Sampling Trajectory Streams with Spatiotemporal Criteria

Michalis Potamias Kostas Patroumpas Timos Sellis
School of Electrical and Computer Engineering
National Technical University of Athens, Hellas
{*mpotamias, kpatro, timos*}@*dbnet.ece.ntua.gr*

Abstract

Monitoring movement of high-dimensional points is essential for environmental databases, geospatial applications, and biodiversity informatics as it reveals crucial information about data evolution, provenance detection, pattern matching etc. Despite recent research interest on processing continuous queries in the context of spatiotemporal data streams, the main focus is on managing the current location of numerous moving objects. In this paper, we turn our attention onto a historical perspective of movement and examine trajectories generated by streaming positional updates. The key challenge is how to maintain a concise, yet quite reliable summary of each object's movement, avoiding any superfluous details and saving in processing complexity and communication cost. We propose two single-pass approximation techniques based on sampling that take advantage of the spatial locality and temporal timeliness inherent in trajectory streams. As a means of reducing substantially the scale of the datasets, we utilize heuristic prediction to distinguish which locations to preserve in the compressed trajectories. A comprehensive experimental study verifies the stability and robustness of the proposed techniques and demonstrates that intelligent compression schemes are able to act as effective load shedding operators achieving remarkable results.

1 Introduction

Research in data compression has grown rapidly over the past few years producing a large number of lossless and lossy compression methods. Data compression is possible because of redundancies found in information collected from various sources and in several domains, such as multimedia systems, astronomy, environmental and other scientific databases, geospatial information systems, as well as conventional databases hosting several terabytes of data. In particular, positioning applications have recently become extremely popular, e.g., for fleet management or location-based services, thanks to recent advances in telecommunications and geopositioning reporting devices (GPS, PDA etc.). Spatiotemporal data from multiple sources is then being sent via a wireless network to a central processor as an unbounded *data stream* of timestamped positions that cannot be managed within a traditional DBMS. A typical monitoring application must be able to provide real-time responses to multiple *continuous queries* that may refer to or even take advantage of the spatiotemporal features pertinent to streams of recorded locations.

Such a systematic observation of spatiotemporal phenomena related to moving humans, animals, vehicles etc. yields trajectory data that keep track of their movement. Informally, the *trajectory* of a moving object may be considered as a sequence of point locations at consecutive time instants. If multiple objects are monitored and their current locations are recorded very frequently, a large volume of data is expected to arrive for processing at an unpredictable and possibly high rate. Further, processing every single recording does not necessarily convey significant movement changes (e.g., if an object moves along a straight line), at the expense of considerable processing overhead.

The aforementioned restrictions illustrate the demanding algorithmic framework imposed by properties that characterize streaming trajectories. As a rule in data stream processing, single-pass algorithms are the most adequate means to effectively summarize massive data items into concise *synopses*. Essentially, there is always a trade-off between approximation quality and both time and space complexity. In the special case of trajectory streams, an additional requirement is posed: not only exploit the timely spatiotemporal information, but also take into account and preserve the sequential nature of the data. Therefore, in order to efficiently maintain trajectories online, there is no other way but to apply *compression* techniques, thus not only reducing drastically the overall data size, but also speeding up query answering (e.g., identify pairs of trajectories that have recently been moving close together).

This paper examines several variants of *sampling* applied over streams of positional updates collected from moving

objects. By intelligently dropping some points with negligible influence on the general movement of an object, a simplified yet quite reliable trajectory representation may be obtained. Such a procedure may be used as a filter over the incoming spatiotemporal updates, essentially controlling the stream arrival rate by discarding items before passing them to further processing. Besides, since each object is considered in isolation, such item-at-a-time filtering can be applied directly at the sources, with substantial savings both in communication bandwidth and in computation overhead at the central processor.

Of course, even by dropping incoming items blindly (e.g., "coin toss") rough outlines of trajectories can be kept. However, there are certain properties that a trajectory sample must preserve; primarily, the sample should be able to catch any significant changes in speed and direction that indicate alterations at the known pattern of movement. Therefore, these points could be used to reconstruct the original trajectory; discarded locations can be derived via linear interpolation with small error. Clearly, there is a need for techniques that can successfully maintain an online sample of the most significant trajectory locations, with minimal process cost per point. The situation is illustrated in Figure 1, where actual points have been recorded at constant intervals (e.g., every 10 seconds). The result of a uniform sampling technique is depicted, which retains 20% of the dense original locations. Notice how one of our techniques (STTrace) takes advantage of the spatiotemporal features that characterize movement, successfully detecting changes in speed and orientation. Threshold-guided sampling, also proposed in this paper, achieves comparable results, by deciding whether the current location can be safely predicted from the recent past.

Our contributions can be summarized as follows:

- We propose two sampling techniques, namely Thresholds and STTrace, in order to maintain a flexible and condense representation of streaming trajectories.
- Both techniques exploit the notions of spatial locality and temporal timeliness inherent in trajectory streams.
- Multiple trajectory compression is also feasible by dynamically redistributing memory resources.
- An extensive experimental study provides concrete evidence that even after applying sampling schemes we may still attain high-quality synopses useful in approximate query evaluation.

The remainder of this paper proceeds as follows. In Section 2, we review related work on sampling data streams and efficient trajectory maintenance. In Section 3, after sketching out a streaming framework for trajectory preservation, we utilize a well-known uniform sampling technique for managing massive trajectories. In Section 4, we present threshold-guided sampling, while in section 5 algorithm STTrace is introduced. Experimental results are discussed in Section 6. Finally, Section 7 draws conclusions and indicates directions for future research.

