A Trajectory Splitting Model for Efficient Spatio-Temporal Indexing

Slobodan Rasetic Jörg Sander James Elding Mario A. Nascimento

Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
{rasetic, joerg, elding, mn}@cs.ualberta.ca

Abstract

This paper addresses the problem of splitting trajectories optimally for the purpose of efficiently supporting spatio-temporal range queries using index structures (e.g., R-trees) that use minimum bounding hyper-rectangles as trajectory approximations. We derive a formal cost model for estimating the number of I/Os required to evaluate a spatio-temporal range query with respect to a given query size and an arbitrary split of a trajectory. Based on the proposed model, we introduce a dynamic programming 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 to be used in a dynamic case where trajectory points are continuously updated. Our experimental evaluation confirms the effectiveness and efficiency of our proposed splitting policies when embedded into an R-tree structure.

1. Introduction

Producing and collecting large volumes of spatio-temporal data has become more practical in recent years, leading to increased availability and consequently the need for efficient management of this type of data. For many applications, it is important to track, store, and query data about moving objects, for instance, to deliver real time services to clients based on spatial and temporal context. Application areas include fleet control, wireless communication networks, and mobile computing.

In this paper, we focus on spatio-temporal queries over historical trajectory data. Trajectories are often used to represent the path of moving objects. The authors in [12] distinguish two main types of spatio-temporal queries: *coordinate-based queries* and *trajectory-based queries*. *Coordinate-based queries* return only the ids 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 to determine possibly complex topological relationships (e.g., whether they cross or bypass an 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 the important class of trajectory-based queries efficiently, specialized index structures that support spatio-temporal range queries are needed. Virtually all spatio-temporal index structures proposed in the literature are derived from spatial index structures such as R-trees [9]. These approaches are based on the intuition that spatio-temporal data can be viewed as spatial objects in an extended spatial domain, where time is treated as an additional dimension. Trajectories are then represented by minimum bounding rectangles (MBRs).

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 [12], 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

**Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005**

alternative is to split trajectories and approximate the resulting sub-trajectories independently by MBRs, balancing between the size of the index and the approximation quality, which may lead to a better overall performance.

The problem of splitting trajectories optimally with the goal of minimizing the expected number of I/Os has not yet been treated rigorously. To our best knowledge, the only work that addresses this 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 paper are the following:

- We derive an analytical cost model for evaluating the split of a trajectory into segments (in terms of expected I/Os), given a spatio-temporal range query.
- Based on this model, we introduce a dynamic programming solution for splitting a given set of trajectories optimally.
- In order to deal with dynamic cases where trajectories are updated incrementally, we derive another cost model that estimates an optimal length for segments when "incrementally" splitting a trajectory.
- Combining our cost models, we develop a linear time trajectory splitting algorithm, which in practice performs close to the optimal algorithm, and which can be used in dynamic cases.
- Finally, we demonstrate through an extensive experimental evaluation that our algorithms are both efficient in practical situations and significantly outperform other trajectory splitting approaches.

The rest of the paper is organized as follows. Section 2 provides background and motivation for the paper. In Section 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 Section 4, a linear time algorithm for splitting trajectories heuristically is formally derived. A thorough experimental performance evaluation and comparison is presented in Section 5. Section 6 discusses related work and Section 7 concludes the paper.

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. The leaf nodes store the MBRs of data objects and pointers to the object and their exact geometry. Internal nodes store a sequence of pairs consisting of an MBR and a pointer to a child node. These MBRs enclose all entries stored in the sub-tree having the referenced child node as its root. 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.

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

In order to index trajectories using an R-tree, each trajectory (or each of its segments if the trajectory is split) is approximated by a 3-dimensional MBR. It is easy to see that splitting trajectories, offers a great potential for improving the performance of spatio-temporal range queries. When splitting a trajectory, the total volume of the approximating MBRs decreases, and consequently the approximations may less likely intersect range queries, leading to an overall reduction of the number of data pages that have to be retrieved when processing these queries. The actual amount of volume reduction when splitting a trajectory depends not only on the number of splits but also on the split points, as illustrated in Figure 1.

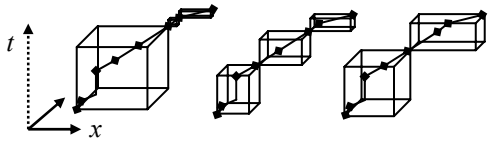


Figure 1. MBRs for different trajectory splits.

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 called *DPSplit* is proposed for splitting a single trajectory T using l splits so that the total volume of T's approximations is minimized. The complexity of this algorithm is $O(t^2l)$, 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 n 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 $O(ns^2)$. 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. These approaches have several drawbacks:

- Minimizing the total volume of trajectory approximations does not necessarily minimize the number of expected I/Os when processing range queries. Introducing more splits necessarily reduces the total volume of the trajectory approximations. However, it also increases the number of segments that may simultaneously intersect a query range, resulting in unnecessary I/Os. Figure 2(a) shows a scenario where a trajectory has an unnecessarily large number of splits for the given query size; Figure 2(b) shows that 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 continuously grow 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.

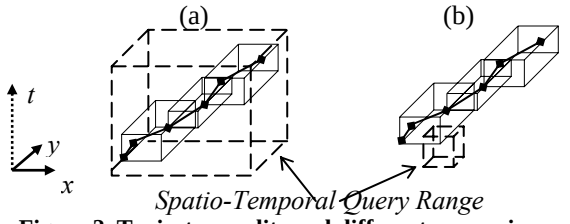


Figure 2. Trajectory splits and different query sizes.

