Information Systems 38 (2013) 331–350

Contents lists available at SciVerse ScienceDirect

Information Systems

journal homepage: www.elsevier.com/locate/infosys

Mining neighbor-based patterns in data streams

Di Yang [a,*], Elke A. Rundensteiner [b], Matthew O. Ward [b]

[a] 1 Oracle Dr, Nashua, NH 03062, United States
[b] WPI, United States

ARTICLE INFO

Article history:
Received 15 September 2011
Received in revised form
2 June 2012
Accepted 13 August 2012
Recommended by: K.A. Ross
Available online 26 September 2012

Keywords:
Streaming
Pattern mining
Clusters
Outliers
Algorithms

ABSTRACT

Discovery of complex patterns such as clusters, outliers, and associations from huge volumes of streaming data has been recognized as critical for many application domains. However, little research effort has been made toward detecting patterns within *sliding window* semantics as required by real-time monitoring tasks, ranging from real time traffic monitoring to stock trend analysis. Applying static pattern detection algorithms from scratch to every window is impractical due to their high algorithmic complexity and the real-time responsiveness required by streaming applications. In this work, we develop methods for the incremental detection of neighbor-based patterns, in particular, density-based clusters and distance-based outliers over sliding stream windows. Incremental computation for pattern detection queries is challenging. This is because purging of to-be-expired data from previously formed patterns may cause birth, shrinkage, splitting or termination of these complex patterns. To overcome this, we exploit the "predictability" property of sliding windows to elegantly discount the effect of expired objects with little maintenance cost. Our solution achieves guaranteed minimal CPU consumption, while keeping the memory utilization linear in the number of objects in the window. To thoroughly analyze the performance of our proposed methods, we develop a cost model characterizing the performance of our proposed neighbor-based pattern mining strategies. We conduct an analysis study to not only identify the key performance factors for each strategy but also show under which conditions each of them are most efficient. Our comprehensive experimental study, using both synthetic and real data from domains of moving object monitoring and stock trades, demonstrates superiority of our proposed strategies over alternate methods in both CPU processing resources and in memory utilization.

1. Introduction

We present a new framework for detecting "neighbor-based" patterns in streams covering two important types of patterns, namely density-based clusters [18,17] and distance-based outliers [24,5] applied to sliding windows semantics [7,8]. Many applications providing monitoring services over streaming data require this capability of real-time pattern detection. For example, to understand the major threats of an enemy's airforce, a battle field commander needs to be continuously aware of the "clusters" formed by enemy warcrafts based on the objects' most recent positions extracted from the data streams reported from satellites or ground stations. We evaluate our techniques for this class of applications by mining clusters in the ground moving target indicator data stream [16]. As another example, a financial analyst monitoring stock transactions may be interested in the "outliers" (abnormal transactions) in the transaction stream, as they are potential indicators for new trends in the market. We evaluate our techniques for this class of application by mining outliers in the NYSE transaction stream [23].

Background on neighbor-based patterns: Neighbor-based pattern detection techniques are distinct from global clustering methods [32,22], such as *k*-means clustering. Global clustering methods aim to summarize the main characteristics of

* Corresponding author. Tel.: +1 508 243 9636.
E-mail addresses: di.yang@oracle.com, matt@cs.wpi.edu (D. Yang).

0306-4379/$ - see front matter © 2012 Published by Elsevier Ltd.
http://dx.doi.org/10.1016/j.is.2012.08.001

huge datasets by first partitioning them into groups (e.g., in Fig. 1, the objects in the same circles are considered to be in the same cluster), and then provide abstract information about the identified clusters, such as cluster centroids, as output. In these works, the cluster memberships of individual objects are not of special interest and thus not determined. In contrast, the techniques presented in this work target a different scenario, namely when individual objects belonging to patterns are of importance. For example, during the battlefield monitoring scenario, the commander may need to drill down to access the specific information about individual objects in the clusters formed by enemy warcraft. This is because some important characteristics of the clusters, such as the composition of each cluster (e.g., how many bomb carriers and fighter planes each cluster has) and the positions of the "super threats" in each cluster (e.g., the bomb carriers with nuclear bombs) can be learned from this specific information. Similarly, specific details about each outlier in the credit card transactions scenario may point to a credit fraud that may cause serious loss of revenue.