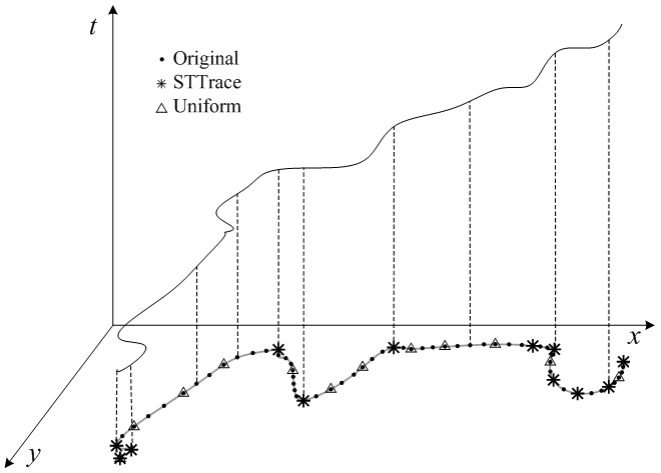

Figure 1. Sampling variations on a trajectory

2 Related Work

Several summarization techniques have been proposed for traditional databases, but recent work has focused particularly on data stream synopses [1], such as sampling [2], histograms [6], wavelets [5], and sketches [4]. Of these, random sampling is perhaps the most simple and intuitive algorithm for data reduction, aiming to capture the most significant features of the append-only stream. Reservoir algorithms such as those in [12] provide a *uniform sample* and make one pass over the data, taking online decisions on whether to accept or reject an item. However, as we further explain in Section 3, applying such a technique over streaming trajectories does not take into consideration spatiotemporal peculiarities pertaining to the movement of distinct objects.

The necessity of compression techniques for moving objects has been set forth in [7]. Inspired by line generalization that has been widely accepted on spatial data, several off-line and online spatiotemporal extensions were suggested to approximate trajectories, also providing a detailed analysis of compression quality. Their notion of synchronous distance between original and approximated locations is similar to ours, but their online algorithms clearly fail to comply with the low-complexity requirements of the data stream model. In fact, although some of the techniques carefully maintain good samples along trajectories, their time complexity is $O(N)$ per position update, where N is the current size of a trajectory. A similar approach was

taken in [3], offering analytical results also based on line simplification, but this time with respect to spatial query answering (e.g., spatial join, nearest neighbor search) and with bounded error guarantees. These techniques, although quite satisfactory for moving objects databases, generally entail high cost when applied on streaming locations.

The insightful idea of *amnesic* approximation for streaming time series was introduced in [10], considering that the importance of each measurement generally decays with time. Their time complexity is independent of the total stream size, but they only handle one-dimensional time series. In this work, apart from handling multi-dimensional points, we argue that the case for trajectories is different due to characteristics inherent in movement: not only spatial locations, but speed and orientation must also not be overlooked when approximating trajectories.

SCUBA [9] is a recently suggested technique that maintains clusters of closely located objects and queries moving along a network. Although it is stated that trajectory queries can be supported, this method is clearly geared towards current status without retaining the positional footprints of multiple distinct objects over long time periods. Other query-oriented techniques [11, 8] hash incoming points to efficient structures, discarding data irrelevant to the specific query, while our techniques act at a lower level, maintaining the original form of the data.

3 Problem Formulation

3.1 Trajectory Representation

Consider a set of numerous point objects continuously moving over the Euclidean plane. Generally speaking, the trajectory of each distinct object is represented as an evolving time series of point locations. Therefore, each trajectory is approximated by point samples collected from the respective data source at distinct time instants (e.g., a GPS measurement is taken every few seconds). More formally:

Trajectory T of a point object moving over the Euclidean plane is a possibly unbounded sequence of timestamped locations across time, i.e., tuples of values $\langle id, t_i, x_i, y_i \rangle,\ i \in 1, \ldots, N, \ldots$, where t_i stands for the timestamp referring to the time instant a spatial position (x_i, y_i) was recorded for an object identified as id.

Of course, the frequency of positional updates may not be identical for each object; in fact, it may be varying even for a single object over time. In moving object databases, *interpolation* techniques may be utilized to estimate objects' positions in between recorded locations and thus reconstruct an acceptable trace of the entire movement. Usually, it is assumed that line segments connect each pair of successive points in the sequence, although a *curve-based* representation is also possible [13] through higher-order polynomials or splines. This technique is used to provide synchronized trajectory representations by reconstructing original positional measurements at a fixed rate.

In practice, a trajectory is often represented as a *polyline* vector, i.e., a sequence of line segments for each object connecting consecutive pairs of observed locations. We adhere to a representation of trajectories as point sequences across time, implying that locations are linearly connected.

A reasonable assumption is that no updates are allowed to already recorded object positions in order to preserve coherence among streaming trajectory locations. Also, out-of-order tuples are not allowed; trivially, point locations may be temporarily buffered for establishing order according to their timestamps, before given to further processing.

3.2 Uniform Sampling

Sampling over trajectory streams simply suggests retaining a fair fraction of the original points in the sampled set, as a means of load shedding when processing incoming tuples. Since the original data is practically a sample of the actual trajectory, it is an unavoidable approximation controlled by system specifications (bandwidth, accuracy of the reporting device etc.). Therefore, by applying a supplementary sampling stage we can get a more coarse summary of the movement, but this one should engage intelligent techniques so as to maintain in memory the most essential trajectory features. The key intuition is that every timestamped point does not carry the same amount of information. In that sense, a predictable location can be discarded altogether so as to reserve memory for other significant points that reveal relative spatiotemporal changes in trajectory evolution.

