JOURNAL OF SPATIAL INFORMATION SCIENCE
Number 7 (2013), pp. 45–75
doi:10.5311/JOSIS.2013.7.132

RESEARCH ARTICLE

Multi-scale window specification over streaming trajectories

Kostas Patroumpas

School of Electrical and Computer Engineering
National Technical University of Athens, Hellas

Received: March 14, 2013; returned: June 20, 2013; revised: September 19, 2013; accepted: November 11, 2013.

Abstract: Enormous amounts of positional information are collected by monitoring applications in domains such as fleet management, cargo transport, wildlife protection, etc. With the advent of modern location-based services, processing such data mostly focuses on providing real-time response to a variety of user requests in continuous and scalable fashion. An important class of such queries concerns *evolving trajectories* that continuously trace the streaming locations of *moving objects*, like GPS-equipped vehicles, commodities with RFID's, people with smartphones etc. In this work, we propose an advanced windowing operator that enables online, incremental examination of recent motion paths *at multiple resolutions* for numerous point entities. When applied against incoming positions, this window can abstract trajectories at coarser representations towards the past, while retaining progressively finer features closer to the present. We explain the semantics of such *multi-scale sliding windows* through parameterized functions that reflect the sequential nature of trajectories and can effectively capture their spatiotemporal properties. Such window specification goes beyond its usual role for non-blocking processing of multiple concurrent queries. Actually, it can offer concrete subsequences from each trajectory, thus preserving continuity in time and contiguity in space along the respective segments. Further, we suggest language extensions in order to express characteristic spatiotemporal queries using windows. Finally, we discuss algorithms for nested maintenance of multi-scale windows and evaluate their efficiency against streaming positional data, offering empirical evidence of their benefits to online trajectory processing.

Keywords: geostreaming, moving objects, multi-resolution, trajectories, windows

1 Introduction

The increasing popularity of location-based services due to proliferation of smartphones and positioning devices (GPS, RFID, GSM), has significant impact on data management. Apart from various types of information exchanged through service providers (e.g., text, imagery, video, etc.), positional updates incur a considerable part of the network load. Indeed, it becomes harder to sustain massive volumes of rapidly accumulating traces from a multitude of vehicles, ships, containers, etc. Most processing techniques (e.g., [10, 13, 17]) usually focus on spatial relationships among *current* positions of moving objects; for instance, reporting which people are now moving in a specific area (i.e., a continuous range query), or finding friends closest to the current location of a mobile user (i.e., a continuous k-nearest neighbor search).

In contrast, the significance of dynamically updated *trajectories* is rather overlooked. Luckily, continuous tracking of mobile devices offers an evolving trace of their motion across time. As numerous objects relay their locations frequently, voluminous positional information is being accumulated in a streaming fashion [24]. But with real-time processing, it is hardly feasible to keep in main memory the entire, ever-growing motion path of every object. So, most user requests focus on the latest portion of data through repeatedly refreshed *sliding windows* [14, 22, 27] that span a recent time interval. For instance, only data received over the past hour are probed in order to meet real-time deadlines. As fresh positions keep arriving, such evaluation should be repeatedly applied in order to provide up-to-date, incremental response.

In such a *geostreaming* context, the relative weight of each isolated position in a trajectory is inherently *time-decaying*. When it first arrives, a position is most valuable, as it indicates the whereabouts of a moving object (e.g., a person, a vehicle or a container). But as time goes by, its importance is steadily diminishing, until it eventually becomes obsolete and practically negligible in the long motion path of that object. Taking inspiration from such an "amnesic" behavior [24], in this paper we introduce the notion of *multi-scale sliding windows* against trajectory streams. Instead of just restricting the focus on recent past using temporal constraints, we prescribe varying degrees of approximation for diverse portions of each trajectory. In a sense, we suggest a spatiotemporal analogue of the well-known concept of scale in cartography.

Towards this goal, we extend our previous work on multi-granular windows at varying levels of detail [21], and we exploit spatiotemporal properties inherent in evolving trajectories. We deem that windowing can effectively retain several, gradually coarser representations of each object's movement over greater time horizons towards the past; in return, higher precision should be reserved for the most recent segments. This can be achieved through diverse *scaled representations* per time horizon, in order to obtain increasingly generalized, yet always connected motion paths. Thanks to scaling, traces of any object consist of a similar amount of locations per time period, so they are lightweight and comparable irrespectively of their reporting frequency. Consider an application for fleet management, which monitors delivery trucks for a logistics firm. Typically, finest "zigzag" details of each itinerary usually matter for the latest 15 minutes. Suppressing most of them could still reliably convey motion characteristics over the past hour, whereas relatively few waypoints per vehicle suffice to give its general course today. Hence, this novel operator acts as an online simplifier per trajectory and constantly offers multiple views at varying resolutions over the motion history of objects. To reduce maintenance cost, a view at a given reso-

lution can get incrementally updated from the more detailed ones, and not directly from the original sequence (with the exception of the finest view, of course). In a nutshell, the underlying semantics of the proposed window is "drop detail with age." This data reduction paradigm exploits both spatial and temporal features per trajectory; such windows are genuinely sliding with time, but they also employ spatiotemporal criteria for determining qualifying positional updates.

In practice, this idea may be proven advantageous for applications like fleet management, traffic surveillance, wildlife observation, merchandise monitoring, maritime control, soldier tracking in battlefields etc. Typical operations include:

- *Trajectory filtering*: Evaluation of range or k-NN search requests can be boosted by examining contemporaneous, lightweight portions of trajectories at comparable scales instead of the detailed, time-varying original traces. In line with the "filter & refinement" paradigm [26], these reduced trajectories may constitute an index for the filtering phase in order to prune irrelevant candidates.
- *Ageing trajectory synopses*: Each object's course can be smoothly updated with time and suitably compressed with age to offer a reliable approximation. In essence, dropping unnecessary details towards the past can actually highlight the most salient spatiotemporal features. Compression is less lossy for recent traces so as to afford more accurate answers to related queries.
- *Efficient motion mining*: Identifying recent trends at varying resolutions can effectively expose important—or even latent—movement patterns that might be difficult to detect in rapidly accumulating data. For instance, changes in traffic flows across the road network can be instantly outlined from reduced representations without resorting to an expensive inspection of the exact routes.
- *Online multi-grained aggregates*: Crucial spatiotemporal measurements, like heading, speed, travel time, area of coverage, etc. [12] can be calculated per trajectory at several time horizons and resolutions. For instance, it makes sense to monitor average speed of each vehicle over the past minute, quarter, and hour at judiciously scaled simplifications of their traces so as to meet online expectations.
- *Advanced visualization*: Map rendering of trajectory features at diverse zoom levels can provide localized or personalized views of important phenomena. Imagine a traffic controller, who can readily overview vehicle circulation in the city, but can occasionally zoom into a congested junction to check for long queues. Further, customized symbology and suitable annotation can provide the means for versatile portrayal and dissemination through multi-modal, interactive maps.

This paper is an extended and revised version of [20]. Compared to that initial approach, it offers a detailed discussion of windowing semantics and a more careful examination of language constructs that can potentially assist in query formulation. In addition, a novel algorithmic framework has been developed, employing three alternative strategies for incremental window maintenance over trajectories. This is not just a proof of concept, since these techniques have been validated empirically, demonstrating significant advantages in terms of real-time response and reliable approximation. To the best of our knowledge, this is the first attempt to introduce composite windows over streaming trajectories with the following contributions:

(i) We advocate for the use of multi-scale sliding windows as a means of capturing essential trajectory features from an evolving positional stream, and we discuss their parameterized semantics in space and time (Section 3).
(ii) We develop maintenance methods to ensure cohesion of trajectory segments using a series of common articulation points that leave no gaps between trajectory features compressed at varying degrees in successive window levels (Section 4).
(iii) We indicate that typical spatiotemporal predicates and functions are directly applicable to these alternate, compressed representations. In addition, we demonstrate the expressiveness of multi-scale windowing through SQL-like statements for characteristic continuous queries involving trajectories (Section 5).
(iv) We conduct a comprehensive experimental study against synthetic geostreaming data in order to evaluate performance, accuracy, and robustness of the applied multi-resolution approximation (Section 6).

2 Related work

As the proposed framework injects ideas from window-based stream processing and multi-granular semantics into trajectory management, related work is reviewed next.

2.1 Windows over data streams

Continuous query execution has been established as the most renowned paradigm for processing transient, fluctuating, and possibly unbounded *data streams* [1, 7, 14, 27] in many modern applications, like telecom fraud detection, financial tickers, or network monitoring. In order to provide real-time response to multiple *continuous queries* that remain active for long, most processing engines actually restrict the amount of inspected data into temporary, yet finite chunks. Such windows [2, 3, 22] are declared in user requests against the stream through properties inherent in the data, mostly using timestamping on incoming items. Typically, users specify *sliding windows*, expressing interest in a recent time period ω (e.g., items received during last 10 minutes), which gets frequently refreshed every β units (e.g., each minute), so that the window slides forward to keep in pace with newly arrived tuples. At each iteration, the temporary *window state* consists of stream tuples within its current bounds; usually $\beta < \omega$, so state overlaps occur and successive window instantiations may share tuples.

2.2 Multi-granular semantics

The sliding window paradigm dictates a single timeline of instants at similar detail. Yet, time dimension naturally adheres to a hierarchical composition of *granules*, i.e., multiple levels of resolution with respect to a *time domain* $\mathbb{T}$. Each granule γ_k at level k consists of a fixed number of discrete time instants $\tau \in \mathbb{T}$. Taking the union of a set of consecutive granules at level k leads into a greater granule at level $k+1$, thus iteratively defining several levels of *granularity* [5], like seconds, minutes, hours, etc.

Regarding granularity issues in spatiotemporal databases, one approach [19] proposed functionality to support semantic flexibility of multiple representations and cartographic flexibility at multiple map scales. Representations of a real-world phenomenon may vary

according to the chosen perception, i.e., time, scale, user profile, point of view, etc. In another direction, a formal model for multi-granular types, values, conversions and queries has been developed in [4], also handling evolutions due to dynamic changes and events. However, none of these approaches considers management of moving objects and their trajectories, nor their stream-based processing.

2.3 Trajectory management

Several models and algorithms have been proposed for managing continuously moving objects in spatiotemporal databases. In [12], an abstract data model and query language were developed towards implementing a spatiotemporal DBMS extension where trajectories are considered as moving points. Based on that infrastructure, the SECONDO prototype [11] offers several built-in and extensible operations.

Besides, a discrete model proposed in [9] decomposes temporal development into fragments, and then uses a simple function to represent movement along every "slice." For trajectories, each time-slice is a line segment connecting two consecutively recorded locations of a given object, as a trade-off between performance and positional accuracy. Interpolation can be used to estimate intermediate positions and thus approximately reconstruct an acceptable trace of the entire movement.

As for trajectory compression, several simplification techniques have been suggested. Some of them opt for an acceptable approximation in terms of a given error margin, such as those in [6, 15, 16]. Besides, the space available for retaining a compressed sequence may also be crucial in online evaluation schemes [24].

2.4 Stream processing with multi-granular windows

The notion of a multi-level sliding window W was introduced in [21] as a collection of n time *frames* (also termed *subwindows*) at diverse user-defined granularities. Its semantics allow concurrent evaluation of a single continuous query against chunks of varying size from a single stream. By this scheme, subwindow $W_k\langle\omega_k, \beta_k\rangle$ at level k has its own time *range* ω_k and *slide step* β_k, effectively adjusting its rear bound t_k backwards from current time τ_c. As depicted in Figure 1 for a 3-level window over stream S of integer values, the substate of each frame (the shaded boxes) contains its subordinate ones in time dimension, while they are all nested under the widest W_{n-1} and keeping up with current time τ_c. A hierarchy of n subsumed frames can be created when $\beta_{k-1} \leq \beta_k$ and $\omega_{k-1} < \omega_k$ for each level $k = 1, \ldots, n-1$. For smooth transition between successive substates, it is prescribed that $\omega_k = \mu_k \cdot \beta_k$ for $\mu_k \in \mathbb{N}^*$, so a given frame W_k consists of a fixed number of primary granules of size β_k units each.

