ClusterSheddy: Load Shedding Using Moving Clusters over Spatio-Temporal Data Streams

Rimma V. Nehme[1] and Elke A. Rundensteiner[2]

[1] Purdue University, West Lafayette, IN 47907 USA
[2] Worcester Polytechnic Institute, Worcester, MA 01608 USA
`rnehme@cs.purdue.edu, rundenst@cs.wpi.edu`

Abstract. Moving object environments are characterized by large numbers of objects continuously sending location updates. At times, data arrival rates may spike up, causing the load on the system to exceed its capacity. This may result in increased output latencies, potentially leading to invalid or obsolete answers. Dropping data randomly, the most frequently used approach in the literature for load shedding, may adversely affect the accuracy of the results. We thus propose a load shedding technique customized for spatio-temporal stream data. In our model, spatio-temporal properties, such as location, time, direction and speed over time, serve as critical factors in the load shedding decision. The main idea is to abstract similarly moving objects into *moving clusters* which serve as summaries of their members' movement. Based on resource restrictions, members within clusters may be selectively discarded, while their locations are being approximated by their respective moving clusters. Our experimental study illustrates the performance gains achieved by our load-shedding framework and the tradeoff between the amount of data shed and the result accuracy.

1 Introduction

Applications dealing with extremely large numbers of moving objects are becoming increasingly common. These include fleet monitoring [31], location-based services [18] and scientific applications [25]. In such applications, queries are typically continuously evaluated over data streams composed of location updates. At times such data streams may become bursty and thus exceed system capacity.

However, existing load smoothing techniques [16, 24, 30] that store the tuples that cannot be processed into archives (spill them to disk) are not viable options for streaming spatio-temporal data. This is because spatio-temporal applications typically have real-time response requirements [18, 21]. Any delay in the answer to a query would give obsolete results, and with objects continuously changing their locations, make them invalid or useless.

In order to deal with resource limitations in a graceful way, returning approximate query answers instead of exact answers has emerged as a promising approach [3, 8, 9]. Load shedding is a popular method to approximate query answers for stream processing while reducing the consumption of resources. The goal is to minimize inaccuracy in query answers while keeping up with the incoming data load. The current state-of-the-art in load shedding [1, 2, 9, 22, 26, 28, 29] can be categorized into two main approaches. The first relies on syntactic (random) load shedding, where tuples are discarded randomly based on expected system performance metrics such as output rate [9, 27]. The second approach, also known as semantic load shedding, assigns priorities to tuples based on their utility (semantics) to the application and then sheds those with low priority first [6, 27]. However, both of these approaches may suffer from high inaccuracy if applied to spatio-temporal data streams. The reason is two-fold: (1) they do not consider the spatio-temporal properties of the moving objects when deciding which data to load shed, and (2) they do not consider that both queries as well as objects have the ability to change their locations. Hence the results to queries depend on both the *positions of the moving objects* and of *the moving queries* at the time of querying.

To motivate the solution presented in this paper, we consider a scenario from the supply-chain management application – fleet monitoring. We assume that vehicles are equipped with positioning devices (e.g., GPS) and are travelling in convoys, i.e., in close proximity from each other. Using random load shedding, all vehicles' location updates are treated equally. Thus any tuple is equally likely to be discarded and the whereabouts of the vehicle may be unknown for some duration of time. Using semantic load shedding, a user may specify vehicles with the most valuable (e.g., expensive or perishable) cargo having the highest utility. The locations of the vehicles with lower utility may be discarded first, and thus temporarily loosing the location information of those moving objects.

The scenario above illustrates that using current load shedding techniques, the spatio-temporal properties of the moving objects are not taken into account when deciding which data to discard. However, if the workload must be reduced by dropping some data, taking into account such spatio-temporal properties as location, speed, direction, much higher accuracy can be achieved. Another point the scenario above illustrates is the spatio-temporal relationship of several different objects relative to each other, more specifically the *similarity of movement*. Thus intuitively if we can approximate similarly moving objects into clusters, and keep track of spatio-temporal properties of the cluster as a whole, then we could load shed the objects close to the center of the cluster without losing much in the results accuracy. So the decision to discard certain data is not related to only one object, but rather to dynamically formed sets of objects. To the best of our knowledge, no prior work has addressed this thus far.

1.1 Spatio-Temporal Similarity

We observe that large numbers of moving objects often share some spatio-temporal properties, in fact, they often naturally move in clusters for some periods of time [7, 14]. For example, migrating animals, city pedestrians, or a convoy of cars that follow the same route in a city naturally form moving clusters (Fig. 1). Such moving clusters do not always retain the same set of objects for their lifetime, rather some objects may enter or leave over time. For example, new animals may enter the migrating group, and others may leave the group (e.g., animals attacked by predators). While belonging to a particular cluster, the object shares *similar properties* with the other objects that belong to the same cluster. In this case, the spatio-temporal properties of the cluster such as

