Querying the Trajectories of On-Line Mobile Objects

Dieter Pfoser
Department of Computer Science
Aalborg University
Fredrik. Bajers Vej 7E
DK-9220 Aalborg Øst, DENMARK
+45-96359973
pfoser@cs.auc.dk

Christian S. Jensen
Department of Computer Science
Aalborg University
Fredrik. Bajers Vej 7E
DK-9220 Aalborg Øst, DENMARK
+45-96358900
csj@cs.auc.dk

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.

Keywords

trajectory, moving objects, query processing, constrained movement, query window segmentation.

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 [16] 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 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 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. The simplest 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 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 2 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].

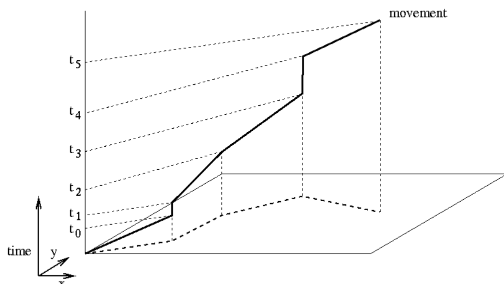

Figure 1: A moving point object trajectory

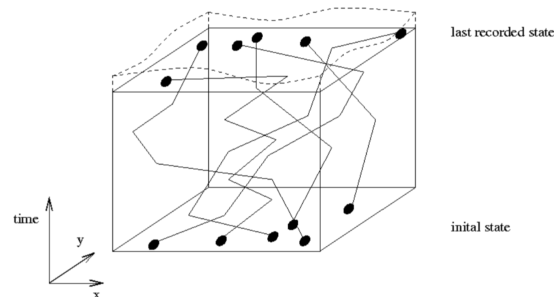

Figure 2: A spatiotemporal space with several trajectories

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 3 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 2. The data was generated using the GSTD tool [15].

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

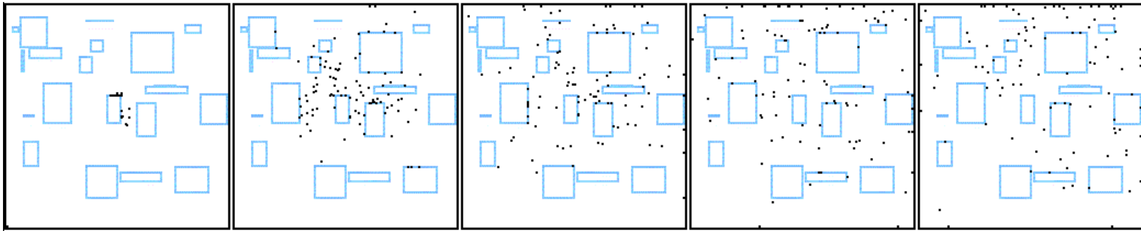

Figure 3: Moving object snapshots and infrastructure

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 OBJECT DATA

Trajectories and the relevant queries require new query processing techniques. In previous work, the focus was on the design of new access methods [16]. In the following, we devise a new query processing technique, which is based on techniques 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 entities in the database have to be examined to decide whether they belong in the final result [3].

Section 2.2 states that where there is infrastructure there cannot be any moving objects. A query window might range over infrastructure, thus requiring us to query "empty" space. It should be noted 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, termed query window segmentation. We only want to consider those parts of the query window, QW, not ranging 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 obtain a set of smaller query windows, qwi, 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 needed 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, qwi.
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 bulk load such an index (cf. [5]).

3.2 Query Window Split Algorithms

An essential part of the three-step technique involves the segmentation of the query window. Before we can devise an algorithm for this task, we have to establish what is an optimal query window, i.e., which shapes of query windows should this segmentation algorithm 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, but is 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 denotes the sum of all the areas of the nodes of the tree, and 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. Having two query windows with the same aerial extent, the one 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 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., as square as possible. In Figure 4 where few but large infrastructure elements are shown as black rectangles, the possible outcome of such a segmentation process is shown as white rectangles.