Such a nesting scheme can be maintained online in an incremental fashion. Thanks to inherent subsumption of window frames, for a subwindow W_k it suffices to retain in a queue g_k only those tuples in interval $(t_k, t_{k-1}]$ not already covered by its subordinating frames. Although a different slide step β_k may be specified per level k, window updates can be managed efficiently. In short, each queue g_k can be combined with an auxiliary one δ_k to buffer items expiring from its subordinating frame at level $k-1$. Figure 2 shows such a "stairwise" scheme of alternating "buffer" δ_k and "core" queues g_k, which can seamlessly maintain the overall window state with no duplicates or any tuples lost in transit between successive levels. Details can be found in [21].

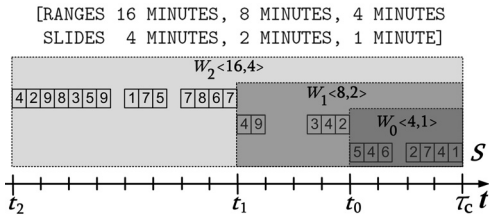


Figure 1: A 3-level sliding window.

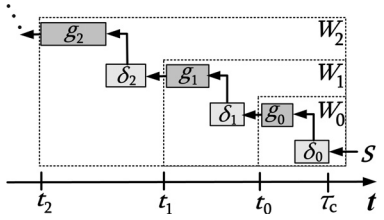


Figure 2: Stairwise processing scheme [21].

Emanating from this concept, we build up a new framework for online trajectory management using windows at multiple temporal extents and spatial resolutions.

3 Semantics of multi-scale windows over trajectories

Without loss of generality, we assume a discrete model with 3-d entities of known identities moving in two spatial and one temporal dimensions, i.e., point objects (not regions or lines) moving in a Euclidean plane across time as illustrated in Figure 3a. For a given object o_i, its successive samples are pairs of geographic coordinates $(x, y) \in \mathbb{R}^2$. Point locations are measured at discrete, totally ordered timestamps τ from a given time domain $\mathbb{T}$ of primitive time instants (e.g., seconds). A large number N of objects are being monitored (e.g., tens of thousands of vehicles), so their relayed locations flow into a central processor as a *positional stream* S of timestamped tuples $\langle o_i, x, y, \tau \rangle$. Thus, each trajectory is considered as an evolving sequence of point samples collected from a given moving source at distinct time instants (e.g., a GPS reading every few seconds). Each object o_i may emit updates at its own rate of ρ_i positions/timestamp, which can be time-varying, e.g., less dense samples when moving straight at steady speed. However, no updates are allowed to already registered locations, so that coherence is preserved among *append-only* trajectory segments.

Thanks to monotonicity of time, trajectories always evolve along the temporal dimension. Thus, the semantics of sliding windows [22] against such positional streams can abstract the recent portion of trajectories and thus provide dense subsequences without gaps. For efficiency and robustness, we suggest that continuous queries could be evaluated against less detailed representations of objects' movement, purposely compressed on-the-fly in an "amnesic" fashion. Overall, the proposed window operator should act as a filter in two successive stages:

(i) *time-based filtering* narrows the examined streaming data down to finite chunks of reported locations at progressively smaller intervals of interest; and
(ii) *trajectory-based filtering* reconstructs subsequences of locations per object and dynamically applies flexible simplifications at each one of them.

3.1 Time-based filtering

We consider a window W specified by a continuous query as a hierarchy of n subsumable sliding frames [21] concurrently applied at time τ_0 over positional stream S, as in Section 2.4. At level k, frame W_k has a fixed-size temporal *range* ω_k always greater than its subordinate ones, whereas it moves forward every β_k time units (its *slide* step). In addition,

it specifies a *scale* factor σ_k, as analyzed next. The actual time bounds of frame W_k at any instant $\tau_c \geq \tau_0$ is its current *scope*. This smoothly moving, fixed-size interval that covers all tuples currently in W_k is determined by

$$scope_k(\tau_c) = [\ \mathtt{max}\{\tau_0,\ \tau_c - \lambda_k - \omega_k + 1\},\ \tau_c - \lambda_k\ ]$$

where $\lambda_k = \mathtt{mod}(\tau_c - \tau_0, \beta_k)$ is a time-varying lag behind current timestamp τ_c. So, each frame neither slides forward at each timestamp nor upon arrival of every position, but discontinuously (once $\lambda_k = 0$) in a deterministic pattern, as discussed in [22]. For instance, $[t_1, \tau_c]$ is the current scope of frame W_1 (Figure 1). Note that for each W_k, the rear bound of its scope is initially τ_0, since the subwindow is "half-filled" as long as its actual range is less than ω_k. Later on, the rear bound becomes $t_k = \tau_c - \lambda_k - \omega_k + 1 > \tau_0$, while the front bound is at $\tau_c - \lambda_k$. Both bounds slide by β_k units in tandem.

Upon sliding, this stage materializes the *time-filtered state* of respective subwindow W_k as a set $\mathcal{C}_k(\tau_c) = \{s \in S : s.\tau \in scope_k(\tau_c)\}$ of stream items having timestamp τ within its actual bounds. For the setting in Figure 1, current state $\mathcal{C}_0(\tau_c)$ of the bottommost frame contains positions over the past 4 minutes; one level above, $\mathcal{C}_1(\tau_c)$ gets those received during last 8 minutes, and widest frame $\mathcal{C}_2(\tau_c)$ includes every location over 16 minutes. Note that each state $\mathcal{C}_k(\tau_c)$ provides a batch of scattered point locations, without any associations regarding their actual sequence as object trajectories.

3.2 Trajectory-based filtering

This stage employs a *demultiplexing* task at every frame W_k. It partitions tuples from previously obtained state $\mathcal{C}_k(\tau_c)$ into distinct subsequences per object. For each object identifier o_i, this subsequence at level k reconstructs a truncated trajectory; it is called *path* and consists of time-ordered positions reported from o_i within frame W_k. Thus:

$$path_k(o_i) = \{s \in \mathcal{C}_k(\tau_c) : s.o_i = o_i \wedge (\forall s' \in \mathcal{C}_k(\tau_c), s'.o_i = o_i : s.\tau < s'.\tau\ \vee\ s'.\tau < s.\tau)\}.$$

For instance, $path_1$ in Figure 3a is the trajectory portion along interval ω_1. At time τ_c a set of such paths (one per object) is obtained for a period ω_k backwards from τ_c, as prescribed for the k-th level. But the bulk of this positional data may still be considerable, especially if windows have wide ranges and many levels. Moreover, evaluation must be repeated for each new slide, and it certainly demands much processing power.

Hence, we advocate for further reduction of truncated paths. Such a process should take into account the actual detail of original trajectories, as objects may not necessarily have a standard reporting frequency. Let $|path_k(o_i)|$ denote the number of points reported from object o_i during past interval ω_k. As it may occur $|path_k(o_i)| \gg |path_k(o_j)|$ for two distinct objects o_i, o_j during the same interval ω_k, this compression should yield comparable traces among trajectories with possibly diverse amounts of relayed samples. Suppose that object o_i has an average rate ρ_i of locations during period ω_k. Then, a total of $\rho_i \cdot \omega_k$ samples have been recently collected from o_i. Ideally, we would like to keep $\sigma_k \cdot \omega_k$ points from each object, in case that all were sending an update per timestamp. Due to fluctuating rates, the intended *smoothing factor* δ_k^i for eliminating superfluous samples from this trajectory must be $\delta_k^i = \frac{\sigma_k}{\rho_i}$. Of course, compression should only be applied if $\rho_i > \sigma_k$. Otherwise, dropping original samples is not necessary, and points relayed by o_i must be left intact.

Example 1. Let a frame W_k specify $\langle \omega_k = 60$ sec, $\beta_k = 10$ sec, $\sigma_k = 0.1 \rangle$ against trajectories of two objects. Object o_1 has relayed 30 points, but o_2 only 12 positions over last $\omega_k = 60$

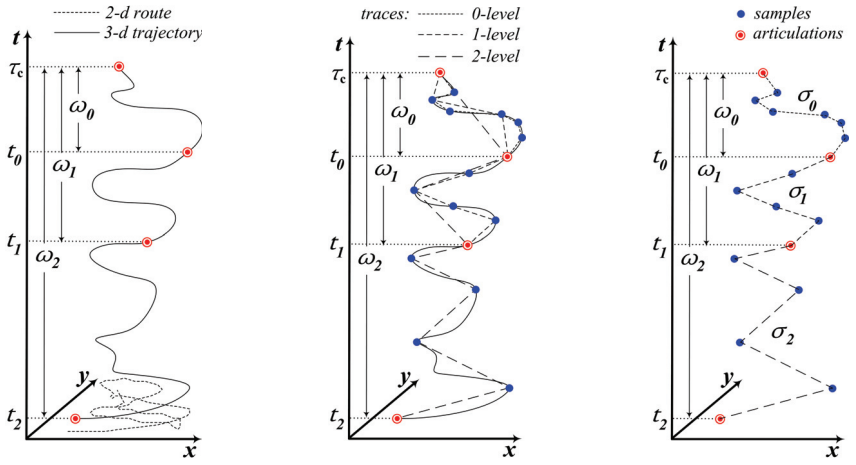


Figure 3: Multi-scale sliding window over trajectory: (a) Original sequence. (b) Traces over diverse time horizons. (c) Unified synopsis composed of non-overlapping paths.

sec, so $\rho_1 = 0.5$ and $\rho_2 = 0.2$ points/sec respectively. According to semantics, reduction is applied with $\delta_k^1 = \frac{0.1}{0.5} = 0.2$, eventually preserving $0.2 \cdot 30 = 6$ points from o_1. Similarly, $\delta_k^2 = \frac{0.1}{0.2} = 0.5$, giving $0.5 \cdot 12 = 6$ locations from o_2 as well, so scaling returns the same amount of locations irrespective of the actual rate of updates. Hence, comparable approximations are obtained from either object for frame W_k. □

By fixing a ratio $\sigma_k < 1$ at level k for all trajectories, accuracy of their representation is restricted at the desired detail. In effect, σ_k acts as a *scale* parameter for frame W_k of the window. It prescribes the maximum degree of detail tolerated amongst its accumulated positions per path. So, all trajectories get separately smoothed at equivalent approximations, with several samples intentionally discarded. This does not necessarily mean that a similar number of locations must be retained per object; depending on the actual motion of a given o_i, less than $\lfloor |path_k(o_i)| \cdot \frac{\sigma_k}{\rho_i} \rfloor$ points may also constitute a reliable trace, e.g., along a straight course at steady speed. Naturally, reduction should be intensified for upper frames and less strong for each subordinate one, hence we establish that $1 \geq \sigma_0 \geq \sigma_1 \geq \ldots \geq \sigma_{n-1} > 0$ among all n levels of window W.

Coupled with parameters for temporal range ω_k and slide β_k, scale σ_k intends to provide an *aging-aware simplification* of trajectories. We defer discussion of reduction options for Section 4, as this problem is orthogonal to window semantics. In general, the less the scaling factor at a given level k, the sparser the locations preserved per trajectory, so the various σ_k values per frame actually control the intensity of approximation. Note that each compressed representation still maintains a cohesive $path'_k(o_i)$, thanks to inherent timestamp ordering of locations in the subsequence (Figure 3b).

