SCUBA: Scalable Cluster-Based Algorithm for Evaluating Continuous Spatio-Temporal Queries on Moving Objects

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

[1]Department of Computer Science, Purdue University
[2]Department of Computer Science, Worcester Polytechnic Institute
rnehme@cs.purdue.edu, rundenst@cs.wpi.edu

Abstract. In this paper, we propose, SCUBA, a Scalable Cluster Based Algorithm for evaluating a large set of continuous queries over spatio-temporal data streams. The key idea of SCUBA is to group moving objects and queries based on common spatio-temporal properties at run-time into moving clusters to optimize query execution and thus facilitate scalability. SCUBA exploits shared cluster-based execution by abstracting the evaluation of a set of spatio-temporal queries as a spatial join first between moving clusters. This cluster-based filtering prunes true negatives. Then the execution proceeds with a fine-grained within-moving-cluster join process for all pairs of moving clusters identified as potentially joinable by a positive cluster-join match. A moving cluster can serve as an approximation of the location of its members. We show how moving clusters can serve as means for intelligent load shedding of spatio-temporal data to avoid performance degradation with minimal harm to result quality. Our experiments on real datasets demonstrate that SCUBA can achieve a substantial improvement when executing continuous queries on spatio-temporal data streams.

1 Introduction

Every day we witness technological advances in wireless communications and positioning technologies. Thanks to GPS, people can avoid congested freeways, businesses can manage their resources more efficiently, and parents can ensure their children are safe. These developments paved the way to a tremendous amount of research in recent years in the field of real-time streaming and spatio-temporal databases [11, 14, 20, 29, 33]. As the number of users of location-based devices (e.g., GPS) continues to soar, new applications dealing with extremely large numbers of moving objects begin to emerge. These applications, faced with limited system resources and near-real time response obligation call for new real-time spatio-temporal query processing algorithms [23]. Such algorithms must efficiently handle extremely large numbers of moving objects and efficiently process large numbers of continuous spatio-temporal queries.

Many recent research works try to address this problem of efficient evaluation of continuous spatio-temporal queries. Some focus on indexing techniques [14, 20, 32, 37], other on shared execution paradigms [24, 29, 39], yet others on special algorithms [34, 27]. A major shortcoming of these existing solutions, however, is that most of them still process and materialize every location update individually. Even in [24, 39] where authors exploit a *shared execution* paradigm among all queries, when performing a join, each moving object and query is ultimately processed individually. With an extremely large number of objects and queries, this may simply become impossible.

Here we now propose a two-pronged strategy towards combating this scalability problem. Our solution is based on the fact that in many applications objects naturally move in clusters, including traffic jams, animal and bird migrations, groups of children on a trip or people evacuating from danger zones. Such moving objects tend to have some common motion related properties (e.g., speed and destination). In [41] Zhang et. al. exploited *micro-clustering* for data summarization i.e., grouping data that are so close to each other that they can be treated as one unit. In [22] Li et. al. extended this concept to *moving micro-clusters*, groups of objects that are not only close to each other at a current time, but also likely to move together for a while. These works focus on finding interesting patterns in the movements. We take the concept of *moving micro-clusters*[1] further, and exploit this concept towards the optimization of the execution of the spatio-temporal queries on moving objects.

We propose the *Scalable Cluster-Based Algorithm* (SCUBA) for evaluating continuous spatio-temporal queries on moving objects. SCUBA exploits a *shared cluster-based execution* paradigm, where moving objects and queries are grouped together into moving clusters based on common spatio-temporal attributes. Then execution of queries is abstracted as a *join-between* clusters and a *join-within* clusters executed periodically (every Δ time units). In *join-between*, two clusters are tested for overlap (i.e., if they intersect with each other) as a cheap pre-filtering step. If the clusters are filtered out, the objects and queries belonging to these clusters are guaranteed to not join at an individual level. Thereafter, in *join-within*, individual objects and queries inside clusters are joined with each other. This two-step filter-and-join process helps reduce the number of unnecessary spatial joins. Maintaining clusters comes with a cost, but our experimental evaluations demonstrate it is much cheaper than keeping the complete information about individual locations of objects and queries and processing them individually.

If in spite of our cheap pre-filtering step, the query engine still cannot cope with the current query workload due to the limited system resources, the results may get delayed and by the time they are produced probably become obsolete. This can be tackled by shedding some data and thus reducing the work to be done. The second contribution of this work is the application of moving clusters as means for intelligent load shedding of spatio-temporal data. Since clusters serve as summaries of their members, individual locations of the members can be discarded if need be, yet would still be sufficiently approximated from the location of the their cluster centroid. The closest to the centroid members are abstracted into a nested structure called *cluster nucleus*, and their positions are load shed. The nuclei serve as approximations of the positions of their mem-

[1] We use the term *moving clusters* in this paper.

bers in a compact form. To the best of our knowledge this is the first work that exploits moving clustering as means to perform intelligent load shedding of spatio-temporal data.

For simplicity, we present our work in the context of continuous spatio-temporal range queries. However, SCUBA is applicable to other types of spatio-temporal queries (e.g., *knn* queries, trajectory and aggregate queries). Since clusters themselves serve as summaries of the objects they contain (i.e., aggregate) based on objects' common properties. This can facilitate in answering some of the aggregate queries. For knn queries, moving clusters that are not intersecting with other moving clusters and contain at least k members can be assumed to contain nearest members of the query object.

The contributions of this paper are the following:

1. We describe the incremental cluster formation technique that efficiently forms clusters at run-time. Our approach assures longevity and quality of the motion clusters by utilizing two key thresholds, namely distance threshold Θ_D and speed threshold Θ_S.
2. We propose SCUBA - a first of its kind cluster-based algorithm utilizing dynamic clusters for optimizing evaluation of spatio-temporal queries. We show how the cluster-based execution with the two-step filtering approach reduces the number of unnecessary joins and improves query execution on moving objects.
3. We describe how moving clusters can naturally be applied as means for intelligent load shedding of spatio-temporal data. This approach avoids performance degradation with minimal harm to result quality.
4. We provide experimental evidence on real datasets that SCUBA improves the performance when evaluating spatio-temporal queries on moving objects. The experiments evaluate the efficiency of incremental cluster formation algorithm, query execution and load shedding.

The rest of the paper is organized as follows: Section 2 is background on the motion model. The essential features of moving clusters are described in Section 3. Section 4 introduces join algorithm using moving clusters. Moving cluster-driven load shedding is presented in Section 5. Section 6 describes experimental evaluation. Section 7 discusses related work, while Section 8 concludes the paper.

2 Background on the Motion Model

We employ a similar motion model as in [22, 34], where moving objects are assumed to move in a piecewise linear manner in a road network (Fig. 1). Their movements are constrained by roads, which are connected by *network nodes*, also known as *connection nodes*[2].

