Trajectory-aware Load Adaption for Continuous Traffic Analytics

Kostas Patroumpas
IMSI, Athena Research Center, Greece
kpatro@imis.athena-innovation.gr

Serafeim Papadias
Technische Universität Berlin, Germany
s.papadias@tu-berlin.de

ABSTRACT

We introduce a framework for online monitoring of moving objects, which takes into account their evolving trajectories and copes smoothly with fluctuating demands of multiple continuous queries for limited system resources. This centralized scheme accepts streaming positional updates from numerous objects, but it only examines recent trajectory segments with expectedly higher utility in query evaluation, shedding the rest as immaterial. We focus on adaptive processing under extreme load conditions, opting to retain salient trajectory segments and possibly sacrifice smaller, frequently observed paths in favor of longer, distinctive routes. We propose heuristics for incremental, yet approximate, query evaluation in order to provide up-to-date traffic analytics using windows that abstract particular regions and time intervals of interest. Finally, we conduct a comprehensive experimental study to validate our approach, demonstrating its benefits in result accuracy and efficiency for almost real-time response to trajectory-based aggregates.

CCS CONCEPTS

• **Information systems → Data streaming**; **Spatial-temporal systems**; **Location based services**.

KEYWORDS

geostreaming, adaption, load shedding, trajectory, traffic analytics

ACM Reference Format:
Kostas Patroumpas and Serafeim Papadias. 2019. Trajectory-aware Load Adaption for Continuous Traffic Analytics. In *16th International Symposium on Spatial and Temporal Databases (SSTD '19), August 19–21, 2019, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3340964.3340967

1 INTRODUCTION

Nowadays, a wide spectrum of monitoring applications collect, exchange, process, and analyze large volumes of *geostreaming* data. From tourist guides to logistics, in geosocial networking, online advertising or wildlife preservation, *Location-based Services* (LBS) have to manage positional updates from numerous moving objects (people, animals, vehicles, merchandise, etc.). Typically, continuous range [3] or k-nearest neighbor queries [11] focus on spatial relationships regarding the *current* location of moving objects.

Yet, evolving *trajectories* of such moving objects also offer precious information by dynamically tracking their updated motion paths across time. As numerous objects relay their locations frequently (e.g., every few seconds), massive positional data is being accumulated in a streaming fashion. But keeping historical, ever-growing trajectories of all objects in main memory so as to quickly answer related queries is hardly affordable. User requests mostly examine their recent portion via *sliding windows* [14], e.g., trajectory segments over the past hour. The bulk of trajectory information may still be enormous with scalable numbers of objects; even worse, any attempt to process it in its entirety may be overwhelming. Not only must existing trajectories be constantly updated with fresh positions, but continuous queries should offer timely response before the next batch of updates arrives. Apart from possibly obsolete or inconsistent results, a series of such delays may have a chain effect that could rapidly exhaust limited system resources (CPU, memory), so performance may steadily degrade and ultimately collapse.

In this paper, we introduce a self-regulated, adaptive mechanism for managing huge volumes of evolving trajectory data. A centralized server maintains and utilizes only *selected trajectories* when evaluating continuous spatiotemporal queries so as to avoid exceeding its system capacity. Our approach is inspired by *load shedding* techniques [1, 7, 12, 16, 19, 21] proposed for effective, yet approximate processing over data streams, by controlling how much input is admitted to the system according to its estimated Quality of Service (*QoS*). Their key idea is to make the most out of available resources, sacrificing certain portions of the incoming tuples without ever processing them, in order to provide answers as timely and accurate as possible to continuous queries.

Our focus is particularly on *continuous queries* over dynamically updated trajectory segments that can provide *traffic analytics* in real time. Employing sliding windows, typical online aggregates like *"Report average speed along each road over the last 15 minutes"* or *"Estimate travel time per road segment over the past half hour"* are valuable for road surveillance, real-time shortest path computation, etc. Of course, accurate vehicle counts per segment are impossible to support even without shedding, since not all actually circulating vehicles may be monitored. Still, such geostreaming samples can rate the *congestion level* (e.g., low, heavy, jam) of roads in comparison with historical records, assess traffic intensity in areas (e.g., city center), etc. Such traffic analytics cover particular *regions of interest* (e.g., roads, city zones), each one stipulating its own functional, filter, and window specifications. Our methodology could also handle other types of network-restricted movement: aircrafts flying along air corridors, packets transmitted via routers in computer networks, etc., provided that similar analytics are requested.

We stress that our approach specifically concerns *trajectory-based* query evaluation, and not simply streaming locations as in previous techniques on spatially-aware shedding [3, 4, 9, 10, 13]. We cannot randomly eliminate a portion from the incoming batch

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SSTD '19, August 19–21, 2019, Vienna, Austria
© 2019 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6280-1/19/08…$15.00
https://doi.org/10.1145/3340964.3340967

of positions at any given time, as this will break the sequence in the maintained trajectories. Instead, we opt to discard trajectory segments less useful in evaluating traffic analytics so as to reduce processing cost and keep in pace with the arrival of fresh positions under intensive load situations. For example, if a vehicle is moving on a road where many other trajectories are maintained, then its trace may be dismissed from the server to make savings in processing cost, since the rest can provide enough information on average speed or travel time. In tandem, aborting trajectories should not put query evaluation at risk, i.e., no query must be left unanswered because relevant data is dropped. Had data reduction been applied close to the sources (i.e., objects reporting their positions less frequently), it would mostly affect data quality (i.e., trajectory simplification) and save in bandwidth rather than improve response latency and scalability *on the server* side, as we aim in this work.

Intuitively, our method resembles a throttling valve that regulates the flow of geostreaming data into the system. To the best of our knowledge, this is the first work on load regulation and adaptive processing over *streaming trajectories* of moving objects. Our contribution can be summarized as follows:

- We introduce a self-regulated, lightweight scheme for effective treatment of evolving trajectories, using heuristics to evict those of presumably minor utility in query answering.
- We propose a simple, yet effective model for predicting load conditions from dynamically updated QoS statistics.
- We employ succinct synopses to keep digested aggregates about trajectories even if they have been temporarily shed.
- We empirically demonstrate that such a mechanism can provide approximate, yet reliable response in real time to continuous aggregates over streaming trajectories.

The remainder of this paper proceeds as follows. Section 2 surveys related work. Section 3 presents the system model. Section 4 introduces an adaptive mechanism for keeping load close to system capacity when calculating trajectory-based aggregates. Section 5 reports performance and quality assessment results from a comprehensive empirical study. Section 6 concludes the paper.

2 RELATED WORK

Various load shedding techniques have been employed in streaming and geostreaming applications. In both cases, the critical questions are *when*, *where* and *how much* load to shed if latency deteriorates because of high arrival stream rates.

Among *stream processing engines* (SPE), pioneering shedding policies for Aurora [19] suggested a precomputed set (*Load Shedding Road Map*) of possible query plans enhanced with data dropping operators. In case of excessive load at runtime, it finds the best plan such that the system utility loss due to the shed input is minimized. Although random dropping can be fairly successful, taking tuple semantics into account can provide even better results. In STREAM [1], random samplers were carefully inserted along query plans to minimize the maximum relative error at output, employing probabilistic bounds over windows of tuples received by aggregates. Since arbitrary tuple-based load shedding can cause inconsistency in windowed aggregates, the approach in [20] suggested sacrificing entire windows of tuples while keeping other windows intact. Based on rate-monotonic scheduling properties, shedding in [7] attempts to minimize the total error in query results caused by data reduction. Control theory techniques like [5, 21] utilize closed-loop control with feedback to provide better result quality at less delay compared with previous shedding policies. Besides, load shedding was also applied to aggregates and mining functions in [12]. DILoS [15] combines query scheduling and load shedding policies when classes of queries have differing levels of priority. Sketches were proposed in [17] to collect processing times per tuple and update a cost model for shedding in query operators. Data triage [16] summarizes excessive data into *synopses* instead of dropping it, so as to improve query results. In a similar manner, we maintain synopses of inactive trajectories (i.e., those having their positions shed) to improve the quality of approximate traffic analytics.

