varying datasets and parameter settings.

## 1 Introduction

The continued advances in hardware and software technologies such as processors, storage media, graphical displays, positioning systems, and wireless communications promise that the coming years will bring about large quantities of online, position-aware mobile objects [1]. Such objects include mobile-phone terminals, a diverse range of personal digital assistants, electronic clothing, and various kinds of vehicles. Estimates are that by the year 2003, there will be 500 million users of mobile-phone terminals [6]. US law will soon require that mobile phones be position aware. A wristwatch with GPS is already available to consumers.

The human users of these objects will employ a range of services made available to them via the Internet that use position data as an essential ingredient. For example, humans wearing smart suits and engaged in action sports may receive warnings of impending dangers, and emergency support may be dispatched when a suit senses that its wearer is in distress.

In order to provide this type of functionality, the services receive samples of the position of each moving object, which enables them to construct a trajectory for each object that represents the object's movement. Trajectories are also termed polylines and consist of connected line segments. Manipulating and querying these representations of movements in space and time is inherently challenging. The amount of collected data is proportional to the elapsed time. In connection with this new type of spatiotemporal data we have to consider new types of queries [15] when designing new indexing techniques and query processing algorithms.

Generally, applications dealing with moving objects may be grouped according to their three *movement scenarios*. We distinguish among *unconstrained movement* (vessels at sea), *constrained movement* (cars, pedestrians), and *movement in networks* (trains and, in some cases, 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 the absolute coordinates. The movement effectively occurs in a different space than for the first two scenarios. All three scenarios may apply to mobile users, since mobile terminals can exist in cars, ships, trains, or can in general be hand held devices.

Objects that constrain movement are termed *infrastructure*. For moving cars, examples of infrastructure are buildings, lakes, and pedestrian zones, but also roadblocks, or slow-moving traffic. From the above examples one can see that infrastructure can be categorized as well. The simplest type is *static* infrastructure, i.e., spatial objects that exist and do not change "throughout time." Conversely, infrastructure may be *dynamic*, in which case elements may appear and disappear (road blocks), as well as change throughout their existence (slow-moving traffic).

In this work, we devise a new technique that utilizes infrastructure when *processing spatiotemporal range queries in constrained-movement scenarios*. To obtain a first assessment of this approach, we only

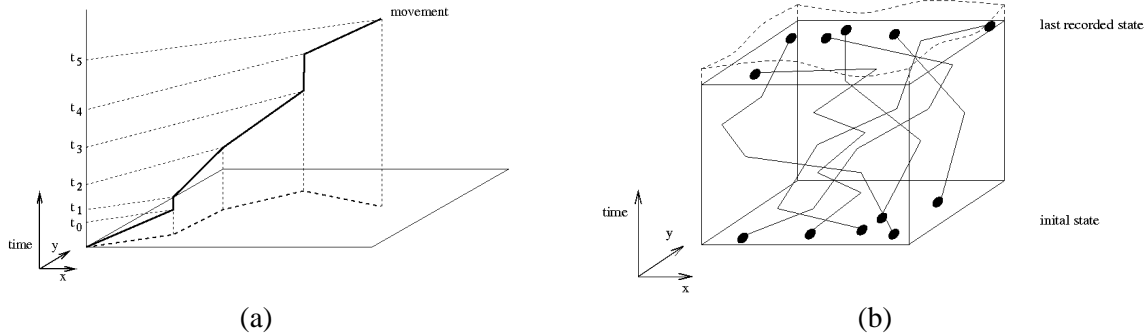


Figure 1: Moving point objects: (a) a trajectory and (b) a spatiotemporal space with several trajectories

consider static infrastructure. We base our approach on a two-step technique used in spatial query processing that utilizes an index containing approximations of the data. In considering infrastructure, we introduce an additional pre-processing step in which we do not actually query the trajectory data itself, but the infrastructure. A spatiotemporal range query, QW, is executed against the infrastructure. Depending on this result, query processing may stop here, i.e., QW is totally covered by infrastructure, or QW is segmented into a set of smaller query windows, qwi, which are either used for querying the trajectory data, or, alternatively, the original range query is used. For query window segmentation, we devise an algorithm that takes the infrastructure and spatiotemporal range query, QW, as arguments and returns a set of smaller query windows, qwi. An important characteristic to be considered in the segmentation process is the shapes of the resulting query windows. Kamel and Faloutsos [5] discuss a formula to predict R-tree performance for a uniform spatial dataset and implicitly devise the shape of an optimal query window as well. It turns out that square query windows are preferable over elongated, rectangular ones. Consequently, the algorithm devised aims to produce query rectangles that are as square as possible.

Previous work exists towards processing multiple query windows. Papadopoulos and Manolopoulos [12] discuss an approach in which they use a Hilbert ordering of the query windows and an LRU-buffer in connection with indexing. This work is based on previous work on multiple query optimization [17]. Leutenegger and Lopez [7] describe a model to predict R-tree performance when using buffering. Their approach is based on the prediction of R-tree performance presented in [5]. In this paper, we adapt the approach of Papadopoulos and Manolopoulos [12] to process the set of segmented query windows, qwi.

To the best of our knowledge, no other work exists that uses query window segmentation based on structural information, i.e., infrastructure, to reduce the query processing cost.

The outline of the paper is as follows. Section 2 describes trajectories and infrastructure in more detail. Section 3 gives the new query processing technique. This includes a discussion of the shapes of query windows and a presentation of the query window segmentation algorithm. Section 4 presents the performance study for various types of trajectories and infrastructure. Section 5 offers conclusions and research directions.

## 2 Moving Objects and Infrastructure

This section briefly introduces spatiotemporal data in the form of trajectories, and it introduces infrastructure, which we will take to refer to static spatial objects that constrain movement.

### 2.1 Trajectories

To record the true movements of objects, we would have to know their positions at all times, or better, on a continuous basis. However, current technology only allows us to sample an object's position, i.e., to obtain the position at discrete instances of time, such as every few seconds.