We conclude that minimizing the volume of trajectory approximations is not enough to minimize the expected number of I/Os for spatio-temporal range queries. We claim that we also need to take into account query sizes. Assuming a query size may seem unusual and limiting at first glance, but we will see that it has several advantages: (1) our cost model will also show that the two straightforward solutions for approximating trajectories in R-trees –not splitting, and splitting at each trajectory point– are in fact just special cases, corresponding to trees that are optimized for an extremely small or extremely large query size, respectively; these correspond to "worst case" assumptions about the query size for most real applications. (2) Having a query size offers a potential for tuning the index which is an option that the other splitting alternatives cannot provide. In practical applications, the assumed query size can be determined as the average query size computed from a workload of queries. (3) An average query size is also a more natural and robust parameter than a user-specified total number of splits as in [7], and is not restricted to static datasets.

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 for data pages is minimized.

3.1 A cost model for splitting trajectories

Given a trajectory $T=\langle p_1, p_2, \ldots, p_t \rangle$ with $p_i = (x_i, y_i, t_i)$, we denote a *segment* of T that starts at point p_u and ends at point p_v by $T[u,v]$ (using this notation $T = T[1,t]$).

A trajectory can be split along its discrete temporal dimension into m segments ($1 \le m \le t-1$) in $\binom{t-2}{m-1}$ possible ways (by choosing $m-1$ split points from T, excluding the endpoints p_1 and p_t). A 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 segment $T[u,v]$. We denote the set of the MBR approximations of all possible decompositions of T into m segments by

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

For our cost model, we assume that segments and their MBRs are stored independently, e.g., under an R-tree. That means that the MBRs of a trajectory are generally stored on different disk pages. Under this assumption and ignoring possible effects of an index directory and caching, each segment's MBR that is intersected by a query will require an independent disk I/O.

Denoting the MBR approximation of a specific decomposition of T into m segments by $B^T = (B_1, \ldots, B_m)$, 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 I/Os due to B^T can be derived as follows. If q intersects B^T, then it intersects exactly k segments simultaneously, where $1 \le k \le m$, yielding exactly k I/Os. The mutually exclusive events that q intersects exactly k segments of B^T (and thus resulting in k I/Os) occur with probability $P(q \cap B^T; k)$. Consequently, the overall expected number of I/Os, $E_{B^T}(q)$, is the sum of the I/Os due to each event, weighted by the probability of the event, i.e.:

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

The following lemma simplifies this expectation.

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

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

That means 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. (A proof of $\sum_k k \cdot P(q \cap B^T; k) = \sum_k P(q \cap B_i)$ for the general case of MBRs for spatial data can be found in [11]).

In order to determine $P(q \cap B_i)$, we consider in a finite data space S the area where a query q can fall and at the same time intersect B_i. This area, denoted by $Ext_q(B_i)$, is given by extending B_i by half of the query extension in each dimension (see Figure 3). Clearly, the query intersects an MBR B_i if and only if the query center is inside the *query extended MBR* $Ext_q(B_i)$.

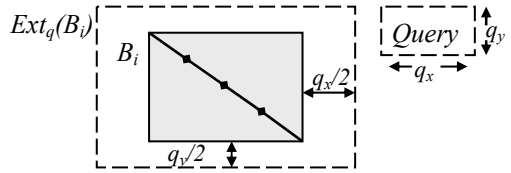


Figure 3. A Query Extended MBR.

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 $Ext_q(B_i)$:

$$P(q \cap B_i) = Vol(Ext_q(B_i)) / Vol(S) \qquad (4)$$

By substituting Equation (4) into (3), we obtain:

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

Minimizing this performance measure for a single trajectory T, means finding among all possible decompositions of T into all possible numbers of segments m, the split with the minimum expected number of I/Os, 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 4 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 more splits will increase the sum of the volumes of the query extended MBRs.

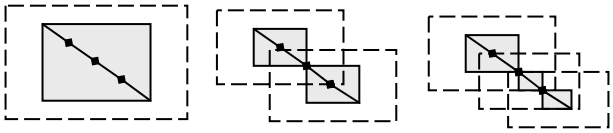


Figure 4. Volumes of approximations using 0, 1, or 2 splits. Grey areas represent segments' MBRs, and dashed lines show the query extended MBRs.

So far, we have only considered how to split a single trajectory optimally. Optimally splitting a *set* of trajectories Θ, theoretically depends also on the directory structure of the particular spatial index used to store the MBRs. This directory structure depends on the page size, the distribution of the trajectories in space and the split algorithm used for splitting overfilled node when constructing the index. Modeling these aspects even for purely spatial data seems infeasible, and only empirically justified heuristics for splitting directory nodes in R-trees have been proposed so far in the literature. Therefore (and to be independent of the used index structure), we restrict our model to the I/O cost due to the data level of an index structure (which dominates the total query cost for most R-trees), and 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 data pages, given a query q, i.e., the expected number of I/Os can be computed as the sum of the individual expectations:

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

Given Equation 6, we can find the optimal splits for a set of trajectories (with respect to q), by minimizing the splits for each trajectory individually.

In general, a trajectory T can be split into m segments in different ways, possibly resulting in a different number of I/Os when processing q. Let $E^{opt}_{T,m}(q)$ be the minimum expected number of I/Os that can be obtained for T by splitting T into m segments:

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

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

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

As a consequence of this cost model, it is easy to see that an R-tree obtained by not splitting any trajectory is equivalent to an R-tree that is optimized for a query size where even splitting the largest trajectory will result in a larger expected number of I/Os than not splitting, which is only possible if the query is very large. Similarly, an R-tree obtained by splitting every trajectory into all its segments is equivalent to an R-tree that is optimized for a query size where every split of a trajectory introduces a gain in the expected number of I/Os, which is only possible if the query is very small. Because of space limitations, we have to omit the analytical details here, but clearly, those query sizes are not representative for most real applications. However, they are implicitly and unchangeably integrated into these respective R-trees.