Load shedding has been also applied in a *geostreaming* context. Concerning mobile continuous queries, LIRA [3] suggests that the monitored area be divided into regions of roughly the same density like a quadtree, and collecting statistics per region so that load shedders are placed in each region with variant shedding percentages. Continuing this line of work, MobiQual [4] employs shedding of both data updates and queries according to QoS specifications. ClusterSheddy [13] performs shedding with data mining techniques. Its key idea is to consider clusters of objects which move in a similar way, and then ignore updates near the centroids of clusters. SOLE [10] utilizes the notion of significant objects, i.e., those participating in many windows and are kept intact. In contrast, positions are dropped from objects that are not deemed relevant in answering many queries. This is analogous to our concept of active trajectories, which marks cohesive traces (not single locations as in [10]) most useful in computing traffic analytics. Specifically for spatial queries in microblogs, the platform in [9] is able to adjust its load according to location distribution and dynamically changing query parameters. To the best of our knowledge, there is no prior work on load shedding from streaming trajectories of moving objects.

Some SPE applications specifically concern *traffic analytics*. InfoSphere Streams [2] aims at scalability and offers various traffic aggregates. GeoInsight [6] supports continuous spatiotemporal queries and ad hoc analytics for traffic prediction. However, these platforms completely lack any load shedding capabilities.

3 SYSTEM MODEL

Next, we present the application setting, we rigorously formulate the problem, and outline our processing scheme.

3.1 Application Setting

Without loss of generality, we assume a LBS application that can monitor up to N objects (e.g., vehicles) equipped with geopositioning devices. Objects are moving across the edges of a fixed network, e.g., roads in a city or a country. This network is represented as a graph (V, E) with directed edges in E (e.g., road segments with a specific traffic flow) connecting pairs of vertices from V. Positional updates are relayed as tuples $\langle o, p, \tau \rangle$. Each object o reports its actual location p frequently enough (say, every few seconds), but not always at a fixed rate; reporting frequency may vary among objects. Thus, the spatiotemporal motion of each object o_i is represented with its *trajectory* $T_i = \{\langle o_i, p_1, \tau_1 \rangle, \langle o_i, p_2, \tau_2 \rangle, \ldots, \langle o_i, p_c, \tau_c \rangle\}$. Clearly, this sequence of positions is ordered by timestamp up to current time τ_c, like the vehicle traces in Fig. 1.

Upon arrival to a centralized server, these locations constitute a single input geostream. We assume no stream imperfections, i.e., no delayed, disordered or missing messages, hence all streaming tuples are considered properly timestamped at the time of their admission. This fresh data is temporarily buffered and then given to processing periodically every δ time units (e.g., every 5 seconds). Each such *execution cycle* is marked by current timestamp τ_c. At cycle τ_c, the server must recognize the trajectory segments traversing each network edge in order to update traffic analytics (e.g., average speed) within deadline δ before the next cycle starts at $\tau_c' = \tau_c + \delta$.

A set of M continuous queries need to make computations over *trajectories* and not on individual locations. Each query is registered as $q : \langle f, R, \omega, \beta, \tau_0 \rangle$ at time τ_0. Unless explicitly suspended, query q remains valid indefinitely and must return fresh results according to its specifications. Typically for sliding time-based windows [14], each query prescribes its own temporal *range* ω of the window that *slides* every β time units. Not all queries get evaluated concurrently due to possibly varying β slides, whereas they may concern different ω ranges. Each query q specifies a function f for computing a *traffic aggregate* (like average speed, travel time, etc.) over evolving trajectory segments collected within its range ω.

In addition, each query q declares a *spatial region* R of interest; e.g., this may be a buffer polygon around the edge(s) in E monitored by query q. Qualifying trajectories report their segments located within region R anytime during the past ω time units. For instance, in a road surveillance application, R may represent the carriageway of a road; depending on its length and importance to traffic, R may only cover particular edges or lanes in a specific direction of traffic flow. In the example shown in Fig. 1, a traffic controller may register query q_1 to estimate average speed across the westbound carriageway between the red dashed lines. Results will be computed from trajectory segments (in green solid lines) collected from vehicles o_2, o_4, and o_5 that traversed region R during the past $\omega = 30$ minutes; estimates are refreshed every $\beta = 5$ minutes. Without loss of generality, we consider every R is wide enough to account for GPS discrepancies, so that each incoming position gets indisputably assigned to a single region. Alternatively, a state-of-the-art *map-matching* algorithm [8] may be used to incrementally match positions with underlying network edges. However, we stress that this is an orthogonal problem and does not affect our proposed methodology for trajectory-aware load adaption.

3.2 Problem Formulation

Suppose a centralized server that monitors N objects moving on a network and evaluates a workload of M continuous queries that compute traffic aggregates. At current execution cycle τ_c, let u_i denote the cost (in CPU cycles) for updating trajectory of object o_i with its recently arrived positions and also identify spatial regions R_j of interest that o_i has traversed. Suppose that c_j is the CPU cost for evaluating a given query q_j at cycle τ_c, i.e., examining all trajectories evolving along its region R_j during window range ω_j and calculating a requested aggregate (e.g., average speed). Thus, the *cumulative actual load* L for updating all trajectories (L_u) and calculating response to all queries (L_c) at execution cycle τ_c is:

$$L = L_u + L_c = \sum_{i=1}^{N} u_i + \sum_{j=1}^{M} c_j \tag{1}$$

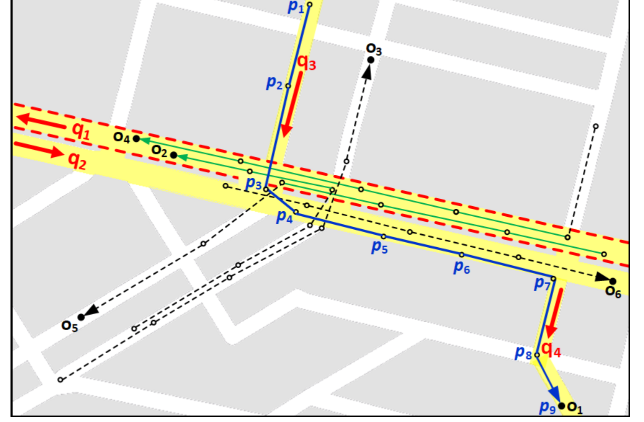


Figure 1: Trajectories of vehicles moving on a road network

But typically, a centralized server has limited processing capabilities. We abstract this limitation by allocating to the monitoring process a preset *capacity* C, i.e., the maximum amount of CPU cycles in which results must be returned at the current execution cycle. Thus, at each execution cycle τ_c,

$$L \leq C \tag{2}$$

must hold so that allocated resources are not exhausted. Ideally, trajectory updates and query evaluation must be concluded within strict period δ. But occasionally, a sudden surge in the amount of incoming locations (e.g., when most of N objects are on the move and report too often) cannot be processed on time for all pending queries. So, it could take more than δ units to finish processing at current cycle τ_c. This may cause incoming tuples queueing up at the admission point, further delays in successive execution cyles, and eventually the system will become unbalanced, failing to provide traffic aggregates from the most fresh object traces on time.