We assume moving objects' location updates arrive via data streams and have the following form (*o.OID*, $o.Loc_t$, *o.t*, *o.Speed*, *o.CNLoc*, o.Attrs), where *o.OID* is the id of the moving object, $o.Loc_t$ is the position of the moving object,

[2] Our solution relies on the fact that objects have common spatio-temporal properties independent of whether objects move in the network or not, and is applicable to both constrained and unconstrained moving objects.

o.t is the time of the update, *o.Speed* is the current speed, and *o.CNLoc* is the position of the connection node in the road network that next be reached by the moving object (its current destination). We assume that an *CNLoc* of the object doesn't change before the object reaches this connection node, i.e., the network is stable. *o.Attrs* is a set of attributes describing the object (e.g., child, red car).

A continuously running query is represented in a similar form (*q.QID*, $q.Loc_t$, *q.t*, *q.Speed*, *q.CNLoc*, *q.Attrs*). Unlike for the objects, *q.Attrs* represents a set of query-specific attributes (e.g., size of the range query)

3 Overall Moving Cluster Framework

3.1 The Notion of Moving Clusters

A *moving cluster* (Fig. 2) abstracts a set of moving objects and moving queries. Examples include a group of cars travelling on a highway, or a herd of migrating animals. We group both moving objects and moving queries into *moving clusters* based on common spatio-temporal properties i.e., with the intuition that the grouped entities[3] travel closely together in time and space for some period. We consider the following attributes when grouping moving objects and queries into clusters: (1) speed, (2) direction of the movement (e.g., connection node on the road network), (3) relative spatial distance, and (4) time of when in that location. Moving objects and queries that don't satisfy conditions of any other existing clusters form their own clusters, *single-member moving clusters*. As objects and queries can enter or leave a moving cluster at any time, the properties of the cluster are adjusted accordingly (Section 3.2).

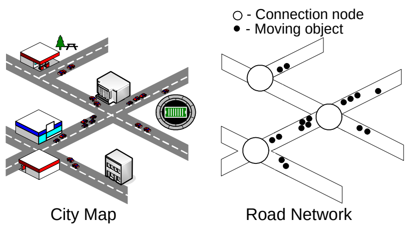


Fig. 1. Road Network Representation

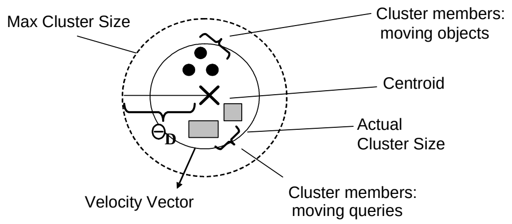


Fig. 2. Moving Cluster in SCUBA

A moving cluster *m* at time *t* is represented in the form (*m.CID*, $m.Loc_t$, *m.n*, *m.OIDs*, *m.QIDs*, *m.AveSpeed*, *m.CNLoc*, *m.R*, *m.ExpTime*), where *m.CID* is the moving cluster id, $m.Loc_t$ is the location of the centroid of the cluster at time *t*, *m.n* is the number of moving objects and queries that belong to this cluster, *m.OIDs* and *m.QIDs* are the collections of id's and relative positions of the moving objects and queries respectively that belong to this moving cluster, *m.AveSpeed* is the average speed of the cluster, *m.CNLoc* is the cluster destination, *m.R* is the size of the radius, and *m.ExpTime* is the "expiration" time of the cluster (for instance, this may be the time when the cluster reaches the *m.CNLoc* travelling at *m.AveSpeed*). The motivation behind the "expiration" time of the cluster is the fact that once a cluster reaches its *m.CNLoc* (which may represent

[3] By entities we mean both moving objects and queries.

a major road intersection) its members may change their spatio-temporal properties significantly (e.g., move in different directions) and thus no longer belong to the same cluster. Alternate options are possible here (e.g., splitting a moving cluster). We plan to explore this as a part of our future work.

Individual positions of moving objects and queries inside a cluster are represented in a relative form using polar coordinates (with the pole at the centroid of the cluster). For any location update point P its polar coordinates are (r, θ), where r is the radial distance from the centroid, and θ is the the counterclockwise angle from the x-axis. As time progresses, the center of the cluster might shift, thus making it necessary to transform the relative coordinates of the cluster members. We maintain a transformation vector for each cluster that records the changes in position of the centroid between the periodic executions. We refrain from constantly updating the relative positions of the cluster members, as this info is not needed, unless a *join-within* is to be performed (Fig. 3).

We face the challenge that with time clusters may deteriorate [15]. To keep a competitive and *high quality* clustering (i.e., clusters with compact sizes), we set the following thresholds to limit the sizes and deterioration of the clusters as the time progresses: (1) *distance threshold* (Θ_D) and (2) *speed threshold* (Θ_S). Distance threshold guarantees that the clustered entities are close to each other at the time of clustering, while the speed threshold assures that the entities will stay close to each other for some time in the future. The thresholds prevent dissimilar moving entities from being classified under the same cluster and ensure that good quality clusters will be formed.

Clusters are *dissolved* once they reach their destination points. So if the distance between the location where the cluster has been formed and its destination is short, the clustering approach might be quite expensive and not as worthwhile. The same reasoning applies if the average speed of the cluster is very fast and it thus reaches its destination point very quickly, then forming a cluster might not give very little, if any, advantages. In a typical real-life scenario though, moving objects can reach relatively high speeds on the larger roads (e.g., highways), where connection nodes would be far apart from each other. On the smaller roads, speed limit, and the proximity of other cars constrains the maximum speed the objects can develop, thus extending the time it takes for them to reach the connection nodes. These observations support our intuition that clustering is applicable to different speed scenarios for moving objects in every day life.

3.2 Moving Cluster Formation

We adapt an incremental clustering algorithm, similar to the *Leader-Follower* clustering [8, 16], to create and maintain moving clusters in SCUBA. Incremental clustering allows us not to store all the location updates that are to be grouped into the clusters. So the space requirements are small compared to the non-incremental algorithms. Also once Δ expires, SCUBA can immediately proceed with the query execution, without spending any time on re-clustering the entire data set. However, incremental clustering makes local one-at-a time decisions and its outcome is in part dependent on the arrival order of updates.

We experimentally evaluate the tradeoff between the execution time and clustering quality when clustering location updates incrementally as updates arrive vs. non-incrementally when the entire data set is available (Sec. 6.4).

We now will illustrate using an example of a moving object how moving entities get clustered. A spatial grid index (we will refer to it as *ClusterGrid*) is used to optimize the process of clustering. When a location update from the moving object o arrives, the following five steps determine the moving cluster it belongs to:

Step 1: Use moving object's position to probe the spatial grid index *ClusterGrid* to find the moving clusters (S_c) in the proximity of the current location (i.e., clusters that the object can potentially join).

