University of Alberta

TRAJECTORY SPLITTING MODELS FOR EFFICIENT SPATIOTEMPORAL INDEXING

by

Slobodan Rasetic ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science.**

Department of Computing Science

Edmonton, Alberta, Canada

Spring 2005

University of Alberta

Library Release Form

Name of Author: Slobodan Rasetic

Title of Thesis: Trajectory Splitting Models for Efficient Spatiotemporal Data Indexing

Degree: Master of Science

Year this Degree Granted: 2005

Date: ____________

University of Alberta

Faculty of Graduate Studies and Research

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled **Trajectory Splitting Models for Efficient Spatiotemporal Data Indexing** submitted by Slobodan Rasetic in partial fulfillment of the requirements for the degree of **Master of Science**.

____

Dr. Joerg Sander

____

Dr. Davood Rafiei

____

Dr. Arturo Sanchez-Azofeifa

Date: ______________

To My Wife

Abstract

In this work we address the problem of splitting trajectories optimally for the purpose of efficiently supporting spatiotemporal range queries using an R-tree framework. In particular, we minimize the expected number of disk I/Os required to answer a query. Previous works have addressed this problem by concentrating on methods that use novel access structures or methods that reduce the volume of trajectory approximations (e.g. MBRs). We show that splitting trajectories with the goal of minimizing volume does not necessarily lead to the best query performance. We derive a model for estimating the number of expected I/Os with respect to a given query size for an arbitrary split of a trajectory. Based on the proposed model, we also introduce a dynamic programming based algorithm for splitting a set of trajectories that minimizes the number of expected disk I/Os with respect to an average query size. In addition, we develop a linear time, near optimal solution for this problem which can be used in a more realistic dynamic case where trajectory points are continuously arriving. Our experimental evaluation confirms the effectiveness of the proposed trajectory splitting policies when embedded into an R-tree structure.

Acknowledgements

Special thanks to Dr. Joerg Sander, my fellow college James Elding and Dr. Mario Nascimento for all the support and contribution to this work.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Producing and collecting large volumes of spatiotemporal data has become more practical in recent years, leading to increased availability and the need for efficient management of this type of data. Therefore, there has been an increasing demand to develop database solutions, which can manage large volumes of spatiotemporal data. For many industries, it is important to be able to track, store, and query information about moving objects over time, for instance, to deliver real time services to clients based on spatial and temporal context. New technologies such as GPS and telecommunication technologies allow us to pinpoint the location of objects in space at a certain time. Examples of such domains include wireless communication networks such as Global Positioning Systems (GPS) and cellular telephones, wildlife tracking experiments, traffic routing problems, fleet control, and mobile computing.

For many industries, it is becoming increasingly important and often critical to track and record the positions of their clientele. An example of such is for the delivery of real time emergency services, such as ON STAR and E911. Within this domain it is imperative to frequently update the last known position of an object as well as continue to maintain all of its historical localities. Object movements, however, tend to be continuous, and from a storage perspective it is not feasible to capture and record all of the positions of its lifetime. Sampling methods can therefore be applied to reduce storage overhead. Sampling the locations of continuously moving objects involves recording their positions at discrete points in time. Therefore, spatiotemporal data in its most general form consists of observations with their timestamps and spatial location. The position of an object can be sampled and recorded as a single point *p* in *n*-dimensional space, i.e. $p=(x_1, x_2,\dots, x_n)$. For simplicity we assume that an object's spatial movements occur over the 2-dimensional plane. Apart from simply recording an object's position it is often equally important to reconstruct the path that an object followed over a given interval of time, which is called a trajectory.

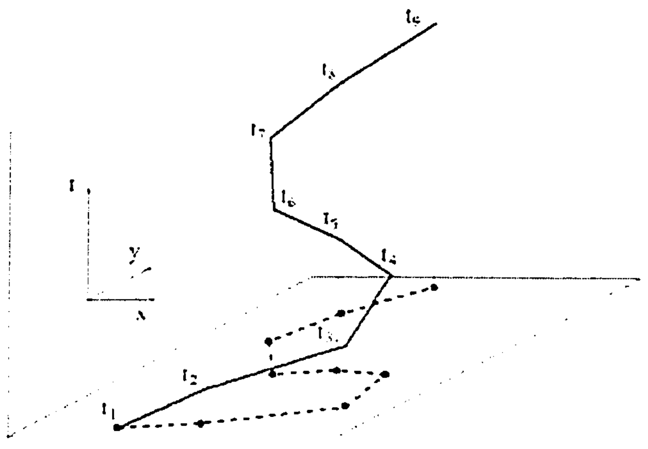


Figure 1. A reconstructed object trajectory composed of its segments [14]

Trajectories of moving objects are sequences of positions recorded at discrete points in time. A linear interpolation between two successive locations is typically assumed. A trajectory T is represented as a sequence of 3-dimensional points $\langle (x_1, y_1, t_1), (x_2, y_2, t_2), \ldots, (x_k, y_k, t_k) \rangle$, where (x_i, y_i) is a spatial location, and t_i is a time instant as illustrated in Figure 1. A consecutive sequence of points of a given trajectory T is called a *segment* of T. A segment of length 1, i.e., consisting of only 2 consecutive points, is called an *elementary segment.*

The rate at which an object's motion is sampled will often guide the quality of reconstructed trajectories. Reconstructing an object's trajectory can be accomplished by applying an interpolation function between successive pairs of sampled points. This operation allows for an objects position to be approximated when its exact location has not been recorded. This method will often lead to some degree of imprecision; however, the amount of data used to represent a trajectory will be dramatically reduced. In our work we assume this model and apply a linear interpolation function between successive pairs of points for the purpose of trajectory reconstruction. For instance, one may have to track the movements of all vehicles in a fleet, sampling their location at regular time intervals. Within each of these domains it is important to store the positions of objects effectively for fast and efficient query support.

Several different types of queries can be expressed over spatiotemporal data. Range-based queries [5][7] can be posed to answer questions about object positions in the past, present, and future. Topological queries [14] can be posed to answer, for instance, questions regarding the traveled distance, heading and average speed of an object during an interval of time. Nearest neighbor queries [20] can be posed to answer questions regarding the similarity based on spatial proximity of moving objects. More complex queries can be constructed by combining the basic query types [14]. Our work concentrates on historical spatiotemporal queries (i.e., we are not concerned about predicting future movements of objects stored in the database). The authors in [14] distinguish two main types of spatiotemporal queries: coordinate-based queries and trajectory-based queries. Coordinate-based queries return only the ids of objects or the count of objects, for instance, the ids of objects whose trajectories intersect a given spatial region during a given time interval. Trajectory-based queries require the exact information about (partial) trajectories, in order to determine possibly complex topological relationships (e.g., whether they cross or bypass a certain area) or navigational information (e.g., what was their top speed and direction within a certain area during a given time interval). To process those queries, usually, one or more range queries are used to extract the relevant trajectory segments from an index.

In order to process spatiotemporal queries efficiently, specialized index structures are needed. It is well understood within the spatiotemporal domain that the type of query, which needs to be supported, will have a direct effect on the make up of the access structure used to index the data. Virtually all spatiotemporal index structures proposed in the literature are derived from spatial index structures such as R-trees. This approach is based on the intuition that spatiotemporal data can be viewed as spatial objects in an extended spatial domain, where time is treated as an additional dimension. In this context, trajectories or trajectory segments are typically represented by minimum bounding rectangles (MBRs). R-Tree based access structures have proven to be effective for supporting range based queries over multi-dimensional spatial data sets [2][6]. Most attempts to efficiently support historical range based queries have focused on modifying the way that trajectories are represented within R-Tree based structures. A primary goal of all approaches is to reduce the expected number of disk accesses required to answer a user query.

One straightforward solution within an R-tree is to approximate each trajectory by a single MBR. This approach, however, yields poor approximations, leading to low query performance in general (except possibly for queries with very large spatial and temporal extent). Another straightforward solution is to approximate each line segment of a trajectory individually by an MBR. Since each line segment can be oriented in only four different ways within an MBR [14], the orientation information and an MBR can be stored within each leaf node entry. In this case, information about each trajectory is completely stored within the R-tree and can be reconstructed without any additional disk I/Os to a separate data level. This type of index is particularly effective for coordinate based queries. However, the size of such an R-tree is, in general, much larger than in the first approach, and the disk I/Os at the directory level are more significant. A more effective alternative is to split trajectories and approximate the resulting sub-trajectories independently by MBRs using a method that tries to overcome the drawbacks of both of the aforementioned solutions. This approach would provide us with a proper balance between the size of the index and the approximation quality, and, as a consequence, should lead to better overall performance.

The problem of splitting trajectories optimally with the goal of minimizing the expected number of I/Os with respect to spatiotemporal queries has not yet been treated rigorously. To our best knowledge, the only work that addresses the problem of splitting a set of trajectories to improve query performance is presented in [7]. The authors assume a predetermined total number of allowed splits for a static set of trajectories, and propose a solution that distributes the splits among the given trajectories so that the total volume of the resulting MBRs is minimized. We argue, and our experiments confirm, that such an optimization goal does not necessarily lead to the best query performance. Our main contributions in this work are the following:

- We derive an analytical cost model for evaluating the split of a trajectory into segments. Given a spatiotemporal range query, this model estimates the expected number of I/Os due to the MBR approximations of the resulting segments.
- Based on the proposed cost model for evaluating trajectory splits, we introduce a dynamic programming algorithm for splitting a set of trajectories so that the number of expected disk I/Os is globally minimized with respect to a given spatiotemporal range query. This method assumes that complete trajectories are available when constructing the index.
- In order to deal with cases where trajectories are incomplete and updated incrementally, we develop another cost model that estimates an optimal length for segments when "incrementally" splitting a trajectory with respect to a given query.

- Combining the two above cost models, we derive a linear time algorithm for splitting trajectories that leads to a query performance which is close to the performance of the optimal algorithm in practice, and can be used in dynamic cases where individual trajectories are incrementally extended.
- Finally, we demonstrate through an extensive experimental evaluation that our algorithms are both efficient in practical situations and significantly outperform previous approaches.

The rest of the thesis is organized as follows. Chapter 2 presents the background and motivation. In Chapter 3, we derive a formal cost model for evaluating the quality of trajectory splits and propose an algorithm for finding the optimal split with respect to this cost model. In Chapter 4, a linear time algorithm for splitting trajectories heuristically is formally derived. An extensive experimental performance evaluation and comparison is presented in Chapter 5 and Chapter 6 concludes the thesis.

Chapter 2

Background and Motivation

The R-tree [6] is typically used to organize multi-dimensional spatial objects using minimum bounding hyper-rectangles (MBRs) as approximations. In this section we discuss the R-Tree indexing structure with respect to spatial and spatiotemporal data in more detail. In addition, we discuss various spatiotemporal indexing structures that have been developed over the past few years. We also motivate trajectory splitting using a spatiotemporal query as a driving parameter with the goal of minimizing the expected number of disk I/Os.

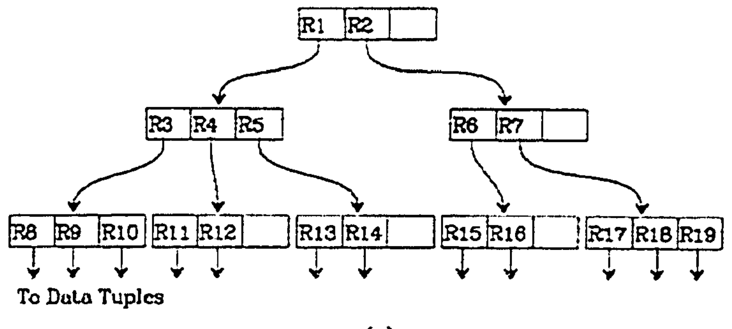
2.1 The R-Tree

The R-tree leaf nodes store the MBRs of data objects and a pointer to the exact object's geometry. Internal nodes store a sequence of pairs consisting of an MBR and a pointer to a child node. To answer a range query, starting from the root, the set of MBRs intersecting the query range is determined, and then the corresponding child nodes are searched recursively until the data pages are reached.

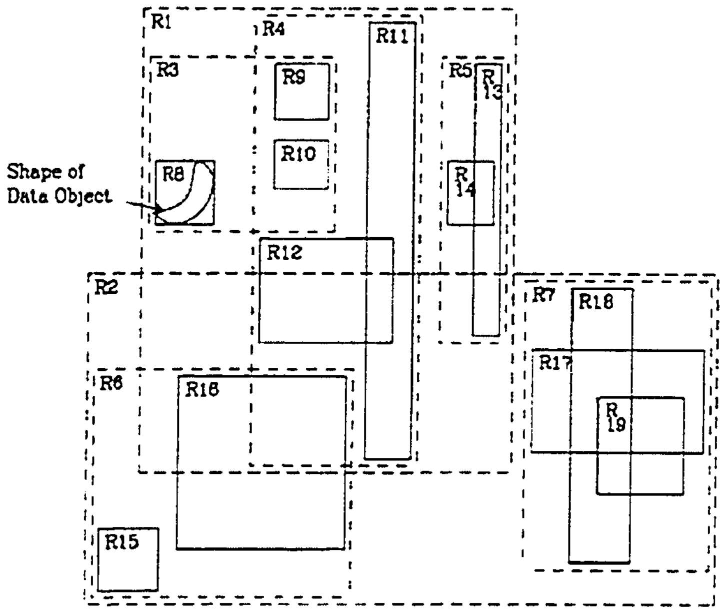
The key issue for efficient query performance is reducing the amount of overlap between MBRs in the tree and the empty space covered by MBRs. The amount of overlap influences the number of sub-trees that have to be traversed at query time. Empty space in MBRs can lead to unnecessary page accesses if the query intersects only the MBRs' empty space but not any part of the objects approximated by those MBRs. In the following, we discuss the R-Tree indexing structure in more detail.

2.1.1 The R-Tree and Spatial Data

The R-Tree is an efficient height balanced access structure capable of storing spatial data objects. The R-Tree was initially intended to support spatial search queries in applications such as computer aided design (CAD). It is well understood that classical one-dimensional database indexing structures cannot efficiently support spatial search over multi-dimensional datasets [6]. One of the difficulties in representing spatial objects directly in a database is that their geometry might be too complex to index. As a result an object's convex hull might be better represented using a simpler approximating polygon. Using less coarse approximations to represent complex spatial objects will have the obvious benefit of reducing the amount of storage space required for the index. A major drawback is that the approximation might not be too accurate; this can lead to discordance between the approximation and the exact object itself. The R-Tree utilizes bounding boxes, recursively throughout the tree, to approximate spatial objects. An R-Tree index used on 2-D spatial objects is illustrated in Figure 2.