To overcome this possible burden in processing, we suggest choosing a certain amount s of trajectories to shed from the system. In effect, this choice classifies trajectories into:

- *active* trajectories that will be updated and subsequently used in query evaluation, as opposed to
- *inactive* trajectories that get temporarily suppressed (i.e., not updated by fresh positions) and ignored in evaluation.

Note that this distinction does not depend on whether an object is stationary or not, but on whether its trajectory will be among the set $\mathcal{T}$ of those available (*"active"*) for probing in query processing. Rendering some trajectories inactive will certainly make savings in execution cost, so that actual load L may be kept below capacity according to (2). However, shedding trajectories randomly may have serious impact in the quality of answers. Instead, we should rather retain trajectories with greater utilization to queries in a *semantic* fashion. Active trajectories should be those that assist most in timely answering as many queries as possible. Intuitively, longer and more fresh traces traversing many edges of the graph should be prioritized. In addition, queries involving a very low number $\leq \xi$ trajectories should also take precedence, as they cannot afford to lose any of their current sample traces; otherwise, results may be inaccurate or even impossible to compute. So, a trajectory T_i required by any such query should be preserved.

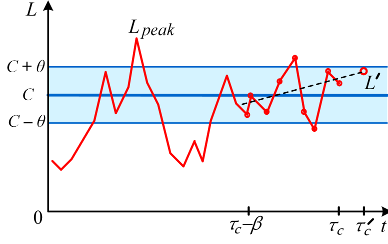


Figure 2: Load fluctuations w.r.t. system capacity C

Obviously, the extra overhead L_a incurred by this readjustment should satisfy $L_a \ll L_u + L_c$ in order not to overspend precious resources on adaption. Still, fluctuations of observed load $L = L_u + L_c + L_a$ around capacity C are inevitable. In addition, it may occasionally happen that too many trajectories are shed, and L drops well below allocated capacity C. Thus, in order to avoid triggering this adaption process in situations where L has only a small deviation from C, we set a *tolerance* θ expressed as a small percentage % of C. As illustrated with the horizontal cyan rectangle in Fig. 2, parameter θ defines a *saturation band* $(C - \theta, C + \theta)$, which sets acceptable levels for actual load L close to capacity C, so that adaption may not be unduly activated. Most importantly, adaption must be decided *proactively* to prevent overloading or underloading, i.e., before the next cycle τ_c' actually starts. This decision can only be based on *expected load* L' at τ_c', which may be estimated according to recent load indications that reflect the Quality of Service (QoS), as illustrated in Fig. 2. Thus, we prescribe that load adaption must be triggered when

$$|L' - C| \geq \theta. \tag{3}$$

Our goal is to get fair estimates for expected load L' at next cycle τ_c' that can lead to suppressing or supplementing a suitable amount $\pm s$ of trajectories, such that *actual load* L at τ_c' ideally is

$$C - \theta < L < C + \theta. \tag{4}$$

Due to volatility in motion patterns and variety in query specifications, overloads or underloads may still happen occasionally (a few spikes just above and below the saturation band as in Fig. 2). The system must also check for unnecessarily inactive trajectories. Yet, when not many objects are on the move (e.g., off-peak hours), it is normal that the system may be underloaded as condition (2) prescribes, so adaption is off and no trajectories are shed.

3.3 System Overview

Figure 3 illustrates the processing flow of the proposed system, which accepts streaming positions from N objects moving on a network graph (V, E) and must provide response to M continuous, trajectory-based aggregate queries. In order to adapt in abrupt load deviations, this system consists of the following basic modules:

- A *Spatial Index* covers the entire monitored area so as to quickly match each fresh position to a region of interest (query). All such regions are indexed in a regular *grid* G.
- *Trajectory Manager* accepts region-aligned positions from the monitored objects and updates their trajectories accordingly. It also maintains digested information on their motion, i.e., lightweight *synopses* of the successive regions they recently traversed, and collects useful *statistics* on the amount of trajectories observed per region.
- *Query Processor* evaluates all active queries at current cycle τ_c by making use of *partial aggregates* eagerly computed on-the-fly by the Trajectory Manager while updating object traces. The resulting traffic analytics are returned to users for enabling shortest path calculations, notification alerts, map visualization, archiving, etc.
- *Load Monitor* examines trends in the QoS time series $\mathcal{L}$ of actual loads over recent execution cycles and anticipates overload or underload situations at next cycle τ_c'.
- *Load Adapter* regulates the amount of objects that will be allowed to update their trajectories at cycle τ_c'. This decision is based on expected load L' as estimated by the Load Monitor with respect to system capacity C, and dictates whether a certain portion of trajectories must be included or excluded from processing at τ_c'.

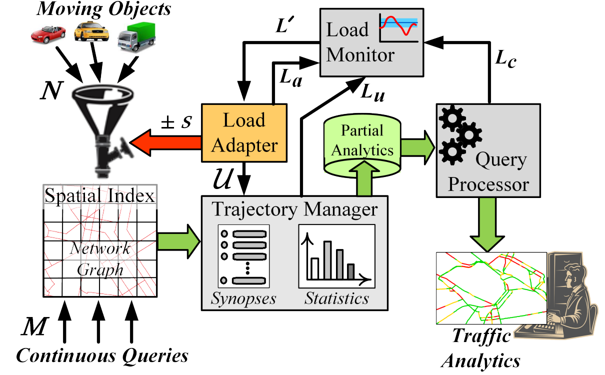


Figure 3: System Architecture

4 ADAPTIVE MECHANISM

At every cycle τ_c, execution proceeds in successive stages, each employing one or more of the aforementioned modules:

i) *Trajectory update* is carried out by the Trajectory Manager assisted by the underlying Spatial Index.
ii) *Query evaluation* is the job of Query Processor thanks to partial analytics obtained from the Trajectory Manager.
iii) *Load adaption* is performed through the Load Monitor and Load Adapter modules.

In the sequel, we analyze each stage in detail.

4.1 Trajectory Update

For each incoming position p from a monitored object o, the system must first identify the edge in the network graph where o currently moves along. In our case, this corresponds to a query q with spatial region $q.R$ responsible to monitor traffic over (or including) this edge. Even in case that this position is very close to a graph vertex (e.g., crossroads), the correct edge is the one most similar to the orientation of *instantaneous velocity* $\vec{v}$ of object o, i.e., the vector connecting its previous p_{prev} to current location p.

For enabling fast identification of candidate regions (those close to p), the entire area of interest is subdivided by means of a uniform

grid partitioning G. Indeed, grids allow fast retrieval, especially for points or frequently changing locations [18]. Grid granularity g controls the number of cells per axis, subdividing the entire area into $g \times g$ equi-sized square cells. In a preprocessing step, each region R is trivially assigned to those cells it overlaps with. At runtime, position p is hashed against the grid and the cell containing p is found. After searching over regions indexed in that cell, p is associated with a single query q whose region $q.R$ contains p and its direction of traffic flow is most similar to that of velocity $\vec{v}$.