Step 2: If there are no clusters in the grid cell ($S_c = \emptyset$), then the object forms its own cluster, with the centroid at the current location of the object, and the radius = 0;

Step 3: If otherwise, there are clusters that the object can potentially join, we iterate through the list of the clusters S_c and for each moving cluster $m_i \in S_c$ check the following properties:

1. Is the moving object moving in the same direction as the cluster m_i ($o.CNLoc == m_i.CNLoc$)?
2. Is the distance between the centroid of the cluster and the location update less than the distance threshold, that is $|o.Loc_t - m_i.Loc_t| \leq \Theta_D$?
3. Is the speed of the moving object less than the speed threshold, that is $|o.Speed - m_i.AveSpeed| \leq \Theta_S$?

Step 4: If the moving object o satisfies all three conditions in Step 3, then the moving cluster m_i *absorbs* o, and adjusts its properties based on o's attributes. The cluster centroid position is adjusted by considering the new relative position of object o. The average speed gets recomputed. If the distance between the object o and the cluster centroid is greater than the current radius, the radius is increased. Finally, the count of the cluster members is incremented.

Step 5: If o cannot join any existing cluster (from Step 4), o forms its own moving cluster.

Critical situations (e.g., each moving cluster contains one object or one big moving cluster contains all moving objects) are rare to happen. If such situation does in fact occur, then our solution can default to any other state-of-the-art moving objects processing technique without any savings offered by our solution.

4 Join Algorithm Using Moving Clusters

In this section, we describe the joining methods utilized by SCUBA to minimize the cost of execution of spatio-temporal queries. The main idea is to group similar objects as well as queries into moving clusters, and then the evaluation of a set of spatio-temporal queries is abstracted as a spatial join, first *between* the moving clusters (which serves as a filtering step) and then *within* the moving clusters (Fig. 3). To illustrate the idea, traditionally each individual query is evaluated separately. In the *shared execution* paradigm the problem of evaluating numerous spatio-temporal queries is abstracted as s spatial join between the set of moving objects and queries [39]. While a shared plan allows processing with only one scan, however objects and queries are still joined individually. With large numbers of objects and queries, this may still create a bottleneck in performance and may cause us to potentially run out of memory. The *shared cluster-based execution* groups moving entities into moving clusters and a spatial join is performed on all moving clusters. Only if two clusters overlap, we have to go to the individual object/query level of processing, or automatically assume that objects and queries within those clusters produce join results (Fig. 4).

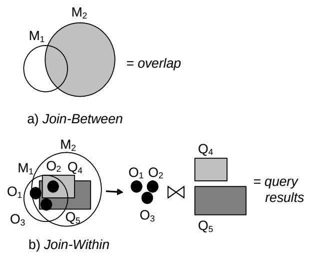
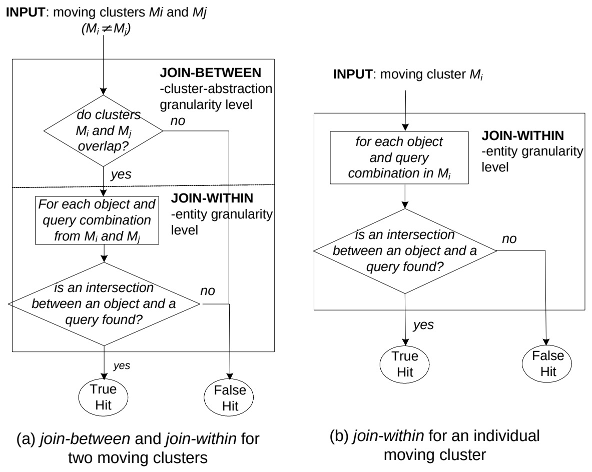


Fig. 3. *Join-Between* and *Join-Within* moving clusters



INPUT: moving clusters Mi and Mj (Mi ≠ Mj)
JOIN-BETWEEN -cluster-abstraction granularity level
do clusters Mi and Mj overlap?
no
yes
For each object and query combination from Mi and Mj
JOIN-WITHIN -entity granularity level
is an intersection between an object and a query found?
no
yes
True Hit
False Hit
(a) join-between and join-within for two moving clusters
INPUT: moving cluster Mi
for each object and query combination in Mi
JOIN-WITHIN -entity granularity level
is an intersection between an object and a query found?
no
yes
True Hit
False Hit
(b) join-within for an individual moving cluster

Fig. 4. *Join-Between* and *Join-Within* workflows

4.1 Data Structures

In the course of execution, SCUBA maintains five in-memory data structures (Fig. 5): (1) *ObjectsTable*, (2) *QueriesTable* (3) *ClusterHome*, (4) *ClusterStorage*, and (5) *ClusterGrid*.

ObjectsTable stores the information about objects and their attributes. An object entry in the *ObjectsTable* has the form (*o.OID*, *o.Attrs*), where *o.OID* is the object id and *o.Attrs* is the list of attributes that describe the object. Similarly, *QueriesTable* stores the information about queries, and has the form (*q.QID*, *q.Attrs*) where *q.QID* is the query id, and the *q.Attrs* is the list of query attributes. *ClusterHome* is a hash table that keeps track of the current relationships between objects, queries and their corresponding clusters. A moving object/query can belong to only one cluster at a time *t*. An entry in the *ClusterHome* table is of the following form (*ID*, *type*, *CID*), where *ID* is the id of a moving entity, *type* indicates whether it's an object or a query, and the *CID* is the id of the cluster that this moving entity belongs to. *ClusterStorage* table stores the information (e.g., centroid, radius, member count, etc.) about moving clusters. *ClusterGrid* is a spatial grid table dividing the data space into N x N grid cells. For each grid cell, *ClusterGrid* maintains a list of cluster ids of moving clusters that overlap with that cell.