(a)



(b)

Figure 2. The R-Tree [6]

The R-Tree is designed to index spatial objects by approximating their exact geometry using a minimum bounding rectangle (MBR). The nodes of the R-Tree contain the spatial coordinates pertaining to each MBR. All the nodes within the R-Tree correspond to a disk page of fixed size, if the index is disk resident. The number of MBRs that are contained in a node depends on the size of the disk page. The leaf nodes of the R-Tree contain MBRs and a pointer to the data tuples, which maintain the exact geometry of the object. Non-leaf nodes are composed of an MBR and a pointer to its child nodes.. In Figure 2 the MBR R8 approximates an object whose geometry is contained within the data tuples that are pointed to from a leaf node. The approximating MBR for R8, R9, and R10 is R3. This approximation scheme continues up the tree until the entire data space is indexed. By observing Figure 2 it should be apparent that the quality of the approximation degrades in the higher levels of the tree. MBRs can be inserted into or deleted from the leaf nodes of the R-Tree. Because disk pages might grow too large or become too small the R-Tree must often reorganize its structure when objects are inserted or deleted. For brevity, we do not discuss insert and delete operations in this work, however they can be found in [2][6]. The primary goal of R-Tree insertion and delete operations is to minimize the total area, covered by all of the resulting MBRs, i.e. a partitioning of MBRs within nodes such that there is minimal dead space. This idea is illustrated in Figure 3 that shows two different ways of splitting a node containing four MBRs. The original node itself is described as an MBR. In Figure 3a, after splitting a node into two MBRs, the sum of the areas of the resulting MBRs is significantly larger than in the case depicted in Figure 3b. Therefore, if the data space is partitioned poorly, the resulting MBRs will cover large fractions of empty space. Both Linear-Cost and Quadratic-Cost R-Tree's node splitting heuristics [6] rely on calculating the areas of the MBRs participating in the splitting as well as the areas of the resulting nodes. The R*-Tree node splitting policy [2] goes a step further in order to find a better split. It takes into account the *area-value*, *margin-value*, and *overlap-value* that can be determined using the extents of MBRs in each dimension. It is worth noting that the insert and delete operations used to maintain the R-Tree are heuristics and are not guaranteed to reduce the dead space optimally. Reducing the dead space is important for efficiently supporting spatial range queries.

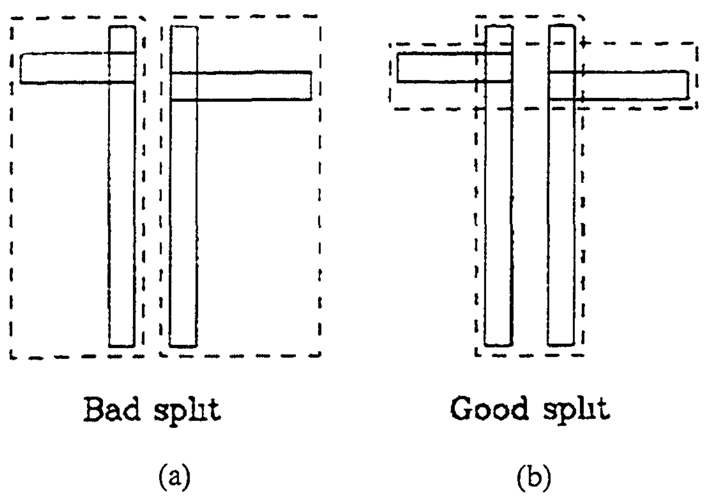


Figure 3. A bad (a) and good (b) split of the spatial objects [6]

For our work, reading a page from a disk is synonymous to making a disk I/O. To process a range query using an R-Tree index, we start from the root node and check all the MBRs contained in the root node for the intersection with the range query. For each of the intersecting MBRs, the search continues recursively down the corresponding sub-tree until there are no longer any intersecting MBRs to check. If a range query intersects an MBR in the leaf node then the data tuples must be examined to determine whether the range query intersects the exact geometry of the corresponding object. In addition to disk I/Os, this operation is often the most expensive process of a range query search since it invokes expensive computational geometry algorithms. Therefore, it is important to approximate the data in a way that reduces the probability that a data tuple must be examined when it is not necessary. In the next section, we show how spatiotemporal data has been incorporated into the R-Tree framework.

2.1.2 The R-Tree and Spatiotemporal Data

Spatiotemporal data can be considered as a general instance of multi-dimensional data. The R-Tree is effective for indexing multi-dimensional spatial datasets [2][6]. While the use of R-Trees and MBRs has proven successful for query support in spatial domains, it remains to be an open question whether these solutions should be directly applied to spatiotemporal datasets. Spatiotemporal data possesses the interesting peculiarity that its spatial boundaries are generally well defined and closed while its temporal domain is strictly increasing and generally unbounded. This property often results in large volumes of dead space when trajectories are approximated by MBRs. Copious amounts of dead space are generally not observed when using MBRs to approximate spatial data because a spatial object's extents are often well defined, closed, and static. Generally, trajectories are modeled as sequences of moving objects positions recorded at discrete points in time. A trajectory is then approximated by an MBR. Therefore, almost all of the space used to approximate a trajectory will be empty. This might lead to a condition where a user query will often only intersect the dead space contained in an MBR and not the trajectory itself resulting in a so-called false hit.

By reducing the dead space of a trajectory approximating MBR fewer false hits should be observed. Analytically and empirically determining the probability of a false hit is beyond the scope of this work. This probability is dependent on the amount of dead space used to approximate the trajectory and the size of the user query. However, a trajectory can be approximated in such a way that minimizes the probability that a user query intersects an MBR. When a user query intersects an MBR the sub-trees of an R-Tree must be examined. In the worst case, the number of disk I/Os will be high at the leaf nodes because the data tuples must then be examined. An ideal goal is to reduce the probability of a query intersecting an MBR at the highest levels of the R-Tree. Consequently, smaller number of sub-trees would have to be inspected thus reducing the number of disk I/Os required to answer the user query.

Many different approaches have been proposed for reducing the number of disk I/Os required to answer a user query. In many works this involves tightly approximating the trajectories themselves, i.e. MBRs of the leaf nodes. The quality of the approximation at the leaf nodes will often dictate the quality of the approximation at the higher levels of the R-Tree. Another approach towards the goal of minimizing the probability of an MBR intersecting a user query is to decouple the temporal and spatial dimensions of the index. In this approach several indexes are maintained. Each index might then contain fewer trajectory approximations; as a result there should be less probability in intersecting an MBR. All of the solutions are oriented towards reducing the number of disk I/Os required to answer a user query. None of the approaches, however, is designed to solve this problem with respect to the size or distribution of the user queries.

2.2 The Spatiotemporal Indexing Structures

Spatiotemporal indexing structures proposed in the literature are virtually all based in one way or another on R-trees [9] and can be classified into three main approaches. In the first approach, time is simply treated as an additional spatial dimension [17]. For trajectories, however, this approach results in inefficient indices since the MBRs tend to be very large, cover large portions of empty space and lead to a high degree of overlap among the MBRs. Another structure that falls under this type of approach is the TB-tree [14]. Its insertion split strategy is oriented towards trajectory preservation such that a leaf node only contains segments that belong to the same trajectory. The main disadvantage of this approach is that "concessions to the most important R-tree property, node overlap" must be made. Indeed, experimental results in [14] show that it is outperformed by a regular R-tree for spatiotemporal range queries, in particular for small queries.

In the second approach, time and space are treated differently within a combined indexing scheme [5]. A two level index is proposed. On the upper level the space is partitioned into non-overlapping cells. Additionally, for each of these cells, a separate R-Tree is used to index the trajectory segments of objects in time dimension. The main drawback of this approach is that it is unclear how to partition the space for practical applications. If the partition is too coarse, the temporal index in each cell is not very selective, whereas if it is too fine, a large number of temporal indices have to be accessed for each query.

The third approach also treats time differently from space. The idea is to have virtual and incrementally maintained 2-dimensional R-trees for each point in time [11]. However, this approach suffers from a prohibitively large overhead when indexing very dynamic scenarios, and is not suited for trajectory data.

Most of the recently published work for indexing trajectories has been aiming at improving the first approach. Two orthogonal strategies have been investigated: replacing MBRs by different geometric approximations, and splitting trajectories. As mentioned earlier, approximating a trajectory using a single MBR typically results in an MBR covering a large portion of empty space. This may lead to a large degree of overlap between the trajectory MBRs, which in turn degrades the R-tree performance. In order to alleviate this problem, a possible approach is to use tighter approximations other than MBRs. This line of reasoning has been investigated for indexing spatiotemporal trajectories in [21]. In that paper the authors propose to trim the corners of the trajectories' MBRs in order to obtain a bounding octagon prism, instead of a box. The experimental results however, do not provide clear evidence that a considerable gain is obtained for spatiotemporal range queries, when compared to the R*-tree. Another strategy to improve approximations of trajectories is via a splitting process. A trajectory is split into trajectory segments by choosing one or more split points. We assume split points to be only at one of the observed (discrete) time points. In this way, a complete trajectory is no longer approximated by a single MBR, but rather by several

MBRs, each approximating one trajectory segment. The advantage of doing that is that one is likely to obtain a large decrease in the amount of indexed empty space, and consequently better query performance. An important question within this approach is the one of how to split a given trajectory. The work presented in [7] addresses this problem by proposing several algorithms to find split points for trajectories, with the goal of reducing the amount of the approximations' empty space, given a number of split points. The paper does not present a formal model for minimizing the number of I/Os per query, instead it only focus on minimizing the amount of indexed empty space. In fact the presented techniques use as a driving parameter the total number of splits one can use. An open question that the paper does not explore is the one of finding the optimal number of splits given a set of trajectories.

Apart from the problem of indexing spatiotemporal trajectories, several other types of spatiotemporal data and queries have been investigated. There has been some work on answering queries with respect to the future. The work presented in [15], [16] and [8] fall into this category. Nearest neighbor queries have also received attention in the spatiotemporal domain, [20] and [3] are examples of this line of work; the problem of reverse nearest neighbor queries in the spatiotemporal setting has also been addressed in [3]. In [19] the authors use an approach based on linear quad-trees to index a sequence of images as spatiotemporal data. An index for supporting OLAP operation over spatiotemporal data has been proposed in [13]. In the following, we discuss some of the previously mentioned approaches in more detail.

2.2.1 Time as Additional Spatial Dimension

In many of the current spatiotemporal indexing approaches a trajectory is usually treated as 3D spatial data where the 3rd dimension is a temporal dimension. The queries that are imposed on this kind of structure typically involve range (window) queries. A number of other types of queries are also possible in this domain. In order to answer these types of queries efficiently the information of the neighboring segments of the same trajectory need to be preserved. Therefore, the authors of [14] introduce two novel indexing approaches that solve the problem of *preservation of trajectories*. The first one is called spatiotemporal R-tree or STR-tree for short. It attempts to group line segments not only based on spatial proximity but also tries to group segments that belong to the same trajectory together. The other approach, trajectory-bundle tree or TB-tree for short, does not take the spatial proximity into account at all. Its leaves contain the line segments that belong to the same trajectory.

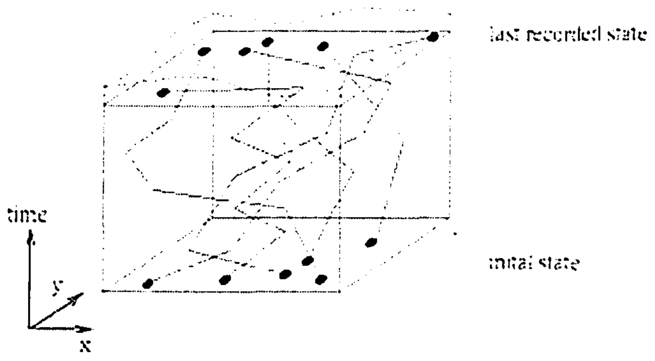


Figure 4. Trajectories of spatiotemporal objects [14]

Representing the movement of objects is illustrated in Figure 4. Each object is represented by a trajectory that grows in time. Each of the objects is sampled at discrete points in time giving us the number of successive positions. Those positions are linearly interpolated to form a trajectory.

As we can see, each trajectory contains both spatial and temporal information of each object. Using this information it is possible to derive additional information such as *speed, traveled distance, heading* etc. Based on this, all kinds of different types of queries could be imposed. These queries can be divided into the following main types:

- *Coordinate based queries*: The typical examples include range, point, time-slice and nearest-neighbor queries.
- *Trajectory based queries*: Those queries are of topological and navigational character and involve derived information such as speed and heading.
- *Combined queries:* The queries that combine *coordinate* and *trajectory* based queries.

To support various query types the authors of [14] introduce a novel indexing structure called the STR-tree which is the extended version of the classical R-tree. It takes into account not only spatial *closeness* but also *trajectory preservation*. The difference between the two is the insertion and split strategy. The insertion algorithm for the STR-tree uses a preservation parameter *p* that represents a number of levels that should be reserved for the trajectory preservation. Using this parameter the line segments that belong to the same trajectory are kept together sacrificing the spatial discrimination. The split algorithm considers four types of different segments:

- *Disconnected segments*: Segments not connected to any other segments
- *Forward (respectively backward) segments*: Segments that are forward, respectively backward connected to other segments belonging to the same trajectory
- *Bi-connected segments:* Segments that are both forwards and backward connected to other segments belonging to the same trajectory.

STR split strategies are illustrated in Figure 5. As we can see from the figure, if the node contains consecutive segments that belong to the same trajectory they are placed in the same node after splitting.

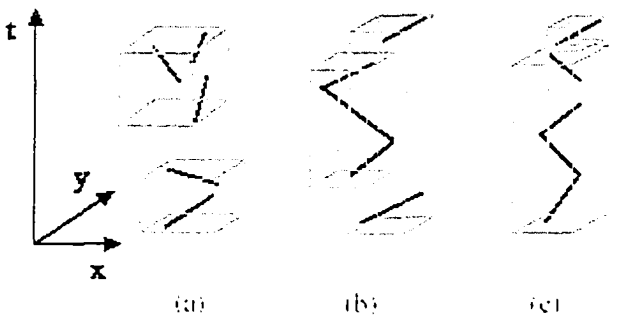


Figure 5. Different Split Strategies [14]

In Figure 5a, all of the segments are *disconnected.* Since none of the segments are connected to any other segment, they are split based on spatial *closeness* only. In Figure 5b and Figure 5c, some of the segments are connected to other segments and after splitting they are placed into the same node consequently sacrificing spatial *closeness* for *trajectory preservation.*

The TB-tree is another indexing structure that takes a more extreme step to preserve trajectories. In this new indexing structure spatial discrimination is sacrificed for *trajectory preservation*. Each of the leaves of the TB-tree contains only segments that belong to the same trajectory. If the trajectory segments span multiple leaves, the leaves belonging to the same trajectory are additionally linked through backward and forward pointers. This structure is illustrated in Figure 6.

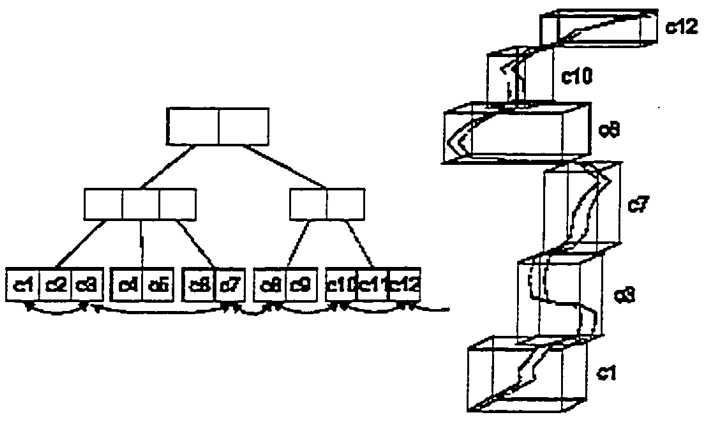


Figure 6. The TB tree structure [14]

Figure 6 shows an example where a whole trajectory that is stored in leaves (C1, C3, C7, C8, C10, C12) is linked through backward and forward pointers. Although the spatial discrimination is sacrificed, the TB-tree structure performs reasonably well even in the case of pure *coordinate-based* queries. In the case of *topological* or *combined* queries, the TB-tree structure allows us to follow the links to extract the additional segments surrounding the segment that was originally returned as a result. In the classical R-tree case we would have to perform additional searches to extract those additional segments.