Thus our techniques focus on identifying specific objects that behave individually (for outliers) or together (for clusters) in some special manner. More specifically, the neighbor-based patterns are composed of object(s) with specific characteristics with respect to their local neighborhoods. Precise definitions of the patterns will be given in Section 2. Fig. 2 shows an example of two density-based clusters and a distance-based outlier in the dataset from Fig. 1.

Motivation for sliding window scenario: Another important characteristic distinguishing our work from previous efforts [13,12] is that we aim to mine for neighbor-based patterns within the sliding window scenario. The sliding window semantics, while widely used for continuous query processing [7], have rarely been applied to neighbor-based pattern mining. Sliding window semantics assume a fixed window size (either a fixed time interval or a fixed number of objects), with the pattern detection results generated based on the most recent data falling into the current sliding window. However, in previous clustering work [20,19,13,12], objects with different time horizons are either treated equally or assigned weights decaying as their recentness decreases. These techniques capture the accumulative characteristics of the full data stream seen so far, rather than isolating and reflecting about the features in the most recent stream portion. Using our earlier example, the position information of the warcraft may only be valid for a certain time period due to the movement of the monitored objects. In such cases, the sliding window technique is necessary as it forces the system to discard the out-of-date information and form the patterns only based on the most recent positions of the moving objects.

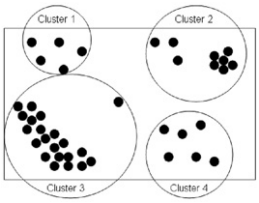

Fig. 1. Four global clusters determined by global clustering algorithms, such as *K*-means.

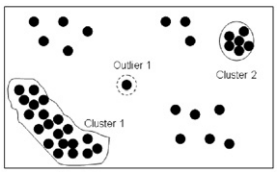

Fig. 2. Two density-based clusters and one distance-based outlier determined by neighbor-based pattern detection algorithms.

Challenges: Detecting neighbor-based patterns for sliding windows is a challenging problem. Naive approaches that run the static neighbor-based pattern detection algorithms from scratch for each window are often not feasible in practice, considering the conflict between the high complexity of these algorithms and the real-time response requirement for stream monitoring applications. Based on our experiments (see Section 11), detecting density-based clusters from scratch in a 50 K-object window takes around 100 s on our experimental platform, which does not meet the real-time responsiveness requirement for many interactive applications.

A straightforward incremental approach, which relies on incrementally maintaining the *exact* neighbor relationships (we will henceforth use the term "*neighborship*" for this concept) among objects, will also fail in many cases. This is because the potentially huge number of *neighborships* can easily raise the memory consumption to unacceptable levels. In the worst case, N^2 *neighborships* may exist in a single window, with N the number of data points in the window. Our experiments confirm that this solution consumes on average 15 times more memory compared to the from-scratch approach when applied to real datasets [16,23].

To overcome this serious strain on memory consumption, while still enabling the incremental computation, we introduce several *neighborship* abstractions that guarantee a linear in N memory consumption. However, designing solutions based on abstracted *neighborships* now come with a new shortcoming. Namely, the absence of exact *neighborships* makes discounting the effect of the expired objects from previously detected patterns become highly expensive in terms of CPU resources. This is because it is difficult to track what pattern structural changes, such as "splitting" or "termination", will be triggered by objects' expiration, without knowing which objects are directly connected to the expired objects and thus are affected.

Proposed methods: To make the abstracted *neighborships* incrementally maintainable in a computationally efficient manner, we propose to exploit an important characteristic of sliding windows, namely the "predictability" of the objects' expiration. Specifically, given a query window with fixed window size and slide size, we can predetermine all the windows in which each object can survive. A further insight gained from this "predictability" property leads us to propose the notion of the "predicted views". Namely given the objects in the current window, we can predict the pattern structures that will persist in subsequent windows and abstract them into the "predicted view" of each individual future window. The "view prediction" technique

enables us to elegantly discount the effect of expired objects and thus to incrementally maintain the abstracted *neighborships* efficiently.