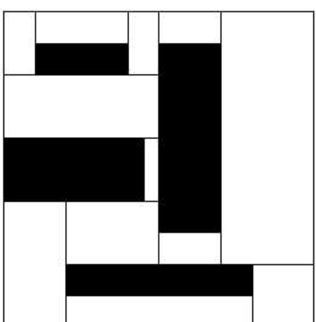

Figure 4: A query window segmentation example

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 5(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. As a result, A and B are the two candidate rectangles for this step.

Seed points are the lower-left corners of all rectangles. Seed points are determined in two stages. Initially, the algorithm finds all seed points on the left and lower side of the query window (they would not be found otherwise). Further, more seed points are found during the course of the algorithm when new rectangles are segmented, i.e., the resulting rectangles of the segmentation process generate new seed points. The algorithm terminates when all seed points have been considered.

3.2.3 Segmentation Algorithm—a Detailed View

In the following, we give a more detailed description of the algorithm. The pseudo code of the algorithm can be found in [15].

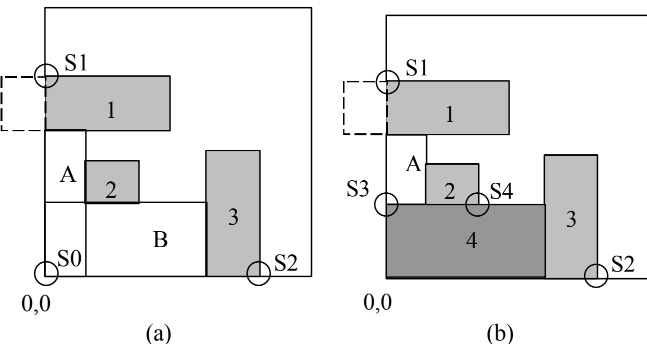

Figure 5: Two steps in the query window segmentation process

Initially, the algorithm determines a first 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. Subsequently, the algorithm iterates over all seed elements, initial ones and newly computed ones, to compute query window segments.

To determine a valid query window segment, the algorithm tries to find infrastructure elements that bound a rectangle whose origin is the current seed point. In the example in Figure 5(a), the algorithm determines three candidate bounds. Elements 1, 2, and 3 constrain a rectangle originating from seed point S0. Those constraints leave us with two possible rectangles, A and B. 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. 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. The outcome of this step can be that the shape is elongated in the x-direction, in the y-direction, or the shape is acceptable, i.e., it is reasonably close to a square.

The type of shape determines how to compute the final rectangle and new seed points. If the rectangle is *elongated in the* x-*direction*, the rectangle is possibly shortened such that it ends with its upper constraint (element 1 in Figure 6(a)) or an element constraining the rectangle from below (element 2 in Figure 6(a)). The algorithm chooses the element that is the most restrictive. In the example of Figure 6(a) that is element 1. If neither element is restrictive, i.e., if both elements have larger y-extensions than our rectangle, the rectangle is left unaltered. A similar approach applies in case the rectangle is elongated in the y-direction.

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 6(a)).

As for new seed points, Figure 6(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.

Remember, seed points are the lower-left corners of future rectangles, thus they can only be 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).

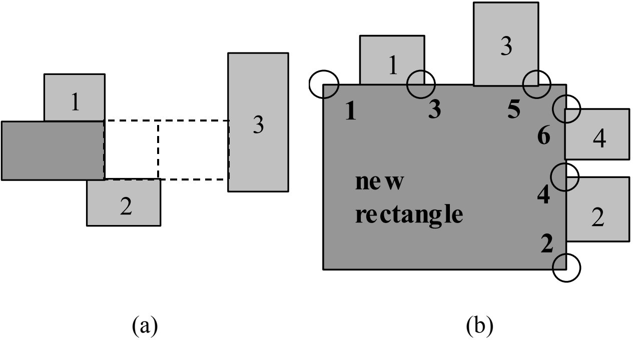

Figure 6: (a) elongated rectangles, and (b) seed points

The algorithm has to check all six alternatives in case of a well-shaped query window. 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.

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, equals Q, 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 $\mathrm{I} \log \mathrm{I}$, whereas the cost for the last is $(\mathrm{Q}+\mathrm{I}-1)\log(\mathrm{Q}+\mathrm{I}-1)$. To compute the total cost we compute the sum of an arithmetic series.

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

The running time is therefore $O(n^2 \log^2 n)$.