This stage returns a *trajectory-filtered state* $\mathcal{W}_k(\tau_c) = \{path'_k(o_i), \forall i, 1 < i < N\}$ for window level k at current time τ_c. It contains a single compressed sequence per object, as opposed to dispersed positions that a conventional sliding window would return.

3.3 Discussion

Original trajectory data itself does not become multi-granular (e.g., alternate sequences in hours or days), and the underlying spatiotemporal model for their representation is kept as simple as possible. Instead, it is the proposed windowing operator that produces a series of n temporary datasets of increasing temporal extents and gradually sparser positional samples. Those repetitively refreshed paths are meant to be utilized primarily in continuous query evaluation, and not necessarily for permanent storage. Parameterizations for range ω_k, slide β_k, and scale σ_k per level k are defined by users in their requests, thereby controlling the desired precision of response.

Granularities strictly refer to levels of detail for window specification and do not affect the underlying data model. Of course, relationships may exist among granularities, i.e., *'finer-than'* and *'coarser-than'* operators defined through inclusion and overlapping [4, 5]. Thanks to such inherent relationships, most granules can be mapped onto the finest one supported by the model (e.g., seconds), and thus simplify calculations.

Multi-scale windowing should be distinguished from *partitioned windows* [2], although they also use attribute values to demultiplex incoming items into disjoint partitions. But here, an important "path creation" step is involved (Section 3.2), which yields sequential paths per object. This policy also differs substantially from *load shedding* techniques that judiciously drop data points upon arrival [10]. In contrast, all locations get admitted into the stream database for precessing. Each query may specify diverse time horizons and scales through windows, eventually discarding superfluous points and producing its own multi-resolution paths over the recent motion history.

4 Online maintenance of multi-resolution motion paths

Next, we present a scheme for efficiently maintaining trajectory states across multi-scale frames. We identify crucial properties that compressed sequences should respect, and we propose three strategies for online evaluation against streaming positions.

4.1 Representation issues for approximated trajectories

The proposed window operator must not solely extract simplified paths per object and offer multi-scale representations for querying, but should perform this task repetitively as trajectories evolve. So, processing must be *incremental* as fresh locations continuously arrive, and also *shared*, by potentially exploiting already computed paths.

Towards these intertwined goals, we opt for a scheme that can digest point locations across many window frames. Since more detailed representations are prescribed for the narrowest frame (i.e., closest to present), selected point samples per trajectory may be progressively discarded when adjusting the compressed segments upwards in the window hierarchy. In effect, fewer and fewer points remain in the coarser frames by eliminating certain motion details, in accordance with their scale factor σ_k.

Preservation of certain *articulation points* per trajectory is our seminal idea for a coordinated maintenance of multiple paths. As shown in Figure 3b, those points signify object locations at time instants that mark frame boundaries (or samples available closest to that time, depending on reporting frequency). Such a methodology can promote a fair share of indicative locations to wider frames by keeping account of such points persistently. Fur-

ther, it may also yield a unified *synopsis* of each trajectory, i.e., a cohesive representation with non-overlapping point sets per level, each spanning consecutive intervals joined at those articulations (Figure 3c). Apart from expectations for optimized state maintenance, this scheme may also prove advantageous for a versatile portrayal of trajectories on maps across multiple scales. In [25], the similar notion of "reference points" was used for data reduction specifically in urban movement scenarios. However, it is questionable whether that compression could be applied in an online fashion, since it draws heavily from geographic context (e.g., points are always along an underlying transportation network) and spatial cognition techniques (e.g., wayfinding); none of these assumptions apply for articulation points.

Retained samples from diverse trajectories may not be necessarily synchronized, i.e., measured at identical time instants. Although synchronization facilitates comparison among trajectories, it might lead to oversimplified paths that could occasionally miss interesting motion details when samples are chosen at a fixed frequency. Instead, samples should keep each compressed trace as much closer to its original course, chiefly by minimizing approximation error as in trajectory fitting methods [6,16]. Most simplification policies incur considerable cost in updates, as they have to probe each location more than once; at best, this cost is $O(m \log m)$, where m is the number of examined points. Although such algorithms can offer good approximation accuracy, they would not be efficient in real-time evaluation. For checking approximation quality, an *error tolerance* ϵ must be also specified, which is rather difficult to assess beforehand for evolving trajectories from numerous objects. Another crucial issue for path maintenance is how to attain a specific smoothing factor per frame W_k. To address these goals, we next propose a processing scheme with three alternative policies for online multi-scale approximation of trajectories over windows.

4.2 Processing mechanism

We consider a centralized processor where users can register their continuous queries using multi-scale windows over timestamped positions from N moving objects, as detailed in Section 3. The maintenance scheme from [21] with a chain of core and buffer nodes can be adapted to work over spatiotemporal features, so that point selection only occurs at transitions between frames (i.e., in buffer nodes). Involving a small fraction of the accumulated samples, such a process can suitably drop less important points in an incremental fashion when aging locations ascend through the stairwise organized frames (Figure 2). Thanks to inherent nesting, each frame handles trajectory segments not already covered by its subordinate ones. Scaling is achieved through online filtering of locations for object o_i expiring from a frame at level $k-1$, so that only a prescribed fraction $\delta_k^i = \frac{\sigma_k}{\rho_i}$ of them propagates up in the window hierarchy.

Example 2. Let a multi-scale window against the trajectory of an object o, which sends its location at every clock tick, i.e., $\rho = 1$ position/timestamp. Suppose that at k-th level, this window specifies a frame W_k ranging over $\omega_k = 20$ timestamps and sliding every $\beta_k = 4$ timestamps, as shown in Figure 4. Scale factor $\sigma_k = 0.5$ signifies that only half of the transmitted locations should be kept in the compressed path. □

Implicitly, such *nested processing* dictates that fresh locations close to current time τ_c are buffered over a period of β_k time units at most. When window frame W_k slides forward, selected positions from object o_i get included in its trajectory path at level k. Those filtered

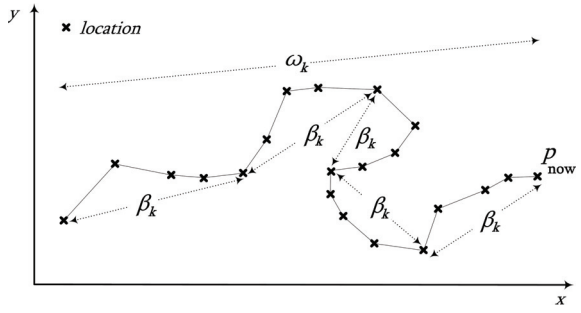


Figure 4: A window frame W_k specifying $\langle \omega_k = 20, \beta_k = 4, \sigma_k = 0.5 \rangle$ over a trajectory.

Algorithm 1 Multi-scale windowing over trajectories

1: **Procedure** *Coordinator* ($\{\langle \omega_k, \beta_k, \sigma_k \rangle, k = 0..n\text{-}1\}$) *//Against a n-level window* $\mathcal{W}$
2: **Input:** Streaming positional updates $\langle o_i, x, y, \tau \rangle$ from $i = 1..N$ moving objects;
3: **Output:** A collection $\mathcal{C}$ of scaled trajectory paths $\mathcal{S}_i = \{s_0, ..., s_{n-1}\}$ from each object o_i;
4: $traj_i$: evolving trajectory of object o_i; ρ_i : reporting rate (positions/timestamp) for o_i;
5: $\mathcal{Q}_i$: a queue buffer per object o_i;
6: **while** input is not exhausted **do**
7: $\mathcal{Q}_i$.push ($\langle o_i, x, y, \tau \rangle$); *// Buffer incoming positions to the corresponding queue*
8: **if** no more positions have arrived for timestamp τ **then**
9: $\mathcal{C} \leftarrow \emptyset$; *//Start a new execution cycle to detect trajectory states for all objects*
10: **for** each object o_i **do**
11: $traj_i \leftarrow$ *updateTrajectory*(o_i); *//Operation depends on scaling strategy*
12: $\mathcal{C} \leftarrow \mathcal{C} \cup$ *adjustFrames*(o_i, τ) ; *//Operation depends on scaling strategy*
13: **end for**
14: **end if**
15: **end while**
16: **End Procedure**

points are the result of scaling applied over candidate positions that had been buffered upon expiration from level $k-1$. Simultaneously, positions referring to the oldest period of β_k timestamps along that path are expiring; so they get buffered for subsequent promotion to level $k+1$. Because changes only occur at the front and rear bounds of frames, no action should be taken against intermediate points, thus avoiding recomputation of isolated subwindow states from scratch.

Algorithm 1 outlines the main task running on the server, which provides the overall state $\mathcal{C}$ of trajectory paths from every object. It accumulates incoming locations in a separate queue $\mathcal{Q}_i$ per object $o_i, i = 1..N$ (line 7). Evaluation takes place in *execution cycles*, either at every timestamp or usually depending on elementary sliding step β_0 of the window. After each cycle, every point sequence $traj_i$ is modified by method *updateTrajectory()* (line 11). In addition, method *adjustFrames()* rearranges window states against each trajectory and a fresh collection of their scaled paths $\mathcal{S}_i = \{s_0, ..., s_{n-1}\}$ (one per frame) is returned per object o_i (line 12). We stress that the functionality of both methods depends on the approximation strategy employed for scaling. One option is to randomly take a given percentage of the buffered samples. Another approach is based on velocity vectors and aims to pick positions that signify important changes in each object's course. Yet another

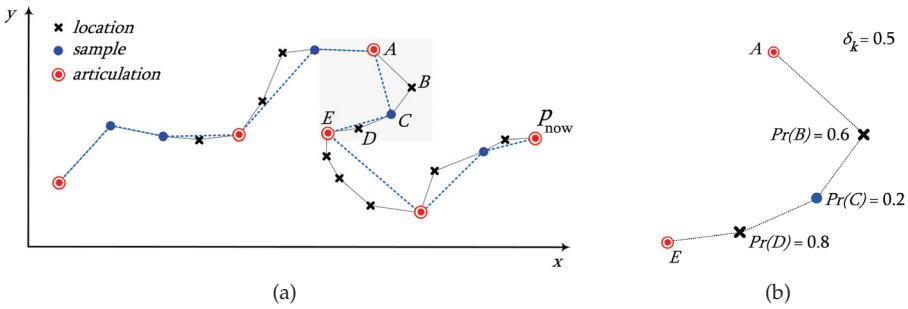


Figure 5: Scaling with random sampling: (a) Chosen articulation points and random samples. (b) Taking samples with probability $\leq \delta_k$ once frame W_k slides.

possibility is to discard locations that incur the least distortion in trajectory shape. Next, we analyze each strategy in turn.

4.3 Scaling strategy using random sampling

Typically, systematic sampling involves a random start and then proceeds by selecting every m-th element onwards. To imitate window semantics, the starting location may be the latest position when each frame is evaluated. Then, going backwards in time at level k, we could keep one sample out of every successive batch of m locations, no matter the actual frequency ρ_i of updates received from moving object o_i.

In our scenario, we devise an alternative strategy based on random sampling. Clearly, at every window frame W_k the algorithm needs to examine locations for object o_i buffered every β_k timestamps. First, the most fresh location (i.e., the one with timestamp closest to the front bound of k-th frame) is chosen as articulation point. For each of the remaining candidates, we toss a coin (independently for each point) and keep locations with probability $\delta_k = \frac{\sigma_k}{\rho_i}$ to achieve the desired scaling.