4.2 The SCUBA Algorithm

SCUBA execution has three phases: (1) *cluster pre-join maintenance*, (2) *cluster-based joining*, and (3) *cluster post-join maintenance* as depicted in Fig. 6. The cluster pre-join maintenance phase is continuously running where it receives incoming information from moving objects and queries and applies in-memory

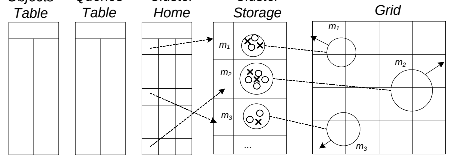


Fig. 5. Data Structures in SCUBA

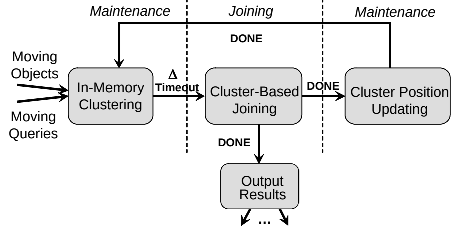


Fig. 6. State Diagram

Algorithm 1 *SCUBA()*

```
1: loop
2:   //*** CLUSTER PRE-JOIN MAINTENANCE PHASE ***
3:   $T_{start}$ = current time //initialize the execution interval start time
4:   while (current time - $T_{start}$) < $\Delta$ do
5:     if new location update arrived then
6:       Cluster moving object o/query q //procedure described in Section 3.2
   // $\Delta$ expires. Begin evaluation of queries
7:   //*** CLUSTER-BASED JOINING PHASE ***
8:   for $c = 0$ to $MAX\_GRID\_CELL$ do
9:     for every moving cluster $m_L \in G_c$ do
10:      for every moving cluster $m_R \in G_c$ do
11:        //if the same cluster, do only join-within
12:        if ($m_L$ == $m_R$) then
13:          //do within-join only if the cluster contains members of different types
14:          if (($m_L$.OIDs > 0) && ($m_L$.QIDs > 0)) then
15:            Call DoWithinClusterJoin($m_L$,$m_L$)
16:        else
17:          //do between-join only if 2 clusters contain members of different types
18:          if (($m_L$.OIDs > 0) && ($m_R$.QIDs > 0)) ||
               (($m_L$.QIDs > 0) && ($m_R$.OIDs > 0)) then
19:            if DoBetweenClusterJoin($m_L$,$m_R$) == TRUE then
20:              Call DoWithinClusterJoin($m_L$,$m_R$)
21:  Send new query answers to users
22:  //*** CLUSTER POST-JOIN MAINTENANCE PHASE ***
23:  Call PostJoinClustersMaintenance() //do some cluster maintenance
```

Algorithm 2 *DoBetweenClusterJoin(Cluster m_L, Cluster m_R)*

```
1: //Check if two circular clusters $m_L$ and $m_R$ overlap
2: if $((m_L.Loc_t.x - m_R.Loc_t.x)^2 + (m_L.Loc_t.y - m_R.Loc_t.y)^2) < (m_L.R - m_R.R)^2$ then
3:   return TRUE; //the clusters overlap
4: else
5:   return FALSE; //the clusters don't overlap
```

clustering. In this phase, depending on the incoming location updates, new clusters may be formed, "empty" clusters may be dissolved, and existing clusters may be expanded. The cluster-based joining phase is activated every Δ time units where join-between and join-within moving clusters is executed. The cluster post-join phase is started by the end of the joining phase to perform a cluster maintenance for the next query evaluation time.

Algorithm 1 shows the pseudo code for SCUBA execution. For each execution interval, SCUBA first initializes the interval start time (Step 3). Before Δ time interval expires, SCUBA receives the incoming location updates from moving objects and queries and incrementally updates existing moving clusters or creates new ones (Step 6).

When Δ time interval expires (location updating is done), SCUBA starts the query execution (Step 8) by performing *join-between* clusters and *join-within* clusters. If two clusters are of the same type (all objects, or all queries), they are not considered for the *join-between*. Similarly, if all of the members of the cluster are of the same type, no *join-within* is performed. The *join-between* checks if the circular regions of the two clusters overlap (Algorithm 2), and *join-within* performs a spatial join between the objects and queries of the two clusters (Algorithm 3). If *join-between* does not result in intersection, *join-within* is skipped.

Algorithm 3 *DoWithinClusterJoin(Cluster m_L, Cluster m_R)*

```
1:  $R = \emptyset$; //set of results
2:  $S_q$ = Set of queries from $m_L \cup m_R$ //query members from both clusters
3:  $S_o$ = Set of objects from $m_L \cup m_R$ //object members from both clusters
    //join moving objects with queries from both clusters
4:  for every moving object $o_i \in S_o$ do
5:      for every moving query $q_j \in S_q$ do
6:          spatial join between object $o_i$ with query $q_j$ ($o_i \bowtie q_j$)
7:      $S_r$ = Set of queries from joining $o_i$ with queries in $S_q$
8:      for each Q $\in S_q$ do
9:          add (Q, $o_i$) to $R$
10: return $R$;
```

After the joining phase, cluster maintenance is performed (Step 23). Due to space limitations, we don't include the pseudo-code for *PostJoinClustersMaintenance()*. The operations performed during post-join cluster maintenance include dissolving "expiring" clusters and re-locating the "non-expiring" clusters (in the ClusterGrid) based on their velocity vectors for the next execution interval time (i.e., $T + \Delta$). If at time $T + \Delta$ the cluster passes its destination node, the cluster gets dissolved.

Example. Fig. 7 gives an illustrative example for the SCUBA algorithm. There are two moving clusters M_1 and M_2 (Fig. 7a). M_1 contains four moving objects (O_1,O_2,O_3,O_5) and no moving queries. M_2 contains one moving object (O_4) and two moving queries (Q_1,Q_2). New moving object O_6 and a new moving query Q_3 send their location updates (Fig. 7b). Q_3 has common attributes with moving cluster M_1 and O_6 has common attributes with M_2. Thus M_1 adds query Q_3 as its member (which causes its radius to expand). M_2 adds object O_6 as its member (no radius expansion here) (Fig. 7c). In Fig. 7d, we differentiate the members of the two clusters using color[4]. At time T the cluster joining phase begins (Fig. 7e). *Join-between* M_1 and M_2 returns a positive overlap. Thus the *join-within* the two clusters must be performed which produces a result (Q_2,O_3). *Join-within* for the cluster M_1 returns a result (Q_3,O_5). After the cluster joining phase, the maintenance on the clusters is performed (Fig. 7f). Based on the velocity vectors, SCUBA calculates the positions of the clusters at the next joining time (T+Δ). Since M_1 still hasn't reached its destination node at time T+Δ, it is not dissolved, but moved to its expected position based on the velocity vector. M_2 will pass its destination at the next join time. It will be dissolved at this stage.

5 Load Shedding using Moving Clusters

Load shedding is not a new idea. It has been well explored in networking [18], multimedia [6], and streaming databases [2, 36, 35]. Typically, there are two fundamental approaches distinguishing which data tuples to load shed, namely random tuples or semantically less important ones [36]. Thereafter, most works thus far primarily focus on the easy case, namely random drops, treating all tuples equally in terms of value to users [28, 35]. We instead here follow the idea of

[4] We do it for visibility purpose for the reader. No such step is executed in SCUBA.