A first approach to represent the movements of objects would be to simply store the position samples. This would mean that we could not answer queries about the objects' movements at times in-between sampled positions. Rather, to obtain the entire movement, we have to interpolate. Here, the simplest

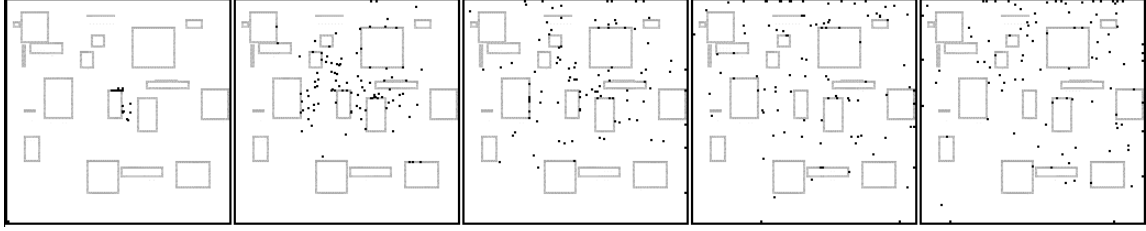


Figure 2: Moving objects snapshots and infrastructure

approach is to use linear interpolation, as opposed to other methods such as polynomial splines [2]. The sampled positions then become the endpoints of line segments of polylines, and the movement of an object is represented by an entire polyline in three-dimensional space. In geometrical terms, the movement of an object is termed a *trajectory* (we will use “movement” and “trajectory” interchangeably). The solid line in Figure 1(a) represents the movement of a point object. Space (x- and y-coordinates) and time are combined to form a single coordinate system. The dashed line shows the projection of the movement on the two-dimensional plane [13]. Figure 1(b) shows the spatiotemporal data space (the cube in solid lines) and several trajectories (the solid polylines). Time moves in the upward direction, and the top of the cube is the time of the most recent position sample. The wavy-dotted lines at the top symbolize the growth of the cube with time. Interpolating trajectories raises questions on the uncertainty associated with a particular representation [13].

## 2.2 Infrastructure

Infrastructure elements obstruct the movements of objects. As we saw previously, depending on the type of the moving object, what constitutes infrastructure might change. Figure 2 gives an example of trajectories affected by infrastructure. The five images represent temporal snapshots of the trajectories, i.e., slices of a cube such as the one shown in Figure 1(b). The data was generated using the GSTD tool [14].

With respect to indexing, trajectories pose a serious challenge. By using an R-tree like access method, we approximate the objects to be indexed. Approximating a line by a Minimum Bounding Box (MBB) introduces a large amount of so-called “dead space.” This means that even in areas where there are no trajectories, the index “believes” that there are.

In terms space and where movement can occur, infrastructure represents “black-out” areas, meaning that there are no trajectories to index where there are infrastructure elements. However, because of dead space, those areas are not empty in the index and will incur unnecessary search in the index as well as produce a certain number of falsely reported answers, which must subsequently be eliminated. Both lead to extra I/O operations and thus negatively affect performance. To eliminate this extra I/O, we can use infrastructure in a pre-processing step, i.e., why should we look for objects, where there cannot be any? The strategy we choose is to *query the infrastructure to save on querying the trajectory data*. Overall, this will turn out to be favorable, since the number of infrastructure elements can be assumed to be very small compared to the trajectory data. Further, the trajectory data is growing with time, whereas the size of the infrastructure data remains more or less constant.

In the following, we devise a query processing strategy to include infrastructure in a pre-processing step.

## 3 Querying Moving Objects Data

Trajectories and the relevant queries require new query processing techniques. In previous work, the focus was on the design of new access methods [15]. In the following, we devise a new query processing technique, which is based on technique known from spatial databases.

### 3.1 Three-Step Query Processing

In spatial databases, a two-step technique is used to process queries. Using approximations of the real spatial objects in the index (minimum bounding rectangles (boxes), MBR (MBB)) requires filtering out false drops from the set of solutions we obtain after processing a query through the index. In many cases, the real spatial entity in the database has to be examined, to decide whether this entity belongs to the final set of solutions [3].

In Section 2.2, we presented infrastructure as the static spatiotemporal objects that hinder objects in their movement, i.e., where there is infrastructure there cannot be any moving objects. A query window might range over infrastructure, thus requiring us to query space where there cannot be any objects. At this point do note that a query window ranges over two spatial and one temporal dimension. We use infrastructure only to limit the two spatial dimensions. The temporal dimension remains unaffected.

We extend the two-step technique to include an additional pre-processing step, namely query window segmentation. We only want to consider those parts of the query window, QW, that do not range over infrastructure. The outcome of this step can be either one of the following three cases. In case (i) the set of segments is empty, we stop processing the query since it only ranges over infrastructure. We get a set of smaller query windows,  $q_{wi}$ , and processing them is (ii) beneficial, or (iii) is not beneficial, in comparison to processing the original query window. Beneficial in this context is defined as a lower number of page accesses to process the query.

The technique for processing spatiotemporal range queries involving infrastructure thus comprises the following three steps.

1. Segmenting the original query window, QW, into a set of smaller query windows,  $q_{wi}$ .
2. Querying the index depending on the outcome of step 1 to retrieve a candidate set of solutions.
3. Evaluating all objects contained in the candidate set of solutions.

To efficiently process step 1, we can index the infrastructure elements by using a spatial access method such as the R-tree. Furthermore, assuming that the entire infrastructure is known in advance, we can even such an index (cf. [5]).

### 3.2 Query Window Split Algorithms

An essential part of the afore-mentioned three-step technique involves the segmentation of the query window QW. Before we can devise an algorithm for this task, we have to establish of what is an optimal query window, i.e., what is the shape of the query window this segmentation algorithm should aim for.