Velocity $\vec{v}$ and query identifier q are appended as extra attributes to each fresh stream tuple, which becomes $\langle o, p, \vec{v}, q, \tau \rangle$. Subsequently, this information is used to update the status of its trajectory, which of course concerns *active* trajectories only. However, for each object (either active or inactive), Trajectory Manager temporarily holds its last reported position p_{prev}, so that its velocity $\vec{v}$ can be calculated, as well as any transition into another region.

In addition, we maintain a *synopsis* for every evolving trajectory, either active or inactive. Essentially, this synopsis keeps account of the spatial regions that an object o has traversed. In fact, an item in such a synopsis identifies the respective query q_j that monitors such a region, as well as the specific time interval $[t^{in}, t^{out})$ that o spent along $q_j.R$, without keeping any position coordinates. For instance, in the example setting in Fig. 1, vehicle o_1 has recently reported positions $p_1, \ldots, p_9$ at respective timestamps $\tau_1, \ldots, \tau_9$. So, its trajectory T_1 (in blue color) passed from three roads monitored respectively by queries q_3, q_2, and q_4. Consequently, its synopsis is $S_1 = \{\langle q_4, [\tau_7, \tau_9) \rangle, \langle q_2, [\tau_3, \tau_7) \rangle, \langle q_3, [\tau_1, \tau_3) \rangle\}$ with its items listed in *reverse chronological order* from most recent traversal to the oldest. Only the first item in the synopsis is subject to updates, as the upper bound of its time interval may change with a fresh position. Once this object enters a region monitored by another query, a new item is appended to the front of the synopsis. For clarity, here we assumed that an object emits its position once it moves into another region. In practice, this is hardly realistic, given the asynchronous fashion in reporting positional updates. Anyway, time indications for such transitions can be fairly approximated via linear interpolation on the server side. As time goes by, obsolete items in synopses may be discarded depending on window specifications of the respective queries. For instance, if query q_2 specifies a window range ω_2 and currently $\tau_c - \omega_2 > \tau_7$, then the respective item from synopsis S_1 may be purged. Overall, the series of such items per object summarize its timespan over the successive query regions it has passed. Approximate estimates from synopses can be readily used to enrich query results (travel time, average speed, etc.) in case that original locations have been dropped.

Finally, at each execution cycle τ_c, auxiliary *statistics* are collected on the number of trajectories traversing spatial region $q_j.R$ of every query $q_j, j = 1, \ldots, M$ registered in the system. As explained in Section 4.3.3, such counts update a histogram $\mathcal{H}$, which can assist in avoiding inactivation of trajectories; otherwise, some queries may be driven to starvation.

4.2 Query Evaluation

Since Trajectory Manager maintains enhanced positional tuples $\langle o, p, \vec{v}, q, \tau \rangle$, it can readily estimate some elementary traffic analytics concerning sucessive trajectory segments per object. In general, once object o exits from region $q_j.R$ monitored by query q_j, it enters into $q_k.R$ monitored by another query q_k. Then, at the trajectory segment retained for o during its traversal from region $q_j.R$, we can assign two pointers: one pointing to the first location of o observed in $q_j.R$, and another one to its last reported location just before exiting from $q_j.R$. Therefore, if object o_i has recently traversed m regions (i.e., m queries), then its trajectory T_i will point out to $2m$ such locations. We adopt a lazy evaluation policy, in which pointers of T_i are adjusted during the trajectory update stage upon transition into another spatial region.

Thanks to these pointers, a *catalogue* $\mathcal{K}$ can be also maintained in the Query Processor, essentially listing for each query q_j the trajectories recently passing through its region $q_j.R$ at successive time intervals. This data structure is updated at each execution cycle, pointing to the most fresh traces of objects. When evaluating a query q_j, catalogue $\mathcal{K}$ promptly indicates all involved objects. For each such object o_i, the aforementioned pointers can access its trajectory T_i and collect precomputed simple aggregates (average speed, travel time, etc.) concerning region $q_j.R$. Thus, answering to query q_j reduces to a trivial aggregation (e.g., average or maximum) over those singleton values collected from all active trajectories along its region $q_j.R$ during its window range $q_j.\omega$.

4.3 Load Adaption

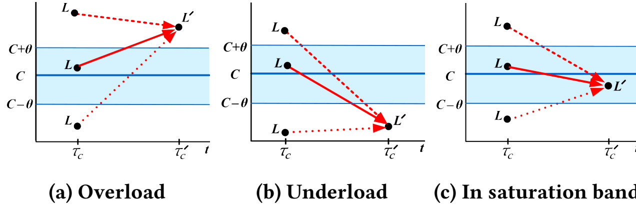
We assume that the monitoring process is allocated with a fixed *capacity* C (in CPU cycles) per execution cycle τ_c. However, completing its task within a period δ is often impossible, given the large number of moving sources, the fluctuating (and occasionally escalating) arrival rate of fresh positions, as well as the demands of continuous queries awaiting for incremental response. At each cycle τ_c, the Load Monitor (Fig. 3) receives QoS measurements concerning: cost L_u for updating trajectories; L_c for evaluating queries; and any overhead L_a charged in the previous cycle for adapting the load at the current one τ_c. Given the scarcity in system resources, cumulative *actual load* L may exceed capacity C, sometimes by far, if all incoming data is examined exhaustively. Then, the Load Adapter must regulate the input, so that the system can still provide reliable answers to queries without risk of thrashing.

4.3.1 *Alternative Policies.*

Load shedding applies to situations where multiple continuous queries vie each other for limited system resources in order to get low-latency response [19]. Shedding does not alter query specifications but affects the amount of data given to processing. In our setting, given a single stream of positional updates, the Load Adapter should be placed close to the admission point so as to regulate the data flow as early as possible (Fig. 3).

Once the system needs adjustment, several alternative policies may be considered with regard to geostreaming data:

i) *Position-level adaption*: The system randomly chooses to keep each incoming position if it exceeds a probability, regardless of the referenced object. So, a given percentage of locations is retained to adjust the load. But such random elimination can yield irregularly sampled trajectories, possibly deviating a lot from original traces.
ii) *Window-level adaption*: As in [20], entire window states are purged, so trajectories get not updated at the respective intervals and not probed by any query. Apart from disrupting



(a) Overload (b) Underload (c) In saturation band

Figure 4: Cases for triggering load adaption at next cycle τ_c'

the sequence in trajectories, this policy also means that some queries are not evaluated at all at some cycles.

iii) *Trajectory-level adaption*: Certain trajectories are selected and temporarily dismissed. This is the most preferable choice in our setting, as it aims to preserve integrity of retained (*active*) trajectories, instead of sparse locations or missing entire subsequences as in the previous two policies.

So, we suggest retaining a suitable portion of active trajectories at each execution cycle, enough to answer all queries, ideally within strict deadline δ. But picking *at random* which trajectories to maintain would yield highly skewed query results. Not all of the selected trajectories may correspond to the spatial regions actually specified by queries, hence some answers may be based on insufficient data; even worse, some queries could be left unanswered. Consequently, we do not consider this naïve random policy. In contrast, similarly to other approaches over data streams [4, 9, 15, 19], we opt for *semantic load adaption*. Crucially, our method is *trajectory-aware* as it keeps object trajectories according to their (presumably superior) utility in query answering. At next cycle τ_c', only active trajectories get updated and take part in evaluation. Each incoming location p that refers to an inactive trajectory T_i may be dropped without consideration, except for the case p indicates that this object has just entered into another spatial region. Marking such transitions is important in maintaining the synopsis for each object (Section 4.1). Indeed, once trajectory T_i becomes active again and used in query evaluation, its recent motion must have been recorded without gaps, even in the condensed form of its synopsis S_i.