To evaluate the performance of the STR-tree and TB-tree, they were compared to a regular R-tree structure [14]. The first experiment was conducted using range queries of varying sizes. As expected, the R-tree shows better performance than the other two methods especially with larger numbers of objects. Similar results were obtained in the second experiment involving time slice queries. In the third experiment that involved combined queries, the TB-tree performed one order of magnitude better than the other two.

2.2.2 Decoupling Spatial and Temporal Dimensions

It has been argued that modeling trajectories using an R-Tree will often result in large amounts of dead space in the approximating MBRs. When many trajectories are introduced into the index, the result will often be an increase in the amount of overlapping dead space between MBRs. From a querying perspective, when the number of trajectories in the data space becomes too large, there might be a high probability of intersecting an MBR, but not necessarily a trajectory, which results in unnecessary disk I/Os. One way that this problem has been dealt with is by decoupling the spatial dimensions from the temporal dimension. In the HR-Tree a spatial index is created for every time stamp [11], i.e. a 2-D R-Tree is created for every time stamp.

In SETI [5] several static spatial partitions (cells) are created over the total data space and for each spatial partition a sparse temporal index is maintained using an R-Tree. This technique separates the space and time dimensions and indexes them separately. It partitions the spatial dimensions into a set of separate partitions that do not overlap. The following is the list of key advantages (augmented by our comments) over the existing indexing structures that the authors of [5] identified:

- The objects are indexed in time dimension only. This prevents the indexing structure performance to decrease rapidly as the number of dimensions grows. This is commonly known as "curse of dimensionality" and it is present in some other indexing structures. However, this is not likely to happen in case of spatiotemporal data indexing since the number of spatial dimensions under normal circumstances should not be greater than three.
- In the experiments it is shown that SETI outperforms the 3D R-tree approach. However, the authors of SETI do not clearly indicate whether they compare SETI to the full split 3D R-tree approach where the complete information about each trajectory is stored within the R-tree.
- In the experiments it is also shown that SETI outperforms the TB-tree.
- SETI allows fast additions of new segments through a *front-line* structure.
- SETI is scalable and performs well with large numbers of objects and trajectory sizes.

- Implementing SETI is easy because it is built using existing indexing structures like R-tree. However, the space partitioning into hexagonal cells requires an implementation of a specialized index structure. In addition, a custom query processing technique needs to be implemented as well.

The insertion procedure in SETI uses an in memory structure called *front-line* structure that keeps the last locations of all moving objects. Figure 7 illustrates a number of moving objects and the movement of the front line.

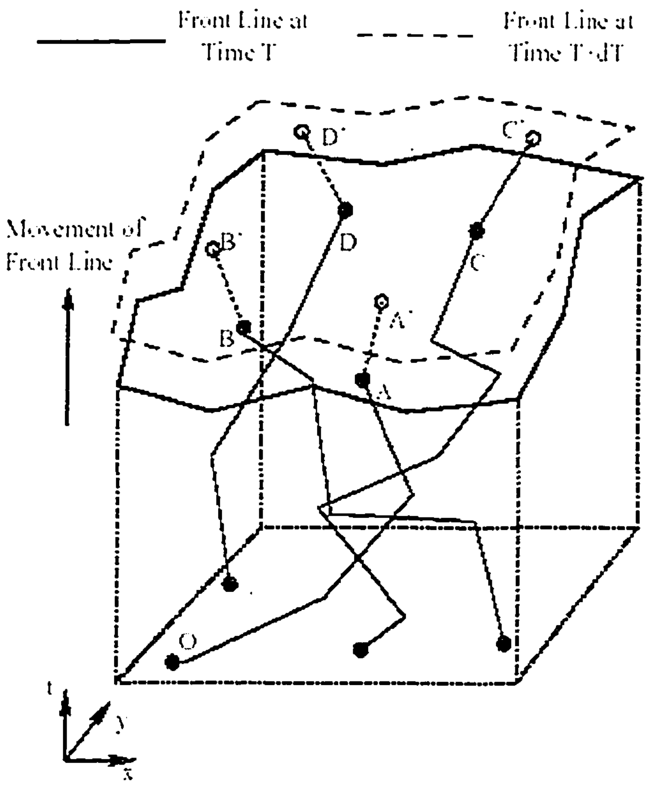


Figure 7. Movement of a front-line structure [5]

Whenever an object location update arrives, its previous location is retrieved and a new segment is formed and inserted into SETI. The space is partitioned into equally sized hexagonal cells. A trajectory segment that is formed needs to be allocated to one of those cells. If the segment spans over multiple partitions, it has to be split into multiple pieces at the border between the cells it spans over. Once split, each piece is inserted in the appropriate cell. Figure 8 illustrates this and shows an original segment AA' on the left side and two pieces that are formed after splitting on the right side AX and XA'. It is important to understand that the splitting is performed in three dimensions even though the splitting in two dimensional spaces is illustrated in Figure 8.

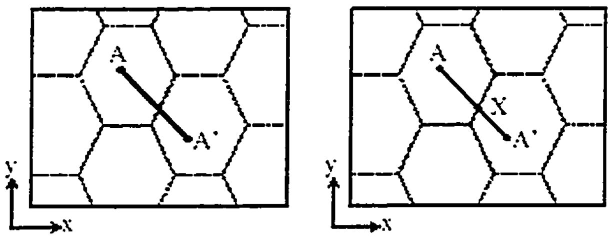


Figure 8. Splitting Segments during Insert [5]

Once the access structure is constructed, search can be performed. The search algorithm is performed in several steps:

- *Spatial Filtering*: Each spatial cell intersecting a query is retrieved and a set of candidates consisting of segments that belong to that cell is formed.
- *Temporal Filtering*: For each of the candidates a temporal component is checked to see whether or not it intersects the query's temporal range.

- Refinement *Step*: Each of the remaining segment in the candidate set is now checked whether it actually intersects the query and the final candidate list is formed
- Duplicate *Elimination*: The final list of segments is checked for segments that belong to the same trajectory (the same moving object) and the duplicates are removed. The produced result set represents actual objects intersecting a given query.

It is evident that by varying the number of cells used to partition the space, the performance of the indexing structure varies as well. If the cells are too big, the spatial discrimination of the index is reduced and the performance suffers. On the contrary, if the cells are too small the number of segments used to represent each trajectory can be large due to the splitting during insertion. There are two main approaches when choosing the number of cells: *static* and *dynamic*. In the *static* approach, the cells are chosen beforehand and are fixed during the lifetime of the indexing structure. In the *dynamic* approach, the cells are readjusted dynamically based on the trajectory distribution of the dynamic dataset. The ideal cell distribution would make a number of segments belonging to each cell equal for all the cells.

In the experiments, SETI is compared to two other common trajectory indexing methods: R-tree and TB-tree. Two different types of data generators are used: GSTD and the Network Data Generator. A number of experiments are conducted comparing the three indexing techniques using data from both data generators. One of the experiments measures absolute execution time of all three indexing structures with respect to the number of cells used. This experiment is repeated for query sizes of 0.01%, 0.1% and 1% of the total query space. Other experiments include comparing the index sizes and insertion performance of SETI, 3D R-tree, and TB-tree. The experiments show that SETI outperforms these indexing approaches.

Decoupling the spatial and temporal indices has the distinct advantage of making the index structure sparse, i.e. fewer trajectories or trajectory segments are maintained within each index. In such approaches a trajectory is no longer logically approximated by a 3-D MBR. The effect of this solution is that there should be less probability of a query intersecting an MBR. However, this will be dependent on the distribution of the data contained in each index. The main drawbacks of such approaches are that more indexing structures need to be maintained and specialized space partitioning and query processing techniques need to be adopted.

2.2.3 Tighter Trajectory Approximations

In the R-Tree, it is important to determine whether a query intersects a trajectory or its segments without searching the data tuples. The probability of a query intersecting a trajectory is related to the size of the query and the size of the approximating MBR. Hence, it is reasonable that one might model a trajectory by using tighter approximations or by using several MBRs to approximate pieces of a trajectory (split accordingly in the time dimension). This should have the effect of reducing the amount of dead space of an MBR.

In [21] a new approximation structure for trajectories, the Minimum Bounding Octagon Prism (MBOP), and a new index structure, the Octagon Prism Tree (OPT) are introduced. An MBOP is generated by cutting off the corners of an MBR. The reduction in volume of an MBOP appears to be substantial; the number of false hits for a region search is about 50% less than using MBRs as approximation structure. A trajectory-splitting heuristic is also explored in an attempt to further reduce the amount of volume utilized by a single trajectory, i.e. splitting trajectories into variable or fixed sized segments.

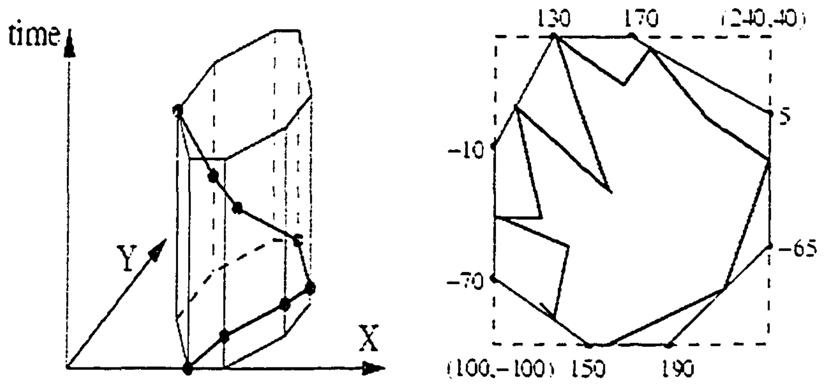


Figure 9. Logical representation of the MBOP [21]

Approximating a trajectory using an MBOP is only a slight departure from using an MBR. This can be observed in Figure 9. MBOPs can be inserted directly into any existing R-Tree structure as the corners of an MBOP are still contained within the boundaries of an MBR. One drawback of the author's work is that their approach does not consider how an OPT will be queried. An interesting extension to the author's work might be to use optimal rotated polygons to approximate a trajectory. In spatial domains this approach has been observed, however the algorithms to achieve this goal are complex and computationally expensive [1].

In order to use spatial access methods such as R-trees to index trajectories, each trajectory (or each of its segments if the trajectory is split) is approximated by a 3-dimensional MBR. In this context it is easy to see that splitting trajectories offers a great potential for improving the performance of spatiotemporal range queries. The intuitive reason is that when splitting a trajectory, the total volume of the approximating MBRs decreases. As a consequence, the approximations cover less empty space and may have less overlap with approximations of nearby trajectories. This may in turn lead to a reduction of the number of MBRs that intersect a given range query, thus reducing the number of pages that have to be retrieved in order to access the corresponding trajectory segments.

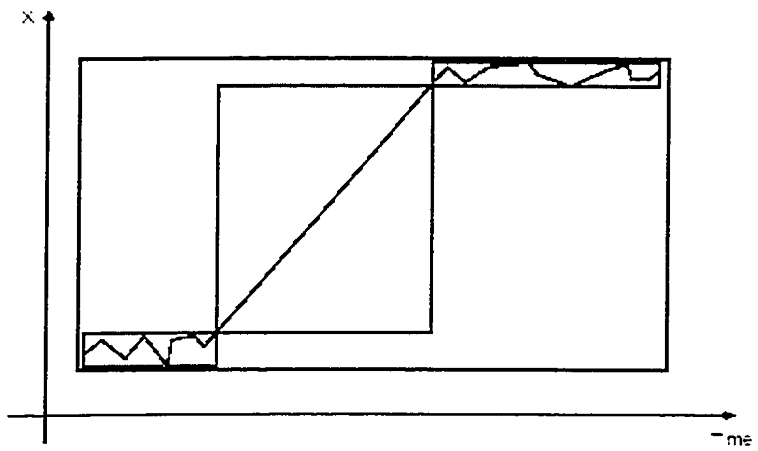


Figure 10. An example of splitting a trajectory using one and three MBRs [7]

To reduce the amount of dead space we could approximate the trajectory of an object using more then one MBR as illustrated on a two-dimensional example in Figure 10. The actual amount of volume reduction when splitting a trajectory depends not only on the number of splits but also on the chosen split points for a given number of splits. Figure 11 illustrates the effect of volume reduction, for different split choices for the depicted trajectory.

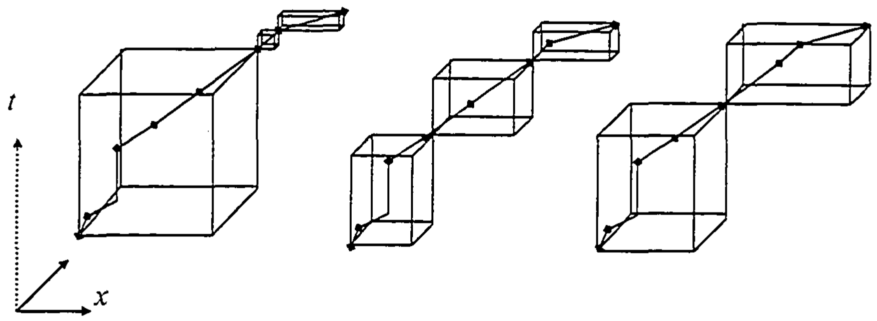


Figure 11. Different trajectory splits and its approximations.

Based on this observation, Hadjieleftheriou et al. [7] proposed several algorithms for minimizing the total volume of trajectory approximations given a user-specified number of splits *s*. First, a dynamic programming algorithm *DPSplit* is proposed that splits a single trajectory *T* using *l* splits so that the total volume of *T*'s approximations is minimized. The complexity of this algorithm is $O(t^2 l)$, where t is the number of trajectory points in *T*. For the same problem, they also described an $O(t \log t)$ greedy heuristic called *MergeSplit*. To split a *set* of trajectories $\{T_1, \ldots, T_n\}$, the authors proposed three algorithms that try to allocate to each trajectory T_i a number of splits l_i (out of the total number s of allowed splits), so that the overall volume of the trajectory approximations is reduced as much as possible. The first algorithm uses a dynamic programming approach, with a time complexity of $O(ns^2)$, to allocate the user-specified number of splits s to the n trajectories. This algorithm produces the optimal solution with respect to volume reduction when combined with *DPSplit*. They also introduce two heuristics, with time complexity $O(s \log n + n \log n)$, that show satisfactory performance in terms of volume reduction. All algorithms assume that the best splits of each trajectory into all possible number of splits are pre-computed and stored, adding the overhead of *DPSplit* or *MergeSplit* (with $l=t-2$) for each trajectory. The main problem with this approach is determining the total number of splits s that is used as an input parameter for the splitting algorithms.

In the experiments, the authors of [7] generate two spatiotemporal datasets. The trajectories are split using different numbers of splits and the resulting MBRs are indexed using the partially persistent R-Tree or PPR-Tree [24] and the regular R*-Tree [2]. The queries are run against the two indexing structures and the average number of disk accesses is recorded when varying the total number of splits parameter and when varying the database size. The experiments show that the PPR-Tree consistently outperforms the regular R*-Tree in both cases.

In the following section, we argue that minimizing the total volume of trajectory approximations does not necessarily lead to the best query performance and motivate the trajectory splitting with respect to the given average query size.