The class of *reservoir sampling* algorithms introduced by Vitter [12] is well suited for online data streams, so it comes up as a natural candidate for trajectory compression. The main idea behind reservoir sampling techniques is to continuously maintain a sample of size $\geq M$, from which a random sample of size M can be easily produced.

Consider a single trajectory T consisting of a sequence of streaming point locations $p_1, p_2, \ldots, p_k, p_{k+1}, \ldots$ and that k of these have been processed so far. In the most simple variant of the method, the first M points of the trajectory have been inserted into the "reservoir". Afterwards, each successive location p_{k+1} is processed sequentially and it has probability $P[p_{k+1}] = \frac{M}{k+1}$ to be selected for insertion into the reservoir, replacing a point chosen randomly among its current M items. Evidently, each time the resulting set of M points is a uniform sample of the original trajectory, making one pass over the sequence without knowledge of its total size. Overall, the time complexity is $O(M(1 + \log \frac{N}{M}))$, where N is the current trajectory size.

Even though uniform sampling may provide a simple

and cost-effective solution, it is clearly insensitive to the spatiotemporal characteristics of the trajectory as well as to its sequential nature. Since the algorithm treats all items according to their succession disregarding their timestamps, it does not take notice at all whether the interarrival time fluctuates between positional updates. Reservoir sampling could be adapted to select items for insertion according to a time-based probability; nonetheless, a similar behavior should be expected. Stratified or weighted variations can also be applied, but these techniques are mostly used to tackle group-by and join queries, respectively [1].

We can distinguish sampling algorithms over streaming trajectories that require (i) constant, (ii) logarithmically increasing, (iii) linearly increasing memory and (iv) ever-increasing memory with non-specific rate. We move next to describe novel algorithms for in-memory trajectory compression with spatiotemporal heuristics of ever-increasing and constant memory requirements per trajectory.

4 Threshold-guided Sampling

The challenge when attempting summarization of moving object trajectories is to exploit spatiotemporal properties in order to produce a representative synopsis as closer as possible to the actual movement. Our key intuition is that a point should be taken into the sample as long as it reveals a relative change in the course of a trajectory.

If the location of an incoming point can be safely predicted (e.g., via interpolation) from the current movement pattern, then this point contributes little information and hence can be discarded without significant loss in accuracy. It is important to note that trajectory locations are not predicted on the basis of spatial positions only, but rather on velocity considerations (e.g., increase in speed, turns etc.). Therefore, a decision to accept or reject a point is taken according to user-defined rules specifying the allowed tolerance ("threshold") in changes in speed and orientation.

We refer to this class of algorithms as *threshold-guided sampling*, because a new point must be appended to the sample when that threshold is exceeded for an incoming location. Observe that under this scheme and in contrast to uniform sampling, the total amount of items in the sample keeps increasing without eliminating any point already stored. Next, we introduce three variants based on location prediction that differ in the way the estimated locations are computed. We exemplify the techniques over a single trajectory, since processing is performed on a tuple basis and in a similar fashion independently for each trajectory.

4.1 Sample-based Thresholds

Consider the trajectory of a moving object as illustrated in Figure 2. Suppose that B, C are the last two points inserted into the current sample. Let pair $\langle v_s, \varphi_s \rangle$ denote the speed and orientation (i.e., azimuth) of the velocity vector $\overrightarrow{v_s}$ calculated according to B, C. Also, let $\langle dv_s, d\varphi_s \rangle$, denote the thresholds specified for the velocity vector; the former refers to the tolerable change in speed (expressed as a percentage), whereas the latter indicates the greatest allowable deviation in orientation (a small angle value). Under normal conditions, it should be expected that the object will continue to move according to this vector for some time. Therefore, it is very easy to check whether the current location can be approximately anticipated by applying velocity vector $\overrightarrow{v_s}$ at the last observed location, which incurs little overhead.

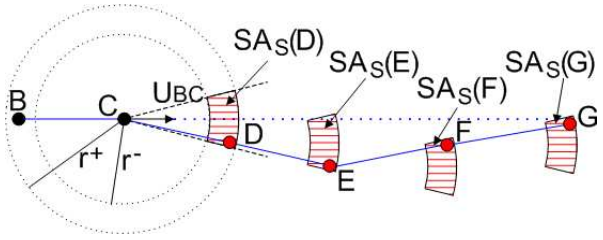

Figure 2. Sample-based Thresholds

In practice, predictions are carried out by constructing a so-called *safe area* SA_S for the last point in the sample, taking into account the time interval dt_s spanning between the last observed trajectory location and the newly arrived one (i.e., the point candidate for insertion). The actual shape of the safe area is determined by constraints that affect changes in speed and deviations in orientation. In particular:

- A circular area determines the locus of all possible positions where the object may be located as long as it maintains its anticipated speed. Taking into account the user-specified threshold dv_s for acceptable speed changes, we may delineate two circles with centers that coincide with the last known point of the trajectory. The radius of the outer circle (upper bound) is $r_s^+ = dt_s \times v_s^+$ and derives from the maximum allowed speed $v_s^+ = v_s \times (1 + dv_s)$. Similarly, the radius of the inner circle (lower bound) is $r_s^- = dt_s \times v_s^-$ and the minimum allowed speed is $v_s^- = v_s \times (1 - dv_s)$. As a result, the speed locus is actually the ring between the two circumferences (Figure 2).