4.3.2 Triggering Load Adaption. At runtime, the Load Monitor can make a fair guess on L' at next cycle τ_c'. We employ a *linear regression* model over actual loads $L(t)$ at recent execution cycles $t \in (\tau_c - \beta_{max}, \tau_c]$ to estimate a best-fitting trendline:

$$L'(t) = \sigma \cdot t + \psi \tag{5}$$

where σ signifies its slope and ψ the intercept. In statistics, the *least square error* linear fit can be obtained with coefficients

$$\sigma = \frac{\sum(t - \bar{t})(L(t) - \bar{L}(t))}{\sum(t - \bar{t})^2} \quad \text{and} \quad \psi = \bar{L}(t) - \sigma \cdot \bar{t} \tag{6}$$

where $\bar{t}$ is the average (a timestamp value) over recent cycles in an interval equal to the greater window slide β_{max} among all M queries. This choice intends to take into account loads potentially fluctuating due to the burden of evaluating a different mix of queries at each cycle because of their varying slides β. Then, $\bar{L}(t)$ is the average over the respective actual loads. Once coefficients are computed with (6), expected load $L'(\tau_c')$ can be estimated from (5), as exemplified in Fig. 2 considering actual load in 9 most recent cycles.

In order to attain a load within (or at least very close to) the saturation band at next cycle τ_c', we estimate the amount s of trajectories to shed or supplement proportionally to the number n of active trajectories utilized at current cycle τ_c. Clearly, s should depend on the relative difference of C and L', as illustrated in Fig. 4, no matter where current load L is w.r.t. the saturation band. Hence, s is *not fixed*, but instead takes fluctuating values (in number of trajectories) according to expected surplus or deficit in load.

The adaption process is triggered in situations of significant load overflow or underflow with respect to system capacity C. More specifically, we distinguish the following cases:

- *Overload* occurs when expected load L' is above the saturation band at τ_c', i.e., $L' \geq C + \theta$, as illustrated in Fig. 4a. Then, the system is expected to be overwhelmed with more data than it can possibly process on time, so the method must keep less active trajectories for evaluating queries. But how many trajectories to shed from those n currently active? Conservatively, this value s should be derived in proportion to n trajectories, so that load at τ_c' would not exceed the band:
$$s = \left\lceil \frac{C + \theta - L'}{L} \cdot n \right\rceil \tag{7}$$
The *negative sign* of the result in (7) means that s out of the n currently active trajectories must be dismissed.
- *Underload* occurs once expected load L' is below the band at τ_c', i.e., $L' \leq C - \theta$ as illustrated in Fig. 4b, also provided that there are some trajectories previously inactivated, which may now become active again. This latter condition means that there is room for admitting extra trajectories and thus improve the quality of answers. How many trajectories to add into those n currently active is dictated by the fact that load at τ_c' should not fall below the band:
$$s = \left\lfloor \frac{C - \theta - L'}{L} \cdot n \right\rfloor \tag{8}$$
The *positive sign* of the resulting integer value in (8) indicates that s more trajectories should be activated.
- When $C - \theta < L' < C + \theta$, load L' is expected *within the saturation band* at τ_c' (Fig. 4c). In this case, no adaption is necessary and the subset of active trajectories is not modified, since load L' is expected at normal levels. Even so, trajectories may start or finish at any time due to changing motion patterns, hence *actual load* L may occasionally fluctuate despite any adaption decisions.
- It may happen that expected load L' is below the band at τ_c', i.e., $L' < C - \theta$, while all trajectories are active (i.e., none dismissed). Then, *no adaption* is necessary, as less and less objects are on the move (e.g., roads at night hours) and load normally falls. Due to self-regulation, adaption will be triggered once overload is noticed again.

Note that a *stepwise* adaption scheme could be used (loosely inspired from [19]), repeatedly increasing or reducing load by a fixed step s until the system becomes stable, i.e., $L \leq C$. In future work, we plan to study a more sophisticated cost model involving operator (i.e., traffic aggregate) selectivity, per tuple processing time and actual load L; however, a proper choice for s may be tricky due to mutability of motion patterns across time.

4.3.3 Semantic Selection of Trajectories.

We choose trajectories to become active at next cycle τ_c' according to heuristics:

<u>Heuristic #1</u>: *Insufficient samples.* A trajectory T_i contributing to queries with few other active trajectories ("samples") should be preserved. Intuitively, not many trajectories are active along such a query q, because objects traverse infrequently its region or have become inactive. Potential inactivation of T_i would worsen or even make impossible a response to q. For instance, if all trajectories along a road are suppressed, no traffic aggregates can be computed.

Selecting a *cutoff limit* ξ to denote insufficient samples per query is based on statistics in the Trajectory Manager. An equi-width *histogram* $\mathcal{H}$ holds b buckets over the various values concerning the number of currently active trajectories per query. Then, the cutoff limit ξ for few samples is the maximum value of the first *b-tile* in $\mathcal{H}$. For example, with $b = 5$, we examine the first quantile containing the lowest 20% of trajectory counts, and ξ is the maximum of these count values. So, this cutoff limit can distinguish queries with very small number ($\leq \xi$) of trajectories utilizable in processing, so as to avoid their inactivation.

<u>Heuristic #2</u>: *Trajectory ranking* can be also used to favor trajectories traversing multiple spatial regions and spanning longer intervals. Such a ranking could be based on the amount of regions that each object has traversed and the interval of its presence along each one. This can be easily achieved thanks to the maintained trajectory synopses (Section 4.1), by considering a weighted sum of their most recent traversals across spatial regions, i.e., the queries that monitor traffic therein. Instead of just relying on the last query most recently affected by a trajectory, an *ageing-aware scoring scheme* could be employed to rank trajectories according to their presumed utility in query answering. A standard *weight* $a \in (0, 1]$ is used to progressively weaken older traversals, i.e., over queries previously affected by a given trajectory T_i. Thus, at current cycle τ_c, trajectory T_i may be assigned an *adjusted score* of

$$\rho_i = \frac{1}{\Delta\tau_i} \cdot \sum_{k=1}^{|S_i|} a^k \cdot (t_k^{out} - t_k^{in}) \tag{9}$$

by only probing its maintained synopsis S_i. Items in S_i are accessed in reverse chronological order; duration of each item (i.e., region traversed by trajectory T_i) is normalized by the total duration $\Delta\tau_i$ of all items in S_i. For example, if $a = \frac{1}{2}$ and object o_1 in Fig. 1 relays its positions regularly every 60 sec, from its synopsis S_1 we can derive its current score $\rho_1 = \frac{1}{480} \cdot (\frac{1}{2} \cdot 120 + (\frac{1}{2})^2 \cdot 240 + (\frac{1}{2})^3 \cdot 120) \approx 0.28$. Clearly, scores are derived from compact trajectory segments (over queries q_4, q_2, q_3 in this example), and not sparse locations.

Intuitively, score ρ_i quantifies the *expected utility* of trajectory segments from object o_i in query answering. Ageing also prevents trajectories from alternating between active and inactive states, by reflecting their motion over longer intervals. Once scores are computed for every trajectory, this heuristic picks up trajectories for activation starting from top ranking ones until it reaches the target amount n of active trajectories for next cycle τ_c'. The higher the attained score ρ_i, the greater the chance that trajectory T_i becomes active. If T_i turns inactive, it stops being updated, but its existing segments (prior of shedding) can still be used in query computations until their expiration from the respective sliding windows.