Finally, along with "predictability", we propose a hybrid *neighborship* maintenance mechanism incorporating two forms of neighbor abstraction and dynamically switching between them when needed. This solution achieves not only linear memory consumption, but now also guarantees optimality in the number of the range query searches. Our proposed technique is able to cluster 50 K objects in a window within 5 s, given 10% of them are new arrivals from the stream. It is on average five times faster than the alternative incremental algorithm using abstract *neighborships* only, while it consumes only 5% of memory space compared to the alternative using exact *neighborships*.

Contributions: The main contributions of this work include: (1) We characterize the key performance bottleneck for incremental detection of neighbor-based patterns over sliding windows lies in handling the expired objects. It consumes either massive memory or CPU processing resources, both critical resources for streaming data processing. (2) We exploit the "predictability" property of sliding windows and further extend it with the notion of "predicted views", which elegantly discounts the effect of expired data from future query results. (3) We present, to the best of our knowledge, the first algorithm that detects density-based clusters in sliding windows. This algorithm is shown to guarantee the minimum number of range query searches needed at each window slide, while keeping the memory requirement linear in the number of objects in the window. (4) We present a new algorithm to detect distance-based outliers for sliding windows. This algorithm covers both count-based and time-based windows and thus is more comprehensive than the only prior solution dealing with count-based windows only [5]. (5) We design a cost model for neighbor-based pattern mining algorithms in streaming windows. (6) We use our cost model to thoroughly analyze the strengths and weaknesses of all the alternative methods under different conditions. (7) Our comprehensive experiments on both synthetic and real streaming data from domains of moving object monitoring and stock trades confirm the effectiveness of our proposed algorithms and their superiority over the existing alternative approaches.

This paper is significantly extended from a previous conference paper [1]. The major extensions that are new contributions include:

(1) We design a cost model for estimating both the CPU and memory resource utilization for neighbor-based pattern mining in sliding windows (Section 10) .
(2) We conduct a thorough cost analysis of the alternative methods (Section 10), comparing their performance under a broad range of parameter settings. This cost analysis identifies the key performance factors of each alternative algorithm, and helps us to establish under which conditions each algorithm performs at its best.
(3) We discuss how our proposed techniques can be naturally extended to handle several important output models, such as incremental output and pattern evolution across different windows (Section 9).
(4) We extend our experimental study section substantially by conducting additional experiments evaluating the scalability of our proposed algorithms (Section 11). These experiments include the scalability test on both the window size and dimensionality, the two key factors affecting scalability of clustering algorithms over streaming windows. Also, the additional test cases are discussed in-depth to better compare and contrast alternative approaches under different situations.
(5) We elaborate in more depth on the design of all proposed algorithms. This includes the addition of pseudo-code for each algorithm, more detailed examples and explanations, and theoretical proofs for all lemmas and theorems (Sections 4, 6 and 8).

The remainder of this paper is organized as follows. Section 2 introduces the preliminaries. Sections 3–7 provide our proposed strategies for the incremental neighbor-based pattern mining. In Section 10, we present a cost model and a cost analysis for all alternative methods. Our experimental studies are given in Section 11. In Section 12, we discuss the related work, while Section 13 offers conclusions.

2. Problem definition

Definition of neighbor-based patterns: We support the detection of "neighbor-based" patterns, in particular, distance-based outliers [18,17] and density-based clusters [24]. In this work, we use the term "data point" to refer to a multi-dimensional tuple in the data stream. Neighbor-based pattern detection uses a range threshold $\theta^{range} \geq 0$ to define the *neighborship* between any two data points. For two data points p_i and p_j, if the distance between them is no larger than θ^{range}, then p_i and p_j are said to be neighbors. Any distance function can be plugged into these algorithms. We use the function $NumNeighbors(p_i,\theta^{range})$ to denote the number of neighbors a data point p_i has, given the θ^{range} threshold.