- In order to form the safe area with respect to orientation, two half-lines are drawn from the last known point of the trajectory. Their slopes are $\varphi_s + d\varphi_s$ and $\varphi_s - d\varphi_s$, obviously defined by the orientation of the velocity vector and the respective deviation threshold. These two straight half-lines divide the plane into two half-planes. The half-plane which includes the velocity vector is the safe area regarding orientation and im-

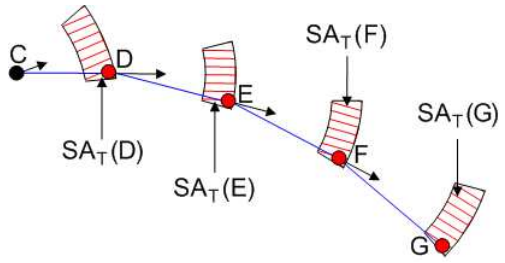

Figure 3. Trajectory-based awkward case

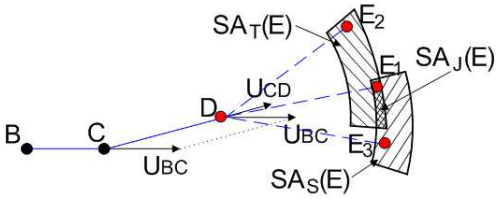

Figure 4. Joint Safe Area

plies the expected direction of movement.

Finally, the sample-based safe area is derived from the intersection of the two aforementioned areas. Given that the incoming point is found within the constructed safe area, we may consider this location redundant and safely ignore it, assuming that it can be anticipated without significant implications. In the opposite case, a decision to insert the point in the sample is taken, suggesting that a considerable change in movement has occurred.

Unfortunately, this policy is vulnerable to error propagation, as a moving object could change its orientation in a way that the algorithm would fail to capture. This awkward case is illustrated in Figure 2. Note that point E is more significant than D and F, since a change in object's direction occurs there; still, E is not inserted in the sample, because it falls within the safe area projected with respect to D.

4.2 Trajectory-based Thresholds

In order to avoid the awkward situation in Figure 2, another variation is possible. Now the safe area SA_T concerning velocity is constructed from the last two locations of the *actual trajectory*, instead of the last two points stored in the sample. The safe area locus is defined from the velocity vector $\overrightarrow{v_t}$ corresponding to the last two observed locations. As before, the speed locus is determined by two concentric circles with radii $r_t^+ = dt_t \times [v_t \times (1+dv_t)]$ and $r_t^- = dt_t \times [v_t \times (1-dv_t)]$, while the orientation locus is determined by lines of slope $\varphi_t + d\varphi_t$ and $\varphi_t - d\varphi_t$.

This approach is also susceptible to error propagation, when consecutive locations exhibit a smooth but significant change in object's orientation as illustrated in Figure 3.

4.3 A Combined Thresholds Approach

One can easily observe that the sample-based approach evades the trajectory-based trap since the constant slight angle augmentations (as illustrated in Figure 3) will be eventually detected. Conversely, trajectory-based approach is able to detect small deviations in orientation not captured by the sample-based approach (as that in Figure 2). Therefore, complications caused by error propagation can be overcome by applying a combination of the two techniques, suggesting that a safe area can be formed as the planar intersection of the sample-based safe area and the trajectory-based one.

The situation is illustrated in Figure 4. Points B and C are the last items stored in the sample. Points C and D are the last observed locations of the actual trajectory, whereas E_i, for $i = 1, 2, 3$ represent three possible locations currently examined for inclusion in the sample. Thus, we form the velocity vectors, $\overrightarrow{v_{BC}}$ for the sample and $\overrightarrow{v_{CD}}$ for the trajectory, respectively. The time interval dt_{DE} is used with the velocity vectors in order to delineate the two safe areas, the sample-based and the trajectory-based one. Finally, their intersection is calculated in order to form the final *joint safe area* (SA_J).

In case the candidate point is located within the safe area (case E_1), it is simply ignored. Otherwise, a relative change has been observed (cases E_2, E_3) and this decision triggers the insertion of a new point in the sample. This insertion involves either the current or the previously observed location of the trajectory. The current-point approach leads to a sample that represents a smooth approximation of the original trajectory, while the previous-point approach results in a more coarse outline, suggesting that in the sample we store the location immediately preceding the one that violated the safe area. In Figure 2, a rule violation will be detected for point F (due to the specified trajectory-based thresholds). The current-point approach would store location F (hence a smoother move), while the previous-point approach would choose location E.

The pseudocode of the combined thresholds algorithm is presented in Figure 5. After the safe area calculations (Steps 2-4), a decision is taken on whether to insert the current point in the sample (Step 6). In Steps 7-8 the insertion occurs, increasing the total sample size for that trajectory.

As it turns out, this combined approach takes under consideration both mean and instantaneous velocities in order to make predictions. The mean velocity comes from the last two points stored in the sample, while the instantaneous derives from the last two observed trajectory locations. The joint safe area (i.e., the intersection of the two loci) is more likely to shrink as the number of points discarded increases after the last insertion into the sample. As soon as no inter-

Algorithm *Thresholds* (Trajectory Stream S, Objects n)
$Sample_{id}$ maintains the sampled points for object id
$sCounter[1..n]$ keeps the sample size for each trajectory

1. **for** each point $p = \langle id, t, x, y \rangle$ **in** S;
2. **calculate** $SA_S(p)$; /*sample-based safe area*/
3. **calculate** $SA_T(p)$;
4. **calculate** $SA_J(p) = SA_S(p) \bigcap SA_T(p)$; /*joint safe area*/
5. **maintain** p in a buffer for the next iteration;
6. **if** $\langle x, y \rangle$ **in** $SA_J(p)$ **then break**;
7. **else** $nextpos \leftarrow ++ sCounter[id]$;
8. $Sample_{id}[nextpos]; \leftarrow p$; /* append p to sample */
9. **end for**;
10. **end** Thresholds.