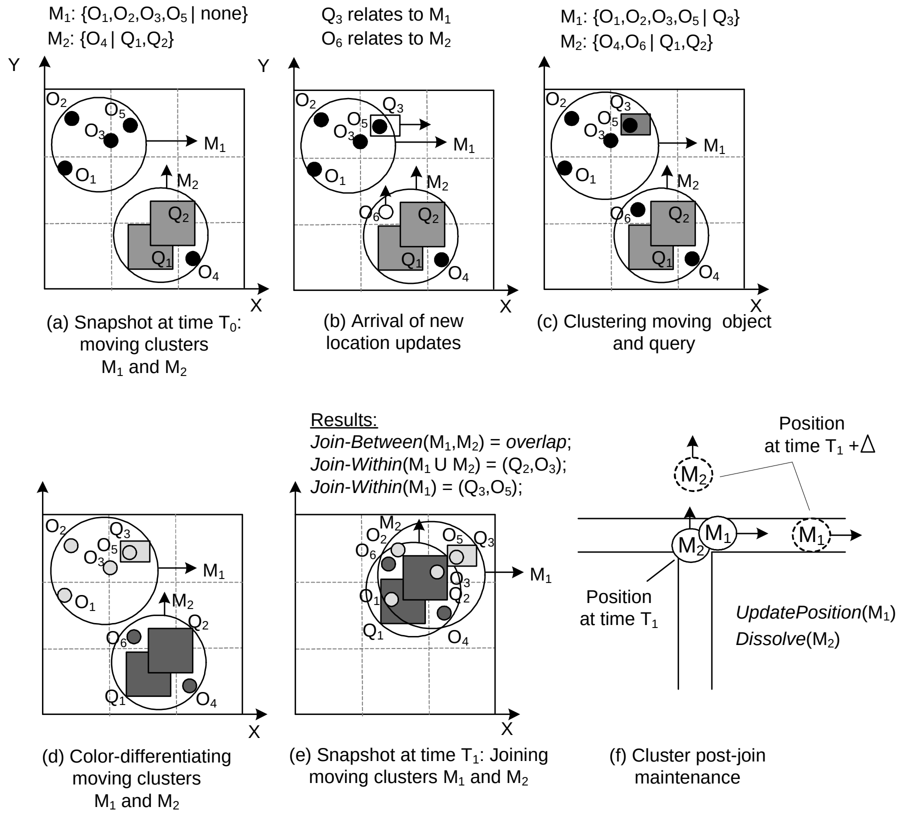

M1: {O1,O2,O3,O5 | none}
M2: {O4 | Q1,Q2}
Q3 relates to M1
O6 relates to M2
M1: {O1,O2,O3,O5 | Q3}
M2: {O4,O6 | Q1,Q2}
(a) Snapshot at time T0: moving clusters M1 and M2
(b) Arrival of new location updates
(c) Clustering moving object and query
Results:
Join-Between(M1,M2) = overlap;
Join-Within(M1 U M2) = (Q2,O3);
Join-Within(M1) = (Q3,O5);
Position at time T1 +Δ
Position at time T1
UpdatePosition(M1)
Dissolve(M2)
(d) Color-differentiating moving clusters M1 and M2
(e) Snapshot at time T1: Joining moving clusters M1 and M2
(f) Cluster post-join maintenance

Fig. 7. SCUBA execution example

semantic load-shedding, however now as applied to the spatio-temporal context. That is, our proposal is to use moving clusters to identify and thus subsequently discard the less important data first (i.e., the data that would cause the minimal loss in accuracy of the answer). Specifically, we consider for this purpose relative positions of the cluster members with respect to their centroids.

Depending on the system load and the accuracy requirements, SCUBA employs the following methods for handling cluster members (Fig. 8). Namely, all cluster members' relative positions are maintained (i.e., no load shedding) (Fig. 8a), none of the individual positions are maintained (i.e., full load shedding) (Fig. 8b), or a subset of relative positions of the cluster members are maintained (partial load shedding) (Fig. 8c). The members near the center are abstracted into a structure inside a cluster called *nucleus*, a circular region that approximates the positions of the cluster members near the centroid of the cluster. The size of the nucleus is determined by its radius threshold, Θ_N parameter where $(0 \leq \Theta_N \leq \Theta_D)$. The larger the value of Θ_N, the more data is load shed.

If the system is about to run out of memory, SCUBA begins load shedding of cluster member positions and uses a *nucleus* to approximate their positions. If memory requirements are still high, then SCUBA load sheds positions of all

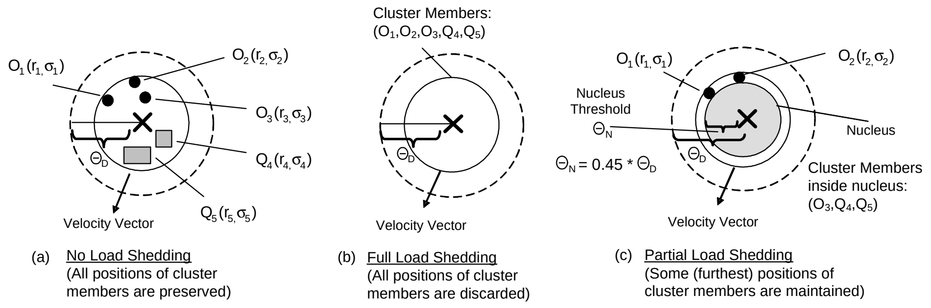

$O_1(r_1,\sigma_1)$
$O_2(r_2,\sigma_2)$
$O_3(r_3,\sigma_3)$
$Q_4(r_4,\sigma_4)$
$Q_5(r_5,\sigma_5)$
Θ_D
Velocity Vector
(a) No Load Shedding (All positions of cluster members are preserved)
Cluster Members: (O_1,O_2,O_3,Q_4,Q_5)
(b) Full Load Shedding (All positions of cluster members are discarded)
Nucleus Threshold Θ_N
$\Theta_N = 0.45 * \Theta_D$
Nucleus
Cluster Members inside nucleus: (O_3,Q_4,Q_5)
(c) Partial Load Shedding (Some (furthest) positions of cluster members are maintained)

Fig. 8. Moving cluster-driven load shedding

cluster members. In this case the cluster is the sole representation of the movement of the objects and queries that belong to it. Such abstraction of cluster members positions by nucleus/cluster corresponds to a tradeoff between accuracy and performance. The accuracy depends on how compact the clusters are. The larger the size of the clusters, the more false positives we might get for answers when performing the join-between. If individual positions are load shed, then when two clusters intersect (in *join-between*), we assume that the objects from the clusters satisfy the queries from both clusters. Making the size of the clusters compact will give more accurate answers, but also will increase the overall number of clusters, hence the join time. Increasing the size of clusters would make the processing faster, but with less accurate results. In Section 6.6 we evaluate all three schemes (i.e., no load shedding, full load shedding, and partial load shedding) in terms of its impact on performance and accuracy.