Definition 2.1. *Distance-based outlier*: Given θ^{range} and a fraction threshold θ^{fra} ($0 \leq \theta^{fra} \leq 1$), a distance-based outlier is a data point p_i, where $NumNeighbors(p_i,\theta^{range}) < N*\theta^{fra}$, with N the number of data points in the data set.

Definition 2.2. *Density-based cluster*: Given θ^{range} and a count threshold θ^{cnt}, a density-based cluster is defined as a group of "connected *core objects*" and the *edge objects* attached to them. Any pair of *core points* in a cluster are "connected" with each other. A data point p_i with $NumNeighbors(p_i,\theta^{range}) \geq \theta^{cnt}$ is defined as a *core point*. Otherwise, if p_i is a neighbor of any *core object*, p_i is an *edge object*. p_i is a *noise* point if it is neither a *core point* nor an *edge point*. Two *core points* c_0 and c_n are connected, if they are neighbors of each other, or there exists a sequence of *core points* $c_0,c_1,\ldots c_{n-1},c_n$, where for any i with $0 \leq i \leq n-1$, a pair of *core objects* c_i and c_{i+1} are neighbors of each other.

Fig. 7 shows an example of a density-based cluster composed of three *core points* (black) and eight *edge points* (grey).

Neighbor-based pattern detection in sliding windows: We focus on periodic sliding window semantics as proposed in CQL [7] and widely used in the literature [5,1]. These proposed semantics can be either time-based or count-based. In both cases, each query has a window with a fixed window size *win* and a fixed slide size *slide* (either a time interval or a tuple count). The query window slides periodically either when a certain number of objects arrive at the system or a certain amount of time has elapsed. During each slide, the query window takes in the new objects and purges the expired objects in the window. For each window W_i, patterns are generated only based on all data points falling into it.

We first focus on the generation of complete pattern detection results. In particular, for distance-based outliers, we output all outliers identified in a window. For density-based clusters, we output the members of each cluster, each with a cluster id of the clusters they belong to. Other output formats can also be supported by our proposed techniques (see Section 9).

3. Naive approach

The naive approach for detecting patterns over continuous windows would be to run the static pattern detection algorithms from scratch at each window. Generally, static neighbor-based pattern detection algorithms [18,24] consume one range query search for every data point in the dataset. This in our case, they need N range query searches at each window W_i, with N the number of data points in W_i.

Considering the expensiveness of range query searches, this naive approach may not be applicable in practice, specially when N is large. Obviously, without the support of indexing, the complexity of each range query search is $O(N)$. The average run-time complexity of a range query search can be improved by using index structures, for instance an R-tree could improve it to $O(\log(N))$ [18]. However, such complexity may still be an unacceptable burden for streaming applications that require real-time responsiveness, not to mention that the high-frequency of data updating in the streaming environments makes the index maintenance expensive. Given these limitations, such naive approach is not the best choice for handling overlapping windows ($Q.slide < Q.win$), where the opportunity for sharing meta-information among windows exists.

4. Exact-neighborship-based solution (Exact-N)

To avoid repeated running of range query searches and recomputing patterns from scratch, our task thus is to design incremental pattern computation algorithms. Now we discuss the first incremental algorithm that detects the neighbor-based patterns based on the exact *neighborships* among data points. We call it the *Exact-Neighborship-Based Solution (Exact-N)*. Exact-N relieves the computational intensity of processing each window by preserving the exact *neighborships* discovered in the previous windows. In particular, Exact-N requires each data point p_i in the window to maintain a list of links pointing to all its neighbors.