Figure 5. Algorithm Thresholds

section is found, an insertion will be prompted regardless of the current spatial location of that object. From the previous discussion, at any step of the algorithm (as well as the other two variations), the following lemma holds :

Lemma 4.1 *The time complexity of threshold-guided sampling is $O(1)$ per incoming tuple.*

All three variations yield samples of unbounded memory requirements; for given thresholds, the size of the sample is a rough indication of complexity in movement pattern (i.e., irregularity in trajectory shape).

5 Sampling based on Synchronous Distances

Although threshold-guided techniques achieve substantial gains with respect to the space required for maintaining complete trajectories, they still have ever-increasing memory requirements. This fact naturally raises a question on whether it is possible to maintain a sample of known and constant memory, and thus more tailored for streaming environments (similarly to uniform sampling presented in Section 3). In this section, we introduce algorithm *STTrace*, which successfully fulfils these specifications. The intuition behind STTrace is to use an insertion scheme based on the recent movement features (as in threshold-guided algorithms), but at the same time also allowing deletions from the sample to make room for the newly inserted points without exceeding allocated memory (as occurs in uniform sampling). However, there is an important differentiation: the sampled point chosen for deletion is not selected randomly over the current sample contents, but according to its significance in trajectory preservation. Admittedly, it is better to discard a point that will produce the less distortion to the current trajectory synopsis. But this comes at a cost: for

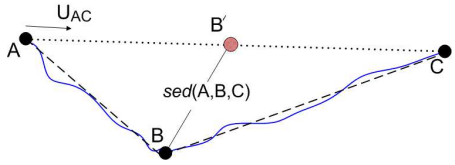

Figure 6. Synchronous Euclidean Distance

every new insertion, the most appropriate candidate point must be searched for deletion over the entire sample with $O(M)$ worst-case cost, where M is the current sample size. Nevertheless, as M should be expected very small and the sampled points may be maintained in an appropriate data structure (e.g., a binary balanced tree) with logarithmic cost for operations (search, insert, delete), normally this is an affordable trade-off.

5.1 Synchronous Euclidean Distance

Being enforced to make deletions from the sample, the most intuitive idea is to choose the point that conveys the less critical information for that trajectory. Preferably, a scalar metric should be used to measure the influence of a point in the sample, since threshold-guided techniques cannot be easily adapted due to the 2-dimensional criteria (speed, orientation) used to determine the sample.

Therefore, we follow a different perspective and define a metric that is based on the notion of *Synchronous Euclidean Distance* (*sed*). For any point in the sample, this is the distance between its actual location and its synchronous position estimated via interpolation between its predecessor and successor points in the sample (Figure 6). Formally:

Synchronous Euclidean Distance: Let A, B, C be three successive (i.e., $t_A < t_B < t_C$) spatiotemporal locations recorded for a trajectory T. Then, the *Synchronous Euclidean Distance* between the actual location B and its predicted location B' according to A and C is defined as:

$$sed(A, B, C) = \sqrt{(x'_B - x_B)^2 + (y'_B - y_B)^2}$$

where coordinates $x'_B = x_A + v^x_{AC} \times (t_B - t_A)$ and $y'_B = y_A + v^y_{AC} \times (t_B - t_A)$ are calculated with respect to the velocity vector corresponding to points A and C, hence $v^x_{AC} = \frac{x_C - x_A}{t_C - t_A}$, $v^y_{AC} = \frac{y_C - y_A}{t_C - t_A}$.

Since this is essentially the distance between the actual point and its spatiotemporal *trace* along the line segment that connects its immediate neighbors in the sequence, the sampling algorithm based on this metric is called STTrace.

5.2 Single-Trajectory Sampling

Let nM be the available memory, where n is the number of moving objects (i.e., trajectories). When handling single

trajectories, this algorithm will devote memory M to each one separately. The stored points are represented by tuples $\langle id, t, x, y, sed \rangle$, where the new attribute sed stands for the synchronous Euclidean distance of the location with respect to its predecessor and successor in the sample. The sed attribute of a stored tuple will be updated only if one of the three points involved in its calculation gets removed.

Initially, every incoming point will be stored in the sample of that trajectory until the available memory is filled. The sed for every newly inserted point is calculated after its successor is stored in the sample. As soon as the allocated memory gets exhausted and a new point is examined for possible insertion, the sample is searched for the item with the lowest sed. That item represents the least possible loss of information in case it gets discarded, so its sed is essentially the lowerest allowable distortion.

Next, the algorithm calculates a *probing* sed, using the last two points of the sample and the currently processed point. If the probing sed is larger than the minimum sed found already in the sample, the currently processed point is inserted into the sample at the expense of the point with the lowest sed. This causes recalculation of the sed attributes for the neighboring points of the removed one, whereas a search is triggered in the sample for the new minimum sed. If the probing sed is smaller than the minimum sed the current point is completely ignored and the streaming process continues. It is obvious that the minimum sed (lower bound of the allowable distortion) generally increases with time, hence the search operation over the sample is triggered more and more sparsely as time goes by.