3.3 Query Window Segmentation and Indexing

The second step of the three-step technique uses 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) [16]. The TB-tree possesses special capabilities in processing spatiotemporal query types (cf. [16]). Segments in the TB-tree are grouped together based on the trajectories they belong to. The R-tree does not preserve trajectories, but 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, where a newly referenced page replaces the "has-not-been-referenced-for-the-longest-time" page. This scheme exploits the 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 using a space-filling curve, namely the Hilbert curve. Basic properties of space-filling curves are: (i) they cover an "area" completely, where area might also refer to higher dimensional volumes, (ii) each point in space is visited exactly 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 ordering and Gray-code curve. Further experiments [4] give similar results.

The Hilbert curve seen in Figure 7 is constructed in a self-similar way by using rotation and mirroring. Algorithms for the

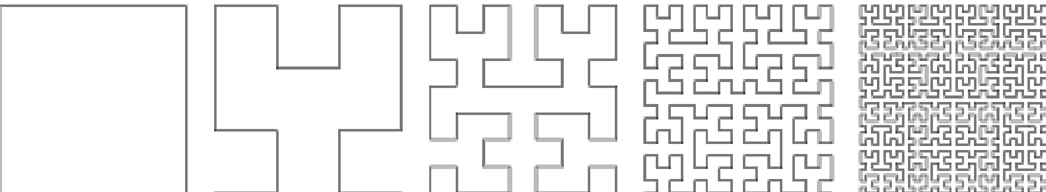

Figure 7: Hilbert space filling curves

Table 1: Experimental results: various query window sizes and datasets

Experiment	1		2		3	
QW (R/TB-tree)	175	196	123	167	89	141
N	1		1		1	
qwi (R/TB-tree)	118(0)	142(0)	100(0)	134(0)	87(0)	122(0)
Visualization						
Experiment	4		5		6	
QW (R/TB-tree)	319	289	166	172	486	368
N	7		56		7	
qwi (R/TB-tree)	319(393)	289(678)	166(1742)	172(4584)	436(400)	321(646)
Visualization						

construction of space filling curves can be found on the Web [9] and in the literature [4].

4. Experimental Studies

The goal of the 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) is the best for processing spatiotemporal range queries. Second, segmenting query windows might prove to be more or less beneficial for different access methods. Here, we consider 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.

Initially, we use an artificial set of infrastructure elements as shown in Figure 8. The real-world correspondence of this infrastructure composition could be a city with city blocks. We create trajectories for 500 moving objects that are uniformly distributed over the whole data space. A trajectory itself consists 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 8 shows a temporal snapshot of the trajectory data used in the following experiments. The infrastructure elements are shown as gray rectangles. Again, we use GSTD++ [15] to generate trajectory data. The parameters are chosen 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.

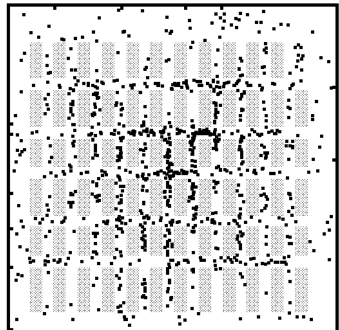

Figure 8: Trajectory dataset snapshot

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 observe that with an increasing query window size, the advantage of segmentation decreases. Also, it seems that only infrastructure elements that are on the border of the query window matter. Experiments 1, 2, and 3 show that although N is the same in all three cases, because the corners of the infrastructure coincide with the query window in experiment 1, the gap between using QW and qwi is larger than in experiments 2 and 3, where the boundary of QW is inside the infrastructure elements. The larger the part of the query window boundary that is inside the infrastructure, the smaller is the advantage of segmentation (experiments 2 and 3). In experiment 4, we extend QW such that no infrastructure intersects with the boundary of the query window. Here, segmentation offers no advantage any more.

In comparing the two access methods, we see that segmentation is beneficial for the TB-tree in more situations than for the R-tree. In experiment 3, 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 more 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 [15]. The parameters of the trajectory data are, again, 250k segments stemming from 500 moving objects uniformly distributed over the data space. Experiment 5 in Table 1 shows the experimental outcome. The infrastructure scenario consists of many (900), but small elements. Consequently, the segmentation process produces many, small query windows (56). In this case, choosing qwi over QW offers no advantages in terms of query processing performance. This shows that the number and size of infrastructure elements determine the efficiency of our approach.