At each window slide, the expired data points are removed along with the exact *neighborships* they are involved in, namely all the links pointing from or to them. Then Exact-N runs one range query search for every new data point p_{new} to discover the new *neighborships* to be established in the new window. For distance-based outliers, Exact-N simply outputs the data points with less than $N \times \theta^{fra}$ neighbors. For density-based clusters, Exact-N constructs the cluster structures by a Depth First Search (DFS) on all *core points* (with no less than θ^{count} neighbors) in the window. Exact-N offers the advantage of conducting only N_{new} range query searches at each window, with N_{new} equal to the number of new data points in the window. The specific algorithm of Exact-N is shown in Figs. 3 and 4.

Discussion: Compared with the N (total number of data points in the window) range query searches needed by the naive approach at each window, Exact-N offers the advantage of conducting only N_{new} range query searches at each window, with N_{new} the number of *new* data points in the window.

```
Exact-N (θ^range, θ^cnt / θ^fra)
1 At each window slide
//Purging
2  For each expired data point p_exp
3    For each p_i in p_exp.neighbors
4      remove p_exp from p_i.neighbors;
5    purge p_exp;
//Loading
6  For each new data point p_new
7    load p_new into index
//Neighborship Maintenance
8  For each new data point p_new
9    Neighbors =
       RangeQuerySearch(p_new, θ^range)
10         For each p_j in Neighbors
11           add p_j to p_new.neighbors
12           add p_new to p_j.neighbors
//Output
13     OutputPatterns (pattern type);
```

Fig. 3. Pseudo-code for Exact-N Part 1.

```
OutputPatterns(Distance-Based Outliers)
1  For each data point p_i in the window
2    If p_i.neigbhors.size () ≤ θ^fra * N
3      Output (p_i)
OutputPatterns(Density-Based Clusters)
1  ClusterId=0;
2  For each p_i with ≤ θ^cnt neighbors
3    If p_i is unmarked;
4      OutputCore(p_i, ClusterId);
5      ClusterId++;
OutputCore(p_c, ClusterId)
1  mark p_c with ClusterId;
2  output(p_c);
3  For each data point p_i on p_c.neighbors
4    If p_i is unmarked
5      If p_i.neigbhors.size () ≤ θ^cnt
6        OutputCore(p_i, ClusterId)
7      Else
8        mark p_i with ClusterId;
9        Output(p_i);
```

Fig. 4. Pseudo-code for Exact-N Part 2.

Lemma 4.1. *For each query window W_i, the minimum number of range query searches needed for detecting neighbor-based patterns in W_i is N_{new}.*

Proof. At each new window W_i, each new data point falling into W_i needs a range query search to discover all its neighbors in the window, otherwise we cannot obtain all new *neighborships* in W_i introduced by the participation of the new data points. Missing *neighborships* between data points may cause wrong clustering results. For example, a missing *neighborship* between two *core points* p_i and p_j, may cause a single cluster containing both of them to be identified as two separate clusters, each containing one of them. This proves the necessity of the N_{new} range query searches. Since we can preserve all *neighborships* inherited from W_{i-1}, we will not miss any prior *neighborships* existing in W_i. This proves its sufficiency. □

However, Exact-N suffers from a major shortcoming, namely its huge memory consumption, as it requires storing all exact *neighborships* among data points. In the worst case, the memory requirement may be *quadratic* in the number of all data points in the window. Such a demand on the memory may make the algorithms impractical for huge window sizes N, given that the real-time response requirement of streaming applications necessitates main memory resident processing. Our experimental results in Section 11 confirm the memory-inefficiency of Exact-N.

5. "Predictability" property in sliding windows

Next, we summarize the "predictability" property of periodic sliding windows.

Definition 5.1. Given the slide size $Q.slide$ of a query Q and the starting time of current window $W_n.T_{start}$, the *life-span* $p_i.lifespan$ of a data point p_i in W_n with time stamp $p_i.T$, is defined by $p_i.lifespan = \lceil p_i.T - W_n.T_{start}/Q.slide \rceil$, indicating that p_i will survive in windows W_n to $W_{n+p_i.lifespan-1}$ before its expiration.

This property determines the expiration of existing data points in future windows. Thus it enables us to pre-handle their impact on the pattern detection results in future windows. For this basic concept of "predictability" to be exploitable for tackling our neighbor-based pattern detection problem, we have developed an observation based on it as given below.