Lemma 5.1 *The time complexity per positional update of algorithm STTrace is* $O(\frac{1}{N}\log\frac{N}{M}\log M)$*, where* N *denotes the current trajectory size and* M *its allocated memory.*

Proof A point insertion into the sample introduces a computational overhead which costs $O(M)$ due to deletion and adjustment. Assuming that items candidate for insertion will be uniformly distributed across the trajectory (i.e., STTrace will choose any point for the sample with equal probability), we expect a total of $\frac{1}{2}(M+\frac{M}{2})$ deletions when the size of the trajectory is $2M$ points, a total of $\frac{1}{2}((M+\frac{M}{2})+(\frac{M}{2}+\frac{M}{3}))$ deletions for $3M$ points, and eventually $\frac{M}{2}+\sum_{i=2}^{L}\frac{M}{i} < (\ln L+1)M$ insertions for a trajectory size $N = LM$. Thus, $O((\log L+1)M^2)$ is the overall time and it can be decreased to $O(M(\log L+1)\log M)$ using appropriate index for sample contents (e.g., a binary search tree). Therefore, expected processing time for a single point is $O(\frac{M}{N}(\log L+1)\log M) = O(\frac{M}{N}\log\frac{N}{M}\log M)$. ∎

The full algorithm is presented in Figure 7. Step 2 calculates the probing sed of the current point. Step 3 is the decision step. Steps 4-9 handle the sample in case of deletion (i.e. when memory is full), recalculating affected sed values and making room for the new point. Step 10 increases the sample counter until it reaches the maximum allowed sample size. In Steps 12-13, the insertion occurs and the sed for the previously inserted point is calculated. At any time of the streaming process, this technique provides a time-ordered representative sample for each trajectory.

Algorithm *STTrace* (Trajectory Stream S, Objects n, Memory m)
$sample[1..n][]$ is a list of points for the sample of each trajectory
$sCounter[1..n]$ stores the number of points in the sample
$min[id]$ (initially 0) index to current threshold-point of each trajectory

1. **for** each point $p =< id, t, x, y >$ **in** S;
2. **calculate** $sed(A, B, C)$; /*with $B = sample[id][sCounter]$ $A = sample[id][sCounter-2]$ and $C = p$*/
3. **if** $(sed < min[id])$ **break**; **end if**;
4. **if** $([sCounter[id]] >= \frac{m}{n})$ */* sample is full */*
5. $delete(sample[id][min[id]])$;
6. **calculate** $sed(sample[id][min[id]-1])$;
7. **calculate** $sed(sample[id][min[id]+1])$;
8. $shiftLeft(1, sample[id][min[id]])$;
9. $find(min[id])$;
10. **else** $sCounter[id]++$; */* sample is not full */*
11. **end if**;
12. $sample[id][sCounter] := p$;
13. **store** $sed(sample[id][sCounter[id]-1])$
14. **end for**;
15. **end** STTrace.

Figure 7. Algorithm STTrace

5.3 Multi-Trajectory Sampling

By now, each trajectory is assigned a specific amount of space M for its sample. For multiple trajectories, the available memory is initially filled with the incoming points regardless of the trajectory they belong to, hence the available space is distributed unevenly amongst them. The lower bound on the allowable sed is calculated globally for all objects and it represents the least significant point for all trajectories, so this location will be the candidate for replacement. Thus, instead of calculating a separate minimum sed for each distinct trajectory, just one such distance is maintained for all n monitored objects. This algorithm is a generalization of the one in Subsection 5.2 in all other aspects.

5.4 Discussion

An issue involves which points to consider when checking for probing synchronous distances. As shown in Figure 6, we need three points to take a decision: that is, A and B can be any two previously observed locations, whereas C

is always the point at hand. Besides, in case of insertion, either B or C may be chosen to be stored in the sample.

Further, threshold-guided sampling as well as algorithm STTrace can perform compression over *multiple trajectories*, as the available space can be distributed unevenly among them by allocating more memory to objects of greater agility. The former produces a sample for each distinct trajectory and the sample size is an indication of its shape complexity. Algorithm STTrace is more powerful; for multiple trajectories, it favors some trajectories that will obtain more space (i.e., more samples) due to many "sharp" changes in their movement, at the expense of other trajectories that will be roughly represented as they follow a more regular (hence predictable) motion pattern.

6 Experimental Evaluation

We implemented the threshold-guided and STTrace sampling algorithms and compared them to Uniform sampling. We conducted three sets of experiments for performance evaluation, in order to (i) validate the consistency of samples with respect to original trajectories, (ii) assess the computation time required for each technique, and (iii) verify whether the sampled trajectories could be useful in approximate query answering.

In the following, note that we have chosen the *previous-point* combined approach for threshold-guided sampling, which is superior than *current-point* approach in terms of errors (Section 4). This technique takes as parameters user-defined thresholds concerning speed and orientation for both the sampled and the original trajectory. Also, only single-mode STTrace execution has been tested for a fair comparison with the threshold-based technique, since both apply to each trajectory separately. The only parameter given to STTrace is the available memory allocated to every trajectory; this also applies to uniform sampling.

6.1 Experimental Setting

All experiments have been performed on an Intel Pentium-4 2.5GHz CPU running WindowsXP with 512 MB of main memory. We generated synthetic datasets for trajectories of moving objects traveling on the road network of greater Athens that covers an area of 250 km^2. No spatial or any other indexing technique is utilized and all processing takes place in main memory.

Objects move at various speeds during their course, each time according to the average speed assigned to the road segment they enter. This means that all objects move at the same speed along a specific road segment. We run simulations for several time intervals (up to 10,000 timestamps) and number of objects (up to 100,000). The interarrival time was constant for all trajectories in a dataset, assuming that all objects reported their positional updates concurrently at specific time periods, hence agility was always set to 100%.