#### 3.2.1 The Ideal Query Window

Kamel and Faloutsos [5] derive a formula to determine the number of disk accesses needed to process an arbitrary range query  $q_i$ . Their formula assumes an R-tree based index, it is however independent of a particular construction method. It is assumed that the centroids of all query ranges are uniformly distributed over the data space, which is the unit square. The number of disk accesses,  $P$ , for a query window,  $q_i$  with extents  $q_{ix}$  and  $q_{iy}$  in the respective dimensions, is computed as follows.

$$P(q_{ix}, q_{iy}) = \text{Total Area} + q_{ix} \cdot L_y + q_{iy} \cdot L_x + N \cdot q_{ix} \cdot q_{iy} \quad (1)$$

In Formula (1), Total Area stands for the sum of all the areas of the nodes of the tree,  $L_x$  and  $L_y$  are the sums of the  $x$  and  $y$  extents of all the nodes in the index.

In our case, the assumption that the data is uniformly distributed over the whole data space does not hold because of the infrastructure. Still, if we assume that the data is uniformly distributed in the data space not occupied by infrastructure, we can use the above formula as a first approximation.

What we can see from Formula (1) is the importance not only of minimizing the area of the query window, but its perimeter as well. I.e., having two query windows with the same aerial extent, the one

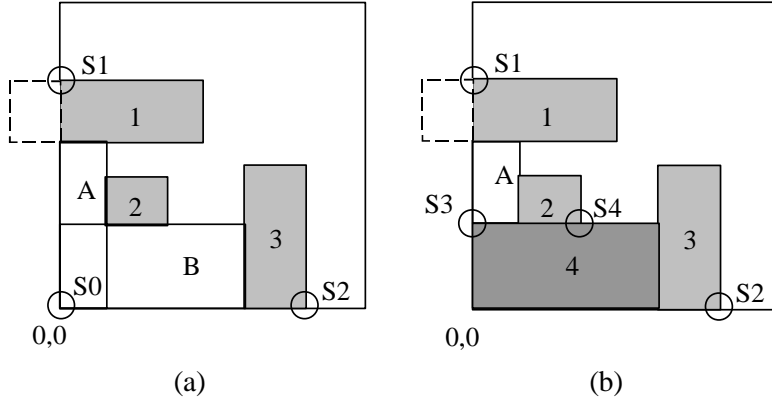


Figure 3: Query window segmentation algorithm

with the smaller perimeter requires fewer disk accesses. The shape that minimizes its perimeter for a given area size is the square. Consequently, what we can derive for the query segmentation algorithm is that the resulting shapes should resemble a square as much as possible. Similar results on the shape of a query window were reported by Pagel et al. [10].

### 3.2.2 Segmentation Algorithm - the Principle

We proceed to devise a segmentation algorithm for query windows. The parameters of the algorithm are a *query window*, and a *set of infrastructure elements*. The output of the algorithm is a *set of query windows*, i.e., rectangles.

The general principle behind our approach is to decompose a given query window based on the infrastructure contained in it. The intuition is, to “chop” the parts of the query window not occupied by infrastructure into well-shaped rectangles, i.e., possibly squares. In the examples of Figure 5(a), few but large and (b), many but small infrastructure elements, where infrastructure elements are shown as black rectangles, the possible outcome of such a segmentation process is shown as white rectangles.

To segment the query window, i.e., to determine the rectangles, the algorithm proceeds from the lower-left corner of the query window to the upper-right. Given a seed point, i.e., the lower-left corner, we try to span a rectangle as far towards the upper right corner as possible. Consider the situation of Figure 3(a). Here, we want to span a rectangle from seed point S0 (seed 0) to the upper right corner. Infrastructure elements 1, 2, and 3 constrain us.

Seed points are the lower-left corners of all rectangles. Seed points are determined in two stages. Initially, the algorithm determines all seed points on the left and lower side of the query window (they would not be found otherwise). Further, more seed points are determined during the course of the algorithm when new rectangles are segmented.

### 3.2.3 Segmentation Algorithm – a Detailed View

The following, more detailed description of the algorithm is based on the pseudo-code shown in the Appendix. In the code, the three main parts of the algorithm are grouped together by boxes and labeled 1,2,and 3 using large light gray digits. Part 3, in turn, consists of subparts 3a, 3b, to 3c.

In the algorithm, part 1 determines the initial set of seed elements. Those include (i) the origin of the query window (lower-left corner), (ii) the upper-left corners of all infrastructure elements touching with the left side of the query window, and (iii) the lower right corners of the elements touching the bottom side of the query window. To clarify, if an infrastructure element intersects with the query window, the element's part outside is truncated for the purpose of segmenting the query window.

The second and third part of the algorithm is contained in a loop, which iterates over all seed elements. The second part determines the query window segments and the third part computes new seed points.

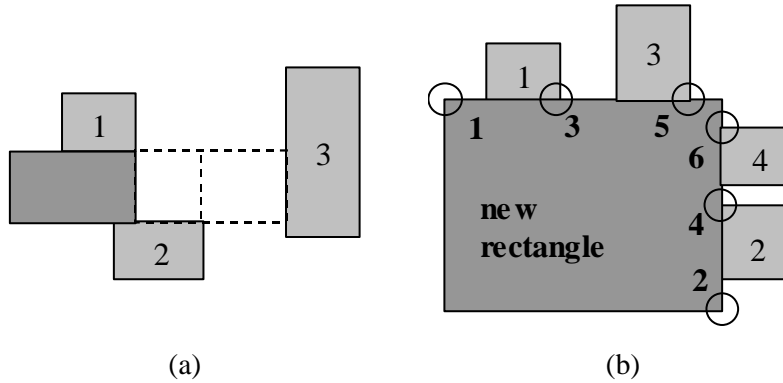


Figure 4: (a) elongated rectangles, and (b) seed points

In part 2, the algorithm tries to find infrastructure elements that bound a rectangle whose origin is the current seed point. In the example shown in Figure 3(a), the algorithm determines three candidate bounds. Elements 1, 2, and 3, constrain a rectangle originating from seed point  $S_0$ . Those constraints leave us with two possible rectangles, A and B (lines 16 to 42 in the code). The algorithm evaluates both and chooses element B because of better proportions (cf. Section 3.2.1). The criterion used is the perimeter ratio of the possible rectangles, i.e., longer side divided by smaller side. For a square this ratio is 1, for rectangles this ratio is larger (lines 46 to 59 in the code). The rectangle with the smallest ratio is selected.