Observation 5.2. Given the slide size $Q.slide$ of a query Q and the starting time of the current window $W_n.T_{start}$, a *neighborship* $Neighbor(p_i,p_j)$ between two data points p_i and p_j in W_n will hold for totally $Neighbor(p_i,p_j).lifespan = Min(\lceil p_i.T - W_n.T_{start}/Q.slide \rceil, \lceil p_j.T - W_n.T_{start}/Q.slide \rceil)$ windows, namely, it will exist in all windows from W_n to $W_{n+Neighbor(p_i,p_j).lifespan-1}$ until either p_i or p_j expires.

Observation 5.2 express the impact of the "predictability" property on the *bilateral* relationships among two data points, namely the neighborships between them. Specifically, Observation 5.2 combines the basic concept of "predictability" with the fact that a neighborship between a pair of data points is guaranteed to hold as long as both of its point participants are still valid. This observation helps us to determine the "life span" of the *neighborships* between any pair of data points in a window. This observation will be used in our proposed algorithms to elegantly maintain the neighborhood of each data point.

The two observations above allow us to "foresee" part of the patterns in the future windows based on the data points and the *neighborships* among them that will surely appear in these windows. The complete patterns in these future windows can later be decided solely by the insertion of new data points. The key point here is that by predetermining the partial members of the future windows, we can eliminate the needs for handling the impact of expired data points on the pattern detection result when the window slides. We call this the "*View Prediction*" technique. This principle will be used by our proposed algorithms to maintain neighbor-based patterns over windows.

6. Abstracted-neighborship-based solution using counts (Abstract-C)

Different from Exact-N, Abstract-C aims to maintain a compact summary of the neighborships, namely the count of the neighbors for each data point.

6.1. Challenges

Maintaining the neighbor counts for each data point appears to be not computationally cheaper than the maintenance of their neighbor lists. This is because, although we can easily determine the increases of the neighbor count of each data point, when to decrease them as their neighbors are expired becomes a hard problem. As in Abstract-C, the data points no longer maintain the exact *neighborships* they are involved in, namely, the list pointing to their neighbors, they cannot directly inform their neighbors to decrease their respective neighbor count when they are expired.

6.2. Solution

Fortunately, the "predictability" property introduced in Section 5 can help us to tackle this problem. The key idea is that since we can exactly predict the expiration of any data point p_i, we can pre-handle the impact of p_i's expiration on its neighbors' neighbor counts, at the time when they are first identified to be neighbors.

Data structure: To accomplish this, we now introduce the notion of a "lifetime neighbor counts" (lt_cnt). The "lifetime neighbor count" of a data point $pi.lt\_cnt$ maintains a sequence of "predicted neighbor counts", each corresponding to the number of "predicted neighbors" p_i has in any of the future windows that p_i will participate in. For example, at a given window W_i, a data point p_i has three neighbors, which are p_1, p_2 and p_3. By using the "predictability', we could figure out the lifespan of each of these neighbors as well as that of p_i. Let us assume p_1 will expire after W_i. p_2 and p_3 will expire after W_{i+1}. p_i will expire after W_{i+2}. Then, at W_i, $p_i.lt\_cnt = (W_i : 3 - W_{i+1} : 2 - W_{i+2} : 0)$ indicates that p_i currently has 3 neighbors in W_i, while at (W_{i+1}), two of these three neighbors, namely p_2, and p_3 will still be its neighbors. p_1 will

no longer be p_i's neighbor then as it will expire after W_i. In other words, at W_i, p_i has two "predicted neighbors" in W_{i+1}. The length of $p_i.lt\_cnt$ is kept equal to $p_i.lifespan$, and thus decreases by one after each window slide by removing the left most entry. In this example, the $W_i : 3$ entry will be removed after the window slide. Here we note that all the "predicted neighbor counts" in $p_i.lt\_cnt$ are calculated based on p_i's neighbors in the current window and will later be updated when new data points join its neighborhood.