2.3 Trajectory Splitting and the Query Size

The trajectory splitting approaches with the goal of minimizing the total volume of trajectory approximations given a user-specified number of splits, in general, have several drawbacks:

- Minimizing the total volume of trajectory approximations without considering actual query sizes does not necessarily minimize the number of expected I/Os when processing range queries. Obviously, introducing more splits for a trajectory necessarily reduces the total volume of the trajectory approximations. However, introducing more splits also increases the number of segments for the same trajectory that may simultaneously intersect a query range, resulting in unnecessary disk I/Os. Figure 12 illustrates two possible cases. Figure 12(a) depicts a scenario where the given trajectory has an unnecessarily large number of splits for the given query size. The query intersects, in this extreme case, all approximating MBRs that may be located on different disk pages, and thus result in a larger number of I/Os. On the other hand, the situation in Figure 12(b) shows a case where the same splits are appropriate for a smaller query size.
- The methods require as input parameter the total number of allowed splits for the whole set of trajectories. This parameter is difficult to determine even for a static set of trajectories. For the important dynamic case, where

trajectories can grow continuously and new trajectories are added over time, a fixed overall number of splits is not meaningful.

- Even knowing a good number of possible splits, the proposed algorithms are very time consuming and have a large storage overhead: all possible splits of all trajectories are pre-computed and stored for re-use in the search algorithms that distributes the given total number of splits, which typically is itself quite large.

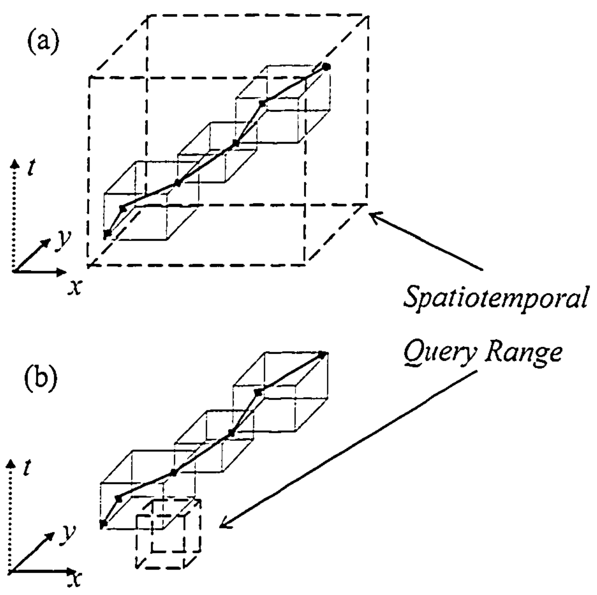


Figure 12. Relation between query size and trajectory splits.

We conclude that minimizing the volume of trajectory approximations is not enough to minimize the expected number of I/Os for spatiotemporal range queries. We claim that in order to optimize query performance, we also need to take into account actual query sizes. Our solution to this problem provides an algorithm for optimally splitting a set of trajectories based on a given query size, which in most practical cases can be determined as the average query size. An average query size is also a more natural and robust parameter than a user-specified total number of splits and it is not restricted to static data sets.

Chapter 3

Optimal Trajectory Splitting

In this section, we derive an analytical cost model that estimates the expected number of I/Os, yielded by a given split of a trajectory and a given query size. Based on this model, we introduce an algorithm for splitting all trajectories in a set of trajectories so that the total number of expected disk I/Os is minimized with respect to the query size.

3.1 A Cost Model for Splitting Trajectories

A trajectory T is given by a sequence of points $T=\langle p_1, p_2, \ldots, p_t \rangle$, where each point p_i is a tuple of spatial and temporal coordinates. We denote a segment of a trajectory T that starts at point p_u and ends at point p_v by $T[u,v]$ (note that consequently $T \equiv T[1,t]$).

A trajectory can be split along its discrete temporal dimension into m segments ($1 \leq m \leq t-1$). For a given m, there are $\binom{t-2}{m-1}$ possible ways of splitting T into m segments. Each decomposition of T into m segments involves choosing $m-1$ split points from T, excluding the endpoints p_1 and p_t. A given decomposition of T into m segments, $T=(T[1,i_1], \ldots, T[i_{m-1}, t])$ for a sequence of split positions $i_1, \ldots, i_{m-1}$, will be approximated by a sequence of MBRs $B^T=(MBR(T[1,i_1]), \ldots, MBR(T[i_{m-1}, t]))$, where $MBR(T[u,v])$ denotes the MBR for a segment $T[u,v]$. We denote the set of the MBR approximations of all possible decompositions of T into m segments by $Decomp(T, m)$, i.e.

$$\begin{aligned} Decomp(T,m) = \{(B_1,\ldots,B_m) \mid \exists i_1,\ldots,i_{m-1} :\\ B_1 = MBR(T[1,i_1]),\\ B_2 = MBR(T[i_1,i_2]),\\ \ldots, B_m = MBR(T[i_{m-1},t])\} \end{aligned} \tag{1}$$

For our cost model, we assume that segments and their respective MBRs are stored independently, e.g., under an R-tree. That means that the MBRs of a trajectory may be stored on different disk pages, and each segment's MBR that is intersected by a query may thus require an independent disk I/O, ignoring possible effects of an index directory and caching. The objective of a trajectory splitting algorithm is therefore to minimize the number of expected disk I/Os required to answer a given query q. In the following let $B^T = (B_1, \ldots, B_m)$ be the MBR approximation of a specific decomposition of T into m segments.

As discussed before, the number of expected disk I/Os required to answer a query q is related not only to the total volume of the MBRs in B^T but also to the size of q. The size of a query determines the probability that q intersects some $B_i \in B^T$, which in turn determines the expected number of I/Os that B^T contributes to the total I/O cost of processing q.

Given a range query q the expected number of disk I/Os due to B^T can be derived as follows. If q intersects B^T, then it intersects exactly k segments simultaneously, where $1 \leq k \leq m$, accordingly we would have exactly k I/Os. The event that q intersects exactly k segments of B^T (thus resulting in k I/Os) occurs with a probability $P(q \cap B^T;k)$. Since these events are mutually exclusive, the overall expected number of I/Os for query q, $E_{B^T}(q)$, is consequently the sum of the number of I/Os due to each event, weighted by the probability of the event:

$$E_{B^T}(q) = \sum_{k=1}^{m} k \cdot P(q \cap B^T;k) \qquad (2)$$

The following lemma can be used to simplify this expectation expression.

Lemma 1. Let $P(q \cap B_i)$ be the probability that a query q intersects the i^{th} segment in B^T. Then it holds that

$$E_{B^T}(q) = \sum_{i=1}^{m} P(q \cap B_i) \qquad (3)$$

Proof. A proof of $\sum_{k=1}^{m} k \cdot P(q \cap B^T;k) = \sum_{i=1}^{m} P(q \cap B_i)$ for the general case of MBRs for spatial data can be found in [12]. ■

Lemma 1 states that the expected number of I/Os can be computed by simply summing up the probabilities of the query q intersecting the MBRs for the trajectory segments independently of each other.

To determine the probability $P(q \cap B_i)$, we consider the area where a query q can fall and at the same time intersect B_i. This area is given by extending B_i by half of the query extension in each dimension, as illustrated in a 2-dimensional example in Figure 13. The rationale for this query extended MBR is that the query intersects an MBR if and only if the query center is within the query extended MBR. We denote the query extended MBR for an MBR B_i by $Ext_q(B_i)$.

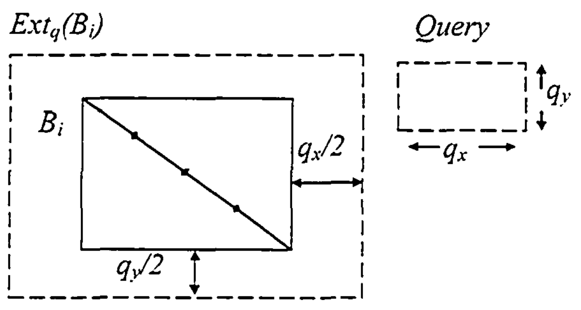


Figure 13. A Query Extended MBR.

Assuming a uniform distribution of queries, and ignoring boundary effects, the probability of a query q intersecting a segment MBR B_i is proportional to the normalized volume of the query extended MBR $Extq(B_i)$, i.e.:

$$P(q \cap B_i) = Vol(Ext_q(B_i)) / Vol(S) \tag{4}$$

where $Vol(S)$ is the volume of the whole data space S.

By substituting Equation (4) into (3), we obtain the expected number of I/Os due to the approximations in B^T given the size of a query q as:

$$E_{B^T}(q) = \sum_{i=1}^{m} Vol(Ext_q(B_i)) / Vol(S) \tag{5}$$

In order to minimize this performance measure for a single trajectory T, T must be split so that the sum of the volumes of the resulting query extended MBRs is minimized, which means finding the minimum expected number of I/Os among of all possible decompositions of T into all possible numbers of segments m, i.e., finding $\min_{1 \le m \le t-1, B^T \in Decomp(T,m)} \{E_{B^T}(q)\}$.

While splitting a trajectory always reduces the total volume of the MBRs approximating the segments, this is not true for the query extended MBRs. Figure 14 illustrates a 2-dimensional case where the sum of the volumes of the query extended MBRs is minimized when splitting the trajectory only once. Introducing a third (or more) split(s) will increase the sum of the volumes of the query extended MBRs, and therefore increase the expected number of I/Os for this trajectory and the given query size. Note that only minimizing the volume of the MBRs would be a misleading measure in this case depicted in Figure 14.

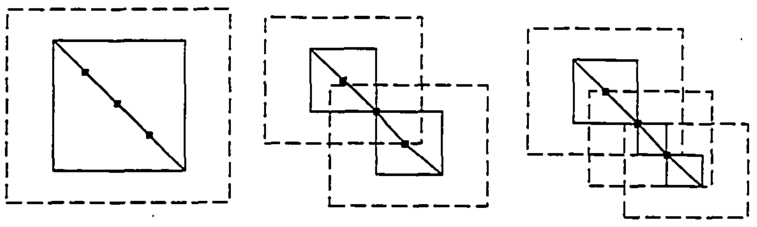


Figure 14. Volume of query extended MBRs using 0, 1, or 2 splits

So far, we have only considered how to split a single trajectory optimally. Optimally splitting a *set* of trajectories Θ, theoretically depends on a number of varying parameters such as the distribution of the trajectories in space, the page size, as well as the directory structure and the split algorithm of the particular spatial index used to store the MBRs. To simplify our model and to be independent of the used index structure, we ignore the possible effect of an index directory and the distribution of the trajectories in space. In this case, each trajectory T in Θ contributes independently towards the total number of expected

I/Os for accessing data pages, given a query q, which means that the expected number of I/Os can simply be expressed as the sum of the individual expectations:

$$E_{total}(q) = \sum_{T \in \Theta} E_{B^T}(q) \tag{6}$$

Equation 6 tells us that, given a query q, we can find the optimal splits needed for a set of trajectories, by minimizing the splits for each trajectory individually.

In general, a trajectory T can be split into m segments in different ways, where each such split may result in a different number of I/Os when processing a given query q. Let $E^{opt}_{T,m}(q)$ be the minimum expected number of I/Os for T that can be obtained by splitting T into m segments, i.e.

$$E^{opt}_{T,m}(q) = \min_{B^T \in Decomp(T,m)} \{E_{B^T}(q)\} \tag{7}$$

(In the special case where m=1, i.e., if there is only one segment, we can drop the superscript "opt", since there is only one choice, which is trivially optimal.)

A trajectory can be split into different numbers of segments, ranging from 1 to t–1. Consequently, the minimal number of I/Os for T over all possible splits, expressed using the notation $E^{opt}_{T}(q)$, is given by the best possible split of T for m ranging from 1 to t–1, i.e.,

$$E^{opt}_{T}(q) = \min_{1 \le m \le t-1} \{E^{opt}_{T,m}(q)\} \tag{8}$$

3.2 Dynamic Programming Algorithm

To solve Equation 8, we propose a dynamic programming solution, which basically finds the best possible split of T for each value of m. Using our notation

$T[u,v]$ to denote a subsequence of T from point p_u to point p_v, we can re-write $E^{opt}_{T,m}(q)$ as $E^{opt}_{T[1,t],m}(q)$. In order to apply dynamic programming to our problem, we have to show that the following property holds.

Theorem 1. Given a trajectory $T=\langle p_1, p_2, \ldots, p_t \rangle$ and a query q, it holds that

$$E^{opt}_{T[1,t],m}(q) = \min_{1<u<t} \{E^{opt}_{T[1,u],m-1}(q) + E_{T[u,t],1}(q)\} \qquad (9)$$

Proof. Using equation 5, equation 7 can be re-written as

$$E^{opt}_{T[1,t],m}(q) = \min_{B^T \in Decomp(T[1,t],m)} \left\{ \sum_{i=1}^{m} \frac{Vol(Ext_q(B_i))}{Vol(S)} \right\}.$$

Expanding the sum in this equation gives us

$$E^{opt}_{T[1,t],m}(q) = \min_{B^T \in Decomp(T[1,t],m)} \left\{ \sum_{i=1}^{m-1} \frac{Vol(Ext_q(B_i))}{Vol(S)} + \frac{Vol(Ext_q(B_m))}{Vol(S)} \right\}$$

Assuming that the start position of the last segment of an *optimal* decomposition of T is u, and using the notation $B^{opt}_m = MBR(T[u,t])$ to denote the MBR of this last segment, we could re-write this equation as

$$E^{opt}_{T[1,t],m}(q) = \min_{B^T \in Decomp(T[1,u],m-1)} \left\{ \sum_{i=1}^{m-1} \frac{Vol(Ext_q(B_i))}{Vol(S)} \right\} + \frac{Vol(Ext_q(B^{opt}_m))}{Vol(S)}$$

$$= E^{opt}_{T[1,u],m-1}(q) + E_{T[u,t],1}(q)$$

This equation holds where the start position u of the last segment of an optimal decomposition is known since then, the last segment is fixed, and the remaining prefix of T, i.e., $T[1,u]$ must consequently be split into $m-1$ segments so that the sum of volumes of the extended MBRs for the first $m-1$ segments is minimal, in order for the whole sum to be minimal. In order to find the optimal decomposition of T in general, we just have to consider all possible values of start positions u in the range $1 < u < t$ for the last segment of T, as originally stated in the theorem:

$$E^{opt}_{T[1,t],m}(q) = \min_{1<u<t}\{E^{opt}_{T[1,u],m-1}(q) + E_{T[u,t],1}(q)\} \quad \blacksquare$$

Theorem 1 is the important property that allows us to use a dynamic programming approach to optimize the expected number of I/Os for a trajectory. It states that in order to find the optimal solution for a trajectory T using m segments, it is sufficient to consider all the optimal sub-solutions using $m-1$ segments for the prefixes $T[1,u]$, $1 < u < t$, (which can be found by recursively applying Equation 12), and combine them with the solution for the remaining segment $T[u,t]$.

The running time of the dynamic programming algorithm to determine the split of one trajectory T into m segments (i.e., $m-1$ splits) is $O(t^2(m-1))$ where t is the number of points in T. Consequently, to find the best possible split for T among all possible values of m, we have to apply the algorithm for the maximum possible value of m, i.e., for $m = t-1$. To split a set of n trajectories optimally, we have to apply this algorithm n times. This time complexity is the same as the time complexity for the *DPSplit* pre-computation step used in Hadjieleftheriou et al.'s algorithms [7]. Note, however, that, in contrast to these algorithms, we don't need to execute an additional time consuming and storage intensive search algorithm to obtain a globally optimal solution with respect to our cost model.