Example 3. For the trajectory in Figure 4, sampling is applied every $\beta_k = 4$ timestamps. The shaded box in Figure 5a highlights one such transition (magnified in Figure 5b), which includes fresh points $\{B, C, D, E\}$, plus the insofar latest location A. Since E now becomes the most recent position, it is by default a new articulation point. After taking probabilities of the rest, it turns out that only C should be kept, since $Pr(C) < \delta_k = \frac{\sigma_k}{\rho_i} = 0.5$. Hence, the path gets updated with points C and E. □

Apart from systematic maintenance of articulation points, note that there is no spatio-temporal rule for choosing locations, e.g., according to motion patterns as in trajectory fitting methods [6, 16]. But, as this selection is entirely probabilistic, it absolutely fits for online processing. Algorithm 2 provides the pseudocode of such a single-pass process. Indeed, upon admission of a new position from object o_i, it can be readily decided whether to include it or not as a sample in any window frame. Each choice is made according to the smoothing factor $\delta_k = \frac{\sigma_k}{\rho_i}$ calculated for level k, and depends on the actual frequency ρ_i of positional updates for object o_i (line 3).

Task *updateTrajectory()* annotates each point with a n-sized bitmap $\mathcal{B} = b_{n-1}..b_1b_0$, having a bit b_k per window frame W_k. If a point is randomly selected for the k-th frame, its b_k bit is accordingly set (line 10). Articulation points are temporally closest to frame transitions (line 7), and thus have their respective bit always set (line 8). If a point is discarded from the bottommost frame, it is instantly suppressed from all upper levels (lines 12–14). This "multi-tier" sampling is performed once per incoming point; a location is dropped from the trajectory representation, unless it participates in at least one frame according to its associated bitmap $\mathcal{B}$ (lines 16–18).

Algorithm 2 Scaling with *random sampling*

```
1: Function updateTrajectory (object o_i)      //Variant for random sampling
2: B ← b_{n-1}..b_1b_0 : a n-size bitmap (initially all reset) assigned to each location;
3: Update ρ_i according to positions buffered in Q_i;
4: while Q_i is not empty do
5:    ⟨o_i, x, y, τ⟩ ← Q_i.pop();      //Consume candidate locations in chronological order
6:    for each frame W_k, k ∈ {0 .. n-1} do
7:       if mod(τ, β_k) = 0 then
8:          b_k ← 1;   //Position closest to front bound of this frame becomes articulation point
9:       else
10:         b_k ← tossCoin(σ_k/ρ_i);     //Decide probabilistically if the point is kept for this frame
11:      end if
12:      if b_0 = 0 then
13:         break;    //If not included in lowest frame, do not consider this location anymore
14:      end if
15:   end for
16:   if OR(b_{n-1}..b_1b_0) = 1 then
17:      traj_i ← traj_i ∪ ⟨o_i, x, y, τ, B⟩;   //Only chosen locations get appended into trajectory
18:   end if
19: end while
20: return traj_i;
21: End Function

22: Function adjustFrames (object o_i, timestamp τ)      //Variant for random sampling
23: S_i ← ∅;  //Prepare to construct new trajectory state for o_i
24: for each frame W_k, k ∈ {0 .. n-1} do
25:    if mod(τ, β_k) = 0 then
26:       t_rear ← τ − ω_k;          //Rear bound of k-th frame
27:       if k = n-1 then
28:          Expunge locations with timestamps < t_rear from traj_i;
29:       else
30:          b_k ← 0 for locations with timestamps < t_rear;  //Reset k-th bit to mark expiration
31:       end if
32:       s_k ← { positions of traj_i within range ω_k having b_k = 1};  //ordered by timestamp
33:       S_i ← S_i ∪ {s_k};          //Append scaled path s_k to trajectory-filtered state for o_i
34:    end if
35: end for
36: return S_i;                        //Report current trajectory-filtered state for object o_i
37: End Function
```

Once frame W_k slides forward, bit b_k should be reset for locations expiring from it. For a given object o_i, method *adjustFrames()* is called and at each frame transition (line 25) it checks for points older than the current rear bound of that subwindow (line 26). Those expiring from the topmost frame are discarded altogether from o_i's trajectory, whereas for subordinate frames only the respective bits should be reset (lines 27–31). As soon as the k-th frame is adjusted, scaled path s_k is updated to reflect the current sequence of points participating in W_k. This collection of paths for the n applied subwindows is the current *trajectory-filtered state* S_i for object o_i (lines 32–33).

Clearly, this technique utilizes lightweight bitmaps with almost negligible overhead for controlling complex window operations over trajectories. Memory footprint for retaining samples is constant per object, as it depends entirely on window specifications. Since this is a single-pass process and each point is checked once and for all, the cost is $O(1)$ per location. This is important in terms of robustness and timeliness, especially when a system has to monitor numerous objects and to provide answers fast. Nonetheless, this policy raises subtle issues, as verified in our experiments. Positional updates may be arriving at fluctuating rates and objects can make arbitrary turns, so this kind of sampling could often miss important trajectory changes. Since choices obey no deterministic rules, it may occur that processing two identical trajectories may lead to approximations of equal size, but with different shapes, as samples are taken at random. Even worse, sampling may occasionally lead to deviating or largely distorted traces for several objects, with increased approximation error.

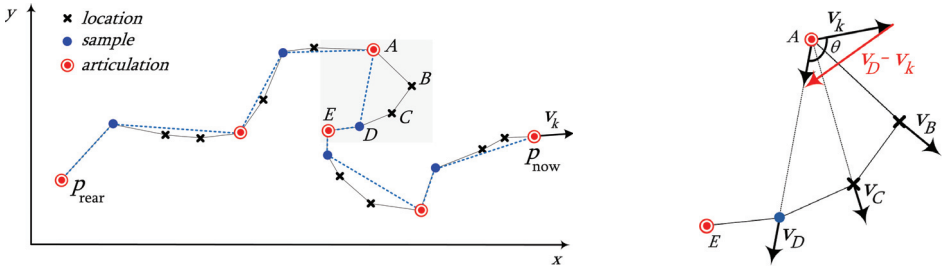
4.4 Scaling strategy based on velocity vectors

Although random sampling can effectively reduce trajectory volumes down to the prescribed sizes per frame, it chooses locations blindly, irrespective of their significance to objects' movement. But scaling could take advantage of motion features inherent in such sequential data. Of them, *velocity* is the principal magnitude that characterizes how each object moves, by measuring its actual speed and heading.

Using such velocity vectors is reminiscent of linear *dead reckoning* techniques [30]. In that case, a moving object and the server both share a linear prediction function for determining object's current position. When an object identifies that its current vector deviates significantly from the predicted movement, it has to relay a new position and to modify its linear function accordingly. Yet, dead reckoning assumes that objects carry some processing capabilities, whereas it is mostly geared towards communication savings by eliminating less important positional updates. Overall, the main concern is data storage with acceptable accuracy rather than continuous query evaluation, as we consider here using sliding windows over trajectories.

In our centralized approach, the server maintains velocity vectors per window frame independently for each monitored object. As illustrated in Figure 6, velocity $\vec{v}_k$ captures the general course of object o_i within the temporal range of window frame W_k. Each $\vec{v}_k$ translates into two real values for average speed and heading during a recent period ω_k. When required, $\vec{v}_k$ can be readily computed from the two extreme locations p_{rear} and p_{now} in the retained subsequence. Not surprisingly, $\vec{v}_k$ is based on articulation points that explicitly reflect the bounds of respective frame W_k.

For a given object o_i, when it comes to choosing samples buffered in a queue $\mathcal{P}_k$ for admission to frame W_k, we opt for a policy that favors selection of positions incurring



(a) (b)

Figure 6: Scaling using velocity vectors: (a) Retain samples that deviate most from currently known velocity $\vec{v}_k$ across window frame W_k. (b) For each floating point in current batch (the shaded box in (a)), find its velocity vector w.r.t. anchor point A. Then, choose samples by decreasing difference of their vectors from velocity $\vec{v}_k$.

larger deviations from its currently known vector $\vec{v}_k$. More specifically, the last location p_{anchor} of object o_i within range ω_k (just before a slide starts) is fixed as its *anchor point* for the k-th frame. Among locations waiting in $\mathcal{P}_k$, the one with the latest timestamp becomes a new *articulation point*, as this will be closest to the new front bound of frame W_k after the slide. For the remaining positions in $\mathcal{P}_k$, each is considered as a *floating* point p_f and coupled with a velocity vector $\vec{v}_f$ connecting it with p_{anchor}. This temporary $\vec{v}_f$ signifies the possible course of o_i if p_f were chosen as its next sample. In Figure 6b, A is anchor point, E becomes a new articulation, whereas points B, C, and D are floating. We are interested in picking up samples that signify substantial diversions from the course predicted by $\vec{v}_k$. In order to quantify such diversions, we measure the vector difference between $\vec{v}_f$ and $\vec{v}_k$, according to the law of cosines:

$$\gamma = \|\vec{v}_f - \vec{v}_k\| = \sqrt{v_f^2 + v_k^2 - 2v_f v_k \cos\theta}$$

where θ is the angle between the two vectors, as depicted in Figure 6b for floating point D. Thus, it suffices to select points in descending order of their γ values.

The number m_k of samples to choose when W_k slides forward and starts consuming items buffered in $\mathcal{P}_k$ is another concern. It is safe to choose $m_k = \lfloor |\mathcal{P}_k| \cdot \frac{\sigma_k}{\rho_i} \rfloor$ locations at this frame transition, and thus leverage actual arrival rate ρ_i for object o_i against the desired scale σ_k. Given that frame W_k consists of a fixed number of primary granules, each one spanning β_k timestamps and providing a scaled subsequence of m_k locations, it easily turns out that W_k would contain no more than $\lfloor |path_k(o_i)| \cdot \frac{\sigma_k}{\rho_i} \rfloor$ points in total, exactly as stipulated by scale factor σ_k.

Note that points assigned as articulations at level k, may later lose this status when promoted upwards. Clearly, each frame chooses suitable articulations among points buffered for admission to it. The sole purpose of articulations is to guarantee cohesion between trajectory paths at successive levels in window hierarchy (Figure 3b).

Example 4. For the trajectory in Figure 4, its scaled representation according to velocity vectors is illustrated in Figure 6. Articulation points are chosen when frame W_k slides forward. Consider an earlier stage, when the last known point is A and the window is

ready to choose locations among points buffered in $\mathcal{P}_k = \{B, C, D, E\}$ (inside the shaded box). Given the window specification and the rate of positional updates, it turns out that $m_k = \lfloor 4 \cdot \frac{0.5}{1} \rfloor$, so two samples should be taken. As shown in detail in Figure 6b, it is straightforward that E becomes a new articulation; for the rest, velocity vectors are derived having A as their anchor. Since D is the location with the greater velocity difference, it is picked as the second sample at this transition. □

In terms of actual evaluation, this strategy requires modified versions for methods *updateTrajectory()* and *adjustFrames()*, as listed in Algorithm 3. The former is practically reduced to expand a trajectory with fresh positions (lines 4–6) and then push them only into the bottommost frame to refresh its respective path s_0 (line 7). This is done on purpose, in order to avoid duplication of points shared between paths at overlapping window frames. As for *adjustFrames()*, it mainly collects in a timestamp-ordered buffer $\mathcal{P}_k$ all points expiring from level k and adjusts trajectory path s_k accordingly (lines 34–39). The most crucial task is undertaken by routine *promotePoints()*, which handles admission of locations at every window level. As already explained, for each point waiting in queue $\mathcal{P}_k$, it measures its vector difference γ from the known velocity $\vec{v}_k$ at level k. In order to judiciously pick points that may incur larger deviations, candidates are inserted into a max-heap structure $\mathcal{H}_k$ by descending γ values (lines 17–21). Since articulation points are by default chosen according to timestamps (line 16), it suffices to repetitively pick locations from $\mathcal{H}_k$, until their total number reaches the desired m_k (line 23). All these locations returned in timestamp order (line 24) can then be used to update the respective trajectory path s_k.