Lemma 6.1. *In any given window W_i, the entries in lt_cnt obey a "monotonic" decreasing function pattern.*

Lemma 6.1 holds because more and more data points in the current window will expire as the window slides.

Algorithm: Now we discuss the initialization and update of *lt_cnt*. For a data point p_i, its *lt_cnt*, $p_i.lt\_cnt$, is initialized when p_i arrives at the system. In particular, the length of $p_i.lt\_cnt$, $Len(p_i.lt\_cnt)$ is equal to $p_i.lifespan$ which can be determined by "predictability". Then we initialize $p_i.lt\_cnt$ as a vector of zeros.

At each window slide, each new data point runs a range query search to build its own *lt_cnt* and update those of its neighbors. In particular, when a new data point p_i finds a neighbor p_j, p_i first calculates the life-span of their *neighborship* $Neighbor(p_i,p_j).lifespan$ based on Observation 5.2. Then we increase the corresponding entries on p_i and $p_j's$ *lt_cnt* s by 1, namely $p_i.lt\_cnt[0]$ to $p_i.lt\_cnt[Neighbor(p_i,p_j).lifespan]$ and $p_j.lt\_cnt[0]$ to $p_j.lt\_cnt.[Neighbor(p_i,p_j).lifespan]$.

This update process pre-handles the impact of a data point's expiration on its neighbors' neighbor counts, because it is not counted as a neighbor in the windows in which it will not participate. This finishes the establishment of the abstracted *neighborship*.

Lemma 6.2. *No neighborship lt_cnt maintenance effort is needed when purging the expired data points.*

Proof. For any data point p_i, we pre-handle the impact of the expiration of $p_i's$ neighbors on $p_i.lt\_cnt$ by not counting them in the windows in which they will not participate. Thus, no maintenance effort is needed for $p_i.lt\_cnt$ when $p_i's$ neighbors expire. □

We now can update the neighbor counts for all the data points in the current window by just running one range query for each new data point. This single pass lookup provides sufficient information for detecting distance-based outliers. For each data point p_i, we simply need to compare $p_i.ln\_cnt[0]$ with $\theta^{fra} \times N$ to decide whether it is an outlier or not. Similarly, the *core objects* for the density-based clusters can be found by comparing $p_i.ln_cnt[0]$ with θ^{cnt}.

However, *lt_cnt* does not provide sufficient knowledge to generate the density-based clusters. This is because, although we could know all the *core objects* in the window, we do not know who their *edge points* are and which of them are within the same cluster. Abstract-C acquires such information by running an extra range query for each *core object* in the window in a Depth First Search manner at output stage. This is similar to the density-based clustering algorithm for static The pseudo code of Abstract-C is shown in Figs. 5 and 6.

```
Abstract-C (θ^range ,θ^cnt / θ^fra)
1 At each window slide
//Purge
2  For each expired data point p_exp
3    purge p_exp ;
//Load
5  For each new data point p_new
6    Initialize_lt_cnt (p_new )
7    load p_new  into index
//Neighborship Maintenance
8  For each new data point p_new
9    Neighbors
       = RangeQuerySearch(p_new , θ^range )
10   For each data point p_j in Neighbors
11     Updatelt_cnt (p_new , p_j )
//Output
12 OutputPatterns(pattern type);
Initialize_lt_cnt (p_i )
1 For n=1 to p_i .lifespan − 1
(p_i .lifespan = ⌈(p_i .T − Window.T_start)/Window.Slide⌉)
2  p_i .lt _cnt [n ] = 0 ;
Updatelt_cnt (p_i , p_j )
1    For n=1 to Len (p_j .lt _cnt )
2      p_i .lt _cnt [n ]++ ;
3      p_j .lt _cnt [n ]++ ;
```