Having determined the best rectangle, we have to judge whether its shape is appropriate, i.e., it could be elongated in the x or y direction. An example check would be that the length in the x-direction is  $n$  times longer than in the y-direction, where  $n$  is a threshold parameter of the algorithm. In the algorithm this procedure is contained in the function `JudgeShape` in line 62.

The type of shape, i.e., the outcome of `JudgeShape` determines the behavior in part 3 of the algorithm, in which it computes the final rectangle and new seed points. Part 3a is executed if the rectangle is elongated in the x-direction. In this case the rectangle is possibly shortened such that it ends with its upper constraint (element 1 in Figure 4(a)) or an element constraining the rectangle from below (element 2 in Figure 4(a)). The algorithm chooses the element that is more restrictive. In the example of Figure 4(a) that is element 1. If neither element is restrictive, i.e., both elements have larger y extensions than our rectangle, the rectangle is left unchanged.

Part 3b is executed if the rectangle is elongated in the y-direction and part 3c if the rectangle is accepted by the function `JudgeShape`.

The rationale behind this approach is that by disallowing extensively elongated rectangles, we allow for a possibly better choice of a rectangle at a later step in the algorithm (cf. Figure 4(a)).

As for new seed points, Figure 4(b) illustrates all possible candidates. Generally, we can encounter six different types of new seed points.

1. the upper left corner of the newly found rectangle,
2. the lower right corner,
3. meeting point with the upper constraint,
4. meeting point with the right constraint,
5. meeting point with an additional upper bounding element that was not considered as a constraint, and
6. meeting point with an additional right bounding element.

For cases 5 and 6, since there can be more than one upper (right) bounding element, the algorithm can also find more than one seed point in each case.



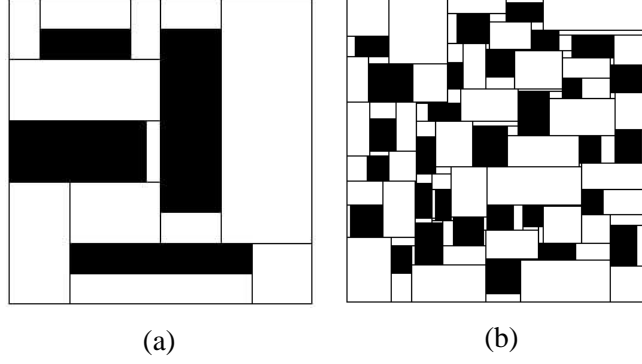


Figure 5: Segmented query windows

Remember, seed points are the lower-left corners of future rectangles, thus they can be only found on the upper side (cases 1, 3, and 5), and on the right side of a newly found rectangle (cases 2, 4, and 6).

The algorithm has to check all six alternatives in case of a well-shaped query window (part 3c of the code). In case the rectangle was elongated (part 3a and 3b), we only have to consider cases 1 and 2. The reasons can be derived from the definition and the three different scenarios and the definition of the seed point cases.

Figure 5 shows two examples outputs of the segmentation algorithm. The infrastructure elements are drawn in black. The infrastructure was created using a random rectangle generator [14].

### 3.2.4 A Word on Running Time

The running time of the algorithm is determined by the number of infrastructure elements,  $I$ , and the number of query windows,  $Q$ , i.e., the result of the segmentation process. Assuming a uniform distribution of the infrastructure elements over the data space,  $Q$  is found to be two to three times  $I$ . This factor, so far, is only empirically established.

The main body of the algorithm is a loop over the set of seed points. The number of seed points,  $S$ , is  $Q - I$ , i.e., for every created query window, there has to be one seed point. The costliest operation in this loop is to sort all existing elements (infrastructure elements plus already created rectangles) once in the  $x$  and then in the  $y$  direction for each seed point. Assuming sorting is of cost  $n \log n$ , the cost of sorting for the first segmented query window is  $I \log I$  whereas the cost for the last is  $(Q+I-1) \log(Q+I-1)$ . To compute the total cost we compute the sum of an arithmetic series.

$$\begin{aligned}
 & 0.5 \cdot ((Q+I-1) \cdot \log(Q+I-1) + I \cdot \log I) \cdot ((Q+I-1) \cdot \log(Q+I-1) - I \cdot \log I) = \\
 & 0.5((Q+I-1)^2 \cdot \log^2(Q+I-1) - I^2 \cdot \log^2 I)
 \end{aligned} \tag{2}$$

The running time is therefore in the order of  $O(n^2 \log^2 n)$ .

### 3.3 Query Window Segmentation and Indexing

The three-step technique uses in the second step an index to process the query. Two access methods for trajectory data are a modified version of the R-tree and the TB-tree (Trajectory Bundle) [15]. The TB-tree possesses special capabilities in processing spatiotemporal query types (cf. [15]). Segments in the TB-tree are grouped together based on the trajectory they belong to. The R-tree does not preserve trajectories and uses purely spatial characteristics such as proximity. Thus, nodes in the TB-tree are larger and “more wasteful” with respect to space. Consequently, such an index has a higher degree of overlap with respect to infrastructure.