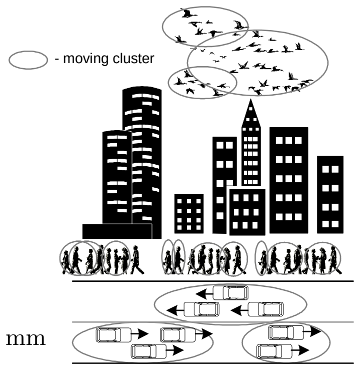

Fig. 1. Conceptual View of Moving Clusters

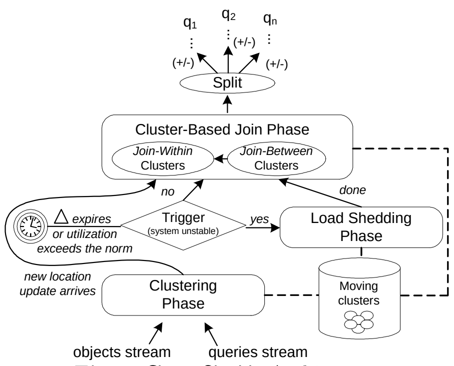

Fig. 2. ClusterSheddy Architecture.

speed, direction[3], and relative proximity of other moving objects summarize *how* these objects are moving and *where* they currently are.

A cluster can in some sense serve as a "common-feature-abstraction" of a group of moving objects sharing spatio-temporal properties. We now postulate that this abstraction can be exploited for efficient load shedding.

1.2 Our Contributions: ClusterSheddy Framework

In this paper, we present the ClusterSheddy framework for processing spatio-temporal queries on moving objects. ClusterSheddy is equipped with novel methods for spatio-temporal load shedding based on motion semantics. Moving clusters, abstracting similarly moving objects and queries, serve as summaries of their members and preserve their location, even if approximate, when their individual positions are load shed. The novelty of our method is that it uses dynamic clusters, together with the knowledge of the current system resources to determine *when*, *how much* and *which* data to load shed. Inside each moving cluster, a nested data structure, termed *nucleus*, abstracts the positions of the cluster members whose positions are load shed. In other words, the load shedding in ClusterSheddy takes a "from-inside-out" approach, where objects/queries closest to the center of the cluster are load shed first. The motivation is that the closer cluster members are to the centroid, the more accurately the cluster approximates their individual locations. The sizes of the moving clusters' nuclei are resource-sensitive, meaning that the nucleus size of a moving cluster changes depending on the current resource availability. We measure the quality of a load shedding policy in terms of the deviation of the estimated answers produced by the system from the actual answers. Experimental results illustrate that ClusterSheddy is very effective in quickly reducing the load while maintaining good accuracy of the results.

Roadmap: Section 2 provides an overview of the ClusterSheddy framework and the moving cluster abstractions over spatio-temporal streams. Section 3 describes the load shedding technique based on moving clusters and the different policies for shedding clusters. Section 4 presents our experimental results. Section 5 discusses related work, and Section 6 concludes the paper.

[3] We measure direction using a counterclockwise angle of rotation with due East. Using this convention, a vector with a direction of 30 degrees is a vector which has been rotated 30 degrees in a counterclockwise direction relative to due east.

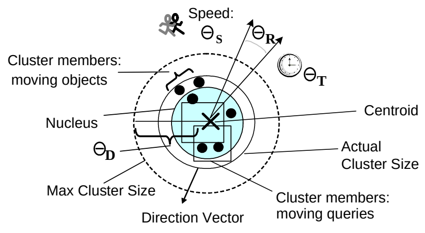

Fig. 3. A Moving Cluster Representation.

Notation	Property	Description
m.Cid	Cluster ID	Numeric cluster identifier
m.oCount	Object Members	Number of moving objects
m.qCount	Query Members	Number of moving queries
m.Loc	Centroid Location	Location of the centroid of the cluster
m.R	Radius	Radius of the cluster
m.AveSpeed	Average Speed	Average of the speeds of all cluster members
m.AveDir	Direction	Movement Direction
m.UpdTime	Update Time	Last update/check time
m.MaxSize	Max Size	Maximum area that the cluster can increase up to
m.Size	Actual Size	Current area of the cluster, m.Size = *m.R2
m.Ans	Results	Query results associated with the queries in the cluster

Fig. 4. Moving Cluster Properties.

2 ClusterSheddy Framework

2.1 Query Evaluation in ClusterSheddy

ClusterSheddy is encapsulated into a physical non-blocking pipelined query operator that can be combined with traditional operators in a query network plan. The input to ClusterSheddy consists of moving objects and spatio-temporal query streams. Moving objects' location updates arrive in the form (oid, Loc, t), where oid is an object id, and Loc is its location at time t. Continuous queries arrive in the form (qid, Loc, t, $qType$, $qAttrs$), where qid is a query id, and Loc is its location at time t, $qType$ is a query type (e.g., knn, range), and $qAttrs$ represents query attributes (e.g., a value of k for a knn query).