Overall, this strategy integrates spatiotemporal conditions when choosing samples. At each frame slide, computation of velocity vectors per object costs $O(|\mathcal{P}_k)|$ at level k, i.e., is linear to the number of points considered during this transition. Sorting vector differences using heaps costs an additional $O(|\mathcal{P}_k| \cdot \log |\mathcal{P}_k|)$ at level k. Despite the widening slides, buffers at higher window levels contain diminishing fractions of the original locations. Thanks to the scaling effect of each successive level, fewer and fewer points propagate across the window hierarchy, thus offering substantial space savings. As verified in the experimental study, the cost seems tolerable. On the downside, picking locations independently on the basis of their deviation from a fixed anchor point, wrongly ignores any intermediate smaller variations in object's course that cumulatively might be significant. In the example of Figure 6, points B and C are dropped in favor of D. However, if C were selected, the approximate path would be closer to the actual route. In addition, this policy may end up selecting several consecutive positions on the basis of their high γ values, even if they occur along a straight line. This drawback clearly shows that the algorithm disregards the importance of elementary displacements between successive points in the sequence.

4.5 Scaling strategy using synchronous Euclidean distance

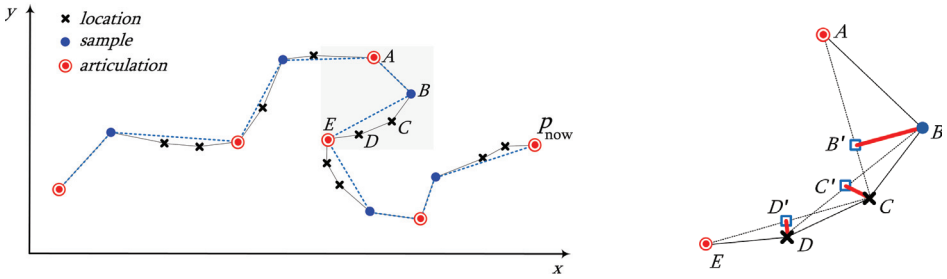
In an attempt to alleviate shortcomings of velocity vectors, we propose another scaling strategy, which eliminates points that would incur the smallest change in the shape of each trajectory. Towards this goal, locations with minimal distances from locally approximated segments are discarded. Similar techniques have also been suggested for managing trajectories in moving object databases. Among them, the spatiotemporal variants of Douglas-Peucker simplification algorithm [16] would incur significant cost, as multiple passes may

Algorithm 3 Scaling with *velocity vectors*

1: **Function** *updateTrajectory* (object o_i) *//Variant for velocity vectors*
2: Update ρ_i according to items buffered in Q_i;
3: **if** mod$(\tau, \beta_0) = 0$ **then**
4: **for** each location $\langle o_i, x, y, t\rangle \in Q_i$ **do**
5: $traj_i \leftarrow traj_i \cup \langle o_i, x, y, t\rangle$; *//Append fresh locations into that trajectory*
6: **end for**
7: $s_0 \leftarrow s_0 \cup$ *promotePoints*(0, Q_i); *//Lowest frame slides to accept batch of latest locations*
8: **end if**
9: **return** $traj_i$;
10: **End Function**

11: **Function** *promotePoints* (level k, locations $\mathcal{P}_k$) *//Variant for velocity vectors*
12: $\mathcal{H}_k \leftarrow \emptyset$; *//Heap of candidate locations sorted by γ*
13: $m_k \leftarrow \lfloor |\mathcal{P}_k| \cdot \frac{\sigma_k}{\rho_i} \rfloor$; *//Max number of points to be admitted into k-th frame*
14: $p_{anchor} \leftarrow$ ending point of current path s_k;
15: $\vec{v}_k \leftarrow$ velocity vector for the portion of $traj_i$ within range ω_k;
16: $p_{artl} \leftarrow \mathcal{P}_k$.pop_back(); *//Latest location becomes articulation point*
17: **for** each buffered location $p_f = \langle o_i, x, y, t\rangle \in \mathcal{P}_k$ **do**
18: $\vec{v}_f \leftarrow$ velocity vector from p_{anchor} to floating point p_f;
19: $\gamma \leftarrow \|\vec{v}_f - \vec{v}_k\|$; *//Vector difference by the law of cosines*
20: $\mathcal{H}_k$.insert($\langle o_i, x, y, t, \gamma\rangle$); *//Push point p_f into the heap according to its γ value*
21: **end for**
22: $\mathcal{P}_k$.clear(); *//Empty buffer for the next cycle*
23: $\mathcal{L}_k \leftarrow$ pop (m_k-1) items from $\mathcal{H}_k$; *//Locations to be promoted into k-th level*
24: **return** $\mathcal{L}_k \cup p_{artl}$; *//Emit samples in timestamp order*
25: **End Function**

26: **Function** *adjustFrames* (object o_i, timestamp τ) *//Variant for velocity vectors*
27: **for** each frame $W_k, k \in \{0 \,..\, n\text{-}1\}$ **do**
28: $\mathcal{P}_k \leftarrow \emptyset$; *//Buffer for locations of o_i that expire from k-th level*
29: **if** mod$(\tau, \beta_k) = 0$ **then**
30: $t_{rear} \leftarrow \tau - \omega_k$; *//Rear bound of k-th frame*
31: **if** $k = n$-1 **then**
32: Expunge locations with timestamps $< t_{rear}$ from $traj_i$ and s_{n-1};
33: **else**
34: **for** each stored location $\langle o_i, x, y, t\rangle \in s_k$ **do**
35: **if** $t < t_{rear}$ **then**
36: $\mathcal{P}_k$.push($\langle o_i, x, y, t\rangle$); *//Collect expired points from k-th level sorted by timestamp*
37: Remove $\langle o_i, x, y, t\rangle$ from s_k; *//Expunge expired point from scaled trajectory path*
38: **end if**
39: **end for**
40: $s_{k+1} \leftarrow s_{k+1} \cup$ *promotePoints*(k+1, $\mathcal{P}_k$); *//Expand with locations expired from k-th frame*
41: **end if**
42: Replace s_k in S_i; *//New trajectory path for o_i at k-th frame*
43: **end if**
44: **end for**
45: **return** $\mathcal{S}_i$; *//Report current trajectory-filtered state for object o_i*
46: **End Function**



(a) (b)

Figure 7: Scaling using synchronous Euclidean distance: (a) Keeping samples that denote locally significant changes in movement. (b) For each batch of points buffered in a frame transition, at most m_k of those with maximal SED values are retained.

be needed per trajectory. Tracking protocols examined in [15] require that the approximate trace deviates no more than a certain accuracy bound ϵ from the actual path. To check this, positional batches are collected and filtered at the moving sources before relaying any updates to the server, in order to save communication cost. But such protocols cannot be used with windows, as these latter constructs are inherently liaised to continuous queries registered at the server.

Our third scaling strategy works by minimizing *synchronized Euclidean distances (SED)* over triples of successive locations [24], while preserving up to $\lfloor |path_k(o_i)| \cdot \frac{\sigma_k}{\rho_i} \rfloor$ points, which is the memory space reserved for k-th window frame. More specifically, at level k, each candidate position p_f (buffered in queue $\mathcal{P}_k$) is probed together with its predecessor p_{prev} and its successor point p_{next} in the subsequence for o_i. In order to assess whether floating point p_f should be kept, the algorithm calculates the distance from its *synchronized trace* point p'_f along line segment $\overline{p_{prev}p_{next}}$. Assuming constant speed, the coordinates of point p'_f can be found by linear interpolation :

$$x'_f = x_{prev} + \frac{\tau_f - \tau_{prev}}{\tau_{next} - \tau_{prev}} \cdot (x_{next} - x_{prev}) \text{ and } y'_f = y_{prev} + \frac{\tau_f - \tau_{prev}}{\tau_{next} - \tau_{prev}} \cdot (y_{next} - y_{prev}),$$

which clearly depend on velocity between p_{prev} and p_{next}, as well as on timestamp τ_f of the candidate for exclusion. Euclidean distance $SED = L_2(p_f, p'_f)$ between original point p_f and its synchronized trace p'_f serves as an indicator of the local importance of p_f relatively to its neighbors in the trajectory sequence. Points with minimal SED values could be discarded without much error, as their eviction can hardly distort trajectory shape. In contrast, locations with maximal SED values may represent turning points, speed changes, etc. and ought to be preserved in the approximation.

Example 5. Figure 7 depicts a compressed trace after scaling by SED values against the trajectory of Figure 4. Compared to previous examples, it clearly provides the most reliable approximation, as it captures most significant motion changes, while retaining an equivalent number of samples. For the frame transition indicated by the shaded box (magnified in Figure 7b), after taking synchronized traces B', C', and D' for points B, C, and D, it is evident that $\|BB'\| > \|CC'\| > \|DD'\|$. Given that $m_k = 2$ locations per slide must be retained, point B is chosen along with articulation E. □

The main difference of this policy from velocity vectors (Section 4.4), is the criterion for promoting points across window levels. Algorithm 4 lists the adapted function *promotePoints()*. Its only divergence from Algorithm 3 is that candidates are inserted into the max-heap according to their SED values (lines 5–10). In all other aspects, both strategies work similarly, so their computational complexity is the same.

Nonetheless, approximation quality is substantially improved, as empirical results confirm. Indeed, each object position is not probed independently, but along with adjacent points in its local context. True, such locality is restricted to immediate neighbors in the sequence, like a "sliding window" of three successive locations due to online demands. Even though, this heuristic may avoid accidental elimination of critical points with presumably more weight (i.e., larger SED) along an object's course.

Algorithm 4 Scaling with *synchronous Euclidean distance*

1: **Function** *promotePoints* (level k, locations $\mathcal{P}_k$) *//Variant for SED*
2: $\mathcal{H}_k \leftarrow \emptyset$; *//Heap of pending locations sorted by SED*
3: $m_k \leftarrow \lfloor |\mathcal{P}_k| \cdot \frac{\sigma_k}{\rho_i} \rfloor$; *//Max number of points to be admitted into k-th frame*
4: $p_{artl} \leftarrow \mathcal{P}_k$.pop_back(); *//Latest location becomes articulation point*
5: **for** each buffered location $p_f = \langle o_i, x, y, t \rangle \in \mathcal{P}_k$ **do**
6: $p_{prev} \leftarrow$ the location that chronologically precedes floating point p_f in $\mathcal{P}_k$;
7: $p_{next} \leftarrow$ the location that chronologically succeeds floating point p_f in $\mathcal{P}_k$;
8: $SED \leftarrow$ distance of p_f from its synchronized trace along segment $\overline{p_{prev}p_{next}}$;
9: $\mathcal{H}_k$.insert($\langle o_i, x, y, t, SED \rangle$); *//Push point into the heap according to its SED*
10: **end for**
11: $\mathcal{P}_k$.clear(); *//Empty buffer for the next cycle*
12: $\mathcal{L}_k \leftarrow$ pop (m_k-1) items from $\mathcal{H}_k$; *//Locations to be promoted into k-th level*
13: **return** $\mathcal{L}_k \cup p_{artl}$; *//Emit samples in timestamp order*
14: **End Function**

5 Towards multi-scale windowed queries over trajectories

In this section, we discuss the potential impact of multi-scale window constructs into specifying spatiotemporal continuous queries against trajectory streams.

5.1 Enhancing expressiveness in trajectory representation

Every window instantiation provides an updated set of recent paths per monitored object. Although such traces span increasingly wider intervals in the past and may be compressed at diverse degrees per level, the window state always offers contiguous, yet lightweight paths. This is a major benefit for unobstructed evaluation of topological and spatiotemporal predicates, as those established in [8, 12] for moving objects.