4.2 Varying LRU Buffer Sizes

To show the effects of varying LRU buffer sizes, we choose experiment 6 in Table 1 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, the 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, if reducing the LRU buffer size, we reduce the advantage of using the segmented query windows over the original window. Figure 9 shows the number of page accesses and, conversely, the number of buffer hits when varying the LRU buffer size from 1 to 16 Kbytes.

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 the corresponding R-tree. Consequently, the TB-tree benefits more from a larger buffer than the R-tree does.

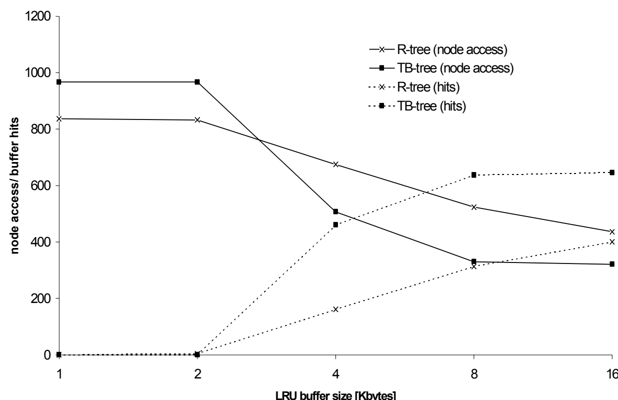

Figure 9: Performance study for varying LRU buffer size

4.3 Summary of Experiments

We can identify the following parameters that determine the effectiveness of query window segmentation. First, the larger the number of segmented query windows, qwi, the smaller the advantage over QW. Second, the more space infrastructure occupies within QW, the better. Third, the more of the infrastructure that is concentrated at the boundaries of QW, the better. The experiments show 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 a larger LRU buffer 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 segment QW. The rationale is that we “chop” away those parts of 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, (ii) QW is segmented into a set of smaller query windows, qwi, 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 reported 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] [18] [19], the work presented in this paper is the

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

This works points to several directions for future research. 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 qwi 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 for when to apply each case. Also, the framework is only empirically validated. Analytical studies should be used to back up the results. Finally, this work only used synthetic trajectory and infrastructure data. It would be interesting to study the performance of this approach using real data sets.

6. ACKNOWLEDGEMENTS

This work was supported by the Chorochronos project, funded by the European Commission DG XII, contract no. ERBFMRX-CT96-0056, and by a grant from the Nykredit Corporation.

The authors wish to thank Yannis Theodoridis and Nectaria Tryfona for many insightful discussions and the anonymous reviewers for their comments, which improved the paper.

7. REFERENCES

[1] Barbará, D.: Mobile Computing and Databases—a Survey. *IEEE Transactions of Knowledge and Data Engineering*, 11(1): 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, 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 2nd 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 14th 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. www.caam.rice.edu/~dougm/twiddle/Hilbert/, current as of April 12, 2001.

[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 6th International Symposium on Spatial Databases*, pp. 111–132, 1999.

[14] Pfoser, D. and Jensen, C. S.: Querying the Trajectories of On-Line Mobile Objects. TimeCenter Technical Report TR-55, www.cs.auc.dk/TimeCenter, current as of April 12, 2001.

[15] 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*, pp. 59–76, 2000.

[16] Pfoser, D., Jensen, C. S., and Theodoridis, Y.: Novel Approaches to the Indexing Moving Object Trajectories. In *Proceedings of the 26th International Conference on Very Large Databases*, pp. 395–406, 2000.

[17] Sellis, T.: Multiple-Query Optimization. *Transactions of Database Systems*, 13(1): 23–52, 1988.

[18] Tzouramanis, T., Vassilakopoulos, M., and Manolopoulos, Y.: Overlapping Linear Quadtrees: A Spatio-Temporal Access Method. In *Proceedings of the 6th International Symposium on Advances in Geographic Information Systems*, pp. 1–7, 1998.

[19] Vazirgiannis, M., Theodoridis, Y., and Sellis, T.: Spatio-Temporal Composition and Indexing for Large Multimedia Applications. *Multimedia Systems*, 6(4): 284–298, 1998.