The ClusterSheddy execution process consists of three phases: (1) clustering, (2) cluster-based join, and (3) load shedding (Fig. 2). When new location data for an object/query arrives, the object/query joins either an existing cluster or forms its own cluster (clustering phase). Similar to our prior work, SCUBA [20], spatio-temporal queries on moving objects are evaluated by first performing a spatial *join between* moving clusters pruning true negatives. If two clusters do not intersect with one other, the objects and queries belonging to these clusters are guaranteed to not join either. Thereafter, in the *join-within* step, individual objects and queries inside the clusters are joined with each other. This two-step filter-and-join process helps reduce the number of unnecessary spatial joins.

Unlike SCUBA, ClusterSheddy implements incremental query evaluation and unlike SINA [19] and SEA-CNN [32], it is done at the coarser level of moving clusters rather than of individual objects and queries. Such incremental approach helps to avoid continuous re-evaluation of spatio-temporal queries. Moreover, ClusterSheddy effectively employs load shedding based on moving clusters, a task not addressed in these prior works.

2.2 Moving Cluster Abstraction

Given the intuition highlighted in Section 1.1 that the moving objects often travel closely together in space for some period of time, we group moving entities[4] into *moving clusters* based on their shared spatio-temporal properties (Fig. 3). Moving entities that don't satisfy conditions of any existing clusters form their own clusters. When moving entities change their spatio-temporal properties, they may enter or leave a moving cluster, and the properties of that cluster (depicted in Fig. 4) are then adjusted accordingly.

[4] By moving entities we mean both moving objects and spatio-temporal queries.

```
PROCEDURE ClusterMovingObject (MovingCluster M)
1. new object o arrives
2.    if distance between o and M.Loc ⩽ distance threshold
3.    and speed difference between o and M ⩽ speed threshold
4.    and direction difference between o and M ⩽ direction threshold
5.    and time difference between o location update and M update ⩽ time threshold
6.        add object o to the moving cluster M, updating cluster properties:
              a) member object count (M.oCount) and total member count (M.Count)
              b) average speed (M.AveSpeed)
              c) average direction (M.Dir)
              d) location of the cluster centroid (M.Loc)
              e) cluster radius (M.R)
              f) record the time of the cluster update (M.UpdTime)
7.    else create new cluster based on the properties of object o
```

Fig. 5. Pseudo Code for Clustering Moving Objects.

Four similarity thresholds play a key role in determining the moving clusters[5]. Using these thresholds, we define the *similarity* among moving entities.

Definition 1. *(Similarity Condition) Let Θ_S be the maximum speed difference threshold, Θ_D the maximum spatial distance threshold, Θ_R the maximum direction difference threshold, and Θ_T the maximum time difference threshold. Let t_k and t_l be the times when moving entities e_i and e_j ($i \neq j$) last updated their spatio-temporal properties. Then if $|e_i.Speed - e_j.Speed| \leq \Theta_S$, and $|e_i.Loc - e_j.Loc|$[6] $\leq \Theta_D$, and $|e_i.Dir - e_j.Dir| \leq \Theta_R$, and $|t_k - t_l| \leq \Theta_T$, the entities are said to be similar, $e_i \overset{s}{=} e_j$.*

Definition 2. *(Moving Cluster) Let E=$\{e_1,e_2...e_i\}$ be a set of moving entities. A moving cluster m is a non-empty subset of E ($m \subseteq E$), with spatio-temporal properties $m_{spt} = (AveSpeed, AveDir, Loc, R, t \ldots)$ which represents the average of the spatio-temporal properties of all entities $e_i \in m$, and where each e_i satisfies the similarity condition with respect to m_{spt}.*

Using Definition 1 above, a moving entity e_i that is found to be in close proximity of a cluster centroid ($m.Loc$) and has similar properties with the cluster, is added to that respected moving cluster. Our clustering method is based on the classic *leader-follower* (LF) algorithm[7] [10, 11]. The LF algorithm can handle incrementally streaming data, producing adequate quality moving clusters in linear time. Fig. 5 gives the pseudo-code for clustering moving objects. Similar processing is done for clustering moving queries. Due to space constraints, we omit the description of the clustering procedure. For more details, we refer the reader to [20].

2.3 Incremental Query Evaluation Using Moving Clusters

ClusterSheddy uses an incremental strategy in evaluating joins *between* moving clusters, and then *within* the overlapping clusters that we describe next.

Incremental Join-Between: Consider two moving clusters m_1 and m_2 (Fig. 6). When performing a join *between* moving clusters, ClusterSheddy distinguishes between four cases as illustrated in Table 1[8]. Column *"Clusters at time t_0"* indicates if m_1 and m_2 intersected at an old time t_0, column *"Clusters at time t_1"*

[5] Deriving threshold values that give you near-optimal clustering is a research area of its own. In our work, we approximated the threshold values that would cluster on average a certain number of objects per cluster based on the properties of the data [12].

[6] The difference is measured by a distance function (e.g., euclidean distance).

[7] However, any alternative clustering algorithm can also be plugged-in, as long as it is efficient.

[8] By $\cap$, we denote intersection.