3.3 Directory Level Node Splitting

So far, we have only considered access to data pages. The cost model estimates, given a query size, the expected number of accesses to a set of MBRs, which correspond to disk accesses if these MBRs are stored on different disk pages. For this estimation, it is not essential that the MBRs enclose trajectory segments. Trajectories only determine the possible points that can be considered when splitting them, resulting in different sets of MBRs.

For R-tree based indices, MBRs for directory pages must be split during index construction and update. Different heuristics have been proposed for that purpose, such as the quadratic and the linear split [6], or the R*-tree split [2]. These algorithms generate a certain subset of all possible splits of an MBR and minimize evaluation functions, which are typically based on area and overlap of the resulting MBRs as already mentioned in Section 2.1.1. The goal of these heuristics is essentially to minimize the probability that queries will intersect both resulting MBRs thus reducing the number of sub-trees that have to be traversed.

The rationale behind our cost model can be applied to directory level splits as well. In our approach, given an average query size, instead if using the MBRs in calculating area, overlap and margin values we propose to use the query extended MBRs and keep the original node splitting algorithms.

Chapter 4

Heuristic Trajectory Splitting

For large datasets containing long trajectories our dynamic programming solution may be inefficient due to the time complexity of the algorithm. Another problem is that the dynamic programming solution requires the complete trajectory to be available in order to find the optimal splits. For many applications, however, trajectories are updated continuously. In order to deal with such applications, we need a more efficient and at the same time incremental method, which can still produce good (ideally close to optimal) results.

In this section, we formally derive an approximation of the optimal decomposition of a trajectory that can be incrementally computed. For this purpose, we first consider special cases of trajectories for which we introduce an I/O cost function that can be minimized by finding an optimal segment size for the whole trajectory. We can then apply this model to pieces of arbitrary trajectories in order to approximate their optimal decomposition using a linear time algorithm.

4.1 A Cost Model for Optimal Segment Size

In this section we want to deal with long trajectories where points are added continuously over a long period of time. Consider first the special case of trajectories with a constant slope, i.e., trajectories for objects moving with constant speed in a constant direction that are sampled at constant time intervals. Sampling at constant time intervals does not really constitute a restriction here since we assume a linear interpolation between sampling points so that constant time intervals can always be achieved by a suitable re-sampling. We will show that for those trajectories the optimal split according to our previous cost model will result in segments of equal size.

Assume a trajectory T consisting of t points, or equivalently, consisting of $t-1$ consecutive elementary segments $s_1, \ldots, s_{t-1}$ as well as a decomposition of T, $B^T=\{B_1, \ldots, B_m\}$. We can express the sum in Equation 5, $E_{B^T}(q) = 1/Vol(S) \cdot \sum_{i=1}^{m} Vol(Ext_q(B_i))$, differently by thinking of the volume of each $Ext_q(B_i)$ as being "generated" by the elementary segments contained in B_i, in the following way. We can define a function f that expresses an equal contribution of each elementary segment to the volume of the query extended MBR it belongs to as:

$$f(s) = \frac{Vol(Ext_q(B_i \ containing \ s))}{\# \ elementary \ segments \ in \ B_i \ containing \ s} \qquad (13)$$

***Lemma** 2*. Let $B^T=\{B_1, \ldots, B_m\}$ be a decomposition of a trajectory T, and let function f be defined as in Equation 13. Then the following equation holds,

$$\sum_{i=1}^{m} Vol(Ext_q(B_i)) = \sum_{i=1}^{m}\left(\sum_{s \text{ contained in } B_i} f(s)\right) = \sum_{i=1}^{t-1} f(s_i) \qquad (14)$$

Proof. Obvious, because by the definition of *f*, since for each B_i in B^T it holds that

$$\sum_{s \text{ contianed in } B_i} f(s) = Vol(Ext_q(B_i)) \quad \blacksquare$$

Lemma 2 tells us that a decomposition that minimizes the left hand side also minimizes the right hand side, and vice versa. Finding an optimal decomposition using the right hand side is, in general, not an easier problem than using the left hand side, since the *f* values for elementary segments depend on where the actual split points in a decomposition of *T* are. However, it is easy to see that for the special case of trajectories with a constant slope the *f* values depend only on the number of elementary segments in an enclosing MBR B_i. In this case, we can compute the volume of a query extended MBR B_i using only the increments in each dimension ($\Delta x, \Delta y, \Delta t$) that define the slope of the trajectory, as illustrated in Figure 15, and the number *c* of elementary segments in B_i as following:

$$Vol(Ext_q(B_i)) = (c \cdot \Delta x + q_x) \cdot (c \cdot \Delta y + q_y) \cdot (c \cdot \Delta t + q_t) \qquad (15)$$

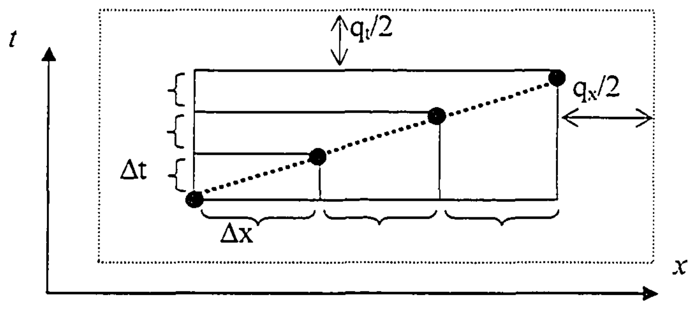


Figure 15. Illustration of a constant-slope trajectory in 2 dimensions.

Furthermore, we can now look at the values of *f* for MBRs of arbitrary size by looking at its definition as a function *g* of *c*,

$$g(c)=\frac{(c\cdot\Delta x+q_x)\cdot(c\cdot\Delta y+q_y)\cdot(c\cdot\Delta t+q_t)}{c} \tag{16}$$

The significance of this function is that we can show that $g(c)$ has a real minimum, which has several important consequences. It means that there is an optimal segment size c_{opt} in the sense that if a trajectory T with a constant slope is decomposed into segments of this size, the value $f(s)$ will be minimal for each elementary segment s. This in turn means that this decomposition of T minimizes the right hand side of Equation 14, giving us an optimal decomposition according to our cost model in Section 3. In this optimal decomposition, all the segments have the same size c_{opt}, and this size is independent of the length of T, and c_{opt} determines where the split point have to be. The value c_{opt} is only dependent on the slope of T and the query size, which also means that we can optimally split trajectories with a constant slope in an incremental manner, i.e., we can already split into optimally sized segments after some points of T have been added. We use this fact later to derive a incremental splitting heuristic for arbitrary trajectories which we approximate by several "constant-slope" sub-trajectories.

Theorem 2. Given a query q, and increments $(\Delta x, \Delta y, \Delta t)$ (that define the slope of a constant-slope trajectory), the function g (Equation 16) has a global, real minimum c_{opt} with respect to c.

Proof. Function g can be re-written as

$$g(c) = \frac{k_1c^3 + k_2c^2 + k_3c + k_4}{c}$$

where $k_1 = \Delta x \Delta y \Delta t$

$$k_2 = \Delta x \Delta y q_t + \Delta y \Delta t q_x + \Delta x \Delta t q_y$$

$$k_3 = \Delta x q_y q_t + \Delta y q_x q_t + \Delta t q_x q_y$$

$$k_4 = q_x q_y q_t$$

Dividing by c gives us the following equation for $g(c)$

$$g(c) = (k_1c^2 + k_2c \quad + k_3 + k_4 \frac{1}{c})$$

Applying the first derivative to find the extreme values, we get

$$\frac{dg(c)}{dc} = (2k_1c + k_2 + (-k_4 \frac{1}{c^2})) = 0$$

which is equivalent to finding the solutions to

$$(2k_1c^3 + k_2c^2 + (-k_4)) = 0$$

Function $g(c)$ has a solution $c = c_{opt}$ in the domain of positive real numbers that can be determined analytically. To show that the function $g(c)$ reaches a minimum at $c = c_{opt}$ we can find second derivative of $g(c)$ with respect to c as following

$$\frac{dg^2(c)}{dc^2} = (2k_1 + 2k_4 \frac{1}{c^3})) > 0$$

Since k_1, k_4, and c are always greater than 0, it is obvious that the second derivative of is $g(c)$ is greater than 0 for all values of c, which means that the function $g(c)$ reaches a global minimum at $c = c_{opt}$. ■

We can use the value c_{opt} that minimizes $g(c)$ to construct a decomposition of a constant-slope trajectory $T = \langle p_1, p_2, \ldots, p_t \rangle$ by trying to divide it into equally sized segments of length c_{opt}. In the following theorem, we assume that t is divisible by c_{opt}. This is justified by the fact that we assume *long* trajectories where points are continuously added (*short* trajectories can be split efficiently using our dynamic programming approach). Since c_{opt} does not depend on the number of trajectory points and can be computed using only the slope of T (and the query size), in practice, we would split T into segments of length c_{opt} continuously as points are added to the given trajectory over time.

Theorem 3. Given a query q, and a trajectory $T = \langle p_1, p_2, \ldots, p_t \rangle$ with constant slope defined by increments $(\Delta x, \Delta y, \Delta t)$, we can find c_{opt} that minimizes $g(c)$ (according to Theorem 2). Assuming that t is divisible by c_{opt}, the decomposition of T (possibly after suitable re-sampling) into equal sized segments determined by c_{opt} is a solution to Equation 8, i.e., a decomposition that minimizes the expected number of I/Os.

Proof. Without loss of generality, we assume that c_{opt} is an integer (if c_{opt} is not an integer, we can re-sample T appropriately so that c_{opt} can be expressed as an integer with respect to the new elementary segment size). Let $B^T_{c_{opt}} = (B_1, \ldots, B_m)$ be the decomposition of T where each B_i contains the same number c_{opt} of elementary segments s. The resulting values $f(s)$ according to Equation 13 for each elementary segment s is by Theorem 2 minimal, i.e., no other MBR size can result in smaller values $f(s)$ for the segments contained in that MBR.

Consequently, the sum $\sum_{i=1}^{t-1} f(s_i)$ is minimal for the given decomposition $B^T_{c^{opt}}$ among all possible decompositions. Using Equation 14 in Lemma 2, it follows that this decomposition must also be a solution to Equation 8. ■

So far, in this section, we have assumed trajectories of constant slope that are sampled at constant time intervals. This assumption is not true for most trajectories in practical applications. However, we can still apply our model to an arbitrary trajectory T by approximating it with constant-slope trajectory $T^{\triangleleft}$ in the following way. We can compute the increments $\Delta x, \Delta y, \Delta t$ that define the slope of $T^{\triangleleft}$ as the average of the corresponding increments of T, e.g., $\overline{\Delta x} = \frac{1}{t-1}\sum_{i=1}^{t-1} \Delta x_i$, where Δx_i represents the difference in x direction between two consecutive points of T. Obviously, the smaller the variance in the increments T is, the better is the approximation $T^{\triangleleft}$. Although the error of the approximation is typically large for *long* trajectories, this is not true for smaller pieces of a trajectory in case of most real world applications since objects usually do not move erratically but keep moving in a similar direction with a similar speed for a certain period of time.

The fact that we can generally approximate a long trajectory well, using several constant-slope sub-trajectories, allows us to design an incremental algorithm for splitting trajectories that shows near optimal performance in practice (unless the objects move extremely erratically).

4.2 Linear Time Trajectory Splitting

For a linear time trajectory splitting algorithm, ideally, the split decisions should be made incrementally as the data points of the trajectories are added. We can incrementally buffer a certain number of incoming points of a trajectory T, say from point p_u to point p_v, and compute the average increments $\overline{\Delta x}, \overline{\Delta y}, \overline{\Delta t}$ for the points in the buffer to obtain a constant-slope approximation $T^{\Delta}[u,v]$ for the trajectory segment $T[u,v]$ in the buffer. Using the proof of Theorem 2, we can then determine the optimal number c_{opt} of elementary segments that should be grouped together in an optimal decomposition of $T^{\Delta}[u,v]$, and then use this number to decompose $T[u,v]$ accordingly.

To apply this method, we have to determine a suitable number of points that should be buffered for a trajectory before applying the split policy. This number may depend on several interacting factors including the average query size, the speed, the direction changes, and the sampling rate of the moving object. For different trajectories, and even for different segments of the same trajectory, a different buffer size may be optimal.

We propose to use our cost model for optimally splitting a trajectory, derived in Section 3, to estimate when to apply our linear splitting method. The intuition is that we can determine when an MBR around the points of a trajectory segment $T[u,v+1]$ is not optimal, according to the following condition.

$$E_{T[u,v+1],1}(q) > E_{T[u,v],1}(q) + E_{T[v,v+1],1}(q) \qquad (18)$$

If the condition is true, it means that the expected number of I/Os using one MBR around the segment of $T[u,v+1]$ is larger (i.e., worse w.r.t. performance) than the number of expected I/Os when introducing a split before the last elementary segment of $T[u,v+1]$. Therefore, it makes sense to consider splitting $T[u,v+1]$. The fact that the condition is true tells us that there is at least one possible split, i.e., before the last elementary segment, that will result in a better I/O expectation. This split is, however, in general not the best possible way of splitting the current segment $T[u,v]$. Iteratively collecting points until Equation 17 becomes true, then introducing a split at exactly that position, and repeating this until the trajectory ends, will, in general, create segments that are consistently larger than the segments obtained by an optimal split. Condition 17 is good at detecting significant changes in speed and direction of a trajectory. For close to constant-slope segments of a trajectory, the condition tends to become true only after several times the optimal segments size has been accumulated. We have confirmed this behavior experimentally, but we can also understand it more formally. Consider the difference between the left hand side and the right hand side of Equation 18.

$$E_{T[u,v+1],1}(q) - \left(E_{T[u,v],1}(q) + E_{T[v,v+1],1}(q)\right) \qquad (19)$$

If this expression is smaller than or equal to 0, condition 18 is false, if it is greater than 0, condition 18 true. For the case of constant-slope trajectories, we can compute the expected I/O values in this expression as the volumes of the query extended MBRs around $T[u,v+1]$, $T[u,v]$, and $T[v,v+1]$ respectively, using

Equation 15. The number of elementary segments c, in $T[u,v]$ is given by the equation $c = v - u$. After simple arithmetic transformations, we obtain:

$E_{T[u,v+1],1}(q)-\left(E_{T[u,v],1}(q)+E_{T[v,v+1],1}(q)\right)=3k_1c^2+2k_2c+3k_1-k_4$, where k_1, k_2, k_4 are defined as in the proof of Theorem 2. Consequently, condition 18 holds if $3k_1c^2+2k_2c+3k_1-k_4>0$ or, equivalently if

$$3k_1c^2+2k_2c+3k_1>k_4 \tag{20}$$

On the other hand, we know from the proof of Theorem 2 that the function g (Equation 16) has a global minimum c_{opt} for the optimal number of elementary segments at $2k_1c_{opt}^3+k_2c_{opt}^2-k_4=0$ or, equivalently if

$$2k_1c_{opt}^3+k_2c_{opt}^2=k_4 \tag{21}$$

Substituting Equation 21 in Equation 20, tell us when the condition in Equation 18 is true in terms of the number of elementary segments for a constant-slope trajectory, i.e., it is true if

$$3k_1c^2+2k_2c+3k_1>2k_1c_{opt}^3+k_2c_{opt}^2 \tag{22}$$

Considering that both c and c_{opt} have to be greater than or equal to 1 since we never collect less than one elementary segment, and we never split less than one segment in practice, it is easy to see that for this inequality to hold, the value of c must be larger than the value of c_{opt} for $c \geq 2$ (i.e., if the buffer contains at least two elementary segments). For $c = 1$, the condition is true for $c_{opt} < 1.5$. However, since we consider only splits at sample points of a trajectory, we will round c_{opt} to the closest positive integer value c_{opt}^{*} (in the case $c = 1$, $c_{opt}^{*} = 1$). In summary, this means that the number of elementary segments c, collected up to the point where condition 18 becomes true, is always a multiple of the optimal segment size.