Dynamic Programming Algorithm

To solve Equation 8, we propose a dynamic programming solution, which finds the best possible split of T for each value of m. Using our notation $T[u,v]$ to denote a segment 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 the following property:

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. (*Sketch*) Using Equation 5, Equation 7 can be rewritten 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

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

Let the start position of the last segment of an *optimal* decomposition of T be u, and let $B^{opt}_m = MBR(T[u,t])$ denote the MBR of this last segment. Then we obtain

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

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

This equation holds since the last segment (starting at u) is fixed by assumption, and the remaining prefix of T, $T[1,u]$ 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. To find the optimal decomposition of T without knowing u, we have to consider all possible values of start positions u in the range $1 < u < t$ for the last segment of T, as stated in the theorem:

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

Theorem 1 states that in order to find the optimal solution for a trajectory T using m segments, it is sufficient to consider all optimal sub-solutions using m–1 segments for the prefixes $T[1,u]$, $1 < u < t$, (which can be found by recursively applying Equation 9), and combine them with the solution for the remaining segment $T[u,t]$.

The runtime of a dynamic programming algorithm that accordingly determines 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, the algorithm has to be applied for the maximum possible value of m, i.e., for $m=t$–1. To split n trajectories optimally, the algorithm has to be applied n times. This time complexity is the same as the time complexity for the *DPSplit* precomputation step used in Hadjieleftheriou et al.'s algorithms [7]. Our algorithm, however, does not need to execute an additional, time consuming and storage intensive search algorithm on top of this solution 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. Assuming the MBRs approximating trajectory segements are stored independently on 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, directory pages may be split during index construction and during updates. Different heuristics have been proposed for that purpose, such as the quadratic and the linear split [6], or the R*-tree split [1]. These algorithms generate a certain subset of all possible splits of an MBR and minimize evaluation functions, which are typically based on volume and overlap of the resulting MBRs. The goal of these heuristics is essentially to minimize the probability that queries will intersect both resulting MBRs thus reducing the number of subtrees that have to be traversed.

The rationale behind our cost model can be applied to directory level splits as a heuristic evaluation function as well: Given the average query size used to split trajectories, we can choose among the possible splits of a directory node the split that minimizes the volume of the resulting query extended MBRs using Equation 5. More precisely, we replace the node split evaluation function using our cost model, keeping the original algorithm for generating candidate node splits.

4 Heuristic Trajectory Splitting

The above split strategy requires complete trajectories to be available in order to find the optimal splits. For many applications, however, trajectories are updated continuously. Another limitation is that for large datasets containing long trajectories, even if they were completely available, the dynamic programming solution may be too inefficient to be practical. For such applications, a more efficient and incremental method is needed, which ideally can produce near optimal results.

4.1 A Cost Model for Optimal Segment Size

In this section we assume trajectories where points are continuously added over time. We formally derive an approximation of the optimal split of a trajectory that can be computed incrementally.

Consider first the special case of trajectories for objects moving with constant speed in a constant direction that are sampled at constant time intervals[1] (see Figure 5 for a 2*d* illustration). For brevity, we call such trajectories "constant-slope trajectories". We will show that the optimal split of these trajectories, according to our previous cost model, will result in segments of equal size. Using this property, pieces of arbitrary trajectories can be approximated by constant slope trajectories and split "near-optimally" in linear time.

[1] 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.

Assume a trajectory T consisting of t points, or equivalently, consisting of t–1 consecutive *elementary* segments s_1, …, s_{t-1} as well as a decomposition of T, $B^T=\{B_1, \ldots, B_m\}$. The sum in Equation 5 can be expressed differently by thinking of the volume of each $Ext_q(B_i)$ as being "generated" by the elementary segments contained in B_i, via a function f that expresses an equal contribution of each elementary segment s to the volume of the query extended MBR it belongs to:

$$f(s) = \frac{Vol(Ext_q(B_i \text{ containing } s))}{\#\text{ elementary segments in } B_i \text{ containing } s} \quad (10)$$

Lemma 2. Let $B^T=\{B_1, \ldots, B_m\}$ be a decomposition of a trajectory T, and f be defined as in Equation 10. Then,

$$\sum_{i=1}^{m} Vol\,(Ext_q(B_i)) = \sum_{i=1}^{m}\left(\sum_{s \text{ contained } \text{ in } B_i} f(s)\right) = \sum_{i=1}^{t-1} f(s_i) \quad (11)$$

Proof. Trivial. By "construction" of f, it holds that $\sum_{s \text{ contained in } B_i} f(s) = Vol(Ext_q(B_i))$ for each B_i in B^T. ■

The significance of Lemma 2 is that a decomposition that minimizes the right hand side also minimizes the left hand side. Minimizing the right hand side is in general not an easier problem, since the f values for elementary segments depend on where the actual split points for a split of T are. However, for constant-slope trajectories, the f values depend only on the number of elementary segments in an enclosing MBR B_i, i.e., we can compute the volume of $Ext_q(B_i)$ using only the increments in each dimension $(\Delta x, \Delta y, \Delta t)$ that define the slope of the trajectory, and the number c of elementary segments in B_i:

$$Vol(Ext_q(B_i)) = (c \cdot \Delta x + q_x) \cdot (c \cdot \Delta y + q_y) \cdot (c \cdot \Delta t + q_t) \quad (12)$$

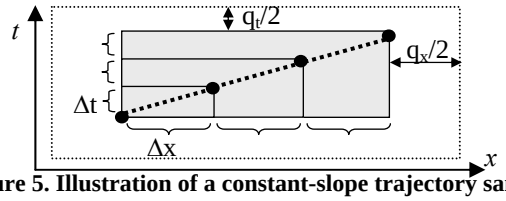


Figure 5. Illustration of a constant-slope trajectory sampled at regular time intervals.