Case	Clusters at time t_0	Clusters at time t_1	Join-Within Needed	Result Updates	Illustration
1.	$m_1 \cap m_2 = \emptyset$	$m_1 \cap m_2 = \emptyset$	-	-	Fig. 6a
2.	$m_1 \cap m_2 = \emptyset$	$m_1 \cap m_2 \neq \emptyset$	✓	*positive*	Fig. 6b
3.	$m_1 \cap m_2 \neq \emptyset$	$m_1 \cap m_2 = \emptyset$	-	*negative*	Fig. 6c
4.	$m_1 \cap m_2 \neq \emptyset$	$m_1 \cap m_2 \neq \emptyset$	✓	*positive/negative*	Fig. 6d

Table 1. Cases for Incremental Evaluation: ***Join-Between*** Moving Clusters

Case	Clustering at time t_1	$m.Ans$ at time t_0	o,q at t_1	Update Answer Set	Result Updates	Illustration
1.	$o \in m$ and $q \in m$	$(q, o, m.Cid) \in m.Ans$	$o \in q$	-	-	Fig. 7b
2.	$o \in m$ and $q \in m$	$(q, o, m.Cid) \in m.Ans$	$o \notin q$	$(q, o, m.Cid) \notin m.Ans$	(q,-o)	Fig. 7c
3.	$o \in m'$ and $q \in m$	$(q, o, m.Cid) \in m.Ans$	$o \in q$	$(q, o, m'.Cid) \in m.Ans$	-	Fig. 7d
4.	$o \in m'$ and $q \in m$	$(q, o, m.Cid) \in m.Ans$	$o \notin q$	$(q, o, m.Cid) \notin m.Ans$	(q,-o)	Fig. 7e
5.	$o \in m'$ and $q \in m$	$(q, o, m.Cid) \notin m.Ans$	$o \in q$	$(q, o, m'.Cid) \in m.Ans$	(q,+o)	Fig. 7f
6.	$o \in m'$ and $q \in m$	$(q, o, m.Cid) \notin m.Ans$	$o \notin q$	-	-	-

Table 2. Cases for Incremental Evaluation: ***Join-Within*** Moving Clusters

describes if they are currently intersecting (at time t_1). *"Join-Within Needed"* column specifies if *join-within* needs to be performed after this current *join-between*, while the *"Results Updates"* column describes the types of result updates that would be sent to the output stream. Column *"Illustration"* names the figure that graphically illustrates this case. For example, consider Case 1. If at time t_0 and at time t_1 two clusters m_1 and m_2 are not overlapping, no further processing is needed and no result updates are sent[9].

Incremental Join-Within: To illustrate incremental *join-within* moving clusters, consider a query q which belongs to a cluster m and a moving object o which at time t_0 was in the same cluster m as q, but at time t_1 may belong to either the same cluster (m) or to any moving cluster currently intersecting with m (e.g., m') (Fig. 7). ClusterSheddy, at time t_1, distinguishes among six cases (Table 2)[10]. Column *"Clustering at time t_1"* describes which moving clusters the object o and query q belong to at time t_1. *"$m.Ans$ at time t_0"* column describes the result set associated with cluster m for query q. Whether o still satisfies q at time t_1 is depicted in column *"o, q at t_1"*. Column *"Update Answer Set"* describes which cluster result set has to be updated at time t_1. Finally, *"Result Updates"* depicts the types of result updates that would be sent, and *"Illustration"* names the figure for that case. For example, in Case 1, o and q are in the same cluster m at both time t_0 and t_1. At time t_1 o still satisfies q. Since only the updates of the previously reported results are processed, o is neither processed nor is any result sent to the output stream[11].

3 Cluster-Based Load Shedding

Load shedding of moving clusters is an optimization problem composed of three sub-problems: (1) how to effectively estimate the current system load. This implicitly determines when we have to load shed to avoid system overload; (2) how to determine which clusters to load shed. Is it better to shed *all* moving clusters equally or to target a *subset* of moving clusters to more effectively reduce the load, while minimizing the overall inaccuracy; (3) how much to shed per cluster,

[9] For brevity of discussion, we skip the detailed discussion of each case.
[10] We did not include the figure for case 6 as it is trivial.
[11] We omit the detailed discussion of each case, as Table 2 illustrates the main ideas of each case.

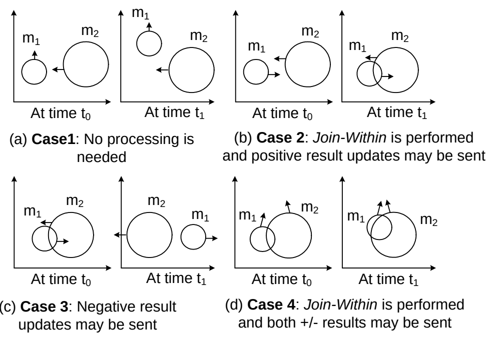
m1
m2
At time t0
At time t1
(a) **Case1**: No processing is needed
(b) **Case 2**: *Join-Within* is performed and positive result updates may be sent
(c) **Case 3**: Negative result updates may be sent
(d) **Case 4**: *Join-Within* is performed and both +/- results may be sent

Fig. 6. ***Join-Between*** Clusters.