6 Experimental Evaluation

In this section, we compare SCUBA with a traditional grid-based spatio-temporal range algorithm[5], where objects and queries are hashed based on their locations into an index, say a grid. Then a cell-by-cell join between moving objects and queries is performed. Grid-based execution approach is a common choice for spatio-temporal query execution [9, 24, 27, 39, 29]. In all experiments queries are evaluated periodically (every Δ time units).

6.1 Experimental Settings

We have implemented SCUBA inside our stream processing system CAPE [31]. Moving objects and queries generated by the *Network-Based Generator of Moving Objects* [5] are used as data. The input to the generator is the road map of Worcester, USA. All the experiments were performed on Red Hat Linux (3.2.3-24) with Intel(R) XEON(TM) CPU 2.40GHz and 2GB RAM. Unless mentioned otherwise, the following parameters are used in the experiments. The set of objects consists of 10,000 objects and 10,000 spatio-temporal range queries. Each evaluation interval, 100% of objects and queries send their location updates every time unit. No load shedding is performed, unless noted otherwise. For the *ClusterGrid* table we chose a 100x100 grid size. Δ is set to 2 time units. The

[5] For simplicity, we will refer to it as *regular execution* or *regular operator*.

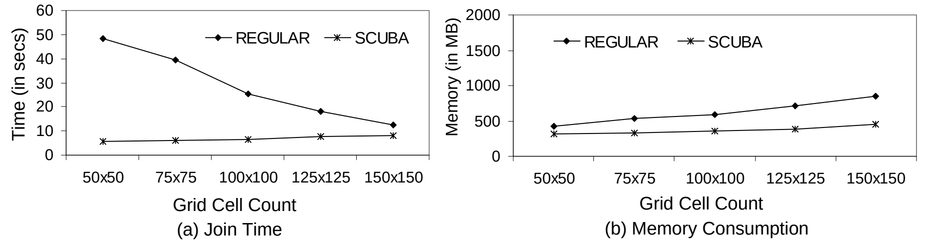


Fig. 9. Varying grid size

distance threshold Θ_D equals 100 spatial units, and the speed threshold Θ_S is set to 10 (spatial units/time units).

6.2 Varying Grid Cell Size

In this section, we compare the performance and memory consumption of SCUBA and the regular grid-based algorithm when varying the grid cell size. Fig. 9 varies the granularity of the grid (on x-axis). Since the coverage area (the city of Worcester) is constant, by increasing/decreasing the cell count in each dimension (x- and y-), we control the sizes of the grid cells. So in the 50x50 grid the size of a grid cell is larger than in the 150x150 grid. The larger the count of the grid cells, the smaller they are in size and vice versa.

From Fig. 9a, the join time decreases for the regular operator when decreasing the grid cell size. The reason for that is that smaller cells contain fewer objects and queries. Hence, fewer comparisons (joins) need to be made. But the fine granularity of the grid comes at a price of higher memory consumption, because of the large number of grid cells, each containing individual location updates of objects and queries.

The join time for SCUBA slightly goes up as the grid cell sizes become smaller. But the change is minimal, because the cluster sizes are compact and even as the granularity of the cells is increasing, the size of grid cells is frequently larger than the size of clusters. So unless many clusters are on the borderline of the grid cells (overlapping with more than one cell), the performance of SCUBA is not "hurt" by the finer granularity of the grid. Moreover, only one entry per cluster (which aggregates several objects and queries) needs to be made in a grid cell vs. having an individual entry for each object and query. This provides significant memory savings when processing extremely large numbers of densely moving objects and queries.

6.3 Varying Skewness to Facilitate Clustering

Now we study the impact of the skew in the spatio-temporal attributes of moving objects and queries. By varying the cluster-related attributes causing objects and queries to be very dissimilar (no common attribute values) or very much alike (i.e., clusterable). This determines the number of clusters and the number of cluster members per cluster.

In Fig. 10, the *skew factor* on x-axis represents the average number of moving entities that have similar spatio-temporal properties, and thus could be grouped

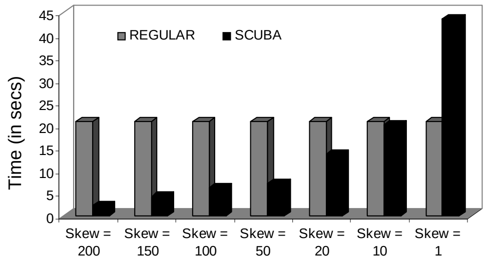


Fig. 10. Join time with skewing factor

into one cluster. For instance, when skew factor = 1, each object and query moves in a distinct way. Hence each forms its own cluster. When the skew factor = 200, every 200 objects/queries send their updates move in a similar way. Thus they typically may form a cluster. In Fig. 10, when not many objects and queries are clusterable, the SCUBA performance suffers due to the overhead of many single-member clusters. Thus more join-between clusters are performed as the number of clusters goes up. If many single member clusters spatially overlap, the join-within is performed as well. This increases the overall join time. In real life this scenario is highly unlikely as with a large number of moving objects the chance increases that common motion attributes for some duration of time are present (e.g., increase in traffic on the roads). As the skew factor increases (10-200), and more objects and queries are clusterable, the join time for SCUBA significantly decreases. The overall join time is several orders of magnitude faster compared to a regular grid-based approach when the skew factor equals 200, i.e., approximately 200 moving entities per cluster.

6.4 Incremental vs. Non-incremental Clustering

In this section we study the tradeoff between the improved quality of the clusters which can be achieved when clustering is done non-incrementally (with all data points available at the same time) and the performance of SCUBA. As proposed, SCUBA clusters location updates incrementally upon their arrival. We wanted to investigate if clustering done offline (i.e., non-incrementally, when all the data points are available) and thus producing better quality clusters and facilitating a faster join-between the clusters outweighs the cost of non-incremental clustering. In particular, we focus on the join processing time, and how much improvement in join processing could be achieved with better quality clusters.

We implemented a *K-means* (a common clustering algorithm) extension to SCUBA for non-incremental clustering. The K-means algorithm expects the number of clusters specified in advance. We used a tracking counter for the number of unique destinations of objects and queries for a rough estimate of the number of clusters needed. Another disadvantage is that K-means needs several iterations over the dataset before it converges. With each iteration, the quality of clustering improves, but the clustering time increases. We varied the number of iterations from 1 to 10 in this experiment to observe the impact on quality of clusters achieved by increasing the number of iterations.