Furthermore, we can now look at the values of f for MBRs of varying number of contained elementary segments c by looking at its definition as a function g of c,

$$g(c) = \frac{(c \cdot \Delta x + q_x) \cdot (c \cdot \Delta y + q_y) \cdot (c \cdot \Delta t + q_t)}{c} \quad (13)$$

The significance of this function is that $g(c)$ has a real minimum, which means that there is an optimal *segment length* c_{opt} (i.e., optimal number of consecutive elementary segments that form the segments) for constant slope trajectories T, in the sense that if T is decomposed into segments of this length, the value $f(s)$ will be minimal for each elementary segment s. This in turn means that this decomposition minimizes the right hand side of Equation 11, giving us an optimal decomposition according to our cost model in Section 3. In this optimal decomposition, all the segments have the same length c_{opt}, which is independent of the length of T. The value c_{opt} determines where the split points have to be; it is only dependent on the increments in each dimension in T and the query size, which also means that we can optimally split constant-slope trajectories in an incremental manner, i.e., after some points of T have been added. We will use this fact later to derive a heuristic incremental splitting algorithm for arbitrary trajectories by conceptually approximating them with 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 13) 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 $g(c) = (k_1c^2 + k_2c + k_3 + k_4\frac{1}{c})$

Applying the first derivative to find extremas, 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

$$(2k_1c^3 + k_2c^2 + (-k_4)) = 0 \quad (14)$$

This cubic equation has an analytical solution c_{opt} in the domain of positive real numbers. The second derivative is always greater than 0, so $g(c)$ has a minimum at $c= c_{opt}$. ■

Intuitively, we can use the value c_{opt} that minimizes $g(c)$ to construct an optimal decomposition of a constant-slope trajectory T (according to the cost model in section 3) by dividing it into segments of equal length, containing c_{opt} many elementary segments. In practice, since c_{opt} does not depend on the number of trajectory points and can be computed using only the increments $\Delta x, \Delta y, \Delta t$ of T and the query size, we can split T into segments of length c_{opt} continuously as points are added to T over time.

Theorem 3. Given a query q, and a constant-slope trajectory $T = \langle p_1, p_2, \ldots, p_t \rangle$ 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[2] by c_{opt}, the decomposition of T into segments of equal length 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, let c_{opt} be an integer.[3] 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 seg-

[2] This is justified by the fact that we assume *long* trajectories where points are continuously added.

[3] If c_{opt} is not an integer, we can re-sample T so that c_{opt} can be expressed as an integer w.r.t. the new elementary segment size.

■

ments s. The resulting values $f(s)$ (Equation 10) for each elementary segment s is by Theorem 2 minimal, i.e., no other MBR size can give smaller $f(s)$ values. Consequently, the sum $\sum_{i=1,\ldots,t-1} f(s_i)$ is minimal for the decomposition $B^T_{c^{opt}}$ among all possible decompositions. By Lemma 2, this decomposition is also 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 a constant-slope trajectory T^{Δ} in the following way. We can compute the increments $\Delta x, \Delta y, \Delta t$ that define the slope of T^{Δ} as the average of the corresponding increments of T, e.g., in x direction: $\overline{\Delta x} = \frac{1}{t-1}\sum_{i=1}^{t-1}\Delta x_i$, where Δx_i represents the difference in x direction between two consecutive points of T. Obviously, the smaller the variance in the increments of T is, the better is the approximation T^{Δ}. Although the error of the approximation can be large for *long* trajectories, this is not true for sub-trajectories in case of most real world applications since objects usually keep moving in a similar direction with a similar speed for certain periods of time. The fact that we can typically approximate a long trajectory well, using several constant-slope sub-trajectories, allows us to design an incremental splitting algorithm that performs nearly optimal in practice (unless objects move extremely erratically).

4.2 Linear Time Trajectory Splitting

To split a trajectory T incrementally, we can buffer a certain number of incoming points of 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 before applying the split policy. This number may depend on several 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 appropriate. The cost model for optimally splitting a trajectory from Section 3 can be used as a heuristic to determine dynamically a suitable buffer size. We can determine when an MBR around a trajectory segment $T[u,v+1]$ is not optimal, according to the following condition.

$$E_{T[u,v+1],1}(q) > E_{T[u,v],1}(q) + E_{T[v,v+1],1}(q) \qquad (15)$$

This condition is true, when 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 $T[u,v+1]$. In this case, it makes sense to consider splitting $T[u,v+1]$ since there is at least one possible split (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 15 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. Equation 15 is good at detecting significant changes in speed and direction of a trajectory. For nearly constant-slope segments of a trajectory, the condition tends to be 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 15.

$$E_{T[u,v+1],1}(q) - [E_{T[u,v],1}(q) + E_{T[v,v+1],1}(q)] \qquad (16)$$

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 12. The number of elementary segments in $T[u,v]$ is $c = v - u$.