We can further define auxiliary functions and predicates against such sequences so as to abstract particular aspects from the streaming spatiotemporal features. We particularly advocate for two utilities intended to repetitively return a compact series of point locations per moving object. At each iteration, taking the trajectory-filtered states as input (i.e., subsequences of object positions within the various ranges of a multi-scale window), they offer updated polyline representations, as follows:

- Function `trace()` may reconstruct distinct paths against each subwindow. As illustrated in Figure 3b for locations relayed by a single object, a separate polyline of timestamped points is returned per specified window frame. Note that such partial representations of an object's course may have common vertices; but in general, they are not expected to completely coincide, not even along overlapping time ranges among nested frames. Besides, each particular polyline gets updated at diverse frequency; the larger the scope, the longer the obtained trace, but with more sparse and less frequently refreshed vertices.
- Function `synopsis()` yields a "merged" path composed from successive multi-scale segments. As shown in Figure 3c, a single such polyline is derived per object. Essentially, it combines disjoint, yet consecutive parts from all its traces, but each at the best available resolution. Articulation points are most valuable, acting as connectors between segments from diverse scales. For any such synopsis, its most recent vertices are obtained from the finest window frame, hence frequently updated and providing a finer approximation. In contrast, its older segments come from wider frames with more aggressive reduction, so the synopsis gets progressively coarser towards the past.

Against such evolving timestamped polylines per object, it is possible to apply typical spatiotemporal functions (e.g., `speed`, `duration`) or predicates (like `WITHIN`, `INTERSECTS`, `CROSSES`, etc.). But now these operations concern reduced paths at multiple resolutions and not original trajectories, so their results may generally be meaningful, albeit approximate at varying degrees. Depending on the operation, false negatives or false positives may arise as well, usually in topological checks. For instance, a query asking for trajectories in a rectangular region may receive certain inaccurate answers, as some qualified trajectories could have been overly smoothed due to smaller scaling. If an original trajectory intersects that rectangle but all related positions are dropped, then its approximation leads to a false negative answer. Similarly, a highly smoothed path could erroneously qualify, although the original route bypasses the region. However, this is a typical side-effect of lossy approximations that cannot be entirely avoided, no matter which data reduction method is actually applied. Query evaluation methods against windowed trajectories are left for future work.

5.2 Multi-scale window declaration in spatiotemporal queries

SQL extensions for stream processing typically include some functionality for specifying several types of windows, not only in early academic prototypes like Aurora [1], STREAM [2] or TelegraphCQ [7], but also in commercial platforms [18, 28, 29]. Time-based sliding windows [22] are usually specified against a streaming source with a clause `[RANGE` ω `SLIDE` β`]` in continuous query language (CQL) [2]. When it comes to multi-level sliding windows [21], specification should better not be done using an independent clause per level. Since each frame W_k is concurrently applied over the same stream, yet covering different portions of the received items (*range* ω_k) and moving forward at its own pace (*slide* β_k), a more compact form is preferable. And as we intend to perform approximation on each trajectory, such a clause must prescribe *scale* factors σ_k per level, along with a discriminator attribute for object identifiers. Hence, the general form of declaring a n-level window should be equivalent to:

[RANGES $\omega_0, \omega_1, \ldots, \omega_{n-1}$ SLIDES $\beta_0, \beta_1, \ldots, \beta_{n-1}$ SCALES $\sigma_0, \sigma_1, \ldots, \sigma_{n-1}$ BY $< object\_id >$]

Typically for time-based windows, values for ω_k and β_k are in time units (seconds, minutes, hours, etc.), whereas scales are real numbers $\sigma_k \in (0..1)$. As in cartography, a larger σ_k close to 1 signifies that more features should be preserved in trajectory paths, whereas smaller σ_k values approaching 0 incur more lossy compressions. But here the concept of scale goes beyond its cartographic analogue; it stands for the precision of a trajectory approximation across time. So scale essentially becomes a spatiotemporal notion, as it takes into account speed and heading, but it is also dependent on inherently streaming characteristics such as frequency of updates and window sizes.

The syntax of this clause is hybrid, borrowing terms from sliding and partitioned windows over streams [2, 22]. It is specifically tailored to streaming sequences (like trajectories), where some degree of approximation is also required. Listing of window parameters starts from the lowest level and ends up to the widest frame at level $n - 1$. In a potential implementation, the query parser should check correspondence of the specified parameters per frame and validate that the entire construct abides by the constraints of Section 3. Compared to multiple local views (one per frame) eventually combined into a `SELECT` statement for a given continuous query, the aforementioned SQL-like rendition is far more concise and excels in expressiveness.

Such sliding windows enhanced with the utilities proposed in Section 5.1 may prove valuable for expressing several types of continuous spatiotemporal queries. An important distinction is that `SELECT` statements may return a multiset of answers, each referring to a different frame. For instance, a user could ask for the average speed over a multi-scale window specifying various periods of interest (e.g., past 15 minutes, one hour, etc.). To properly annotate results, we suggest a function `WSCOPE(*)` that can be used as an additional expression in `SELECT`. Calling this function would issue a window identifier per answer and thus indicate the respective scope (i.e., interval in timestamp values) over which results were actually computed.

To offer more insight, we examine two characteristic continuous queries (CQ) for a fleet management scenario. Let a stream `S` of tuples `<id, pos, ts>` that denote positional updates from vehicles as relayed into a traffic control center. For simplicity, we consider windows of $n = 3$ frames, yet each one specifies its own values for $\langle \omega_k, \beta_k, \sigma_k \rangle$. We stress that the proposed SQL-like syntax is only indicative, as no actual implementation currently supports declarative submission of such composite queries.

Example 6. CQ #1: "Indicate vehicles that have been recently on the move in the city center."

```
SELECT S.id
FROM  S [RANGES 1 MINUTE, 5 MINUTES, 20 MINUTES
         SLIDES 15 SECONDS, 1 MINUTE, 10 MINUTES
         SCALES 0.4, 0.2, 0.1 BY S.id],
      (SELECT region FROM Districts WHERE name = 'Athens center') D
WHERE duration(Intersection(synopsis(S.pos), D.region))>= '10 MINUTES';
```

This statement joins the spatiotemporal stream of positions S and a spatial relational table `Districts`, which contains the geometries of certain areas in the city. Nested subquery with alias `D` simply provides a polygon for the area of interest. In order to get the portion of each trajectory inside that region, an `Intersection` operation is employed. Its result is also a spatiotemporal trajectory and not a spatial polyline, exactly as prescribed

in [12]. Then, function `duration` can be applied over the cropped path to return the respective time interval of movement within that region. If such duration spans more than, say, 10 minutes for a given vehicle, then it qualifies for the answer at that time. Note that results get updated at the pace of the smaller frame, i.e., every 15 seconds. For a timely response, evaluation may only involve synopses and not distinctive traces per frame, in order to examine a single approximate path per object. But false or missing answers cannot be excluded, as the topological check may not be always correct for the coarser trajectory segments due to scaling. □

Example 7. CQ #2: "For each major junction in the network, continuously inspect traffic variations according to vehicle counts."

```
SELECT T.junction_id, T.win_ref, COUNT(T.vehicle_id)
FROM ( SELECT J.id AS junction_id, S.id AS vehicle_id, WSCOPE(*) AS win_ref
       FROM S [RANGES 10 MINUTES, 30 MINUTES, 1 HOUR
               SLIDES 1 MINUTE, 5 MINUTES, 15 MINUTES
               SCALES 0.5, 0.3, 0.1 BY S.id ],
            Junctions J
       WHERE CROSSES(trace(S.pos), J.region) ) T
GROUP BY T.junction_id, T.win_ref ;
```

This query intends to offer some sort of traffic analytics. Spatial table `Junctions` is supposed to list important crossroads of the network as polygonal areas (attribute `region`). Nested subquery with alias `T` identifies trajectory traces that topologically `CROSS` any of such regions. Note that such probing of traces refers to each of the three frames; if a path qualifies, then the nested subquery provides this vehicle's id and the junction it has actually traversed, as well as a window annotation `win_ref` with a call to `WSCOPE(*)`. It is important to mark results with the respective time period `win_ref`, as a distinct vehicle may have passed through a junction 15 minutes ago, so just two of its traces qualify (i.e., paths spanning 30 and 60 minutes). Finally, the outer query simply counts vehicles by junction and time period in order to emit aggregates every minute, as stipulated by the sliding pace of the bottommost frame. Up to three counts should be expected per junction (i.e., as many as the window levels) offering the potential of identifying traffic fluctuations for any junction across the network. □

6 Experimental evaluation

In this section, we empirically validate the alternative strategies for multi-scale sliding window maintenance against streaming trajectories of moving objects.

6.1 Experimental setup

Due to lack of massive, streaming real motion data, we generated a synthetic dataset simulating $N = 100\,000$ vehicles moving at diverse speeds in the road network of Athens. By calculating shortest paths between nodes chosen randomly across the network, positional samples at 200 timestamps were taken from each route. This point set is ordered by time, effectively representing concurrently evolving trajectories.

All algorithms were implemented in C++ and executed on an Intel Core 2 Duo 2.40GHz CPU running GNU/Linux with 3GB of main memory. In accordance with the data stream

processing paradigm, this implementation adheres to online computation in main memory, excluding any disk-bound techniques (e.g., spatial indexing).

Table 1: Experimentation parameters.

Parameter	Values
Object count N	**100 000**
Window levels n	**3**, 4, 5, 10
Window ranges ω (timestamps)	4,8,**10**,12,16,**20**,24,28,30,32, 36,**40**,50,64,80,90,**100**,160
Window slides β (timestamps)	**2**,**4**,**5**,6,**8**,**10** 18,**20**,25,32,50
Scale factor σ	0.1, **0.2**, 0.25, **0.3**, 0.4, **0.5**, **0.6**, 0.8, 1

Table 2: Custom 3-level window settings.

Parameter	Sets of values
Ranges (timestamps)	$\omega 1 = \{5, 10, 20\}$, $\omega 2 = \{10, 20, 40\}$, $\omega 3 = \{20, 40, 80\}$, $\omega 4 = \{10, 50, 100\}$
Slides (timestamps)	$\beta 1 = \{2, 5, 10\}$, $\beta 2 = \{5, 10, 20\}$, $\beta 3 = \{10, 10, 20\}$ $\beta 4 = \{5, 25, 50\}$
Scales	$\sigma 1 = \{0.5, 0.3, 0.2\}$, $\sigma 2 = \{0.6, 0.3, 0.2\}$, $\sigma 3 = \{0.8, 0.4, 0.2\}$, $\sigma 4 = \{1, 0.5, 0.25\}$

We ran simulations using diverse parameter settings for each experiment. Table 1 summarizes experimentation parameters and their respective values. Note that each window specification is comprised of several triplets $\langle \omega_k, \beta_k, \sigma_k \rangle$, one per level k. Many such combinations have been tested, always conforming to the window constraints as defined in Section 3. Values recurring in several experiments are shown in bold. Table 2 shows some particular sets of frame specifications especially for 3-level windows; for each set, values are listed from bottommost to topmost frame. For instance, set $\omega 1 = \{5, 10, 20\}$ specifies $\omega_0 = 5$ for the lower frame, $\omega_1 = 10$ for the middle one, and $\omega_2 = 20$ for the topmost frame. As detailed in the results, we used these sets to perform tests with one window parameter varying, but having fixed values on the other two (e.g., varying ω_k, and fixed β_k and σ_k). To verify robustness of the proposed techniques, all experiments were conducted with $\rho = 100\ 000$ tuples/timestamp, so all objects concurrently report a new location at every timestamp.