With a dynamically determined buffer size according to these considerations, we propose the following linear time splitting algorithm for trajectories which we call LinearSplit. The algorithm collects points of a trajectory consecutively. For each new point p_{v+1}, it determines whether the new point should be merged into the current buffer $T[u,v]$ or whether a split at this point should be introduced, using Equation 18. If the condition is true, we compute the optimal segment size c_{opt} according to Theorem 2 (with a constant-slope approximation of the current trajectory segment $T[u,v]$), and we round it to the nearest positive integer c^{*}_{opt}. We split as many segments of size c^{*}_{opt} as possible from $T[u,v]$ and insert the corresponding MBRs into the index. This procedure is repeated as long as new points are added. When a trajectory is completed, the last segment is still in the buffer and has to be inserted as well.

Obviously, this algorithm splits a trajectory in O(t) time where t is the number of points of the trajectory. The pseudo code for the algorithm, LinearSplit, is presented on the next page.

Algorithm LinearSplit

```
u := 1, v := 2; //after the first two points of T
while (there is a next point $p_{v+1}$ in trajectory T)
    if $E_{T[u,v+1],1}(q) > E_{T[u,v],1}(q) + E_{T[v,v+1],1}(q)$
        find $c_{opt}$ for T[u,v] using Theorem 2;
        c* = round($c_{opt}$);
        extract the first (v-u)/c* segments from
        T[u,v] and insert their MBRs into the index;
        u:=u+k * c*;
    v++;
//end of T is reached
insert last MBR(T[u,v]) into the index;
```

Chapter 5

Experimental Results

In order to evaluate the proposed approaches for splitting spatiotemporal range queries we used two types of datasets, one produced by the Network Data Generator [22] and another one produced by GSTD [18]. The network data generator simulates different classes of objects, e.g., vehicles and people, moving through the streets of a real city. Therefore, different objects have different speeds and lifetimes, producing a very rich and realistic dataset. GSTD, on the other hand, allows generating more random patterns, allowing us to investigate how well the proposed algorithms perform under more extreme situations.

For each generator, we produced datasets containing 10,000, 20,000, and 50,000 trajectories, respectively. For each trajectory in the network datasets using the map of the city of Oldenburg, a varying number of observations ranging between 50 and 345 were recorded, resulting in 97 observations on average per trajectory. We set GSTD's parameters so that trajectories were formed by objects, uniformly distributed in the data space, changing speed and direction randomly at any point in time (the maximum speed was limited though, so that an object could not cross more than 20% of the total space from one time stamp to the next). This scenario, when objects are moving extremely erratically, is particularly challenging for our LinearSplit algorithm. Unlike for the network dataset, exactly 100 observations were recorded for each trajectory. Therefore, all our datasets had between 1,000,000 and 5,000,000 observations in total. All experiments were performed on a 1900+ AMD Athlon PC with 512 Mb of RAM.

In our experiments we used the quadratic R-tree implementation provided by the XXL library [23], using a page size of 4Kb for all algorithms which resulted in a capacity of 70 entries per node. For our algorithms, we replaced the split evaluation function by our cost model as described in Section 3.3.

We evaluate the quality of the proposed algorithms by measuring the performance of the generated indices using different trajectory split policies. We measure the number of disk I/Os on the index's directory and data level per query, averaged over 10,000 uniformly distributed queries, without considering buffering. We also measure the time required to pre-process a dataset, i.e., the time required to split the trajectories, create the MBRs and create the index tree.

In all forthcoming figures we refer to our dynamic programming-based algorithm as "OptimalSplit". The linear time algorithm is referred to as "LinearSplit". We also use "HKTG-*k*%" to refer to the volume oriented split policy proposed in [7] (using the *DPSplit* algorithm for splitting trajectories individually), where k is defined according to [7], i.e., k% means that $N \cdot k/100$ total number of splits are used for splitting a dataset with N trajectories. Similar to [7], we used k equal to 50, 100 and 150. We also used two baseline algorithms. First, an R-tree where each trajectory is approximated by a single MBR, i.e., trajectories were not split at all. This algorithm is referred to as "NoSplit".

Second, an R-tree where each elementary segment of a trajectory is approximated by an MBR, i.e., trajectories were split at each observation point. This algorithm is referred to as "FullSplit".

5.1 Robustness with Respect to Query Size

In the first experiment, we used datasets of 50,000 trajectories to investigate the robustness of our algorithms with respect to the proposed cost models. Since the cost models aim primarily at minimizing disk accesses to data pages, we measured the average number of disk I/Os at the data level for both datasets. For each test we built trees that are optimized for a particular query size of S% of each spatial dimension and T time points. In particular, we used S = 1%, 2%, ..., 16% and T = 1, 2, ..., 16. Note that while the spatial dimension of the query is given as percentage of the total 2*d* space, the temporal dimension is given in absolute time points since time is unbounded. We use $I_{i,j}$ to denote the index that is optimized for the query size with spatial extensions given by S=*i*% and temporal extension T=*j*. Similarly, $Q_{i,j}$ denotes the size of the queries that were executed against the different indices. The numbers in Tables 1 through 4 represent the average performance for different indices and different query sizes. Each row in the tables represents the performance of a particular query $Q_{i,j}$ using all the constructed indices (given by the columns). Ideally, given a query size, the best performance should be when using the index built for that query size, i.e., in the diagonal of the tables. Note that in this experiment, we compare in the rows the performance of different trees for a given query. Comparing different queries for the same tree here (i.e., looking at columns) is not very meaningful since the columns only show the obvious fact that smaller queries result in smaller numbers of I/Os than larger queries, simply because they intersect less MBRs. Tables 1 and 2 show results for the network dataset. Tables 3 and 4 show results for the GSTD dataset.

Table 1. Robustness of OptimalSplit for Network Data

# I/Os		Tree – optimized for S(%) and T (duration)				
		$I_{1,1}$	$I_{2,2}$	$I_{4,4}$	$I_{8,8}$	$I_{16,16}$
Queries	$Q_{1,1}$	0.56	0.64	1.01	2.11	5.59
	$Q_{2,2}$	2.37	2.18	2.52	3.95	8.22
	$Q_{4,4}$	13.02	9.97	8.73	10.1	15.9
	$Q_{8,8}$	83.85	56.76	39.32	34.62	40.13
	$Q_{16,16}$	572.9	358.5	212.4	150.1	131.9

Table 2. Robustness of LinearSplit for Network Data

# I/Os		Tree – optimized for S(%) and T (duration)				
		$I_{1,1}$	$I_{2,2}$	$I_{4,4}$	$I_{8,8}$	$I_{16,16}$
Queries	$Q_{1,1}$	0.57	0.63	1.01	2.14	5.44
	$Q_{2,2}$	2.65	2.23	2.57	3.99	7.97
	$Q_{4,4}$	15.65	10.61	8.88	10.32	15.63
	$Q_{8,8}$	105.27	62.15	40.28	35.33	39.95
	$Q_{16,16}$	738.6	400	218.9	155.1	135.3

The performance of both algorithms on both datasets is qualitatively very similar. The best performance (shaded cells) occurs exactly where expected, in the diagonal of the tables, except for one case when using the LinearSplit heuristic and GSTD data (see $Q_{2,2}$ in Table 4), due to the erratic movement in that dataset.

Even in this single case, the value in the diagonal is very close to the minimal value in that row.

Table 3. Robustness of OptimalSplit for GSTD Data

# I/Os		Tree – optimized for S(%) and T (duration)				
		$I_{1,1}$	$I_{2,2}$	$I_{4,4}$	$I_{8,8}$	$I_{16,16}$
Queries	$Q_{1,1}$	1.01	1.02	1.39	3.89	10.39
	$Q_{2,2}$	2.96	2.95	3.33	6.27	13.71
	$Q_{4,4}$	11.65	11.46	10.90	13.50	22.25
	$Q_{8,8}$	59.63	57.9	48.59	40.60	47.56
	$Q_{16,16}$	355.5	342.4	264.8	162.9	137.3

Table 4. Robustness of LinearSplit for GSTD Data

# I/Os		Tree – optimized for S(%) and T (duration)				
		$I_{1,1}$	$I_{2,2}$	$I_{4,4}$	$I_{8,8}$	$I_{16,16}$
Queries	$Q_{1,1}$	1.07	1.11	1.93	3.40	6.91
	$Q_{2,2}$	3.24	3.25	4.14	5.91	10.00
	$Q_{4,4}$	13.59	13.24	12.88	13.90	18.50
	$Q_{8,8}$	74.57	70.87	55.86	45.86	46.84
	$Q_{16,16}$	469.6	439.7	304.3	200.6	161.8

Looking at the rows of all tables, we can also observe that the query performance degrades on average only by 13% when queries were run against indices optimized for queries two times smaller or larger than the used query size. This indicates that the algorithms are very robust with respect to the assumed average query size. This is an important property for the case where the query load contains queries of significantly varying sizes, which we will explore in the following experiments. Note also that the LinearSplit algorithm in all cases performs very similarly to the OptimalSplit algorithm, at a much lower computational cost.

5.2 Number of Disk I/Os

5.2.1 Varying Query Size

In our second set of experiments we used the same databases as above, containing 50,000 trajectories, to study the performance of different query types: snapshot queries and range queries. While all queries are assumed to have varying spatial areas, snapshot queries have a temporal duration of one time interval, whereas range queries have durations of varying time intervals. Table 5 and Table 6 define the sizes of both types of queries, similarly to those used in [7].

Table 5. Snapshot query sizes

	Spatial extent in each dim. (S%)	Duration (T)
Small (SS)	1 – 3	1
Medium (SM)	3 – 9	1
Large (SL)	9 – 27	1

Table 6. Range query sizes

	Spatial extent in each dim. (S%)	Duration (T)
Small (RS)	3 – 9	1 – 3
Medium (RM)	3 – 9	3 – 9
Large (RL)	3 – 9	9 – 27

In this set of experiments we measure the number of I/Os at both the directory level and the data level, in order to provide a thorough analysis of the performance of the indices. Two different approaches are considered with respect to our algorithms: multiple index trees and a single index tree. In the multiple tree approach, we built an index tree for each query type separately (for each of our two algorithms). The trajectories are split for each index, considering the average size of the query type associated with the index, e.g., in case of the RS query type, the average query size is S=6% and T=2. In total, we built six different index trees for each of our algorithms in the multiple tree approach. A query was then run against the index tree that was optimized for the corresponding query type. In the single tree approach, we built only one index for all query types (for each of our two algorithms), i.e., we determine the average query size over all given query types (i.e., S=7.3% and T=4.83), and use this query size when splitting trajectories according to our cost models. The resulting index is then used to answer all types of queries.

The results of our experiments in this and the following section are presented using bar graphs. Figure 16 and Figure 17 show the average number of I/Os per query for the network data and the GSTD data, respectively, using the multiple tree approach. Figure 18 and Figure 19 show the average number of I/Os per query for the network data and the GSTD data, respectively, using the single tree approach. Note that using multiple trees or just a single tree affects only our algorithms OptimalSplit and LinearSplit, and that the values for the other algorithms are consequently the same for the same dataset.

Each bar in the figures represents the average number of I/Os per query and consists of two parts: the bottom (shaded) part corresponds to the average number of hits on the data level while the top (blank) part corresponds to the average number of hits on the directory levels of the indices (except for the FullSplit algorithm where the trajectory information is completely stored in the directory and consequently all hits are directory level hits).

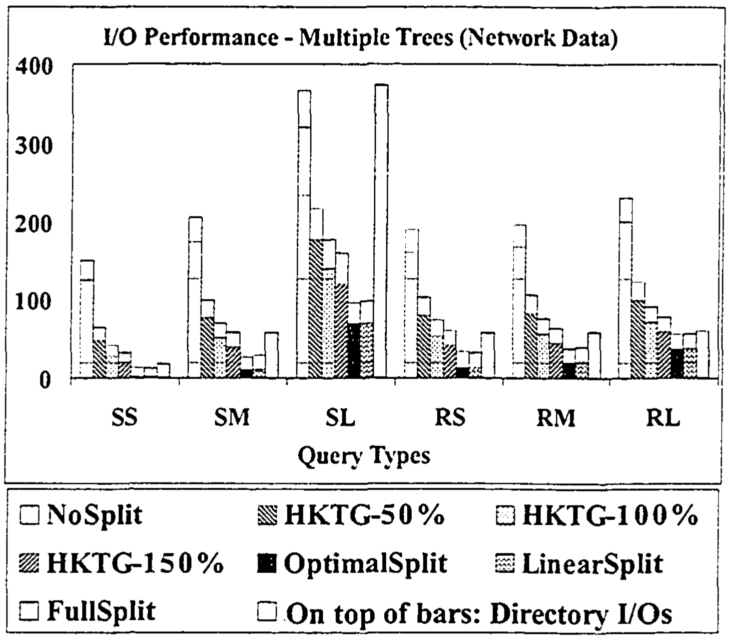


Figure 16. I/O Performance using multiple trees (Network Data)

In all scenarios, our approaches consistently outperform all others, and LinearSplit shows performance close to OptimalSplit, confirming again the suitability of the linear split heuristic. For SS and RL queries on the Network data, the FullSplit algorithm performs competitively to our approaches, however for other query types the performance can be much worse.