After simple arithmetic transformations, we obtain:

$$E_{T[u,v+1],1}(q) - E_{T[u,v],1}(q) - E_{T[v,v+1],1}(q) = 3k_1c^2 + 3k_1c + 2k_2c - k_4$$

where k_1, k_2, k_4 are defined as in the proof of Theorem 2. Consequently, Equation 15 holds if

$$3k_1c^2 + 3k_1c + 2k_2c > k_4 . \qquad (17)$$

On the other hand, we know from the proof of Theorem 2 that the function g (Equation 13) 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

$$2k_1c_{opt}^3 + k_2c_{opt}^2 = k_4 \qquad (18)$$

Substituting Equation 18 in Equation 17, tells us when the condition in Equation 15 is true in terms of the number of elementary segments for a constant-slope trajectory, i.e., it is true if

$$3k_1c^2 + 3k_1c + 2k_2c > 2k_1c_{opt}^3 + k_2c_{opt}^2 \qquad (19)$$

It is easy to see that this inequality holds if the value of c is larger than or equal to the value of c_{opt} for all $c \geq 2$ (i.e., if the buffer contains at least 2 elementary segments, which is required in practice before considering a split).

In summary, this means that the number of elementary segments c, collected up to the point where condition 18 becomes true (i.e., the trajectory buffer at that point), is always a multiple of the optimal segment size.

Using a dynamically determined buffer size according to these considerations, we propose a linear time trajectory splitting algorithm, called *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 would improve I/O expectation according to Equation 15. If the condition is true, the optimal segment size c_{opt} (Theorem 2) is computed (using a constant-slope approximation of the current trajectory segment $T[u,v]$), and we round c_{opt} 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. If a trajectory is completed, the last segment which may still be in the buffer has to be inserted as well.

Obviously, this algorithm splits a trajectory in O(t) time where t is the number of points of a trajectory. The pseudo code for the algorithm *LinearSplit* is given below.

Algorithm LinearSplit

```
u := 1, v := 2; //after the first 2 points of T
while (next point p_{v+1} in trajectory T exists)
   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 k=⌊(v−u)/c*⌋ segments from
      T[u,v]; 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;
```

5. Experimental Results

For the experimental evaluation, we used two datasets, produced by the Network Data Generator [20] and by GSTD [17], respectively. The network data generator simulates different classes of objects, e.g., vehicles and people, moving through streets of a real city (Oldenburg). Different objects have different speeds and lifetimes, giving a rich and realistic dataset. GSTD allows generating more random patterns suitable to investigate the performance of the algorithms 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, 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, was expected to be particularly challenging for the LinearSplit algorithm. Exactly 100 observations were recorded for each trajectory. Therefore, all our datasets had between 1,000,000 and 5,000,000 observations. All experiments were performed on a 1900+ AMD Athlon PC with 512 Mb RAM.

We used the R-tree implementation provided by the XXL library [3], using a 4Kb page size for all algorithms. For our algorithms, we replaced the split evaluation function using our cost model as described in Section 3.3.

We evaluate the quality of all algorithms by measuring 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 actual time required to pre-process a dataset, i.e., the time required to split the trajectories, create the MBRs and create the index tree.

Our dynamic programming-based algorithm is referred to as "OptimalSplit"; the linear time algorithm is referred to as "LinearSplit". "HKTG-*k*%" denotes the volume oriented split policy proposed in [7] (using the *DPSplit* algorithm for splitting trajectories individually), where *k*% means that $Nk/100$ total number of splits are used for splitting a dataset with N trajectories [7]. Similar to [7], we set k equal to 50, 100 and 150.[4] We also compare with two baseline algorithms. First, an R-tree, referred to as "NoSplit", where each trajectory is approximated by a single MBR, i.e., trajectories were not split (as discussed above, this is equivalent to a tree, optimized for a very large query size). Second, an R-tree, referred to as "FullSplit", where each elementary segment of trajectory is approximated by an MBR, i.e., trajectories were split at each observation point (as discussed above, this is equivalent to a tree optimized for an extremely small query size).

5.1 Optimality and Robustness w.r.t. Query Size

In the first experiment, we used datasets of 50,000 trajectories to study the suitability and robustness of the cost models and the associated split algorithms. We measured the average number of disk I/Os at the data level here, since the objective of the models is minimizing data level I/Os. We built trees that are optimized for different query sizes of *S*% of each spatial dimension and *T* time points. In particular, *S* = 1%, 2%,…, 16% and *T*=1, 2,…, 16.[5] 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.

[4] We also set k for the HKTG-*k*% algorithm so that it corresponded to the optimal number of splits found by our optimal split algorithm. However, when using our datasets we could not finish building the 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 when provided with the optimal number of splits, however, the overall I/O performance for queries was much worse than ours due to a larger overhead for directory level I/Os.

[5] The spatial extension of a query is given as percentage of the total 2*d* space since the space is finite; the temporal dimension is given in absolute time points since time is unbounded.

Each row represents the performance of a query $Q_{i,j}$ using all the constructed indices (listed in the columns). If the models are appropriate, given a query size, the best performance should occur when using the index built for that query size, i.e., in the diagonal of the tables. Note that 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) only shows the obvious fact that smaller queries result in smaller numbers of I/Os than larger queries. Note that, if we would include the NoSplit and FullSplit trees, their performance would be shown in columns to the left, respectively right side of the tables, since they correspond to trees optimized for an even smaller, respectively larger query size than the given ones. Their performance is worse for every query than the values for the given trees, which are "intermediate" trees with respect to the degree of trajectory splitting.