Apart from performance, we also assessed approximation quality of each scaling strategy. For each window frame, we estimated deviation between an original trajectory and its derived trace (i.e., after scaling). Deviation between two polylines can be computed from the pairwise distance of their corresponding points, i.e., between each pair of synchronized locations. Suppose that an original point p_i has been evicted due to scaling. To estimate the resulting deviation, we interpolated between the pair of samples retained immediately before and after p_i in order to obtain its synchronized point trace p'_i along the approximate route. Obviously, only such cases contribute to error; otherwise, it holds $p_i \equiv p'_i$ because original locations are strictly used in the approximate trace and no strategy introduces any points not already in the input. Assuming that M original points per object were within scope of the examined window frame, we estimated the root mean square error (RMSE) between original and approximate sequences of locations using this formula:

$$\text{RMSE} = \sqrt{\frac{1}{M} \cdot \sum_{i=1}^{M} (L_2(p_i, p'_i))^2}$$

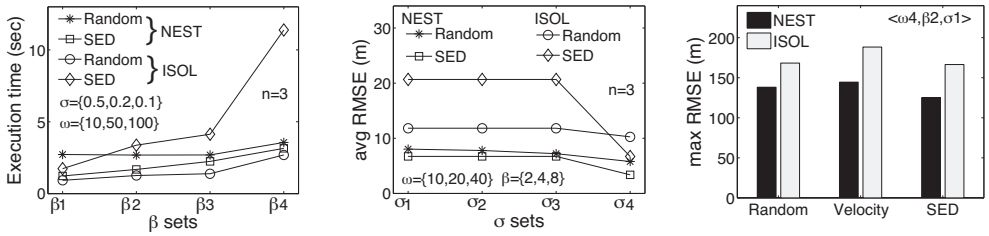
where L_2 denotes Euclidean distance between point coordinates on 2-d plane. Since coordinates in the dataset were expressed in meters, so are distances and RMSE values. This estimation took place upon sliding of a given window frame, i.e., when paths got refreshed. Typically, the measure reported in the graphs is the *average* of RMSE values against transient trajectory synopses per timestamp, one per monitored object. We complement this evaluation study with diagrams for the *maximum* RMSE observed among all

object sequences (indicating worst-case errors), as well as the *reduction ratio* of trajectory representations achieved by the proposed scaling.

6.2 Experimental results

Next, we show diagrams from most representative simulations for windowed operations with diverse parameter settings. With the exception of graphs for maximum RMSE on approximate traces, all other evaluation results are averages of actual measurements per timestamp over complete (i.e., not "half-filled") windows.

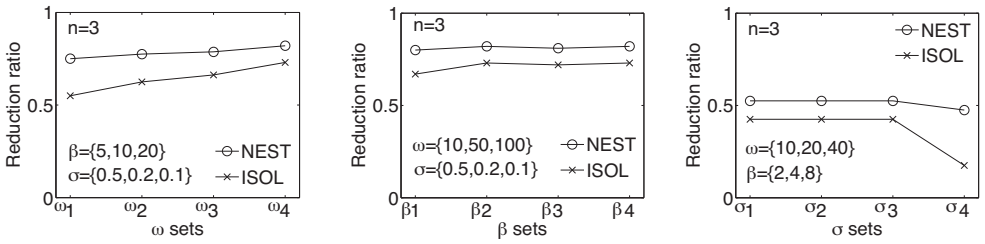
Frame maintenance mode The first set of experiments confirm that our nested framework (NEST) is advantageous over a baseline approach (ISOL) involving separately updated windows, each at a single resolution. As plotted in Figure 8a, only random sampling runs faster in isolated mode, simply because no bitmaps need be maintained or updated. But calculating velocity or SED values separately against longer sequences of points buffered over wider β intervals is absolutely wasteful, as the same vectors or distance values get computed multiple times over points that fall in several, overlapping frames. If windows slide less often (larger β), savings from nested evaluation are considerable especially for SED; results for velocity vectors are similar and not reported for brevity. Figures 8b and 8c show that accuracy of scaled trajectories benefits considerably by deriving them gradually from paths available at better resolutions. Regardless of scale settings, nested processing incurs consistently smaller deviations. Indeed, it is less likely to drop a critical point within a finer frame spanning a brief interval; when promoted upwards to coarser frames, such positions are more fit to epitomize motion than batches of undistinguished raw points utilized in isolated maintenance.



(a) (b) (c)

Figure 8: Nested vs. isolated maintenance of multi-level window frames.

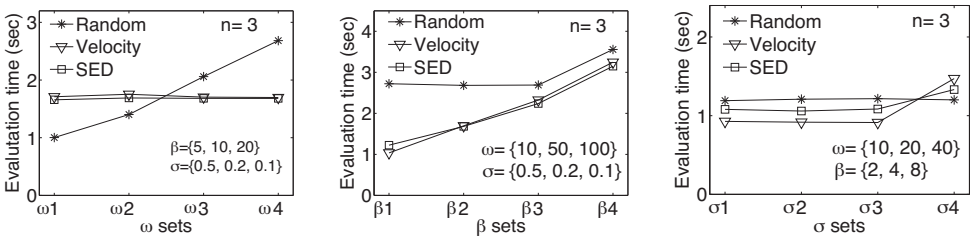
Reduction ratio One of the primary objectives of multi-scale windows over trajectories is to offer reduced, yet reliable approximations. In order to measure the accomplished reduction ratio, we compared the amount of discarded points against the originally relayed locations per trajectory. Figure 9 depicts measurements of this ratio with varying window settings from Table 2, both for isolated computation per frame (ISOL) and nested window maintenance (NEST). Clearly, reduction depends chiefly on scaling, as smaller factors dictate a frame to drop more point locations (Figures 9a and 9b). For the moderate scale settings in Table 2, about half of the total locations in each sequence are dropped with nested



(a) (b) (c)

Figure 9: Reduction ratio under nested and isolated window maintenance.

maintenance (Figure 9c). In all tests, the isolated scheme consumes more space (i.e., less reduction effect), as it keeps separate lists of locations per path; so, duplicates are possible when a location contributes to multiple paths across levels (e.g., articulation points). In contrast, with a nested evaluation of trajectory-filtered states, a chosen point is retained once not already picked up at a subordinate level. Thus, reduction is more pronounced for upper levels in the nested hierarchy. With respect to memory consumption, consider the extreme case of fixed $\sigma = 1$, i.e., no compression at all. Thanks to inherent nesting of its substates in a strairwise scheme, a n-level window W has to retain as many tuples as an autonomous sliding window equivalent to its widest frame W_{n-1}. This constitutes a clear advantage over isolated maintenance of separate windows per level, and such a gain gets definitively improved when scaling comes into play.

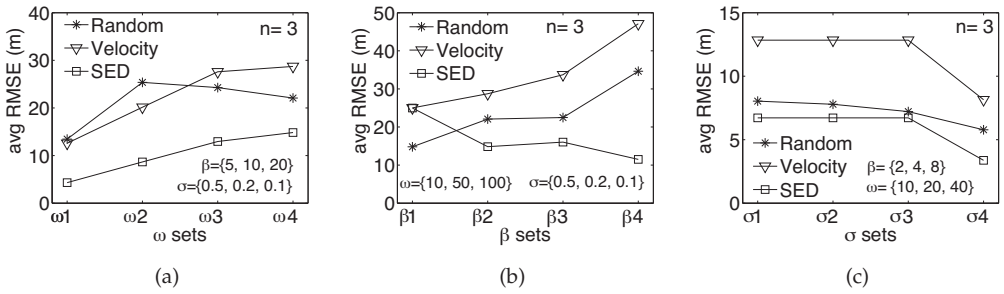


(a) (b) (c)

Figure 10: Nested window maintenance cost under diverse parameterizations.

Window parameterization For the nested processing scheme, we now examine performance of each strategy against window instantiations with $n = 3$ levels. Figure 10 depicts window maintenance cost, i.e., the time required to refresh nested substates and update multi-scaled paths for all objects. We tested diverse sets of each window property as defined in Table 2, using fixed values per level for the other two properties. As illustrated in Figure 10a for several combinations of range values, determining each window state per timestamp remains stable for strategies based on motion features (i.e., SED and velocity vectors). This is plausible, since picking new points occurs upon each slide only, and β values are fixed. As for random sampling, its cost increases with wider temporal ranges, as

this incurs additional overhead due to longer sequences and more bitmap updates. With respect to varying slides β, strategies based on SED and velocity vectors also prove superior to random sampling (Figure 10b). Naturally, these policies require more time to update trajectory paths for increasing slide steps, since they have to deal with more candidate positions at each iteration. With respect to different scale factors, but fixed ranges and slides as shown in Figure 10c, it appears that all three strategies are comparable with little fluctuations. In all experiments, the nested maintenance per timestamp for all window frames and all trajectories regularly took less than 2 seconds and never exceeded 5 seconds. Given the high rate of $\rho = 100\,000$ positions/timestamp of incoming updates, such performance certainly meets real-time expectations of the proposed scheme.



(a) (b) (c)

Figure 11: Average RMSE for nested evaluation of multi-scale windows.

Figure 11 plots average RMSE for the same window settings. One can easily verify that strategy SED offers better trajectory approximations in almost all cases. Note that error increases with the window ranges (Figure 11a), since longer paths are maintained, so more points get inevitably discarded to meet scale restrictions. Surprisingly, random sampling incurs less error than velocity vectors over larger ranges (>80 timestamps). As stated in Section 4.4, using velocity vectors from the same anchor point may lead to choosing several adjacent locations with high γ values, even if just one of them could suffice. If allocated memory is wasted to accommodate such redundant positions among a total of m_k points reserved per level k, then little space is left for other points of less γ, but perhaps more critical along the sequence. In situations like these, even choosing samples randomly may be more appropriate, as this test indicates. With respect to varying window slides, Figure 11b reveals that SED may achieve better approximation quality with more abrupt window slides (i.e., larger β per level). By looking at an increasing number of candidate points, this strategy succeeds to identify more representative positions according to their local impact on trajectory shape. Quite the reverse occurs with velocity vectors; wrongly picking consecutive samples with similar γ values gets all the more accentuated with larger slides. For varying sets of scale values (Figure 11c), velocity vectors incur again serious deviations from original trajectories. Naturally, error drops with greater σ values (i.e., less reduction per level), as scale factors are meant to dominate the degree of approximation and accuracy of compressed trajectory paths is sensitive to scaling. Overall, strategy SED seems quite suitable for most window settings, both in terms of performance and quality of derived approximations. Its average RMSE is below 10 meters in most cases, which is practically more than tolerable in such a streaming context.

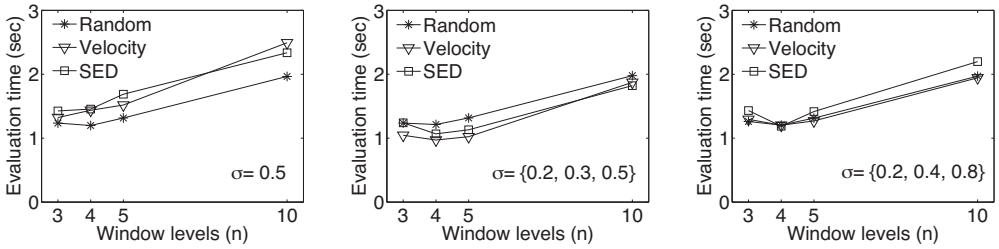


Figure 12: Nested maintenance cost for varying numbers of window levels.

Varying window levels Smooth maintenance of multiple substates is an important benefit from the proposed paradigm. This is reflected in Figure 12, indicating that it takes little time to update trajectory paths. In these tests, we examine multi-level windows with a varying number n of frames, yet setting $\omega_{n-1} = 40$ timestamps for the widest range. We also consider varying scales for each level: either the same factor per level (Figure 12a) or diverse values chosen from the indicated sets (Figures 12b and 12c), always obeying the rule that a subordinate frame deserves better resolution. But, even for up to 10 levels, each additional level incurs little overhead for all strategies, while focusing farther in the past. This happens mainly because we can exploit already computed sequences from subordinate levels.