Algorithm 1: TrajectoryActivation (#trajectories n, #buckets b)

```
1  State: 𝒯 = {T_i : trajectory maintained for moving object o_i}
2  State: 𝒦 = {∀ query q, {o_i : object o_i traversed q during q.ω}}
3  Output: 𝒰 = {u : ∀ object o_i, u=1 if o_i is active; u=0 if inactive}
4  𝒰.resetAll()                 //Initially, mark all trajectories as inactive
5  ℋ ← ∅   //Histogram (with b buckets) of trajectory counts per query
6  for each query q ∈ 𝒦 do
7      ℋ.insert(𝒦[q].size())  //Put trajectory count for q into histogram
8  ξ ← max({values in bucket ℋ[1]})   //Cutoff limit based on first tile
9  for each query q ∈ 𝒦 do
10     if 𝒦[q].size()≤ ξ then  //  Prioritize queries of ≤ ξ samples
11         for each object o_i ∈ 𝒦[q] do
12             if n > 0 then
13                 𝒰[o_i] ← 1     //Activate trajectory T_i for next cycle
14                 n ← n − 1
15 B ← ∅        //Priority queue with trajectories ranked by their utility
16 for each object o_i ∈ 𝒯 \ {o_j : 𝒰[o_j] = 1} do
17     ρ_i ← RankTrajectory(o_i)                    //Calculated by Eq. (9)
18     B.insert(⟨o_i, ρ_i⟩)      //Trajectory rankings in descending order
19 while (n > 0) ∧ (B ≠ ∅) do
20     𝒰[B.top()] ← 1                  //Pick trajectories with top scores ...
21     B.pop()                         //...until target count n is reached
22     n ← n − 1
23 return 𝒰      //Trajectories to utilize in query evaluation at next cycle
```

As listed in Algorithm 1, semantic selection applies Heuristic #1 trying to preserve selected trajectories with distinct motion patterns as active in bitmap $\mathcal{U}$ and avoid starvation of queries (Lines 6–15). Then, the score-based Heuristic #2 activates extra trajectories in $\mathcal{U}$ by their rankings (Lines 16–23) without considering again those already qualifying to the first rule (Line 17). Resulting bitmap $\mathcal{U}$ (Line 24) determines which trajectories will be active at cycle τ_c'.

5 EMPIRICAL EVALUATION

In this Section, we report results regarding performance and approximation quality from a comprehensive evaluation of the proposed load adaption framework over large synthetic trajectory data.

5.1 Experimental Setup

To the best of our knowledge, there is lack of publicly available trajectory data with massive, frequent position updates from a large number of moving objects. Hence, we generated synthetic trajectories simulating vehicles that move on the road network of greater Athens (area ≈250 km^2). Objects are moving at varying speeds during their course, but each time according to the average speed (values assigned according to traffic studies) of the road edge they currently traverse. After calculating shortest paths for up to 1,000,000 vehicles between nodes chosen randomly across the network, position samples were taken every 10 seconds along each such path (the longest trip took 4060 sec). Additional samples were taken once a vehicle makes a turn into another road, so as to capture a possible change in speed. Instead of letting all objects start moving at τ=0, their trajectories were time-shifted to occur within a period T=4200 sec. According to this scheme, at τ=0 no vehicle is moving,

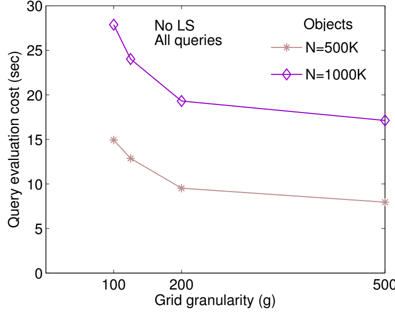
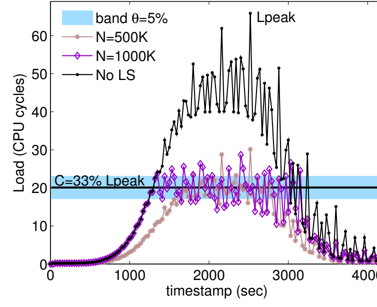
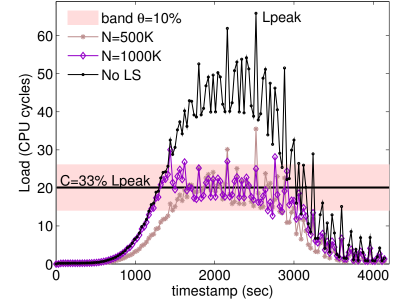


Figure 5: Sensitivity to grid granularity



(a) $\theta = 5\%L_{peak}$



(b) $\theta = 10\%L_{peak}$

Figure 6: Load adaption at fixed capacity $C = 33\%L_{peak}$ with varying object count N

but progressively more and more start circulating. At τ=2100 sec, each vehicle is halfway on its journey; by then, system load is expected to soar as all vehicles are on the move. Eventually, as more and more objects reach their destination, load is diminishing. As a varying number of vehicles is moving at each time, this scenario attempts to simulate a peak effect in traffic conditions, especially during rush hours along major arterials in the network.

Regarding continuous queries, each one corresponds to a particular road and monitors its average speed. The spatial region of a query covers the carriageway of a road, i.e., a buffer polygon in a specific direction of traffic flow; hence, two distinct queries are used for bidirectional roads. Each query specifies its own window (ω, β) assigned according to road classification, as listed in Table 1. E.g., motorways are more important and more susceptible to traffic fluctuations that must be promptly detected, thus they are given shorter range and slide values compared with the rest. For smooth window maintenance at runtime, we set $\frac{\omega}{\beta} = 10$ in all windows, and the various β values are multiples of a basic period $\delta = 30$ seconds between successive execution cycles.

Algorithms were implemented in C++ and experiments with diverse parameter settings were simulated on an Intel Xeon E5-2660 CPU at 2.20 GHz and 96GB RAM. All processing takes place in main memory. Load is shown across time, as the stream is consumed and continuous queries get evaluated; all other performance metrics are averages over execution cycles where all queries had to provide response. Execution cycles occur every $\delta = 30$ sec, as stipulated by the minimum slide step β in queries. Table 2 lists experiment parameters and their ranges; default values are shown in bold.

5.2 Performance Results

In a preliminary test, before considering any load adaption, we searched for a suitable granularity g for the spatial grid. As indexing concerns only the query regions, Figure 5 shows average cost of query evaluation per cycle without load adaption (*No LS*). With a finer granularity, the monitored area is subdivided into more and smaller cells. Although memory footprint increases, each cell covers fewer query regions, incurring reduced evaluation costs. In the sequel, all tests were conducted with a 500×500 grid, so as to reduce query evaluation overhead and focus mainly on adaptivity to load fluctuations.