We modify both access methods to allow for the buffering of retrieved nodes, i.e., pages. We adopt what is known as the “Least Recently Used (LRU)” approach. Here, a newly referenced page replaces the “not

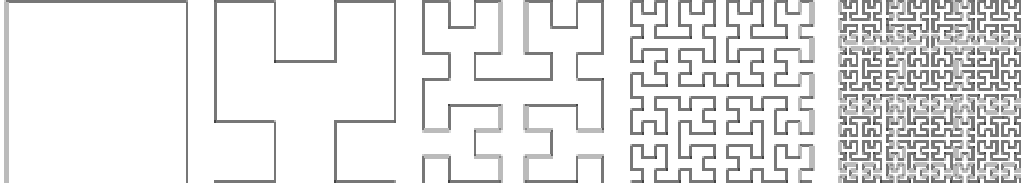


Figure 6: Hilbert space filling curve

been referenced for the longest time” page. This scheme exploits overlap in page retrievals caused by simultaneous execution of spatially close query windows.

To efficiently utilize the LRU buffer, we order the segmented query windows with the help of a space-filling curve, namely the Hilbert curve. Basic properties of a space-filling curve are (i) it covers an “area” completely, where area might also refer to higher dimensional volumes, (ii) each point in space is visited once and only once, and (iii) neighbor points in the native space are likely to be close neighbors on the space-filling curve. Property (iii) is used to measure the quality of the space-filling curve, i.e., its ability to preserve proximity. Moon et al. [8] show analytically and empirically that the Hilbert curve achieves better clustering than the z and Gray-code curve. Further experiments [4] give similar results.

The Hilbert curve seen in Figure 6 is constructed in a self-similar way, by using rotation and mirroring. Algorithms for the construction of space filling curves can be found on the Web [9], or in the literature [4].

## 4 Experimental Studies

The goal of the following experiments is twofold. First, we try to establish the conditions under which query window segmentation is useful. That is to distinguish when case (ii) or (iii) are more beneficial for processing spatiotemporal range queries. Second, segmenting query windows might prove to be more or less beneficial for different access methods. We consider here the R-tree and the TB-tree as mentioned in Section 3.3.

The parameters in our experiments are varying infrastructure datasets, query windows, and LRU-buffer sizes.

### 4.1 Varying Query Window Size and Datasets

In the first set of experiments we compare the cost of querying trajectories using the original query windows, QW, to using the set of segmented query windows, qwi for different query windows and infrastructure datasets.

In our first experiment, we use an artificial set of infrastructure elements as shown in Figure 7. The real-world correspondence of this infrastructure composition could be a city with building blocks. We create trajectories for 500 moving objects that are uniformly distributed over the whole data space. A trajectory consists itself of 500 segments, leading to a total of 250k segments, i.e., the total number of entries in the index.

The size of the LRU buffer is 16 Kbytes, which corresponds to 16 times the page size of the index, which is 1 Kbyte.

Figure 7 shows a temporal snapshot of the trajectory data used in the following experiments. The infrastructure elements are shown as gray rectangles. Again, we used GSTD++ [14] to generate trajectory data, and the parameters were chosen as such that the density of the trajectories is higher towards the center of the data space and the objects move around their initial positions. Using this data, we conduct six experiments with a varying query window size.

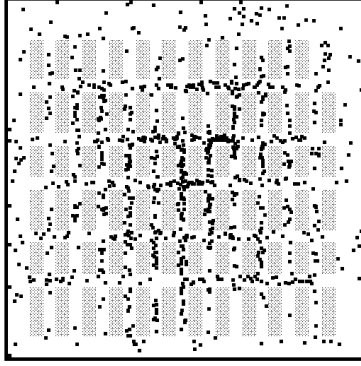


Figure 7: A snapshot of the trajectory dataset

The outcome of the experiments is shown in Table 1. For each query window, we measure the number of node accesses using the original query window (QW) and the set of segmented query windows (qwi). For the latter, the number in parenthesis indicates the LRU buffer hits. The number of query windows that constitute qwi is given as  $N$ .

Assuming the data space is the unit cube, the temporal extent of the queries shown in Table 1 is 0.2 in the midst of the temporal range, i.e., from 0.4 to 0.6. We leave the temporal range constant throughout all of the experiments, since we observed that varying it, only increases/decreases the absolute number of visited nodes but not the relative number, i.e., nodes visit for QW vs. qwi.

We can observe that with an increasing query window size, the advantage of segmenting QW into qwi decreases. Also, it seems that only infrastructure elements that are at the border of the query window matter. In comparing experiments 3, 6, and 7, we can observe that although  $N$  is the same in all three cases, because the corners of the infrastructure coincide with the query window in experiment 3, the gap between using QW and qwi is larger than in experiments 6 and 7, where the boundary of QW is inside the infrastructure elements. The larger the part of the boundary that is inside the infrastructure, the smaller is the advantage of segmentation (experiments 6 and 7). In experiment 8, we extended QW such that no infrastructure intersects with the boundary of the query window. As a result we can observe that segmentation offers no advantage any more.

In comparing the two access methods, we see that segmentation is in more situations beneficial the TB-tree than it is for the R-tree. In experiment 7, while segmentation offers virtually no advantage for the R-tree index (89 vs. 87 node accesses), segmentation for the TB-tree still proves to be beneficial (141 vs. 122 node accesses). This can be explained by the properties of the indices as outlined in Section 3.3, i.e., the TB-tree nodes have a higher volume of dead space.

Next, we perform experiments with a random infrastructure scenario. We compute an arbitrary set of rectangles, where the number as well as the minimum and the maximum extents are input parameters of the data generator [14]. The parameters of the trajectory data are the same as in Figure 7, namely 250k segments stemming from 500 moving objects uniformly distributed over the data space. Table 2 shows the experimental outcome.

Again, the trajectory data is massed around the center of the data space. In experiments 10 to 12, the space occupied by infrastructure is smaller than in the previous experiment. The infrastructure scenario in experiments 10 and 11 consists of fewer (50), but larger elements, which is in contrast to experiment 12, where we encounter many (900), but small elements.