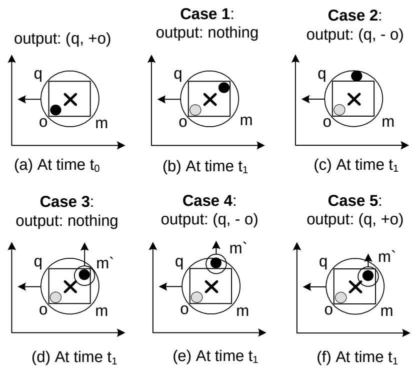
output: (q, +o)
Case 1: output: nothing
Case 2: output: (q, - o)
Case 3: output: nothing
Case 4: output: (q, - o)
Case 5: output: (q, +o)
q
o
m
m`
(a) At time t0
(b) At time t1
(c) At time t1
(d) At time t1
(e) At time t1
(f) At time t1

Fig. 7. ***Join-Within*** Clusters.

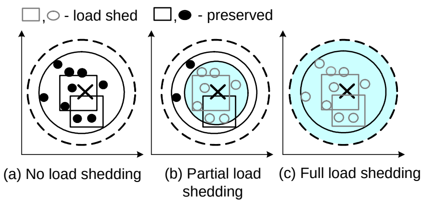

Fig. 8. Effect of Nucleus Size on Load Shedding

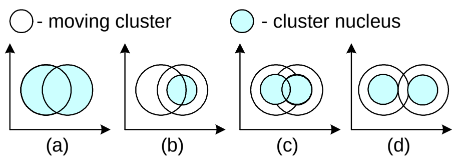

Fig. 9. Affect of Nucleus Size on Join Execution

so that the introduced relative error in the final query results is minimized. We address these three questions next.

3.1 Load Shedding Via Cluster Nucleus

A *nucleus* of a cluster is a circular region that approximates the positions of the members near the centroid of the cluster. Individual positions of objects and queries inside the nucleus are load shed[12]. The imprecision is directly proportional to the size of the nucleus. The larger the nucleus, the more imprecise the query results may become.

Definition 3. *(Cluster Nucleus) Given the nucleus threshold Θ_N where $0 \leq \Theta_N \leq \Theta_D$, let m be a moving cluster. Then the cluster nucleus m.Nucl is the subset of m ($m.Nucl \subseteq m$), where $\forall$ objects $o_i \in m.Nucl$, $|o_i.Loc - m.Loc| \leq \Theta_N$ and $\forall$ queries $q_j \in m.Nucl$, $|q_j.Loc - m.Loc| \leq \Theta_N$.*

Fig. 8 depicts the effect of increasing a cluster nucleus on the amount of objects/queries preserved. In Fig. 8a, no load shedding is performed ($\Theta_N = 0$). In Fig. 8b, Θ_N is increased, and seven objects and one query are load shed. Fig. 8c illustrates the extreme case, when $\Theta_N = \Theta_D$ (maximum possible size of the cluster), where all cluster members are discarded. Even if a new member were to arrive to the cluster, it would not be preserved, but automatically discarded. Nucleus threshold Θ_N may be set either globally for all moving clusters, or individually for each moving cluster, if a finer granularity shedding is needed.

Query Processing With Shedded Clusters: By discarding moving entities from the nuclei and not knowing their precise locations, we make several assumptions when executing a cluster-based join. Objects that fall into the intersection region with a nucleus are assumed to satisfy all the queries from the nucleus. Similar reasoning is applied to queries. Fig. 9 depicts the cases with the intersecting clusters and their nuclei.

[12] To load shed a query, the nucleus must fully overlap with the query region.

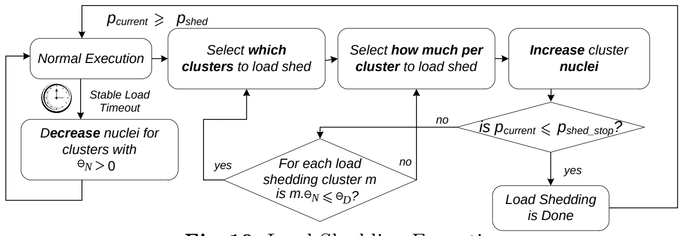
ρcurrent ⩾ ρshed
Normal Execution
Select which clusters to load shed
Select how much per cluster to load shed
Increase cluster nuclei
Stable Load Timeout
Decrease nuclei for clusters with ΘN > 0
no
is ρcurrent ⩽ ρshed_stop?
yes
For each load shedding cluster m is m.ΘN ⩽ ΘD?
yes
no
Load Shedding is Done

Fig. 10. Load Shedding Execution

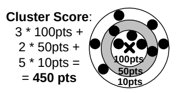

Fig. 11. *Score-Based Policy*: calculating a score for a moving cluster

3.2 Estimating When To Load Shed

Fig. 10 depicts the main steps of load shedding execution. Load shedding is initiated when ClusterSheddy utilization becomes greater than or equal to the ρ_{shed} threshold. First, we determine *which* moving clusters are to be shed (termed *shedding clusters*). We then determine *how much per* each shedding cluster to load shed. This is repeated until the current utilization $\rho_{current}$ becomes less than or equal to ρ_{shed_stop}. We use queuing theory [4] and *Little's Law*[13] to predict system overflow[13]. Queueing theory has been widely used to model and analyze the performance of complex systems involving service. For details of queueing theory see [4, 13].