The first set of adaption tests (Fig. 6) were conducted with a *fixed* capacity $C = 33\%L_{peak}$ (in CPU cycles). L_{peak} reflects the maximum load ever observed in the system, namely when it had to manage 1000K objects without load adaption (respective plot *No LS* is illustrated for comparison). Forcing the system to remain at a steady state when the maximum load can reach a value three times higher, is a stress test for our adaption method. In general, the greater the tolerance θ, the less frequently adaption is invoked. In monitoring 1000K objects, load ascends more sharply and crosses the saturation band earlier in time, hence adaption is also triggered earlier. This means that the system can afterwards make better estimates with regression about the expected load and choose a suitable subset of active trajectories. This is much more evident with a relaxed $\theta = 10\%$, where the load by and large fluctuates within the band in the period where most of trajectories are on the move. With a narrower saturation band ($\theta = 5\%$), adaption seems to fare better with 500K objects; sudden spikes are fewer, although slightly higher. With more (1000K) objects, less deviations are observed in actual loads, but more spikes fall out of the band. However, spikes are inevitable: recall that our windowed queries specify differing slides β (Table 1), hence a varying number of queries must be answered at each iteration. Such spikes occur exactly when all M queries must be given a fresh response, just after several iterations where smaller subsets of queries get evaluated at a diminished cost. In these cases, predicting the expected load is less successful due to the extreme mutability in query workload.

Table 1: Query specifications

Road class	*#Queries*	*Total length* (km)	ω (sec)	β (sec)
1: *motorway*	2	4.1	300	30
2: *highway*	21	62.2	600	60
3: *primary*	575	557.8	900	90
4: *secondary*	1173	689.9	1200	120

Table 2: Experiment parameters

Parameter	Values
Number N of moving objects	500K, **1000K**
Number M of continuous queries	1771
Grid granularity g per axis	100, 125, 200, **500**
System capacity C (as $\%L_{peak}$)	33%, **50%**, 67%
Tolerance θ (as $\%L_{peak}$) around C	**5%**, 10%
Number b of buckets in histogram	**5**, 10, 20

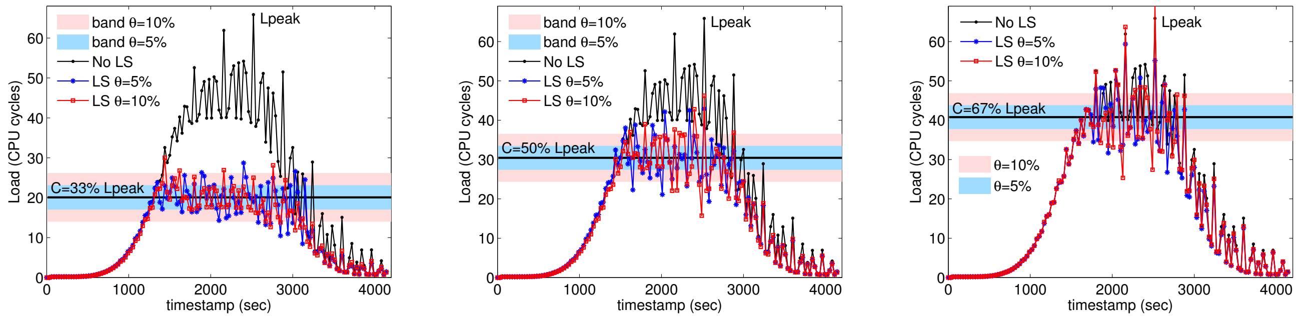


Figure 7: Load adaption against trajectories of $N = 1000K$ objects for varying values in capacity C and tolerance θ

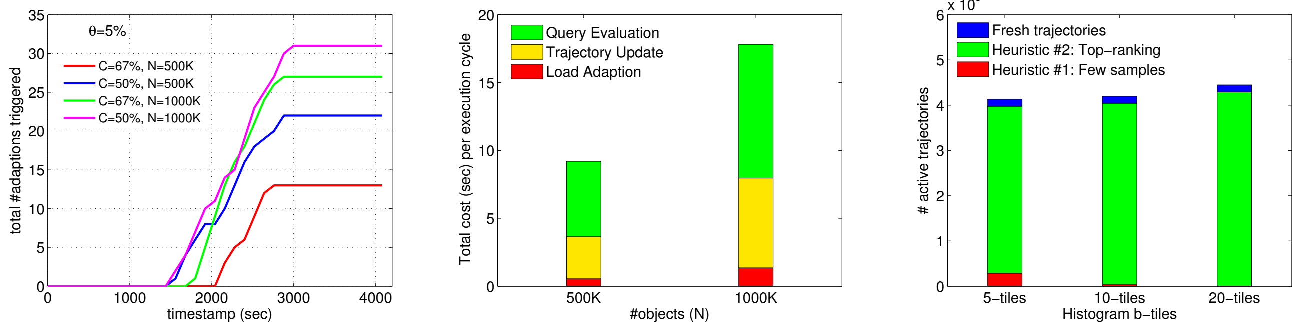


Figure 8: Number of triggered adaptions

Figure 9: Processing cost per stage

Figure 10: Active trajectories per cycle

Figure 7 plots actual loads for varying values in capacity C and tolerance θ over trajectories of $N = 1000K$ moving objects. When $C = 33\%L_{peak}$, adaption achieves its best results, as load practically remains restricted within the band with very few exceptions. Even when spikes do occur due to the varying mix of queries, their deviation from the band is almost negligible. But when more capacity is allowed, more and greater spikes are observed. Curiously, the situation gets worse when more capacity $C = 67\%L_{peak}$ is given. In that case, adaption seems to offer very little compared to no adaption at all (*No LS*). The reason is that, at most iterations, load normally remains close to the band when less than M queries are processed, so no adaption is required. But, when the system must respond to all M queries, it cannot predict their excessive demands with regression, hence it allows a greater number n of active trajectories. The loss in trajectory information is less, but cost escalates beyond the band. Clearly, this is a limitation of the regression model currently employed, which we plan to fine tune in future work.

In all plots concerning load, notice the fluctuations below the saturation band approximately after $\tau = 3000$ sec. The cause is not load adaption, but the varying number of queries that require response at each execution cycle. Figure 8 illustrates how many times adaption was triggered (cumulatively since $\tau = 0$) while consuming the input stream with tolerance $\theta = 5\%$, but varying numbers N of objects and allowed capacities C. Initially, there is no need for adaption, as load remains below saturation. As soon as the system begins to experience overloading, adaption is repeatedly triggered to balance the load (and also correct underloading situations). Of course, the more the monitored objects and the less the allowed capacity, the more frequently adaption gets triggered. But, once many objects reach their destination (when $\tau > 3000$ sec), less trajectories need maintenance and load normally starts to fall. Then, load adaption is not needed anymore.

Figure 9 depicts the average cost for each stage over execution cycles where all M queries receive response (i.e., the most intense situations). The overall cost represents the time for updating all (active) trajectories, answering all pending queries, as well as the adaption overhead (when applied). As intended, this overhead remains low enough, compared to the other two stages that perform user-requested computations. Updating trajectories takes less than evaluating queries, mainly because a significant portion of trajectories is occasionally dismissed, yielding considerable savings to resources. Query evaluation is more expensive because all M queries must be responded, so auxiliary data structures must be frequently accessed. Clearly, the total cost grows linearly with the number N of objects, since the query workload is fixed. But, even with $N = 1000K$ objects, average cost is less than 18 seconds per cycle, well in advance from the next cycle (every δ=30 sec) treating another mix of queries over more fresh data.