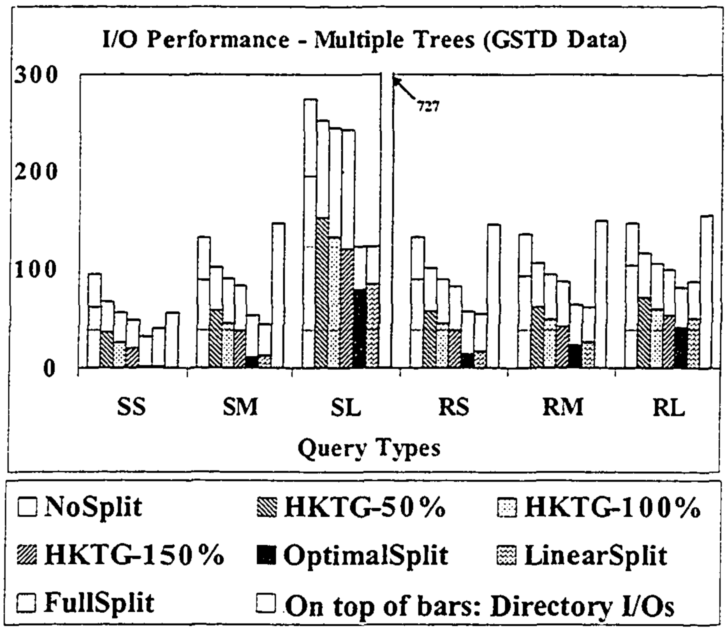


Figure 17. I/O Performance using multiple trees (GSTD Data)

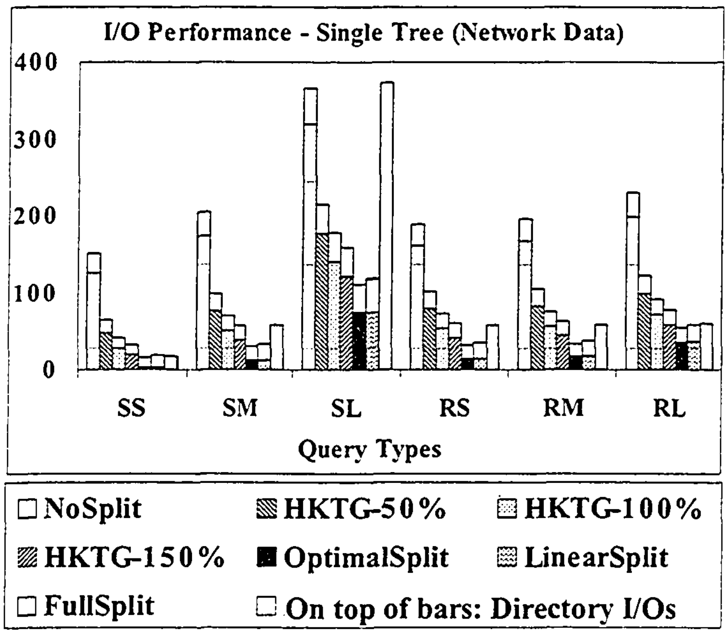


Figure 18. I/O Performance using a single tree (Network Data)

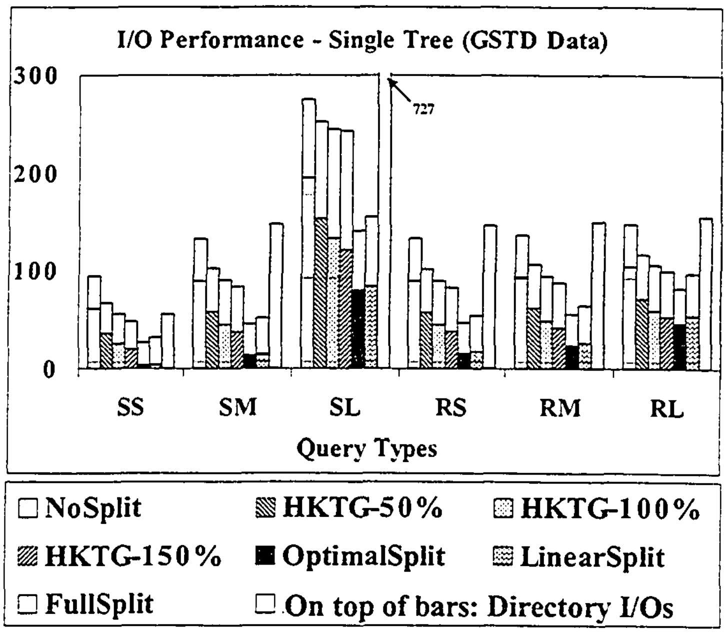


Figure 19. I/O Performance using a single tree (GSTD Data)

As we can see from the figures, our approaches have a significantly lower number of I/Os on the data level than the other algorithms (except again FullSplit since there is no separate data level) thus reducing the number of false hits. In addition to the unnecessary I/Os, false hits also add CPU time since computationally intensive algorithms have to be invoked to determine whether a trajectory segment approximated by the retrieved MBRs actually intersects the given query. Note also that our algorithms in general result in less directory I/Os than the NoSplit and the HKTG-*k*% algorithms even though our trees are typically larger since we introduce more splits, showing the effectiveness of our proposed split evaluation function for internal nodes. The FullSplit algorithm always has the largest possible tree since it introduces the maximum number of splits.

When comparing the performance of our algorithms using multiple trees with the performance using only a single tree for all query types, we can again confirm the robustness of our approach. The performance of a single tree is very close to the performance of multiple trees, which means that for the given datasets and query types, we can safely use only a single tree instead of six trees to efficiently support all given query types. In some cases, however, e.g., if the range of query sizes varies more dramatically, or if we have a multi-modal distribution of query sizes, or if certain types of queries should run as fast as possible (e.g. because of organizational reasons), it may be desirable to generate different trees for different query types. Note that in these cases the trajectory data does not have to be replicated; only different index directory structures have to be created.

5.2.2 Varying Database Size

In our third set of experiments we created indices for different database sizes, ranging from 10,000 to 50,000 trajectories. We ran medium sized range queries (query type RM) against all indices, where our indices where built for the average RM query size. The results are shown in Figure 20 and Figure 21.

The I/O performance or our algorithms is always significantly better than the NoSplit and the HKTG-k% algorithms. Although for the smaller datasets FullSplit performs competitively when compared to our approaches, it is clear that its performance degrades faster with increasing database size. The reason for this degradation is that FullSplit index is much larger than other approaches and at a certain database size a new directory level is introduced within the index.

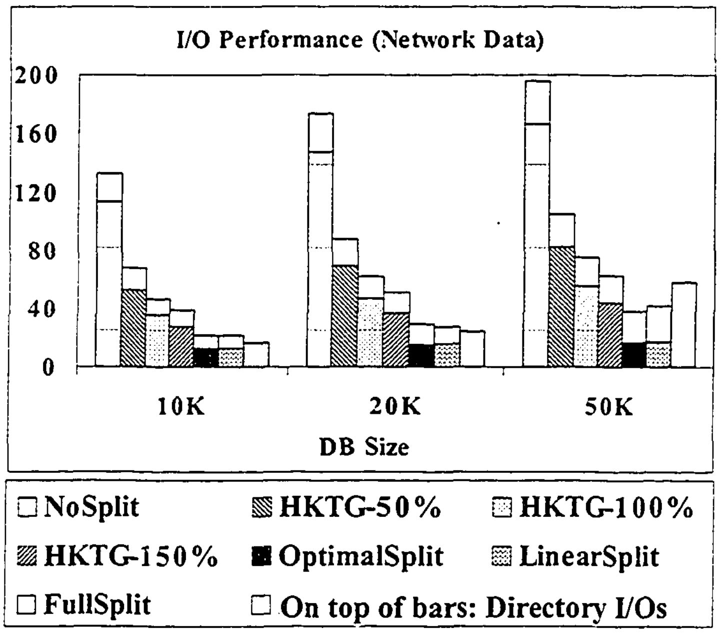


Figure 20. I/O Performance (Network Data)

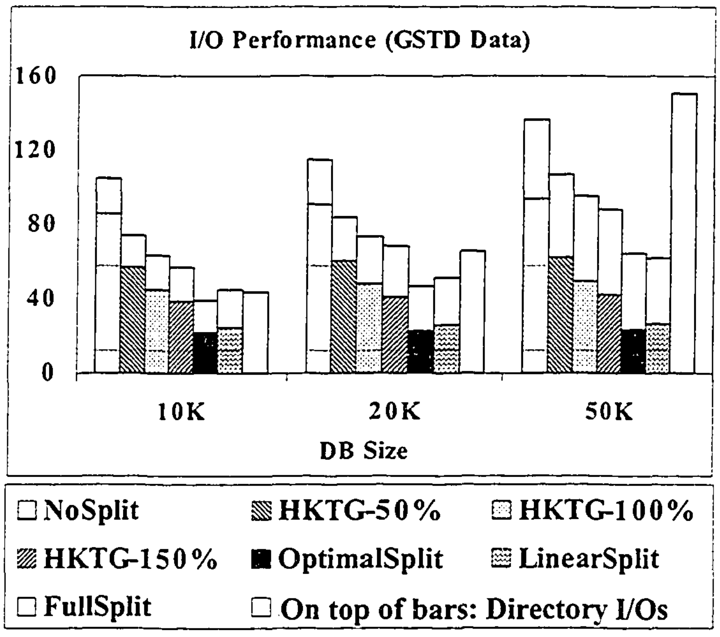


Figure 21. I/O Performance (GSTD Data)

Note that, even though in all experiments the OptimalSplit results, by design, in the smallest number of data level I/Os, this does not guarantee the best overall performance. In some cases LinearSplit exhibits the best performance due to a smaller number of directory level I/Os, which is due to the heuristic nature of our node splitting policy.

5.3 Index Size

In addition to I/O Performance presented in the previous section in case of both varying query and the database size, in this section we compare the index sizes of all of the approaches in both cases. Since the index size is directly proportional to the total number of MBRs that is inserted into the index, we use this number as a measure of index sizes of all of the approaches.

5.3.1 Varying Query Size

Figure 22 and Figure 23 show the number of MBRs of all of the approaches when varying query sizes for Network and GSTD data respectively. In case of our approaches, OptimalSplit and LinearSplit, we show the results obtained by the multiple tree approach. As we can see from the figures, the number of MBRs for our approaches is different for each query type since different indexes are created for each query type. The number or MBRs obtained by other approaches is the same for each query type since they do not take average query size into account. NoSplit creates only one MBR per object so it always results in a smallest index size. The HKTG-*k*% algorithms introduce a smaller number of splits than our approaches for each query type thus resulting in a smaller number of MBRs. FullSplit, however, uses a trivial splitting algorithm that creates an MBR around every two consecutive trajectory points resulting in a maximum possible number of MBRs. Therefore, the number of MBRs obtained by FullSplit approaches 5,000,000 for both Network and GSTD datasets. Although, the FullSplit splitting algorithm is fast and simple, the running time of the preprocessing step in this case is greatly influenced by the number of MBRs since each MBR has to be inserted into an index as shown in Section 5.4.

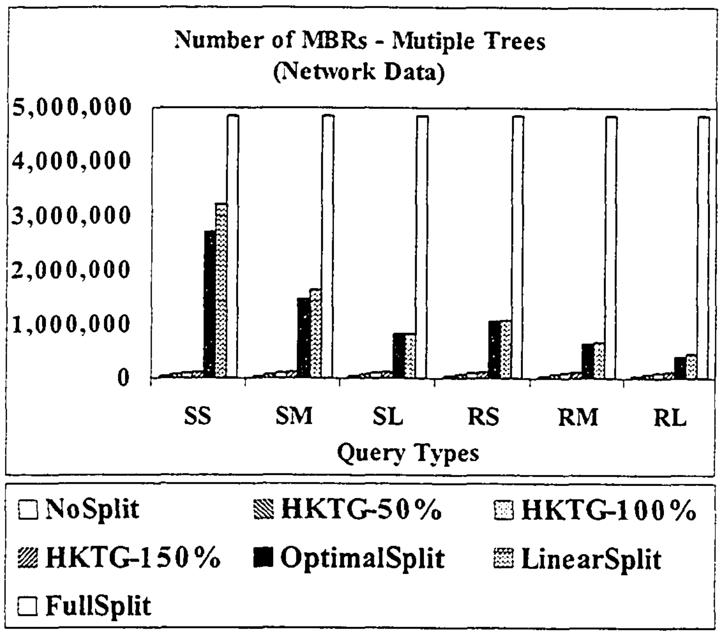


Figure 22. Number of MBRs for each query type (Network Data)

In case of the single tree approach, however, the number of MBRs for our approach is fixed as well since a single average query size is used for all of the query types resulting in a number of MBRs similar to the results for the RM query type. This behavior is expected since an average query size used in a single tree approach is close to the average query size used for RM query type. Therefore, in addition to a good I/O performance obtained by using the single tree approach, the resulting number of MBRs is relatively low reducing both storage requirements and insertion time.

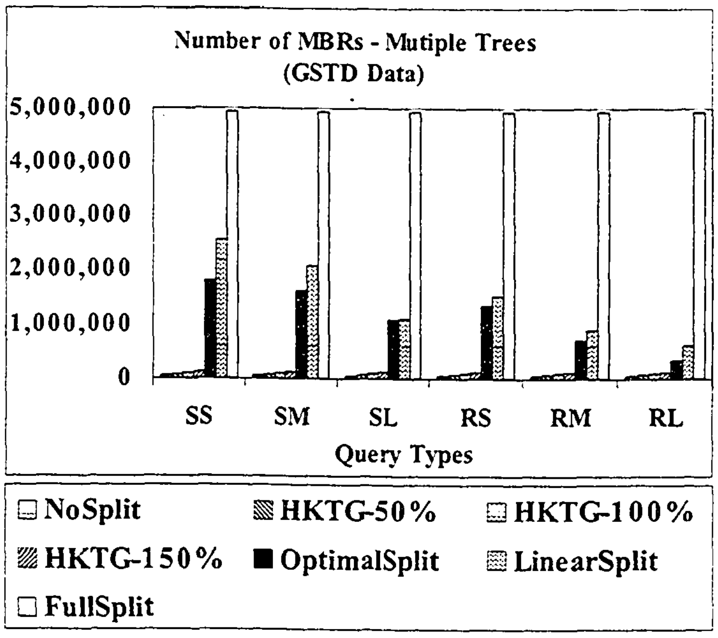


Figure 23. Number of MBRs for each query type (GSTD Data)

5.3.2 Varying Database Size

To measure the scalability of the index size with respect to database size, we used again our indices for the databases containing 10,000, 20,000, and 50,000 trajectories, where our indices where built for the average RM query size. The results in Figure 24 and Figure 25 show that all the algorithms scale linearly with respect to the number of MBRs when increasing database size.

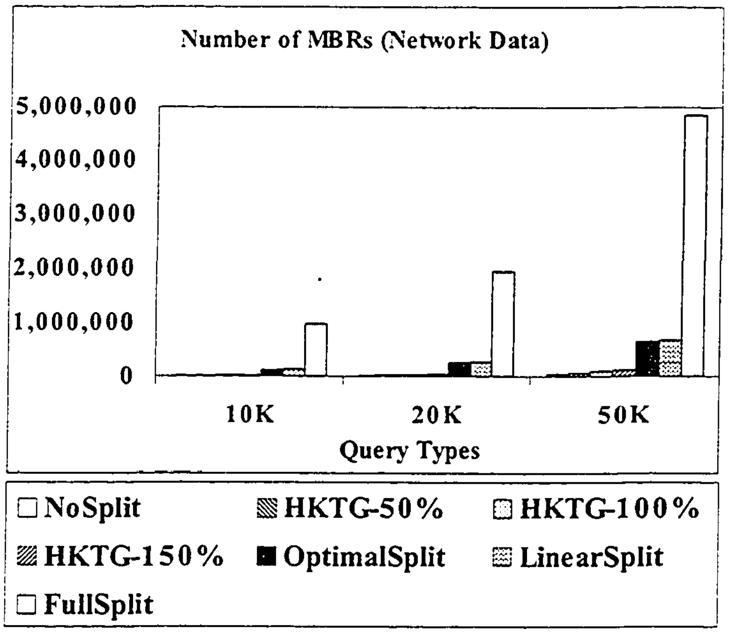


Figure 24. Number of MBRs (Network Data)

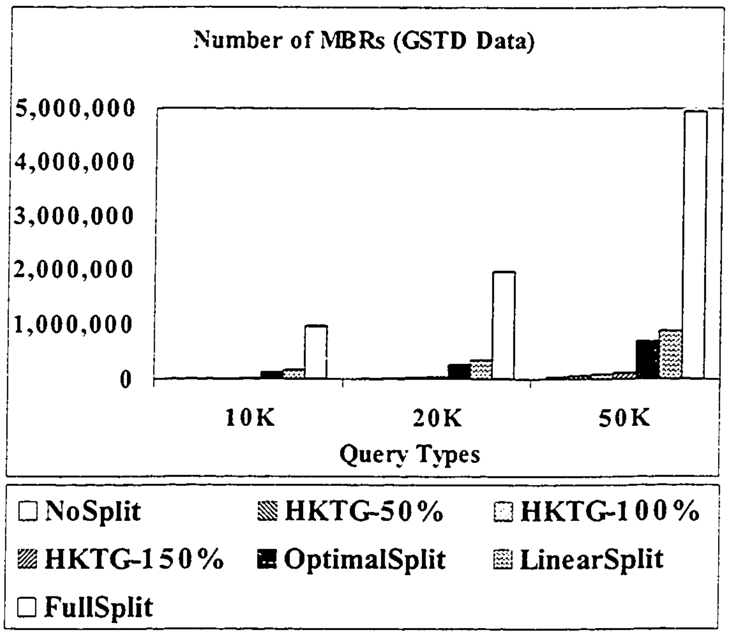


Figure 25. Number of MBRs (GSTD Data)

5.4 Index Building Time

In this section we compare the index building times of all of the approaches for both varying query and the database size.