Fig. 11 presents the join times for SCUBA when clustering is done incrementally vs non-incrementally. The bars represent a combined cost of clustering time

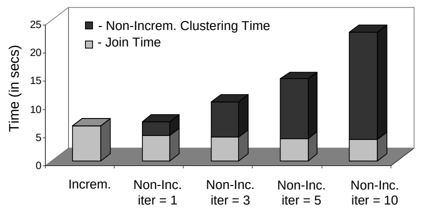


Fig. 11. Incr. vs. Non-Incr. Clustering

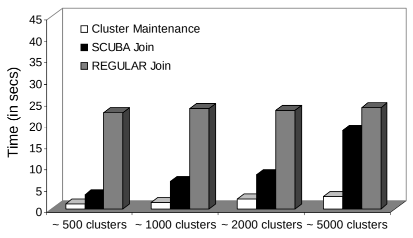


Fig. 12. Cluster Maintenance

and join time. The time to perform incremental clustering is not portrayed as the join processing starts immediately when Δ expires. In the offline clustering scenario, the clustering has to be done first before proceeding to the join. With the increased number of iterations (on x-axis), the quality of clusters improves resulting in faster join execution compared to the incremental case. However, the cost of waiting for the offline algorithm to finish the clustering outweighs the advantage of the faster join. When the number of iterations is 3 or greater, the clustering time in fact takes longer than the actual join processing. The larger the dataset the more expensive each iteration becomes. Offline clustering is not suitable for clustering large amounts of moving objects when there are constraints on execution time and memory space. Even with a reduced number of scans through the data set and improved join time, the advantage of having better quality clusters is not amortized due to the amount of time spent on offline clustering and larger memory requirements. This justifies the appropriateness of our incremental clustering as an efficient clustering solution.

6.5 Cluster Maintenance Cost

In this section, we compare the cluster maintenance cost with respect to join time in SCUBA and regular grid-based join time. Fig. 12 gives the cluster maintenance time when the number of clusters is varied. By cluster maintenance cost we mean the time it takes to pre- and post-process the clusters before and after the join is complete (i.e., form new clusters, expand existing clusters, calculate the future position of the cluster using its average speed, dissolve expired clusters, and re-insert clusters into the grid for the next evaluation interval).

For this experiment, we varied the skew factor to affect the average number of the clusters (the number of objects and queries stays the same). The x-axis represents the average number of clusters in the system. Fig. 12 shows that the cluster maintenance (which is an overhead for SCUBA) is relatively cheap. If we combine cluster maintenance with SCUBA join time to represent the overall join cost for SCUBA, it is still faster than the regular grid-based execution. Hence, even though maintaining clusters comes with a cost, it is superior over processing objects and queries individually.

6.6 Moving-Cluster-Driven Load Shedding

Here we evaluate the effect of moving cluster-based load shedding on the performance and accuracy in SCUBA. Figures 13a and 13b respectively represent the join processing times and accuracy measurements for SCUBA when load shedding positions of the cluster members. The x-axis represents the percent of

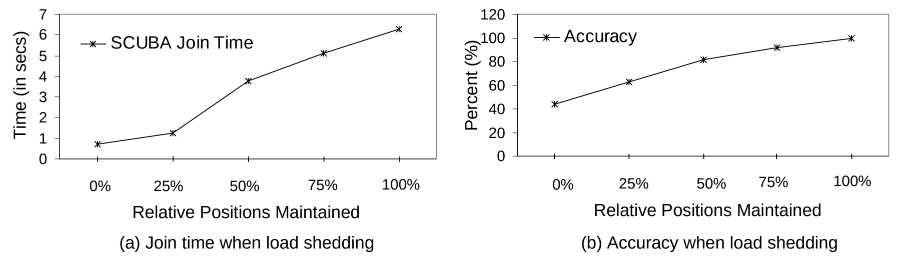


Fig. 13. Cluster-Based Load Shedding

the size of the *nucleus* (i.e., circular region in the cluster approximating cluster members whose positions are discarded) with respect to the maximum size of the cluster. For simplicity, we will refer to percent of the nucleus-to-cluster size as η. When $\eta = 0\%$, no data is discarded. On the opposite, when $\eta = 100\%$, all cluster members' positions are discarded, and the cluster solely approximates the positions of its members. The fewer relative positions are maintained, the fewer individual joins need to be performed when executing a join-within for overlapping clusters.

As expected load shedding comes at a price of less accurate results (Fig. 13b). To measure accuracy, we compare the results outputted by SCUBA when $\eta = 0\%$ (no load shedding) to the ones output when $\eta > 0\%$, calculating the number of false-negative and false-positive results. The size of the nucleus has a significant impact on the accuracy of the results when performing load shedding. Hence it must be carefully considered. When $\eta = 50\%$, the accuracy $\approx 79\%$. So relatively good results can be produced with cluster-based load shedding even if 50 % of a cluster region is shed. If random load shedding were to be performed, the same number of tuples - but just not the same tuples would be load shed. Instead the shedding mechanism would randomly pick which ones to shed, potentially throwing away more important data and significantly degrading the results' accuracy.

7 Related Work

Related Work on Spatio-Temporal Query Processing: Efficient evaluation of spatio-temporal queries on moving objects has been an active area of research for quite some time. Several optimization techniques have been developed. These include Query Indexing and Velocity Constrained Indexing (VCI) [29], shared execution [24, 38, 39], incremental evaluation [24, 39], and query-aware moving objects [17]. Query Indexing indexes queries using an R-tree-like structure. At each evaluation step, only those objects that have moved since the previous evaluation step are evaluated against the Q-index. VCI utilizes the maximum possible speed of objects to delay the expensive updates to the index. To reduce wireless communication and query reevaluation costs, Hu et. al [17] utilize the notion of safe region, making the moving objects query aware. Query reevaluation in this framework is triggered by location updates only. In the similar spirit [29] and [38] try to minimize the number of unnecessary joins between objects and queries by using the *Safe* and *No-Action* regions respectively. In the latter case, the authors combine it with different join policies to filter out the objects and queries that are guaranteed not to join.

The scalability in spatio-temporal query processing has been addressed in [11, 23, 24, 29, 39]. In a distributed environment, such as MobiEyes [11], part of the query processing is send to the clients. The limitations of this approach is that the devices may not have enough battery power and memory capacity to perform the complex computations. The shared execution paradigm as means to achieve scalability has been used in SINA [24] for continuous spatio-temporal range queries, and in SEA-CNN [39] for continuous spatio-temporal kNN queries. Our study falls into this category and distinguishes itself from these previous works by focusing on utilizing moving clusters abstracting similar moving entities to optimize the execution and minimize the individual processing.

Related Work on Clustering: Clustering has been an active field for over 20 years [1, 25, 41]. Previous work typically uses clustering to analyze data to find interesting patterns. We instead apply clustering as means to achieve scalable processing of continuous queries on moving objects. To the best of our knowledge, this is the first work to use clustering for shared execution optimization of continuous queries on spatio-temporal data streams.