Regarding *cutoff limit* ξ for designating insufficiently small samples of trajectories for queries, Fig. 10 illustrates tests with different number b of buckets in the histogram that maintains trajectory counts. Recall that designating a suitable ξ value gives a clue about the magnitude of such a small sample. For example, when $b = 5$, we pick the first quantile which indicates that 20% of the roads have less than ξ trajectories passing through them and cannot afford reducing their samples without harming the resulting traffic esti-

Table 3: Error in speed estimates w.r.t. exact measurements

(a) Error with different system capacity

Capacity C	*avg (km/h)*	*stdev (km/h)*
$50\%L_{peak}$	0.665	1.382
$67\%L_{peak}$	0.346	1.436

(b) Error by road length

Length (km)	*avg (km/h)*	*stdev (km/h)*
< 0.2	0.679	1.517
$0.2 - 0.5$	0.648	1.200
$0.5 - 1.0$	0.660	1.239
$1.0 - 2.0$	0.753	1.501
> 2	0.909	2.023

(c) Error per road class

Road type	*avg (km/h)*	*stdev (km/h)*
1: *motorway*	1.319	3.431
2: *highway*	1.423	3.181
3: *primary*	0.667	1.255
4: *secondary*	0.666	1.272

mates. The less the number of buckets, the more trajectories will be activated because of *Heuristic #1*. Fairly enough, the vast majority of trajectories are activated thanks to their estimated utility scores (*Heuristic #2*). This is desirable, as it keeps longer trajectories spanning wider time intervals along multiple spatial regions of interest. Hence, a single trajectory may contribute its partial analytics to multiple queries. Also note that a small percentage of active trajectories are fresh ones, i.e., objects that are starting to move and are chosen to be retained as active.

5.3 Quality of Approximate Traffic Estimates

In order to measure accuracy of *approximate* traffic estimates resulting from our methodology, we compared speed estimates with respective *exact* results after exhaustive evaluation over all incoming positions from $1000K$ objects (i.e., without shedding any trajectories). As shown in Table 3a for typical settings, the *error* of approximate estimates from exact speed measurements is less than 0.7km/h on average, whereas the standard deviation over this difference is below 1.5km/h. This clearly shows that the estimates can be considered quite reliable with an error margin up to around 2km/h. Of course, such deviations tend to increase across longer roads (Table 3b), where wider fluctuations on speed may occur, especially over roads of length more than a kilometer. Similarly, roads carrying more traffic (motorways and highways) are more susceptible to fluctuations in speed, hence slightly wider deviations may be observed in the respective estimates listed in Table 3c. Given that vehicles regularly move at high speeds in such arterial roads, this difference is practically insignificant.

6 CONCLUDING REMARKS

In this paper, we proposed a methodology for restraining system load at desirable levels when processing multiple queries that compute traffic analytics over streaming trajectories. Our strategy employs semantic filters that discard raw information obtained from less important trajectories and retain only those that mostly contribute to the accuracy in resulting aggregates. Our model for load regulation is adaptive to actual conditions, taking into account the expected cost of updating varying numbers of trajectories, as well as the distribution of the continuous queries in order to yield timely and reliable answers. Extensive tests against scalable data volumes have confirmed the robustness of this method and its ability to be self-regulated by smoothly adapting to fluctuating query demands and dynamically changing motion patterns.

Regarding future extensions, offering guarantees for query accuracy and throughput given a system capacity would greatly add to the robustness and adaptivity of this mechanism. Besides, probabilistic or sampling schemes could be used to recompensate for information shed from inactive trajectories. Finally, a distributed approach with data and query partitioning over multiple processing nodes in modern cluster infrastructures would be worthwhile for smoother adaption and load balancing against increased workloads.

Acknowledgements. This work was partially supported by the EU project *SLIPO* (H2020-ICT-2016-1-731581) and the NSRF 2014-2020 project *HELIX* (grant #5002781).

REFERENCES

[1] B. Babcock, M. Datar, and R. Motwani. Load Shedding for Aggregation Queries over Data Streams. In *ICDE*, pp. 350-361, 2004.
[2] A. Biem, E. Bouillet, H. Feng, A. Ranganathan, A. Riabov, O. Verscheure, H. Koutsopoulos, and C. Moran. IBM InfoSphere Streams for Scalable, Real-time, Intelligent Transportation Services. In *ACM SIGMOD*, pp. 1093-1104, 2010.
[3] B. Gedik, L. Liu, K. Wu, and P. S. Yu. Lira: Lightweight, Region-aware Load Shedding in Mobile CQ Systems. In *ICDE*, pp. 286-295, 2007.
[4] B. Gedik, K.-L. Wu, L. Liu, and P. S. Yu. Load Shedding in Mobile Systems with MobiQual. *TKDE*, 23(2): 248-265, 2011.
[5] E. Kalyvianaki, T. Charalambous, M. Fiscato, and P. Pietzuch. Overload Management in Data Stream Processing Systems with Latency Guarantees. In *Feedback Computing*, 2012.
[6] S. J. Kazemitabar, U. Demiryurek, M. Ali, A. Akdogan, and C. Shahabi. Geospatial Stream Query Processing using Microsoft SQL Server StreamInsight. *PVLDB*, 3(2): 1537-1540, 2010.
[7] D. Kulkarni, C. V. Ravishankar, and M. Cherniack. Real-time Load-adaptive Processing of Continuous Queries over Data Streams. In *DEBS*, pp. 277-288, 2008.
[8] Y. Lou, C. Zhang, Y. Zheng, X. Xie, W. Wang, and Y. Huang. Map-Matching for Low-Sampling-Rate GPS Trajectories. In *ACM GIS*, pp. 352-361, 2009.
[9] A. Magdy, M. F. Mokbel, S. Elnikety, S. Nath, and Y. He. Venus: Scalable Real-Time Spatial Queries on Microblogs with Adaptive Load Shedding. *TKDE*, 28(2): 356-370, 2016.
[10] M. F. Mokbel and W. G. Aref. SOLE: Scalable On-line Execution of Continuous Queries on Spatio-temporal Data Streams. *VLDB Journal*, 17(5): 971-995, 2008.
[11] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *ACM SIGMOD*, pp. 634-645, 2005.
[12] B. Mozafari and C. Zaniolo. Optimal Load Shedding with Aggregates and Mining Queries. In *ICDE*, pp. 76-88, 2010.
[13] R. Nehme and E. Rundensteiner. ClusterSheddy: Load Shedding Using Moving Clusters over Spatio-temporal Data Streams. In *DASFAA*, pp. 637-651, 2007.
[14] K. Patroumpas and T. Sellis. Maintaining Consistent Results of Continuous Queries under Diverse Window Specifications. *Information Systems*, 36(1): 42–61, 2011.
[15] T. N. Pham, P. K. Chrysanthis, and A. Labrinidis. Avoiding Class Warfare: Managing Continuous Queries with Differentiated Classes of Service. *VLDB Journal*, 25(2): 197-221, 2016.
[16] F. Reiss and J. M. Hellerstein. Data Triage: An Adaptive Architecture for Load Shedding in TelegraphCQ. In *ICDE*, pp. 155-156, 2005.
[17] N. Rivetti, Y. Busnel, and L. Querzoni. Load-Aware Shedding in Stream Processing Systems. In *DEBS*, pp. 61–68, 2016.
[18] D. Šidlauskas, S. Šaltenis, C. Christiansen, J. Johansen, and D. Šaulys. Trees or Grids? Indexing Moving Objects in Main Memory. In *ACM GIS*, pp. 236-245, 2009.
[19] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker. Load Shedding in a Data Stream Manager. In *VLDB*, pp. 309-320, 2003.
[20] N. Tatbul and S. Zdonik. Window-Aware Load Shedding for Aggregation Queries over Data Streams. In *VLDB*, pp. 799-810, 2006.
[21] Y. Tu, S. Liu, S. Prabhakar, and B. Yao. Load Shedding in Stream Databases: A Control-Based Approach. In *VLDB*, pp. 787-798, 2006.