The performance of both algorithms on both datasets is qualitatively very similar. The best performance (shaded cells) occurs exactly where expected for the OptimalSplit (Tables 1 and 3). Even for LinearSplit (Tables 2 and 4) this is true, except for surprisingly only one case when using GSTD data ($Q_{2,2}$ in Table 4), since this data sets has by construction a very erratic behavior. Overall, the LinearSplit heuristic approximates the OptimalSplit very well not only in terms of where the optimum is but also in term of absolute numbers of I/Os, bu at a much lower computational cost. Furthermore, 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 quite robust with respect to the assumed average query size, but looking at the performance of a query against even more different trees (which in practical worst cases could be the FullSplit or NoSplit tree), it is clear that the query size parameter offers a great opportunity for tuning an index to a particular workload of query sizes. We will explore this in more detail in the following experiments.

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

Table 3. Robustness of OptimalSplit forGSTD 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

5.2 Number of Disk I/Os

5.2.1 Varying Query Size

To study the performance of different query sizes we used again databases with 50,000 trajectories. The definition of the used queries, similarly to [7], is as following:

***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
***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

To provide a thorough and realistic performance analysis we now measure the number of I/Os at both the directory and the data level, and we assume a query load of queries with varying sizes, where each spatial extent and duration within the limits of a query type is equally frequent within that query type.

Two different approaches are used with respect to our algorithms. In the *multiple tree approach*, we built an index for each query type separately (i.e., 6 per algorithm), using the average size of a query type as input parameter (e.g., for the RS query type, the average query size is *S*=6% and *T*=2).[6] A query was then run against the tree that was optimized for the query's query type. In the *single tree approach*, we built only one index for all query types per algorithm, 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 as parameter for the cost models. The resulting index is then used to answer all queries.

Figure 6(a)–(d) show the average number of I/Os per query for different data sets and approaches. Note that

[6] Note that the trajectory data does not have to be replicated; only different index directory structures were created.

using multiple trees or just a single tree affects only OptimalSplit and LinearSplit, and the values for other algorithms are consequently the same for the same dataset.

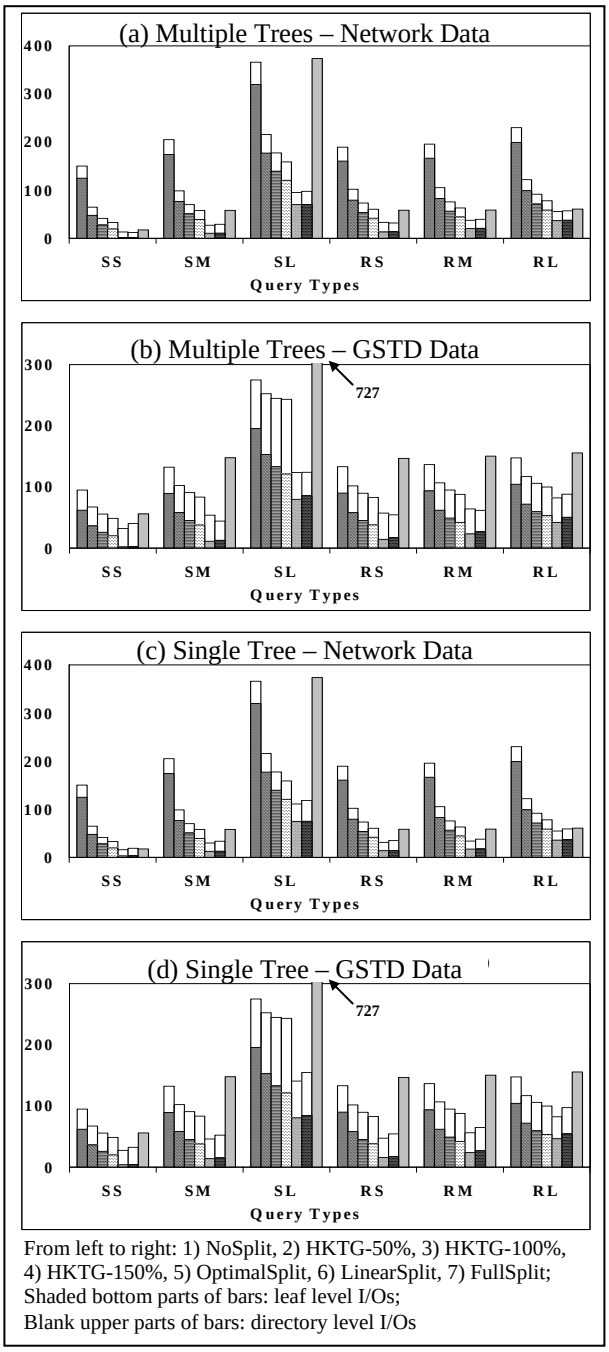


Figure 6. Average #I/Os for different query types.

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.

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.

Our approaches have a significantly lower number of I/Os on the data level than the other algorithms (except FullSplit which has no separate data level).[7] 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 (except FullSplit which always has the largest tree).

The performance of our algorithms using multiple trees is very close to the performance of using only a single tree for a workload of all query types, confirming again the robustness of our approach.

5.2.2 Varying Database Size

To measure sclability, we created indices for different datasets with 10,000, 20,000, and 50,000 trajectories. We ran medium sized range queries of type RM against all indices, where our indices where built for the average RM query size. The results are shown in Figure 7(a) and (b).