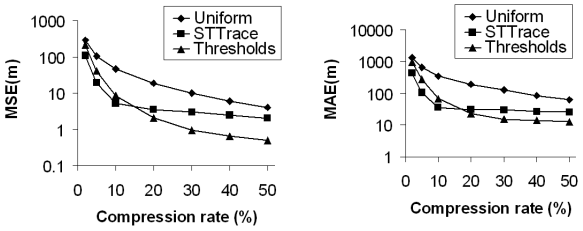

Figure 8. Error vs. compression rate

6.2 Approximation Quality

As a means of quality assessment for our sampling techniques we estimate the error committed, i.e., how much the sampled trajectory deviates from the original. Although several types of distance may be utilized [3], we opt for a *time-uniform (or synchronous) Euclidean distance* that takes into account the spatiotemporal features of trajectories; that is, the actual and the approximated location must correspond to the same time instant. Therefore, the approximated trajectory must be used to reconstruct locations (e.g., via linear interpolation) for all time instants recorded in the original movement. Thus, an *Average Spatiotemporal Distance* (*APD*) can be calculated between the synchronous (i.e., original and reconstructed) timestamped positions per trajectory. In fact, we calculate two error metrics:

(i) *Mean Square Error* (*MSE*) of the APD and

(ii) *Maximum Absolute Error* (*MAE*), i.e., the highest APD for the entire trajectory.

In fact, MSE provides a quality metric based on average performance, while MAE is more essential as it is able to capture sudden deviations due to extreme locations that perhaps cannot be detected during approximation. Next, we will provide simulation results for a synthetic dataset consisting of 100 moving objects each traveling for $10,000$ seconds. Compression rate denotes the ratio in the size of an approximated trajectory with respect to the original.

In Figure 8 (left), MSE measurements for the three techniques is illustrated. Clearly, in all three techniques deviation from the original trajectories drops drastically when more locations are stored, which means that error is sensitive to the size of the sample. However, Thresholds technique outperforms Uniform sampling even by factors greater than 10 for compression rates in the range $10-50\%$. Besides, STTrace is better than Thresholds for compression rates below 15%, while Thresholds performs up to twice as better for any compression rate above 15%. STTrace is 2 to

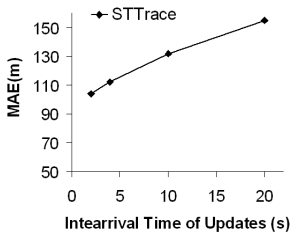

Figure 9. Error vs. frequency of recordings

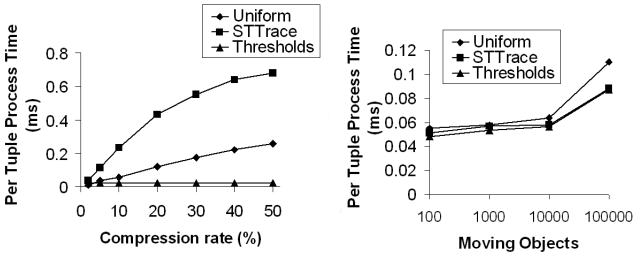

Figure 10. Processing time

5 times better than Uniform for every compression rate.

In terms of MAE, again lower errors are observed for higher compression rates (Figure 8). Thresholds still outperforms Uniform by a factor greater than 10 for compression rates equal to 30%, while STTrace is at least 5 times better than Uniform. STTrace yields a MAE of 100 meters for compression rate 5%, while Uniform gives a comparable MAE for compression rates around 45%. In other words, STTrace maintains a much smaller set of locations extremely critical to trajectory evolution and thus can provide a more reliable outline than the other two techniques.

Note that Thresholds has the additional advantage of distributing available memory unevenly to trajectories according to their complexity, while single-mode STTrace and Uniform devote the same amount of memory to every trajectory. That's why Thresholds overwhelmingly outperforms the other two techniques for higher compression rates (roughly, more than 15%) as shown with its asymptotic behavior towards larger rates, while STTrace is preferable for a more sparse approximation with smaller rates.

To achieve smaller rates, parameters for thresholds must be set to large values which leads to important information loss. For compression rate at 10%, we set thresholds as follows: for trajectory-based, 35^o deviation in orientation and 0.8 for relative speed change, whereas for sample-based 18^o and 0.4 respectively. For compression rate at 20%, the corresponding values were $12^o, 0.2, 12^o, 0.2$. Of course, suitable threshold values have much to do with the motion patterns recognized in the streaming data, and this is rather hard to fine-tune. This is shown in both MAE and MSE diagrams, where STTrace has a more predictable performance for various compression rates than Thresholds.

Figure 9 illustrates the impact (in terms of MAE) of the interarrival time of positional updates on approximation quality. Obviously, the smaller this time period is, the more frequent a position sample arrives for processing. We applied STTrace algorithm four times on a dataset of 100 moving objects, whose location is recorded every 2, 4, 10 and 20 seconds, respectively. The amount of available memory is constant in all four experiments (250 samples per trajectory). As expected, the error increases as the recording frequency becomes more sparse, even though we provide equal amount of memory for the sample. As the input stream rate in STTrace is decreased, the original information becomes less accurate, which results in substantial increase in error.

6.3 Processing Time

In this subsection, we briefly discuss per tuple computation time required for each technique, not surprisingly confirming propositions stated in previous sections. In particular, regarding computation time we should note that Thresholds is an $O(1)$ algorithm, while both STTrace and Uniform have an overhead due to memory manipulation during deletions. Thus, algorithm Thresholds has exactly the same cost regardless of compression rate, while run times for STTrace and Uniform depend on the amount of available memory. This is clearly illustrated in Figure 10 (left).