Experiments 11 and 12 show that given an equal sized query window QW but smaller infrastructure elements results in many segment query windows. The second picture in experiment 12 is an enlargement of the query window. Consequently, the efficiency of our approach is bound to the infrastructure element size.

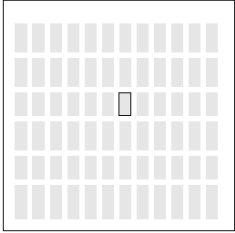
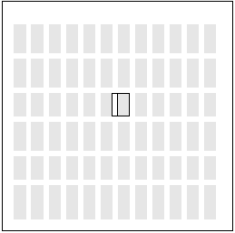
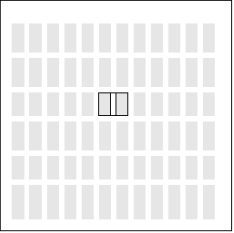
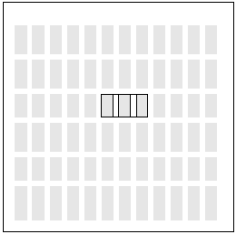
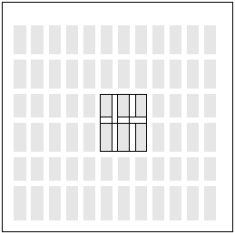
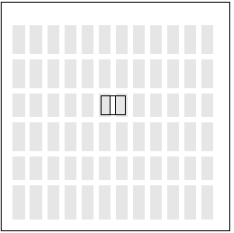
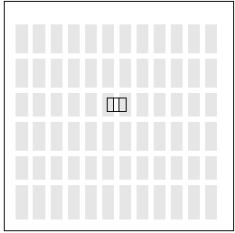
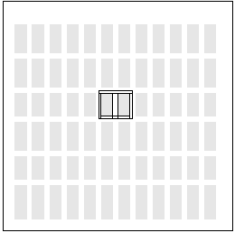
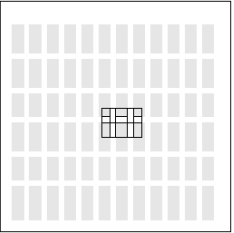
<b>Experiment</b>	1		2		3	
<b>QW (R/TB-tree)</b>	61	143	141	170	175	196
<b>N</b>	0		1		1	
<b>qwi (R/TB-tree)</b>	0	0	118(0)	142(0)	118(0)	142(0)
<b>Visualization</b>						
<b>Experiment</b>	4		5		6	
<b>QW (R/TB-tree)</b>	269	265	486	368	123	167
<b>N</b>	2		7		1	
<b>qwi (R/TB-tree)</b>	196(34)	199(82)	436(400)	321(646)	100(0)	134(0)
<b>Visualization</b>						
<b>Experiment</b>	7		8		9	
<b>QW (R/TB-tree)</b>	89	141	319	289	311	245
<b>N</b>	1		7		7	
<b>qwi (R/TB-tree)</b>	87(0)	122(0)	319(393)	289(678)	306(388)	239(639)
<b>Visualization</b>						

Table 1: Various query window sizes

## 4.2 Varying LRU Buffer Size

To show the effects of a varying LRU buffer size, we choose experiment 5 as a basis. The LRU buffer is used to store retrieved pages in main memory. Thus, revisiting them does not require a disk access. Now, in case of segmenting a query window, all resulting query windows, qwi, are spatially co-located. Naturally, when executing the queries sequentially, many nodes in the index will be accessed multiple times. Thus, in reducing the LRU buffer size, we reduce the advantage of using the segmented query windows over the original window. Figure 8 shows the number of page accesses and, conversely, the number of buffer hits when varying the LRU buffer size from 1 to 16 Kbytes.

<b>Experiment</b>	10		11	
<b>QW (R/TB-tree)</b>	396	357	44	98
<b>N</b>	4		3	
<b>qwi (R/TB-tree)</b>	377(234)	346(391)	36(34)	88(149)
<b>Visualization</b>				
<b>Experiment</b>	12			
<b>QW (R/TB-tree)</b>	166	172		
<b>N</b>	56			
<b>qwi (R/TB-tree)</b>	166(1742)	172(4584)		
<b>Visualization</b>				

Table 2: Queries: various infrastructure and trajectory datasets

We observe that the TB-tree benefits more from using a buffer than the R-tree. Because of the properties of a TB-tree index, for a set of queries that are spatially close, it is more likely to access the same node more often than it is for an equivalent R-tree. Consequently, the TB-tree benefits more from an increased LRU-buffer size, than the R-tree does.

### 4.3 Summary

We can identify the following parameters that determine the effectiveness of query window segmentation. First, the larger the number of segmented query windows, the smaller the advantage over QW. Second, the more space infrastructure occupies within QW, the better. Third, the more infrastructure is concentrated at the boundaries of QW, the better. The experiments showed that infrastructure placed at the center of QW affords query window segmentation less than infrastructure located at the boundary.

In comparing the R-tree and the TB-tree, we saw that the latter benefits in more cases from query window segmentation. Further, it benefits more from an increased LRU buffer size, than the R-tree. The reasons here can be found in how the respective access methods construct the indices.