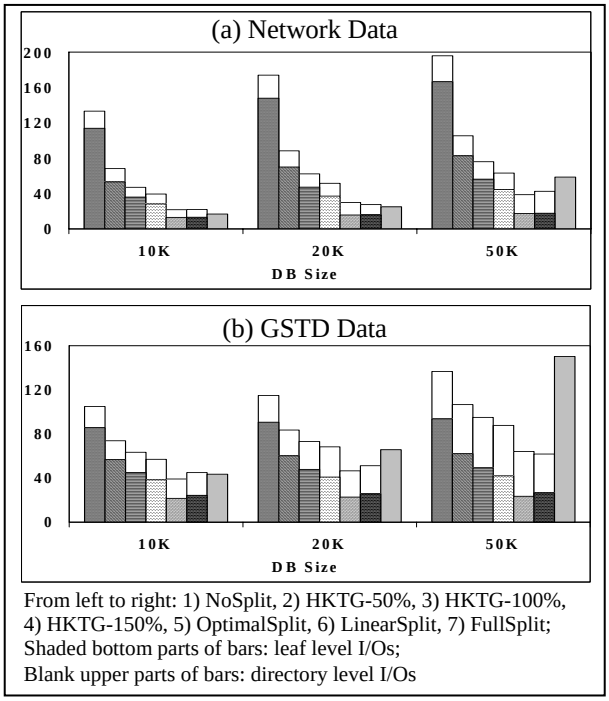


Figure 7. Average #I/Os per query, varying DB size

The I/O performance or our algorithms is always significantly better than the NoSplit and the HKTG-k% algorithms. For the smaller datasets FullSplit performs com-

7 Reducing data level I/Os also reduces false hits, which also saves CPU time for computationally intensive algorithms that are invoked to determine whether a trajectory segment approximated by an intersected MBRs actually intersects a given query.

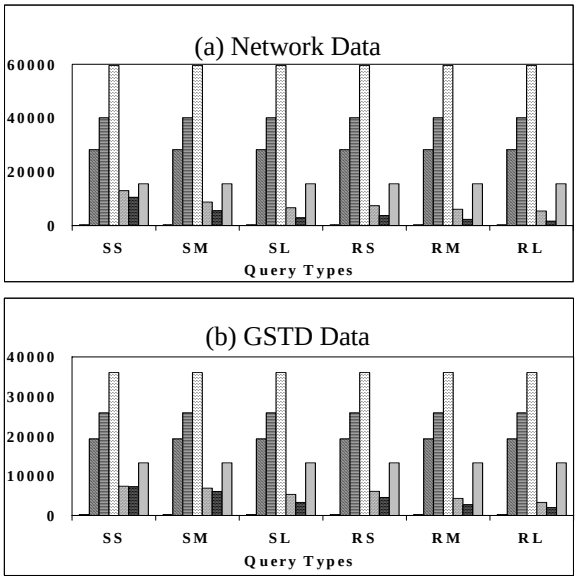
petitively to our approaches, but its performance degrades much faster with increasing database size.

Note that, even though the OptimalSplit always 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 directory node splitting policies.

5.3 Index Building Time

5.3.1 Varying Query Size

The index building times for all algorithms are shown in Figure 8(a) and (b), where we use a tree for every query type. NoSplit is clearly the fastest since it has to insert only one MBR per trajectory. For the HKTG-*k*% algorithms, most of the time is spent splitting the trajectories. FullSplit has to insert one MBR per elementary segment of each trajectory consuming also a significant amount of time. Our algorithms exhibit a good balance between trajectory splitting time and insertion time, outperformed only by the trivial NoSplit. Obviously, as the query size increases, our index building times decrease since trajectories are split less and fewer MBRs are inserted.

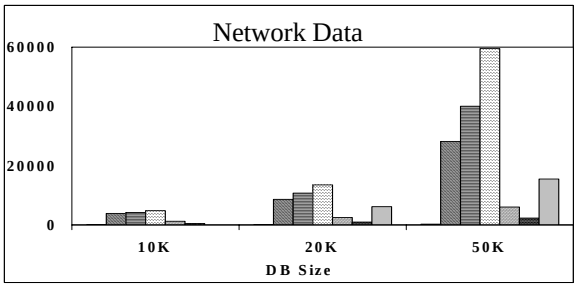


From left to right: 1) NoSplit, 2) HKTG-50%, 3) HKTG-100%, 4) HKTG-150%, 5) OptimalSplit, 6) LinearSplit, 7) FullSplit

Figure 8. Preprocessing time in seconds.

5.3.2 Varying Database Size

To measure the scalability of the index building time 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 are shown in Figure 9. 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).



From left to right: 1) NoSplit, 2) HKTG-50%, 3) HKTG-100%, 4) HKTG-150%, 5) OptimalSplit, 6) LinearSplit, 7) FullSplit

Figure 9. Preprocessing time in seconds

6. Related Work

Most spatio-temporal index structures proposed in the literature are based in one way or another on R-trees [9]. They can be classified into three main approaches. In the first approach, time is simply treated as an additional spatial dimension [16]. For trajectories, this leads to inefficient indices since the MBRs tend to be very large, covering large portions of empty space and leading to a high degree of overlap among the MBRs. Another structure under this approach is the TB-tree [12]. Its insertion split strategy is oriented towards trajectory preservation so that leaf nodes only contain segments that belong to the same trajectory. The main disadvantage, however, is that "concessions to the most important R-tree property, node overlap" must be made. Indeed, experimental results show that it is outperformed by a regular R-tree for spatio-temporal range queries, in particular for small queries.