3.3 Estimating Which Clusters To Load Shed

We divide the approaches for picking *which* moving clusters to load shed into two broad categories: *uniform* and *selective*.

Uniform Load Shedding Policies: With uniform load shedding, we load shed across all moving clusters in the system. *Each* moving cluster gets affected by the load shedding procedure, and some or all data within the cluster is discarded. Uniform load shedding policy does *not* mean that all cluster nuclei will increase by *equal* amount. It merely means that all clusters will participate in load shedding and will shed something.

Selective Load Shedding Policies: Alternatively, in selective load shedding, some clusters are chosen for load shedding to minimize the overall inaccuracy of the answers. The different selective policies may include: (1) *Random Policy* – selecting clusters at random; (2) *Count-Based Policy* – selecting clusters with the highest number of members; (3) *Size-Based Policy* – selecting clusters with the smallest size to minimize the overall inaccuracy; (4) *Score-Based Policy* – selecting clusters with the highest scores (see Fig. 11), thus favoring clusters where members are distributed near the centroid regardless of the cluster size; and (5) *Volatility Policy* – selecting clusters with the lowest volatility (i.e., clusters undergoing fewer changes to their properties). The motivation is that stable clusters, once load shed, can accurately approximate their members for longer time intervals, thus amortizing the load shedding overhead in the long term.

3.4 Estimating How Much Per Cluster to Shed

Once clusters have been selected for load shedding, the next question that needs to be addressed is *how much per cluster* to discard, which is determined by the increase in Θ_N. Clusters may either discard all (*total drop*) or only a subset (*partial drop*) of their members.

[13] However, other models for predicting system overload could be similarly be used.

Total drop policy causes Θ_N to expand to its maximum (Θ_D), and all members inside the clusters are discarded. Total drop is a simple way to quickly reduce the load. However it may significantly impact the accuracy of the results, as we may be discarding data unnecessarily. Partial drop policy is a more prudent way of discarding data. It allows to drop just enough to alleviate the burden on the system without shedding more than necessary.

In the face of detected overload, ClusterSheddy increases cluster nuclei in the *shedding clusters*. The algorithm begins in the exponential growth phase, and increases the nucleus size exponentially[14]. Once ClusterSheddy perceives that there is no longer an overload, it starts to *decrease* the nuclei. The rationale for this is that the utilization is below the load shedding threshold, and ClusterSheddy could be storing precise locations and producing more accurate query results. It does this by decreasing nuclei a little each time after a periodic timeout, termed *stable load timeout*. For simplicity of presentation, we assume that nuclei radii are decremented by some constant spatial unit amount k each time. Thus ClusterSheddy *multiplicatively increases* cluster nuclei when it detects that utilization is aproaching the overload threshold, and *additively decreases* the cluster nuclei when the utilization is low.

4 Experimental Study

Our experiments are based on a real implementation of ClusterSheddy in the Java-based CAPE continuous query engine [23] running on Intel Pentium IV CPU 2.4GHz with 1GB RAM on Windows XP and 1.5.0.06 Java SDK. We use the *Moving Objects Generator* [5] to generate continuously moving objects in the city of Worcester, MA, USA in the form of data streams. We begin with 20K of moving objects and each time unit 1K of new moving objects enter the system. Without loss of generality, all presented experiments are conducted using spatio-temporal range queries. We control skewness of the data and set the *skew factor* to 100. Hence, on average 100 objects are in a cluster. The values of the threshold parameters were set as follows: speed threshold $\Theta_S = 10$ spatial units/time units, distance threshold $\Theta_D = 100$ spatial units, direction threshold $\Theta_R = 10$ degrees, and time threshold $\Theta_T = 1$ time unit. All experimental runs begin with the nucleus threshold set to zero ($\Theta_N = 0$), i.e., no load shedding.

4.1 Incremental vs. Non-incremental Query Evaluation

In this experiment, we compared SCUBA and ClusterSheddy (without load shedding). We wanted to see how the performance and memory consumption are affected when executing spatio-temporal queries using incremental versus non-incremental cluster-based technique (Fig. 12a). We observe that in the long term, ClusterSheddy incremental strategy gives a better performance and requires less memory. The advantage that ClusterSheddy has over SCUBA is that if dense clusters overlap for long durations of time and objects and queries don't change their relative positions within the clusters, the re-evaluation of the contained queries in those clusters is not needed. This translates into significant savings in processing time. Memory-wise, ClusterSheddy also has an advantage over SCUBA, because fewer computations are made and no redundant answers are produced.