## 5 Conclusions and Future Work

In this paper, we present a new query processing technique for trajectory data stemming from a constrained movement scenario. We extend the well-known two-step technique from spatial query processing to include an additional pre-processing step prior to the filter step. Given an arbitrary spatiotemporal range query, QW, the aim of this step is to segment QW into a set of smaller query windows, qwi. We exploit infrastructure information, i.e., spatial objects that constrain movement, to

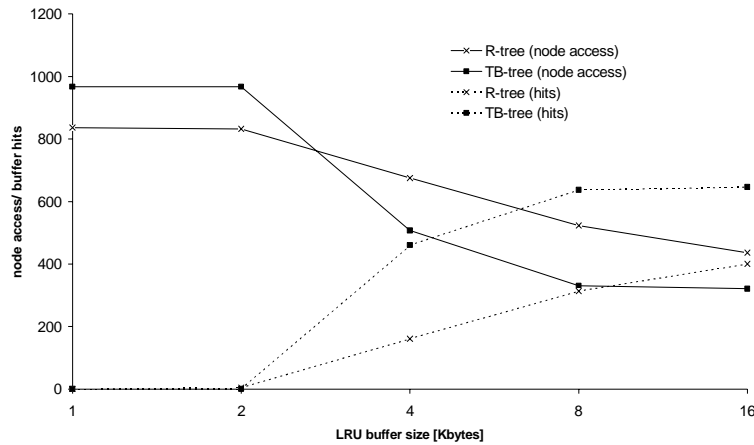


Figure 8: Query processing under varying LRU buffer size

segment QW. The rationale is that we “chop” those parts away from QW that range over infrastructure, i.e., those parts of the data space that do not contain trajectory data.

We devise an algorithm for segmenting the QW based on infrastructure. This segmentation can have three outcomes. Query processing can be (i) stopped after the pre-processing step, i.e., QW is totally covered by infrastructure, or (ii) QW is segmented into a set of smaller query windows,  $q_{wi}$ , which is used for querying the trajectory data, or (iii) the original range query is used. Case (i) is easy to decide. For cases (ii) and (iii), we depend on heuristics that are based on the outcome of the segmentation process. The results of the performance studies give a first indication for such heuristics.

Although recent literature includes work on indexing trajectories of moving objects by maintaining the complete history of object movement [10] [17] [18], the work presented in this paper is the first

- to propose a query processing technique tailored towards trajectory data stemming from objects moving in scenarios constrained by infrastructure, and
- to use a pre-processing step that is based on data other than approximations of the trajectory data (infrastructure vs. approximation).

This work points to the following future research directions. Using the outcome of the segmentation process directly might not be the most favorable choice. We can extend the segmentation algorithm to combine various query windows of  $q_{wi}$  into larger ones. This will combine query window segmentation with the simultaneous execution of query windows [12].

Although we distinguish three cases as the outcome of the segmentation process, clear heuristics have to be derived when to apply each case. Also, the framework is only empirically validated. Analytical studies should be used to back up the results.

In this work, we only used synthetic trajectory and infrastructure data. It would be interesting to study the performance of this approach using real data sets.

## References

- [1] Barbará, D.: Mobile Computing and Databases – a Survey. *IEEE Transactions of Knowledge and Data Engineering*, 11(1), pp. 108-117, 1999.
- [2] Bartels, R., Beatty, J., and Barsky, B.: *An Introduction to Splines for Use in Computer Graphics & Geometric Modeling*. Morgan Kaufmann Publishers, Inc., 1987.
- [3] Brinkhoff, T., Kriegel, H. P., Schneider, R., and Seeger, B.: Multi-Step Processing of Spatial Joins. In *Proceedings of the 1994 ACM SIGMOD Conference on Management of Data*, pp. 197 - 208, 1994.

- [4] Jagadish, H. V.: Linear Clustering of Objects with Multiple Attributes. In *Proceedings of the 1990 ACM SIGMOD Conference on Management of Data*, pp. 332-342, 1990.
- [5] Kamel, I. and Faloutsos, C.: On Packing R-trees. In *Proceedings of the 2<sup>nd</sup> Conference on Information and Knowledge Management*, pp. 490-499, 1993.
- [6] Karppinen, J.: Wireless Multimedia Communications: a Nokia View. In *Proceedings of the Wireless Information Multimedia Communications Symposium*, Aalborg University, 1999.
- [7] Leutenegger, S. T. and Lopez, M. A.: The Effect of Buffering on the Performance of R-Trees. In *Proceedings of the Fourteenth International Conference on Data Engineering*, pp. 164-171, 1998.
- [8] Moon, B., Jagadish, H.V., Faloutsos, C., and Saltz, J. H.: Analysis of the Clustering Properties of the Hilbert Space-Filling Curve, *IEEE Transactions on Knowledge and Data Engineering*, to appear, 2000.
- [9] Moore, D.: Fast Hilbert Curve Generation, Sorting, and Range Queries.  
URL:< <http://www.caam.rice.edu/~dougm/twiddle/Hilbert/>>, current as of September, 2000.
- [10] Nascimento, M., Silva, J., and Theodoridis, Y.: Evaluation of Access Structures For Discretely Moving Points. In *Proceedings of the International Workshop on Spatio-Temporal Database Management*, pp. 171-188, 1999.
- [11] Pagel, B. U., Six, H. W., Toben, H., and Widmayer, P.: "Towards an Analysis of Range Query Performance in Spatial Data Structure." In *Processings of the ACM Conference on Principals of Database Systems*, pp. 214 – 221, 1993.
- [12] Papadopoulos, A. and Manolopoulos, Y.: Multiple Range Query Optimization in Spatial Databases. In *Proceedings of the Second East European Symposium on Advances in Databases and Information Systems*, pp.71-82, 1998.
- [13] Pfoser, D. and Jensen, C. S.: Capturing the Uncertainty of Moving-Object Representations, In *Proceedings of the 6<sup>th</sup> International Symposium on Spatial Databases*, pp. 111-132, 1999.
- [14] Pfoser, D. and Theodoridis, Y.: Generating Semantics-Based Trajectories of Moving Objects. In *Proceedings of the International Workshop on Emerging Technologies for Geo-Based Applications*, Ascona, Switzerland, 2000.
- [15] Pfoser, D., Jensen, C. S., and Theodoridis, Y.: Novel Approaches to the Indexing of Moving Object Trajectories. In *Proceedings of the 26<sup>th</sup> International Conference on Very Large Databases*, 2000.
- [16] Sellis, T.: Multiple-Query Optimization. *Transactions of Database Systems*, 13(1), pp. 23-52, 1988.
- [17] Tzouramanis, T., Vassilakopoulos, M., and Manolopoulos, Y.: Overlapping Linear Quadtrees: A Spatio-Temporal Access Method. In *Proceedings of the 6<sup>th</sup> International Symposium on Advances in Geographic Information Systems*, pp. 1-7, 1998.
- [18] Vazirgiannis, M., Theodoridis, Y., and Sellis, T.: Spatio-Temporal Composition and Indexing for Large Multimedia Applications. *Multimedia Systems*, 6(4), pp. 284-298, 1998.