In the second approach, time and space are treated differently within a combined indexing scheme, e.g., [4] and [14]. In [4] a two level index is proposed. Where the space is first partitioned into non-overlapping cells, and for each cell, an R-tree is used to index the temporal intervals at which objects were in those cells. In [14], the space is first partitioned into zones, and the locations of objects are only represented by zone ids, resulting in a less accurate but more efficient representation, managed by the SEB tree. These types of approaches are not compatible with our cost models since they don't use MBRs. 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 [10]. This approach, however, suffers from a prohibitively large overhead when indexing very dynamic scenarios, and is not suited for trajectory data.

Recent work aiming at improving the first approach has proposed two orthogonal strategies: replacing MBRs by different approximations, and splitting trajectories. In [19], the authors propose to trim the corners of trajectories' MBRs in order to obtain a bounding octagon prism, instead of a bounding hyper-rectangle, which is a tighter approximation. The experimental results, however, do not provide clear evidence that a considerable gain is obtained for spatio-temporal range queries, when compared to an

R*-tree using MBRs. The work presented in [7] proposes several algorithms to find split points for trajectories, with the goal of reducing the amount of the approximations' empty space, given a total number of allowed splits for a whole set of trajectories. (see Section 2 for more detail).

In [5], the authors present a trajectory splitting heuristic for trails of one-dimensional time-series in a multidimensional feature space. Trajectories are split "incrementally" whenever the latest point increases the "marginal cost" (~the expected I/Os per point) in the current subtrail. This heuristic is similar in spirit to a split heuristic for spatio-temporal trajectories that would simply split a trajectory whenever Equation 15 evaluates to true.

Apart from the problem of indexing spatio-temporal trajectories, several other types of spatio-temporal data and queries have been investigated, and are loosely related to this paper. The work presented in [13], [14] and [8] studies answering queries with respect to the future. Nearest neighbor queries have also received attention in the spatio-temporal domain (e.g., [18] and [2]); the problem of reverse nearest neighbor queries in spatio-temporal setting has been addressed in [2].

7. Conclusions

In this paper we investigated the problem of splitting spatio-temporal trajectories in order to improve the performance of queries using MBR-based access structures to index these trajectories. 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 a trajectory splitting algorithm based on this model, which minimizes the expected data page accesses, given an average query size. Using our cost model and approximating trajectories by constant-slope segments, we formally derived a linear time splitting algorithm, which can be applied in dynamic cases.

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, They are robust in the sense that the built indices efficiently support a much wider range of query sizes. Having a query size as a parameter of the model allows tuning indexes according to application requirements and query loads. In some cases, if the range of query sizes varies dramatically, or or if certain types of queries should run as fast as possible (e.g. because of organizational reasons), different directory structures for different query types can easily be constructed for an underlying data set.

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 computational cost, and can be used on dynamic environments.

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 spatio-temporal indices. For this, we also will explore the effect of different distributions of trajectories in space and time.

References

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

[2] Benetis, R. Jensen, C.S. Karciauskas, G. and Saltenis, S: Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. IDEAS 2002, pp. 44-53.

[3] 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

[4] Chakka, V., Everpaugh, A., Patel, J.: Indexing Large Trajectory Sets with SETI. CIDR 2003, online proceedings.

[5] Faloutsos, C., Ranganathan, M., Manolopoulos, Y.: Fast Subsequence Matching in Time-Series Databases. ACM SIGMOD 1994, pp. 419-429.

[6] Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. ACM SIGMOD 1984, pp. 47-57.

[7] Hadjieleftheriou, M., Kollios, G., Tsotras, V., Gunopulos, D.: Efficient indexing of Spatiotemporal Objects. EDBT 2002, pp. 251-268.

[8] Kollios, G., Gunopulos, D., Tsotras, V.: On Indexing Mobile Objects. In ACM PODS 1999, pp. 261-272.

[9] Mokbel, M., Ghanenm, G., Aref, W.: Spatio-Temporal Access Methods. IEEE Data Engineering Bull.,26(1), pp. 40-49, 2003

[10] Nascimento, M., Silva, J.: Towards Historical R-trees. ACM SAC 1998, pp. 235-240.

[11] Pagel, B., Six, H., Toben, T., Widmayer, P.: Towards an Analysis of Range Query Performance in Spatial Data Structures. ACM PODS 1993, pp. 214–221.

[12] Pfoser, D., Jensen, C., Theodoridis, Y.: Novel Approaches to the Indexing of Moving Object Trajectories. VLDB 2000, pp. 395-406.

[13] Saltenis, S., Jensen, C., Leutenegger, S., Lopex, M.: Indexing the Positions of Continuously Moving Objects. ACM SIGMOD 2000, pp.331-342.

[14] Song, Z., Roussopoulos, N., SEB-tree: An Approach to Index Continuously Moving Objects, MDM 2003, LNCS 2574, pp. 340–344, 2003.

[15] Tao, Y., Papadias, D. and Sun, J.: The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. VLDB 2003, pp. 790-801.

[16] Theodoridis, Y., Vazirgiannis, M., Sellis, T.: Spatio-Temporal Indexing for Large Multimedia Applications. IEEE ICMCS 1996, pp. 441-448.

[17] Theodoridis, Y., Silva, R., Nascimento, M.: On the Generation of Spatiotemporal Datasets. SSD 1999, pp. 147-164.

[18] Vlachos, M., Kollios, G., Gunopulos, D.: Discovering Similar Multidimensional Trajectories. IEEE ICDE 2002, pp. 673–684.

[19] Zhu, H., Su, J., Ibarra, O.: Trajectory Queries and Octagons in Moving Object Databases. ACM CIKM 2002, pp. 413-421.

[20] Brinkhoff, T.: Generating Network-Based Moving Object, IEEE SSDBM 2000, pp. 253-255.