We also conducted experiments increasing the number of moving objects in order to test the scalability of our techniques. Figure 10 (right) illustrates the results. As expected, the update cost is linear to the number of moving objects, with STTrace and Thresholds doing better than Uniform. The slightly increased cost in the case of 100,000 moving objects is attributed to resource limitations.

6.4 Evaluation of Spatiotemporal Joins

We also demonstrate the robustness of our techniques using their results in spatiotemporal queries and comparing approximate to actual answers. In particular, we concentrated on *trajectory self-joins*, i.e., on identifying pairs of trajectories that move very closely (within a user-defined distance) to each other during a time interval. We set three parameters: the spatial tolerance for the search distance (20 meters), the recent time interval examined (20 seconds) and the distinct time instants at which we evaluated the query (every 100 seconds). We measured the rates FalseNegatives/Positives and FalsePositives/Positives as illustrated in Figure 11. In both diagrams, STTrace and Thresholds clearly outperform the Uniform approach in approximation quality. They yield an error of 2% of false negatives for

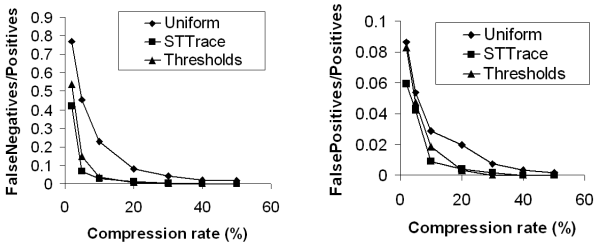

Figure 11. Evaluation of trajectory self-joins

a compression rate of 10%, while Uniform yields the same error for a compression rate at 40%. STTrace is slightly better than Thresholds for compression rates below 15%; the converse holds for higher compression rates. Interestingly enough, the response accuracy for query approximation follows the pattern of the MSE error presented in Figure 8.

6.5 Discussion of Results

Threshold-guided sampling emerges as a robust mechanism for semantic load shedding, filtering out negligible locations with minor computational overhead. The actual size of the sample it provides is a rough indication for the complexity of each trajectory, and the parameters can be fine-tuned according to trajectory characteristics and memory availability. Besides, it can be applied close to the data sources instead of a central processor, sparing both transmission cost and processing power.

STTrace always maintains a small representative sample of a trajectory of unknown size. It outperforms Thresholds for small compression rates, since it is not easy to define suitable threshold values in this case. STTrace incurs some overhead in maintaining the minimum synchronous distance and in-memory adjustment of the sampled points. However, this cost can be reduced if STTrace is applied in a batch mode, i.e., executed at consecutive time intervals; for instance, if we keep the 20 most significant points every 1000 timestamps and then merge the results.

7 Conclusions

In this paper we studied techniques for effective data reduction in streaming trajectories generated from moving objects, stemming not only from storage considerations but also intended for approximate query answering. We introduced two novel sampling algorithms that exploit spatiotemporal features inherent in trajectories; the former is based on spatiotemporal thresholds regarding speed and orientation of objects, whereas the latter (STTrace) attempts to preserve the shape of trajectories to the extent possible. Both techniques have shown quite remarkable and promising results for online compression, with affordable computational overhead and reliable approximation quality.

In the future, we plan to develop variations based on suitable threshold selection in order to achieve constant or predictable memory consumption. Also, dynamic adjustment of thresholds according to current movement characteristics seems a very challenging topic. Finally, applying successive compression filters or even a hierarchy of sampling stages is also an interesting idea for further investigation.

References

[1] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems. In *ACM PODS*, pp.1-16, May 2002.

[2] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *ICDE*, pp. 350-361, March 2004.

[3] H. Cao, O. Wolfson, and G. Trajcevski. Spatio-temporal Data Reduction with Deterministic Error Bounds. In *DIALM-POMC*, pp. 33-42, September 2003.

[4] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi. Sketch-Based Multi-Query Processing over Data Streams. In *EDBT*, pp. 551-568, March 2004.

[5] A. Gilbert, Y. Kotidis, S. Muthukrishnan, M. Strauss. One-pass Wavelet Decompositions of Data Streams. *IEEE TKDE*, 15(3): 541-554, May 2003.

[6] S. Guha and N. Koudas. Approximating a data stream for querying and estimation: Algorithms and performance evaluation. In *ICDE*, pp. 567-576, February 2002.

[7] N. Meratnia and R. de By. Spatiotemporal Compression Techniques for Moving Point Objects. In *EDBT*, pp. 765-782, March 2004.

[8] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *ACM SIGMOD*, pp. 634-645, June 2005.

[9] R. Nehme and E. Rundensteiner. SCUBA: Scalable Cluster-Based algorithm for Evaluating Continuous Spatio-Temporal Queries on Moving Objects. In *EDBT*, pp. 1001-1019, March 2006.

[10] T. Palpanas, M. Vlachos, E. Keogh, D. Gunopulos, and W. Truppel. Online Amnesic Approximation of Streaming Time Series. In *ICDE*, pp. 338-349, March 2004.

[11] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-Temporal Aggregation Using Sketches. In *ICDE*, pp. 214-226, March 2004.

[12] J. S. Vitter. Random sampling with a reservoir. *ACM TOMS*, 11(1):37-57, 1985.

[13] B. Yu, S.H. Kim, T. Bailey, and R. Gamboa. Curve-based Representation of Moving Objects Trajectories. In *IDEAS*, pp. 419-425, July 2004.