## Appendix

**Segment Query Window** algorithm

**Input:** rectangle QW  
array of rectangles (AOR) infra  
int nof-ielems

**Output:** AOR qws  
int nof-qwelems

**begin**

```
01  Set nof-elems = nof-ielems
02  Set origin = lower-left corner of QW
03  // find degenerated seeds
04  for nof-ielems do
05      if infra[i] touches left side of QW then
06          add upper-left corner of infra to seeds
07      if infra[i] touches bottom side of QW then
08          add lower-right corner of infra to seeds
09  end-for
10  // try to include origin in seeds
11  if origin is free then
12      add origin to seeds
13  end-if
14  // find segments as long as there are seeds
15  while seeds not empty do
16      Set seed = closest of seeds to origin
17      for nof-ielems do
18          x-dist[i] = distance in the x-direction from seed to
infra[i]
19          y-dist[i] = distance in the y-direction from seed to
infra[i]
20      end-for
21      // find solutions in the x-direction
22      Sort infra according to smallest x-dist
23      add to solutions infra with smallest x-dist
24      for nof-elems do
25          if infra[i] has larger x-dist but smaller y-dist then
26              add to solutions infra[i]
27          end-if
28      end-for
29      // find solutions in the y-direction
30      Sort infra according to smallest y-dist
31      for nof-elems do
32          if y-dist of infra[i] is valid and x-dist is invalid
33              if nof-solutions == 1
34                  add to solutions infra[i]
35              end-if
36          else for nof-solutions do
37              if y-dist of infra[i] < y-dist of solutions[j]
38                  delete from solutions solutions[j]
39                  add to solutions infra[i]
40          end-if
```



```

41     end-for
42 end-for
43 Sort solutions with respect to y-dist
44 // model for rectangle type;
45 // [0]->x1, [1]->y1, [2]->x2, [3]->y2
46 for nof-solutions - 1 do
47     rect[0] = seed[0]
48     rect[1] = seed[1]
49     rect[2] = solutions[i+1][0]
50     rect[3] = solutions[i][1]
51     p-ratio[i] = rect[2] - rect[0] / rect[3] - rect[1]
52     if p-ratio[i] < 1 then p-ratio[i] = 1/p-ratio[i]
53 end-for
54 // determine segment using the pair of solutions causing
55 // the smallest p-ratio
56 Determine pair i with smallest a/p-ratio
57 upper = solutions[i]
58 right = solutions[i+1]
59 rect[2] = infra[right][0]
60 rect[3] = infra[upper][1]
61 // take certain precautions that the segment is not too
    elongated
62 // in either the x- or the y-direction
63 JudgeShape(rect)
64 // x elongation
65 if elongated in the x-direction then
66     if FindLower(rect) then
67         rect[2] = infra[lower][2]
68     else
69         rect[2] = infra[upper][2]
70     end-if
71     // determine additional seed points
72     // 6 candidates
73     // 1) upper left corner
74     // 2) lower right corner
75     // 3) intersection with upper constraint
76     // 4) int. with right constraint
77     // 5) int. with add'l upper constraints
78     // 6) int. with add'l right constraint
79     // only cases 1, 2, and 5 apply here
80     // 1)
81     if rect[0] < infra[upper][0] then
82         add to seeds upper-left corner of rect
83     end-if
84     // 2)
85     if rect[2] < QW[2] then
86         add to seeds lower-right corner of rect
87     end-if
88     // 5)
89     for nof-elems do
90         if rect[3] == infra[i][1] AND infra[i][2] < rect[2] then

```

```

91         add to seeds lower-right corner of infra[i]
92     end-if
93 end-for
94 // y elongation
95 else if elongate in the y-direction then
96     if FindLeft(rect) then
97         rect[3] = infra[left][3]
98     else
99         rect[3] = infra[right][3]
100    end-if
101    // 1)
102    if rect[3] < QW[3] then
103        add to seeds upper-left corner of rect
104    end-if
105    // 2)
106    if rect[1] < infra[right][1] then
107        add to seeds lower-right corner of rect
108    end-if
109    // 6)
110    for nof-elems do
111        if rect[2] == infra[i][0] AND infra[i][3] < rect[3] then
112            add to seeds upper-left corner of infra[i]
113        end-if
114    end-for
115 end-if
116 // shape of rect is approved
117 else
118     // 1)
119     if rect[0] < infra[upper][0] then
120         add to seeds upper-left corner of rect
121     end-if
122     // 2)
123     if rect[1] < infra[right][1] then
124         add to seeds lower-right corner of rect
125     end-if
126     // 3)
127     if infra[upper][2] < rect[1] then
128         add to seeds lower-right coner of infra[upper]
129     end-if
130     // 4)
131     if infra[right][3] < rect[3] then
132         add to seeds upper-left corner of infra[right]
133     end-if
134     // 5)
135     for nof-elems do
136         if rect[3] == infra[i][1] AND infra[i][2] < rect[2] then
137             add to seeds lower-right corner of infra[i]
138         end-if
139     end-for
140     // 6)
141     for nof-elems do

```

```
142         if rect[2] == infra[i][0] AND infra[i][3] < rect[3] then
143             add to seeds upper-left corner of infra[i]
144         end-if
145     end-for
146 end-if // from line 63
147 end-while
end.
```