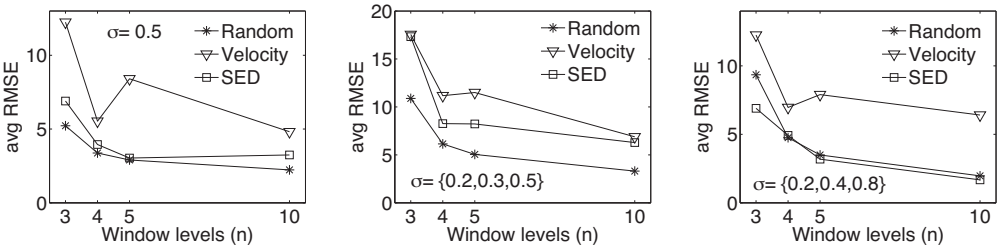


Figure 13: Average RMSE for varying numbers of window levels in nested evaluation.

Regarding approximation error, it generally drops when more periods of interest subdivide a given time horizon. Meanwhile, the derived trajectory shape is highly sensitive to scaling. All this is verified in Figure 13 for a maximal range of $\omega_{n-1} = 40$ timestamps and diverse scale factors. In essence, each additional level attempts to retain a longer path as much closer to the original trajectory. Then, the overall trajectory synopsis glues together consecutive parts of these n subsequences, each offered at the best available resolution ac-

cording to scale factors (as the example in Figure 3c). Quality is moderate when velocity vectors are employed, but quite acceptable (always less than 10 meters) for strategies based on SED and random sampling.

7 Conclusions and future work

In this paper, we set out the foundation for a windowing construct at multiple levels of detail against trajectory data rapidly streaming from numerous moving objects. This novel operator has an inherent spatiotemporal flavor, as it not only returns positional items from the recent past, but it may also offer reliable approximations at varying resolutions. To comply with memory restrictions, it implicitly works in an amnesic fashion, by retaining finer traces for the recent movement at the expense of gradually coarser segments towards the past. Thus, it provides several compressed representations of a given trajectory, each at a "scale" prescribed by user requests.

We explained the semantics of such multi-scale sliding windows and presented certain interesting properties, which enable their efficient, cross-level evaluation. Towards online processing of positional streams, we developed concrete algorithms for nested, incremental maintenance of window states. We also introduced language constructs for windowed trajectories and exemplified their expressiveness in spatiotemporal continuous queries. We conducted a comprehensive empirical study on synthetic datasets, attesting that the proposed framework can boost performance and approximation quality under a variety of window specifications. These experiments indicate that multi-scale window semantics can support reliable trajectory approximations of varying resolutions in near real-time and at reduced space overhead.

This scheme opens up perspectives for improvement and further extensions. First, we plan to investigate other trajectory compression methods that work in online fashion and could offer even better quality. In terms of query evaluation, identifying window specifications shared by multiple user queries and handling them in common, is a challenging topic. In a real monitoring platform, where thousands of requests require immediate response, such policies can provide huge optimization gains [3, 23]. Another research direction concerns quality guarantees for approximate trajectories. In order not to exceed such prespecified error margins, we may opt for policies that gracefully adjust compression degrees per frame at runtime, always conforming to memory limitations. Finally, except for trajectories, this multi-resolution framework might also be attractive for handling other types of online sequential data, e.g., meteorological readings, hydrological timeseries, financial tickers, etc.

References

[1] ABADI, D. J., CARNEY, D., ÇETINTEMEL, U., CHERNIACK, M., CONVEY, C., LEE, S., STONEBRAKER, M., TATBUL, N., AND ZDONIK, S. Aurora: a new model and architecture for data stream management. *The VLDB Journal 12*, 2 (2003), 120–139. doi:10.1007/s00778-003-0095-z.

[2] ARASU, A., BABU, S., AND WIDOM, J. The CQL continuous query language: semantic foundations and query execution. *The VLDB Journal 15*, 2 (2006), 121–142. doi:10.1007/s00778-004-0147-z.

[3] ARASU, A., AND WIDOM, J. Resource sharing in continuous sliding-window aggregates. In *Proc. 30th International Conference on Very Large Data Bases (VLDB)* (2004), VLDB Endowment, pp. 336–347. doi:10.1016/B978-012088469-8.50032-2.

[4] BERTINO, E., CAMOSSI, E., AND BERTOLOTTO, M. Multi-granular spatio-temporal object models: concepts and research directions. In *Object Databases*, M. C. Norrie and M. Grossniklaus, Eds., vol. 5936 of *Lecture Notes in Computer Science*. Springer, 2010, pp. 132–148. doi:10.1007/978-3-642-14681-7_8.

[5] BETTINI, C., DYRESON, C. E., EVANS, W. S., SNODGRASS, R. T., AND WANG, X. S. A glossary of time granularity concepts. In *Temporal databases: Research and practice*, O. Etzion, S. Jajodia, and S. Sripada, Eds., vol. 1399 of *Lecture Notes in Computer Science*. Springer, 1998, pp. 406–413. doi:10.1007/BFb0053711.

[6] CAO, H., WOLFSON, O., AND TRAJCEVSKI, G. Spatio-temporal data reduction with deterministic error bounds. *The VLDB Journal 15*, 3 (2006), 211–228. doi:10.1007/s00778-005-0163-7.

[7] CHANDRASEKARAN, S., COOPER, O., DESHPANDE, A., FRANKLIN, M., HELLERSTEIN, J., HONG, W., KRISHNAMURTHY, S., MADDEN, S., RAMAN, V., REISS, F., AND SHAH, M. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. 1st Biennial Conference on Innovative Data Systems Research (CIDR)* (2003), CIDR.

[8] EGENHOFER, M. J., AND FRANZOSA, R. D. Point-set topological spatial relations. *International Journal of Geographical Information Systems 5*, 2 (1991), 161–174. doi:10.1080/02693799108927841.

[9] FORLIZZI, L., GÜTING, R. H., NARDELLI, E., AND SCHNEIDER, M. A data model and data structures for moving objects databases. In *Proc. ACM International Conference on Management of Data (SIGMOD)* (2000), ACM Press, pp. 319–330. doi:10.1145/342009.335426.

[10] GEDIK, B., LIU, L., WU, K.-L., AND YU, P. S. Lira: Lightweight, region-aware load shedding in mobile CQ systems. In *Proc. IEEE 23rd International Conference on Data Engineering (ICDE)* (2007), IEEE, pp. 286–295. doi:10.1109/ICDE.2007.367874.

[11] GÜTING, R. H., BEHR, T., AND DÜNTGEN, C. SECONDO: A platform for moving objects database research and for publishing and integrating research implementations. *IEEE Data Engineering Bulletin 33*, 2 (2010), 56–63.

[12] GÜTING, R. H., BÖHLEN, M. H., ERWIG, M., JENSEN, C. S., LORENTZOS, N. A., SCHNEIDER, M., AND VAZIRGIANNIS, M. A foundation for representing and querying moving objects. *ACM Transactions on Database Systems 25*, 1 (2000), 1–42. doi:10.1145/352958.352963.

[13] HU, H., XU, J., AND LEE, D. L. A generic framework for monitoring continuous spatial queries over moving objects. In *Proc. 24th ACM International Conference on Management of Data (SIGMOD)* (2005), ACM Press, pp. 479–490. doi:10.1145/1066157.1066212.

[14] JAIN, N., MISHRA, S., SRINIVASAN, A., GEHRKE, J., WIDOM, J., BALAKRISHNAN, H., ÇETINTEMEL, U., CHERNIACK, M., TIBBETTS, R., AND ZDONIK, S. Towards a streaming SQL standard. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1379–1390.

[15] LANGE, R., DÜRR, F., AND ROTHERMEL, K. Efficient real-time trajectory tracking. *The VLDB Journal 20*, 5 (2011), 671–694. doi:10.1007/s00778-011-0237-7.

[16] MERATNIA, N., AND DE BY, R. A. Spatiotemporal compression techniques for moving point objects. In *Advances in Database Technology (EDBT 2004)*, E. Bertino, S. Christodoulakis, D. Plexousakis, V. Christophides, M. Koubarakis, K. Böhm, and E. Ferrari, Eds., vol. 2992 of *Lecture Notes in Computer Science*. Springer, 2004, pp. 765–782. doi:10.1007/978-3-540-24741-8_44.

[17] MOURATIDIS, K., PAPADIAS, D., AND HADJIELEFTHERIOU, M. Conceptual partitioning: an efficient method for continuous nearest neighbor monitoring. In *Proc. 24th ACM International Conference on Management of Data (SIGMOD)* (2005), ACM Press, pp. 634–645. doi:10.1145/1066157.1066230.

[18] ORACLE, INC. Complex event processing CQL language reference. http://docs.oracle.com/cd/E16764_01/doc.1111/e12048/intro.htm, 2009. Last accessed on 15/09/2013.

[19] PARENT, C., SPACCAPIETRA, S., AND ZIMÁNYI, E. The MurMur project: Modeling and querying multi-representation spatio-temporal databases. *Information Systems 31*, 8 (2006), 733–769. doi:10.1016/j.is.2005.01.004.

[20] PATROUMPAS, K. Multi-scale windowing over trajectory streams. In *Advances in Conceptual Modeling*, S. Castano, P. Vassiliadis, L. V. Lakshmanan, and M. L. Lee, Eds., vol. 7518 of *Lecture Notes in Computer Science*. Springer, 2012, pp. 294–303. doi:10.1007/978-3-642-33999-8_35.

[21] PATROUMPAS, K., AND SELLIS, T. Multi-granular time-based sliding windows over data streams. In *Proc. 17th International Symposium on Temporal Representation and Reasoning (TIME)* (2010), IEEE, pp. 146–153. doi:10.1109/TIME.2010.14.

[22] PATROUMPAS, K., AND SELLIS, T. Maintaining consistent results of continuous queries under diverse window specifications. *Information Systems 36*, 1 (2011), 42–61. doi:10.1016/j.is.2010.02.001.

[23] PATROUMPAS, K., AND SELLIS, T. Subsuming multiple sliding windows for shared stream computation. In *Advances in Databases and Information Systems*, J. Eder, M. Bielikova, and A. M. Tjoa, Eds., vol. 6909 of *Lecture Notes in Computer Science*. Springer, 2011, pp. 56–69. doi:10.1007/978-3-642-23737-9_5.

[24] POTAMIAS, M., PATROUMPAS, K., AND SELLIS, T. Online amnesic summarization of streaming locations. In *Advances in Spatial and Temporal Databases*, D. Papadias, D. Zhang, and G. Kollios, Eds., vol. 4605 of *Lecture Notes in Computer Science*. Springer, 2007, pp. 148–166. doi:10.1007/978-3-540-73540-3_9.

[25] RICHTER, K.-F., SCHMID, F., AND LAUBE, P. Semantic trajectory compression: Representing urban movement in a nutshell. *Journal of Spatial Information Science*, 4 (2013), 3–30. doi:10.5311/JOSIS.2012.4.62.

[26] RIGAUX, P., SCHOLL, M., AND VOISARD, A. *Spatial databases: with application to GIS*. Morgan Kaufmann, 2001.

[27] STONEBRAKER, M., ÇETINTEMEL, U., AND ZDONIK, S. The 8 requirements of real-time stream processing. *ACM SIGMOD Record 34*, 4 (2005), 42–47. doi:10.1145/1107499.1107504.

[28] STREAMBASE SYSTEMS, INC. StreamSQL guide. http://docs.streambase.com/sb71/index.jsp?topic=/com.streambase.sb.ide.help/data/html/streamsql/index.html, 2012. Last accessed on 15/09/2013.

[29] SYBASE, INC. Complex event processing using windows. http://infocenter.sybase.com/help/index.jsp?topic=/com.sybase.infocenter.dc01029.0400/doc/html/tbi1263964912506.html, 2010. Last accessed on 15/09/2013.

[30] WOLFSON, O., SISTLA, A. P., CHAMBERLAIN, S., AND YESHA, Y. Updating and querying databases that track mobile units. *Distributed and Parallel Databases 7*, 3 (1999), 257–287. doi:10.1023/A:1008782710752.