[14] This is a similar approach as in TCP congestion control mechanisms [15]

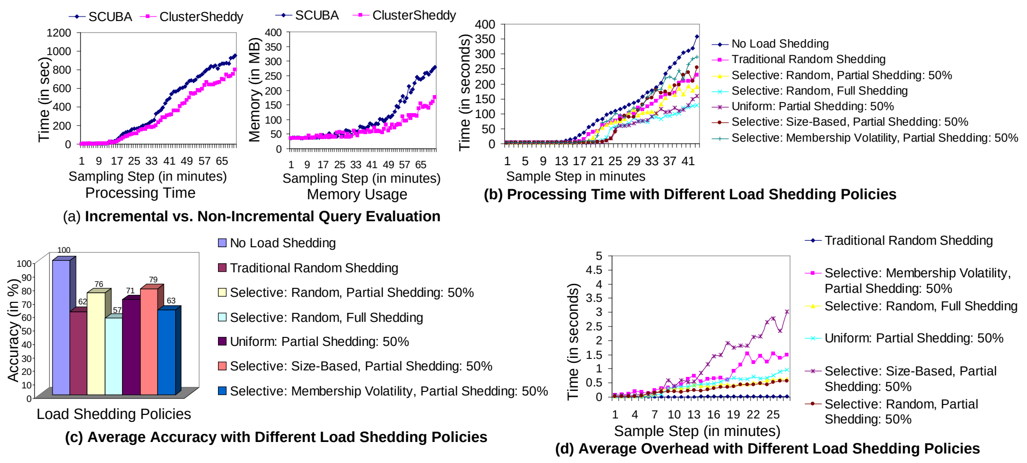

Fig. 12. Experiemental Evaluations

4.2 Load Shedding Policies Comparison

We compared different cluster-based load shedding policies[15] to no load shedding and traditional random load shedding. Figures 12b and 12c respectively represent the join processing times and the accuracy measurements.

Best Performance: We observe that selective random policy with full load shedding gives the best performance but the lowest average accuracy ($\approx 57\%$). With this policy, clusters are randomly selected and all their cluster members are discarded. The processing overhead is small since clusters are chosen randomly, but the accuracy suffers. Any new objects joining the shedded clusters are assumed to satisfy all queries in the cluster. For any new queries joining these clusters, all cluster objects are returned as satisfying these queries.

Best Accuracy: The best accuracy ($\approx 79\%$) was achieved with the selective size-based partial load shedding policy. Here the smallest clusters were selected first and their nuclei were increased by 50%.

Worst Performance: The worst performance was seen when using selective membership volatility policy with partial shedding. With this policy we picked the clusters that were more stable. However, picking stable clusters did not give much advantage. Very dense clusters may be very dynamic, thus we may not be able to reduce load fast. The membership volatility doesn't account for the count and the distribution of the members within the clusters, hence accuracy may suffer as well.

Overall, if both performance and high accuracy are desired, selective random policy with partial shedding or uniform policy with partial shedding may be used. The former gives a better accuracy ($\approx 76\%$), but has slightly worse performance than the latter policy which has a lower average accuracy ($\approx 71\%$).

4.3 Load Shedding Cost

We compared the load shedding overhead for different policies (Fig. 12d). Load shedding overhead cost includes the time to pre-process the clusters before the

[15] For all full shedding strategies, cluster nucleus is increased to the maximum Θ_D. For all partial shedding strategies, cluster nucleus is increased by 50% each time with respect to the *current* (at the time of shedding) size of the cluster.

shedding is initiated. This may include finding the clusters based on the policy selection criteria (e.g., largest, smallest, most dense, etc.), determining and increasing the nucleus size, associating the ids of the shedded objects and queries with the nucleus, etc. Fig. 12d shows that selective size-based policy has the highest overhead compared to other policies. This is due to the fact that we classify clusters into smaller and larger clusters based on the distribution of their members and also determine if a larger cluster may be an *outlier* cluster[16]. Processing the cluster members to see if any latest update caused the cluster to become an "outlier" cluster may require some additional CPU and memory.

5 Related Work

The current state-of-the-art in load shedding includes [1, 2, 9, 22, 26–29]. Load shedding on streaming data has first been presented by Aurora [26, 27]. Aurora shedder relies on a quality of service function that specifies the utility of a *tuple*. This is best suited for applications where the probability of receiving each individual tuple in a query result is independent of the other tuples' values, an assumption that does not hold for spatio-temporal queries.

Load shedding for aggregation queries was addressed in [2]. Babcock et al. describe a load shedding technique based on random sampling. Although sampling works well for aggregation on a traditional data, sampling on location updates without considering their actual values – such as their location – may omit some of the moving entities (when selecting a sample), leading to higher inaccuracy.

The probably most closely related work to ours is the *Scalable On-Line Execution* (SOLE) algorithm [17] performing load shedding on spatio-temporal data streams in PLACE server. In SOLE, specific objects marked as significant are kept, and the other objects are discarded. However, SOLE is not designed to deal accordingly with dense and highly overlapping spatio-temporal data, as objects satisfying many queries are termed as significant and thus are not load shed. ClusterSheddy addresses these shortcomings. In fact, it exploits such spatial closeness to approximate the locations when load shedding is performed.

ClusterSheddy extends our earlier work – SCUBA algorithm [20], which introduced the concept of moving clusters as abstractions on moving objects. While SCUBA only provides full result recomputation, ClusterSheddy now also support incremental query evaluation. Most importantly, ClusterSheddy focuses on load shedding – a topic not addressed by SCUBA.