5.4.1 Varying Query Size

The index building times for all algorithms, and using the average size of each query type for our approaches, are shown in Figure 26 and Figure 27. The NoSplit algorithm is clearly the fastest since it has to insert only one MBR per trajectory into the index. For the HKTG-*k*% algorithms, most of the time is spent splitting the trajectories due to the expensive splitting algorithm. The FullSplit has to insert one MBR per elementary segment of each trajectory consuming a significant amount of time. Our algorithms show a good balance between trajectory splitting time and insertion time, outperformed only by the trivial NoSplit algorithm. Also, as the query size used in our algorithms increases, our index building times decrease since trajectories are split less and therefore less MBRs are inserted.

Note that, in principle, we could provide the optimal number of splits found by our optimal split algorithm as an input parameter value to the HKTG-*k*% algorithm (setting *k* appropriately). However, when using our datasets we could not finish building these trees even after a few days. We did test the HKTG-*k*% with the optimal number of splits for very small data sets, up to 5,000 trajectories. On those dataset, the number of data level I/Os of the HKTG-*k*% algorithm is very close to ours, however, the overall I/O performance for queries is much worse than ours due to a larger overhead for directory level I/Os.

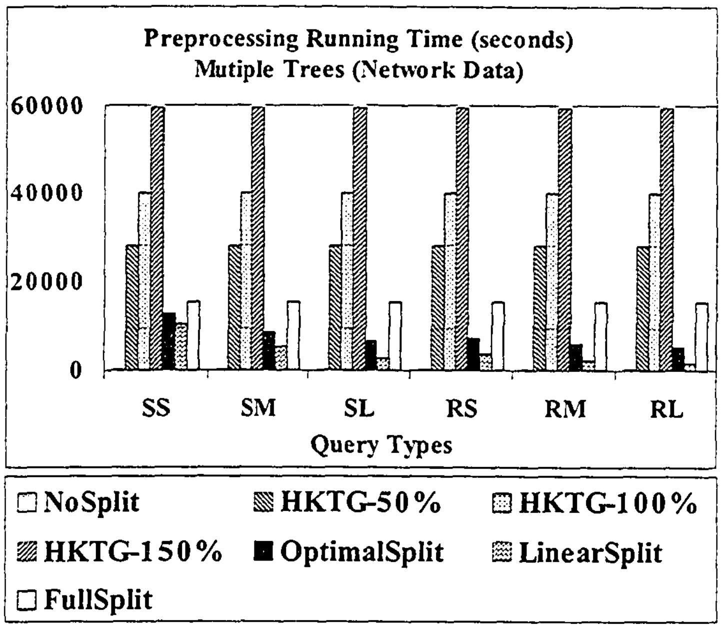


Figure 26. Preprocessing time in seconds for each query type (Network Data)

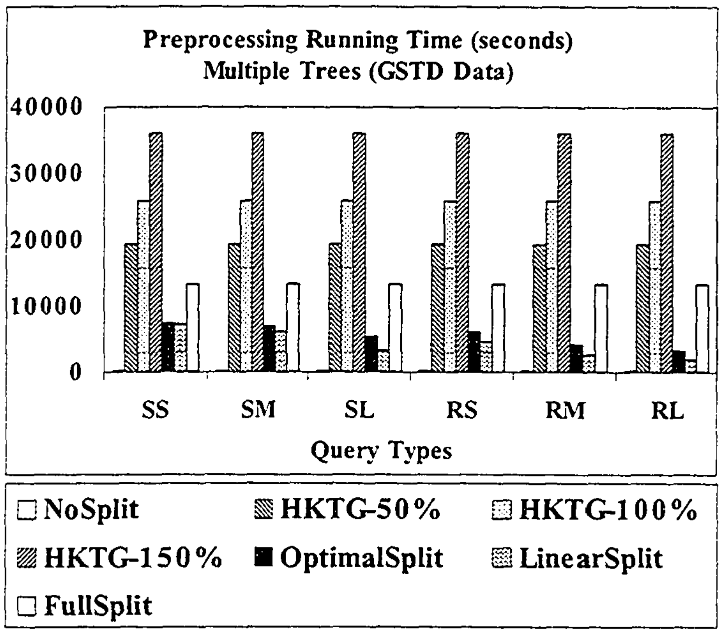


Figure 27. Preprocessing time in seconds for each query type (GSTD Data)

5.4.2 Varying Database Size

In this experiment, we measure the scalability of the index building time with respect to database size. We again use the same datasets with number of trajectories varying from 10,000 to 50,000, and build our indices for the average RM query size. The results are shown in Figure 28 and Figure 29 for Network and GSTD datasets respectively. As expected the index building time for all algorithms increases as the database size increases. Our algorithms scale linearly at a much slower rate than all other ones (again with the exception of the trivial NoSplit algorithm).

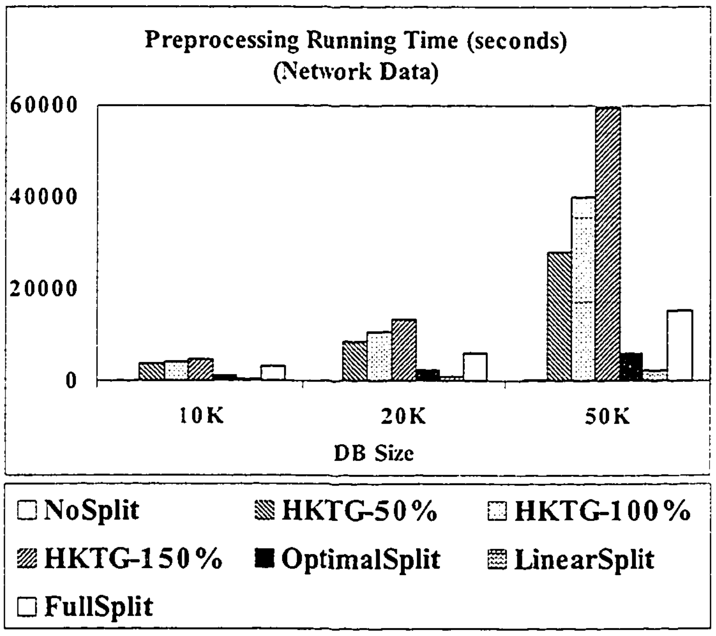


Figure 28. Preprocessing time in seconds (Network Data)

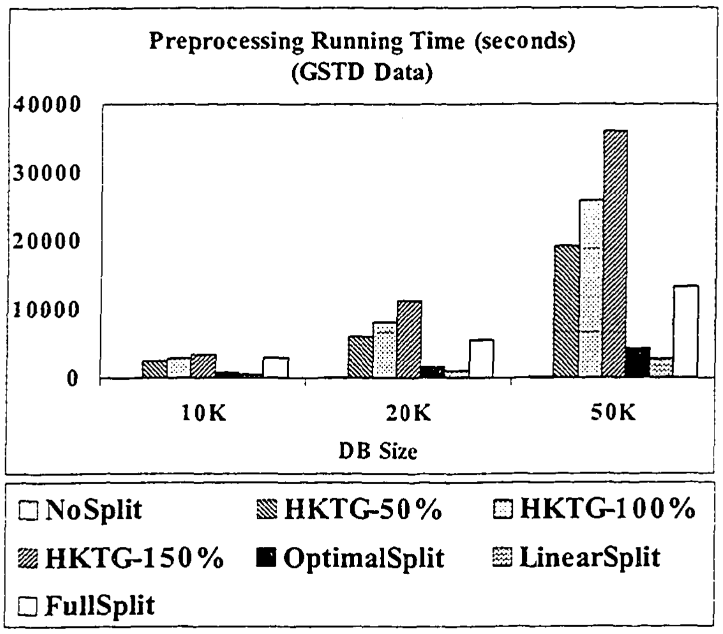


Figure 29. Preprocessing time in seconds (GSTD Data)

Chapter 6

Conclusion

In this work we investigated the problem of splitting spatiotemporal trajectories in order to improve the performance of spatiotemporal queries. To index the trajectories, MBR-based access structures are typically used. We argued that splitting trajectories with the goal of minimizing the volume of the resulting MBRs alone is not the best strategy. A better solution is obtained when taking into account average query sizes. We presented a cost model for predicting the number of data page accesses, and an optimal trajectory splitting algorithm based on this model, which minimizes the expected data page accesses, given an average query size. However, the optimal splitting algorithm may not be suitable for splitting long trajectories due to its time complexity. In addition, the optimal splitting algorithm requires that the complete trajectories are known in advance. Therefore, it is not applicable in a dynamic case where the trajectories are continuously growing. Using our cost model and approximating trajectories by constant-slope segments, we formally derived a linear time splitting algorithm, which can be applied in a dynamic environment.

Using the R-tree as the underlying access structure, our experimental results show that, overall, our proposed trajectory split policies consistently outperform other previously proposed policies, up to 6 times less disk I/Os than FullSplit and up to 5 times less disk I/Os than the approaches proposed in [7]. Although our indices are built assuming a pre-determined query size, our algorithms are robust in the sense that the built indices efficiently support a much wider range of query sizes. Our algorithms scale well with respect to database size for both query performance and index building time. Finally, we also confirmed in our experiments that the LinearSplit algorithm performs similarly to the OptimalSplit algorithm, at a much lower cost.

Directions for future research include extending our cost model to better understand the effect of directory level page accesses and designing optimized split policies for directory pages of spatiotemporal indices. For this, we also will explore the effect of different distributions of trajectories in space and time.

Bibliography

[1] Barequet, G., Har-Peled, S.: Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. In *ACM Symp. on Discrete Algorithms*, pp.829, 1999.

[2] Beckmann, N. , Kriegel, P. , Schneider, R., Seeger, B. The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proc. of the ACM SIGMOD Conf.*, pp. 322-331, 1990.

[3] Benetis, R. Jensen, C.S. Karciauskas, G. and Saltenis, S: Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *Proc. Of the IDEAS 2002*, pp. 44-53, 2002.

[4] Brinkhoff, T., Kriegel, H.P., Schneider, R.: Comparison of Approximations of Complex Objects Used for Approximation-based Query Processing in Spatial Database Systems, In *Proc. of the IEEE 9th Data Engineering Conf*, pp. 40-49, 1993.

[5] Chakka, V., Everpaugh, A., Patel, J.: Indexing Large Trajectory Sets with SETI. In *Proc. of the Conf. on Management of Data*, 2003.

[6] Guttman, A. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proc. of the Intl. Conf. on Management of Data SIGMOD*, pp. 47-57, 1984.

[7] Hadjieleftheriou, M., Kollios, G., Tsotras, V., Gunopulos, D.: Efficient indexing of Spatiotemporal Objects. In *Proc. Of the Intl. Conf. On Extending Database Technology,* pp. 251-268, 2002.

[8] Kollios, G., Gunopulos, D., Tsotras, V.: On Indexing Mobile Objects. In *Proc. Of the 18th ACM Symposium. On Principles of Database Systems*, pp. 261-272,1999.

[9] Mokbel, M., Ghanenm, G., Aref, W.: Spatio-Temporal Access Methods. In *Bull. of the Technical Committee on Data Engineering,*26(1), pp. 40-49, 2003.

[10] Nascimento, M., Silva, J., Theodoridis, Y.: Evaluation of Access Structures for Discretely Moving Points. In *Proc. of the Intl Workshop on Spatio Temporal Data Management,* pp. 177-188, 1999.

[11] Nascimento, M., Silva, J.: Towards Historical R-trees. In *Proc. of the ACM Symp on Applied Computing*, pp. 235-240, 1998.

[12] Pagel, B., Six, H., Toben, T., Widmayer, P.: Towards an Analysis of Range Query Performance in Spatial Data Structures. In *Proc. of ACM On Principles of Database Systems*, pp. 214–221, 1993.

[13] Papadias, D., Tao, Y., Kalnis, P. and Zhang, J.: Indexing Spatio-Temporal Data Warehouses. In *Proc. of the International Conference on Data Engineering*, pp.166-175, 2002.

[14] Pfoser, D., Jensen, C., Theodoridis, Y .: Novel Approaches to the Indexing of Moving Object Trajectories. In *Proc. of the 26th International Conference on very Large Databases*, VLTB, pp. 395-406, 2000.

[15] Saltenis, S. , Jensen, C. , Leutenegger, S. , Lopex, M.: Indexing the Positions of Continuously Moving Objects. In *Proc. Of the ACM Intl. Conf. Of Data, SIGMOD*, pp.331-342, 2000.

[16] Tao, Y., Papadias, D. and Sun, J. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *Proc. of the Conf on Very Large Databases*, pp. 790-801, 2003.

[17] Theodoridis, Y., Vazirgiannis, M., Sellis, T.: Spatio-Temporal Indexing for Large Multimedia Applications. In *Proc. of the 3rd IEEE Conf. on Multimedia Computing and Systems*, pp. 441-448, 1996.

[18] Theodoridis, Y., Silva, R., Nascimento, M.: *On the Generation of Spatiotemporal Datasets.* In *Proc. of the 6th Intl. Symposium on Spatial Databases*, pp. 147-164, 1999.

[19] Tzouramanis, T., Vassilakopoulos, M. and Manolopoulos Y.: Overlapping Linear Quadtrees and Spatio-Temporal Query Processing. *The Computer Journal 43(4)*, pp. 325-343, 2000.

[20] Vlachos, M., Kollios, G., Gunopulos, D.: Discovering Similar Multidimensional Trajectories. In *Proc. of the 18th International Conference on Data Engineering*, pp. 673–684, 2002.

[21] Zhu, H., Su, J., Ibarra, O.: Trajectory Queries and Octagons in Moving Object Databases. In *Proc. of the ACM Conference on Information and Knowledge Management*, pp. 413-421, 2002.

[22] Brinkhoff, T.: Generating Network-Based Moving Object, In *Proc. of the 12th International Conference on Scientific and Statistical Database Management*, pp. 253-255, 2000.

[23] van der Bercken, J., Blohsfeld, B., Dittrich, J.-P., Krämer, J., Schäfer, T., Schneider, M., Seeger B.: *XXL - A Library Approach to Supporting Efficient Implementations of Advanced Database Queries. VLDB 2001*, pp. 39-48.

[24] Tao, Y., Papadias, D.: Mv3r-tree: a spatio-temporal access method for timestamp and interval queries. In *Proc. of the VLDB*, 2001.