In this work we considered clustering algorithms in which clusters have a distinguished point, a center. The commonly used clustering algorithm for such clustering, k-means, is described in [8, 13, 30]. Our concentration was on incremental clustering algorithms only [4], [7], and [40]. Some of the published clustering algorithms for an incremental clustering of data streams include BIRCH[41], COBWEB [10], STREAM [12, 26], Fractal Clustering [3], and the Leader-Follower (LF) [16]. Clustering analysis is a well researched area, and due to space limitations we do not discuss all of the clustering algorithms available. For an elaborate survey on clustering, readers are referred to [19]. In our work, we adapt an incremental clustering algorithm, similar to the Leader-Follower clustering [16]. The extensibility, running time and the computational complexity of this algorithm is such that makes it attractive for processing streaming data.

Clustering of spatio-temporal data has been explored to a limited degree in [21]. This work [21] concentrates on discovering moving clusters using historic trajectories of the moving objects. The algorithms proposed assume that all data is available. Our work instead clusters moving entities at run-time and utilizes moving clusters to solve a completely different problem, namely, efficient processing of continuous spatio-temporal queries.

8 Conclusions and Future Work

In this paper, we propose a unique algorithm for efficient processing of large numbers of spatio-temporal queries on moving objects termed SCUBA. SCUBA combines motion clustering with shared execution for query execution optimization. Given a set of moving objects and queries, SCUBA groups them into moving clusters based on common spatio-temporal attributes. To optimize the join execution, SCUBA performs a two-step join execution process by first pre-filtering a set of moving clusters that could produce potential results in the join-between moving clusters stage and then proceeding with the individual join-within execution on those selected moving clusters. Comprehensive experiments show that the performance of SCUBA is better than traditional grid-based approach where moving entities are processed individually. In particular the experiments demonstrate that SCUBA: (1) facilitates efficient execution of queries on moving objects that have common spatio-temporal attributes, (2) has low cluster maintenance/overhead cost, and (3) naturally facilitates load shedding using motion clusters while optimizing the processing time with minimal degradation in result quality. We believe our work is the first to utilize motion clustering to optimize the execution of continuous queries on spatio-temporal data streams. As future work, we plan to further refine and validate moving cluster-driven load shedding, enhance SCUBA to produce results incrementally and explore further through additional experimentation.

References

1. R. Agrawal and et. al. Automatic subspace clustering of high dimensional data for data mining applications. In *SIGMOD*, pages 94–105, 1998.
2. B. Babcock, M. Datar, and R. Motwani. Load shedding techniques for data stream systems, 2003.
3. D. Barbará. Chaotic mining: Knowledge discovery using the fractal dimension. In *SIGMOD Workshop on Data Mining and Knowl. Discovery*, 1999.
4. D. Barbará. Requirements for clustering data streams. *SIGKDD Explorations*, 3(2):23–27, 2002.
5. T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.
6. C. L. Compton and D. L. Tennenhouse. Collaborative load shedding for media-based applications. In *Int. Conf. on Multimedia Computing and Systems*, 1994.
7. P. Domingos and G. Hulten. Catching up with the data: Research issues in mining data streams. In *DMKD*, 2001.
8. R. O. Duda, P. E. Hart, and D. G. Stork. *Pattern Classification*. Wiley-Interscience Publication, 2000.
9. H. G. Elmongui, M. F. Mokbel, and W. G. Aref. Spatio-temporal histograms. In *SSTD*, 2005.
10. D. H. Fisher. Iterative optimization and simplification of hierarchical clusterings. *CoRR*, cs.AI/9604103, 1996.
11. B. Gedik and L. Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67–87, 2004.
12. S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan. Clustering data streams. In *FOCS*, pages 359–366, 2000.
13. S. K. Gupta, K. S. Rao, and V. Bhatnagar. K-means clustering algorithm for categorical attributes. In *DaWaK*, pages 203–208, 1999.
14. S. E. Hambrusch, C.-M. Liu, W. G. Aref, and S. Prabhakar. Query processing in broadcasted spatial index trees. In *SSTD*, pages 502–521, 2001.
15. S. Har-Peled. Clustering motion. In *FOCS '01: Proceedings of the 42nd IEEE symposium on Foundations of Computer Science*, page 84, 2001.
16. J. A. Hartigan. *Clustering Algorithms*. John Wiley and Sons, 1975.
17. H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD*, 2005.

18. V. Jacobson. Congestion avoidance and control. *SIGCOMM Comput. Commun. Rev.*, 25(1):157–187, 1995.
19. A. K. Jain, M. N. Murthy, and P. J. Flynn. Data clustering: A review. Technical Report MSU-CSE-00-16, Dept. of CS, Michigan State University, 2000.
20. D. V. Kalashnikov and et. al. Main memory evaluation of monitoring queries over moving objects. *Distrib. Parallel Databases*, 15(2), 2004.
21. P. Kalnis, N. Mamoulis, and S. Bakiras. On discovering moving clusters in spatio-temporal data. In *SSTD05*, 2005.
22. Y. Li, J. Han, and J. Yang. Clustering moving objects. In *KDD*, pages 617–622, 2004.
23. M. F. Mokbel and et. al. Towards scalable location-aware services: requirements and research issues. In *GIS*, pages 110–117, 2003.
24. M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD*, pages 623–634, 2004.
25. R. T. Ng and J. Han. Efficient and effective clustering methods for spatial data mining. In *VLDB*, pages 144–155, 1994.
26. L. O'Callaghan and et. al. Streaming-data algorithms for high-quality clustering. In *ICDE*, page 685, 2002.
27. D. Papadias and et. al. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, 2005.
28. Y. C. Philip. Loadstar: A load shedding scheme for classifying data streams.
29. 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.
30. E. M. Rasmussen. Clustering algorithms. In *Information Retrieval: Data Structures & Algorithms*, pages 419–442. 1992.
31. E. A. Rundensteiner, L. Ding, and et.al. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, pages 1353–1356, 2004.
32. S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *SIGMOD*, pages 331–342, 2000.
33. A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *ICDE*, pages 422–432, 1997.
34. Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *SIGMOD*, pages 334–345, 2002.
35. N. Tatbul. Qos-driven load shedding on data streams. In *EDBT '02: Proceedings of the Worshops XMLDM, MDDE, and YRWS on XML-Based Data Management and Multimedia Engineering-Revised Papers*, London, UK, 2002. Springer-Verlag.
36. N. Tatbul, U. Çetintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.
37. J. Tayeb, Ö. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *Comput. J.*, 41(3):185–200, 1998.
38. X. Xiong and M. F. M. et.al. Scalable spatio-temporal continuous query processing for location-aware services. In *SSDBM*, pages 317–, 2004.
39. X. Xiong, M. F. Mokbel, and W. G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, 2005.
40. N. Ye and X. Li. A scalable, incremental learning algorithm for classification problems. *Comput. Ind. Eng.*, 43(4):677–692, 2002.
41. T. Zhang, R. Ramakrishnan, and M. Livny. Birch: An efficient data clustering method for very large databases. In *SIGMOD*, pages 103–114, 1996.