6 Conclusions

This paper addresses an important problem faced in continuous querying of spatio-temporal data streams: system capacity overload. We proposed moving cluster-based load shedding, called ClusterSheddy which uses common spatio-temporal properties to determine which objects' updates would be least sensitive to load shedding and have minimum adverse impact on the accuracy of query answers. The proposed technique is general, because moving objects in

[16] We term a cluster an *outlier* cluster, if the majority of its members are distributed near the centroid with an exception of a single member that is at a far distance, thus causing the size of the cluster to increase.

practice tend to share spatio-temporal properties with other objects for some time intervals. Our experimental results show that ClusterSheddy compared to traditional non-spatio-temporal load shedding is efficient in reducing the load while maintaining good accuracy of results.

References

1. B. Babcock, M. Datar, and R. Motwani. Load shedding techniques for data stream systems. In *MPDS: Workshop on Management and Processing of Data Streams*, 2003.
2. B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. In *ICDE*, pages 350–361, 2004.
3. D. Barbará, W. DuMouchel, and et. al. The new jersey data reduction report. *IEEE Data Eng. Bull.*, 20(4), 1997.
4. G. Bolch and et. al. *Queueing Networks and Markov Chains : Modeling and Performance Evaluation With Computer Science Applications*. John Wiley and Sons, Inc., 1998.
5. T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.
6. D. Carney, U. Çetintemel, and et. al. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.
7. S. Chu. The influence of urban elements on time-pattern of pedestrian movement. In *The 6th Int. Conf. on Walking in the 21st Cent.*, 2005.
8. A. Das, J. Gehrke, and et. al. Semantic approximation of data stream joins. *IEEE Trans. Knowl. Data Eng.*, 17(1):44–59, 2005.
9. A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, pages 40–51, 2003.
10. R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley-Interscience Publication, 2000.
11. J. A. Hartigan. *Clustering Algorithms*. John Wiley and Sons, 1975.
12. A. K. Jain, M. N. Murthy, and P. J. Flynn. Data clustering: A review. Technical Report MSU-CSE-00-16, Department of Computer Science, Michigan State University, East Lansing, Michigan, August 2000.
13. R. K. Jain. *The Art of Computer Systems Performance Analysis : Techniques for Experimental Design, Measurement, Simulation, and Modeling*. John Wiley and Sons, Inc., 1991.
14. P. Kalnis, N. Mamoulis, and et. al. On discovering moving clusters in spatio-temporal data. In *SSTD*, pages 364–381, 2005.
15. J. F. Kurose and K. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
16. B. Liu, Y. Zhu, and E. Rundensteiner. Run-time operator state spilling for memory intensive continuous queries. In *SIGMOD Conference*, pages 347–358, 2006.
17. M. F. Mokbel and W. G. Aref. Sole: Scalable online execution of continuous queries on spatio-temporal data streams. tr csd-05-016. Technical report, Purdue University, 2005.
18. M. F. Mokbel, W. G. Aref, and et. al. Towards scalable location-aware services: requirements and research issues. In *GIS*, pages 110–117, 2003.
19. M. F. Mokbel, X. Xiong, and et. al. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, pages 623–634, 2004.
20. R. V. Nehme and E. A. Rundensteiner. Scuba: Scalable cluster-based algorithm for evaluating continuous spatio-temporal queries on moving objects. In *EDBT*, pages 1001–1019, 2006.
21. S. Prabhakar and et. al. Query indexing and velocity constrained indexing: Scalable techniques for continuous queries on moving objects. *IEEE Trans. Computers*, 51(10), 2002.
22. F. Reiss and J. M. Hellerstein. Data triage: An adaptive architecture for load shedding in telegraphcq. In *ICDE*, pages 155–156, 2005.
23. E. A. Rundensteiner, L. Ding, and et. al. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, pages 1353–1356, 2004.
24. M. Shah, J. Hellerstein, and et. al. Flux: An adaptive partitioning operator for continuous query systems. cs-02-1205. Technical report, U.C. Berkeley, 2002.
25. A. P. Sistla, O. Wolfson, and et. al. Modeling and querying moving objects. In *ICDE*, pages 422–432, 1997.
26. N. Tatbul. Qos-driven load shedding on data streams. In *XMLDM*, pages 566–576, 2002.
27. N. Tatbul, U. Çetintemel, and et. al. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.
28. N. Tatbul and S. B. Zdonik. Window-aware load shedding for aggregation queries over data streams. In *VLDB*, pages 799–810, 2006.
29. Y.-C. Tu, S. Liu, S. Prabhakar, and B. Yao. Load shedding in stream databases: A control-based approach. In *VLDB*, pages 787–798, 2006.
30. T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Eng. Bull.*, 23(2), 2000.
31. O. Wolfson, H. Cao, and et. al. Management of dynamic location information in domino. In *EDBT*, pages 769–771, 2002.
32. X. Xiong, M. F. Mokbel, and et. al. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, 2005.