Fig. 5. Pseudo-code for Abstract-C Part 1.

```
OutputPatterns(Distance-Based Outliers)
1 For each data point p_i in the window
2  If p_i .lt _cnt [0] ≤ θ^fra ∗ N
   //N is num of tuples in the current window
3    Output(p_i ) ;
4    remove p_i .lt _cnt [0] ;
OutputPatterns(Density-Based Clusters)
1 ClusterId=0;
2 For each data point p_i in the window
3    If p_i .lt _cnt [0] ≥ θ^cnt
4      remove p_i .lt _cnt [0] ;
5      If p_i is unmarked"
6        OutputCore(p_i , ClusterId);
7        ClusterId++;
OutputCore(p_c , ClusterId)
1 mark p_c with ClusterId;
2 output(p_c );
3 Neighbors =
     RangeQuerySearch(p_c , θ^range )
4 For each data point p_j in p_c .neighbors
5   If p_j is unmarked
6     If p_j has No less than θ^cnt neighbors
7       OutputCore(p_j , ClusterId)
8     Else
9       mark p_j with ClusterId;
10      output(p_j );
```

Fig. 6. Pseudo-code for Abstract-C Part 2.

Discussion: Abstract-C achieves linear[1] (in the number of data points in the window) memory consumption. This makes it a very efficient algorithm to detect distance-based

[1] The length of *lt_cnt* for each data point is equal to a constant number $C_{ils} = \lceil Q.win/Q.slide \rceil$.

outliers in terms of both memory and CPU. It takes N_{new} (the minimum number) range query searches at each window. However, since Abstract-C takes N_{core} extra range query searches (totally $N_{new}+N_{core}$) for detecting density-based clusters at each window, its performance largely depends on N_{core} the number of *core objects* in the window, which can vary from 0 all the way to N. This instability in CPU performance for the cluster pattern query class is the main shortcoming of Abstract-C, as our experiments confirm in Section 11.

7. Abstracted-neighborship-based solution using membership

Although Abstract-C achieves linear memory consumption, the extra range query searches may make it inefficient in terms of CPU time when detecting density-based clusters. Hence, we now design a third solution, Abstract-M, which aims to reduce the number of range query searches needed for detecting density-based clusters, while still keeping the linear memory utilization.

7.1. What abstract-C suffers from : "amnesia"

We note that the extra range query searches needed in Abstract-C are caused by its "amnesia". At every window, Abstract-C requires each new data point to run a range query to determine the *core points* in the window. Then, in addition, it requires each *core point* to run an extra range query search to produce the exact clusters. Unfortunately, the abstracted *neighborship* maintenance in Abstract-C, namely the *lt_cnt*, does not have the capability to preserve the cluster structures identified in the previous window. Repeatedly running range query searches to re-identify the existing cluster structures is a huge waste in terms of CPU time. So, to relieve the "amnesia" of Abstract-C, we propose to enhance the abstracted *neighborship* maintenance mechanism to capture and preserve the existing cluster structure.

7.2. Enhanced abstracted Neighborship : cluster membership – abstract-M

Abstract-M summarizes the *neighborship* among data points using a higher level abstraction, namely by means of the cluster membership. Specifically, Abstract-M marks the data points found to be in the same clusters with the same cluster IDs, and thereafter preserves such a markings for later windows. As an abstracted *neighborship* maintenance mechanism, such marking strategy avoids the "pair-wise" style *neighborship* storage structure applied in Exact-N that may require quadratic memory.

Although marking cluster memberships for data points at the initial window is straightforward, the maintenance of these memberships must now be carefully examined. Here we first identify all the possible changes on density-based cluster structures that may require updates on the cluster memberships of data points. With the aim to make the cluster memberships incrementally maintainable at any single update to the window, we examine change types based on both types of single updates, namely a removal (expiration) of an existing data point or an addition (participation) of a new data point. We call them *negative changes* and *positive changes* respectively. Further update may cause no change on the existing cluster structures.

Negative changes:

split: The members of an existing cluster now belong to at least two different clusters.

death: An existing cluster loses all its cluster members.

shrink: An existing cluster loses cluster member(s), but no split nor death happens.

Positive changes:

merge: The cluster members of at least two different existing clusters now belong to a single cluster.

expand: An existing cluster gains at least one new member, but no merge happens.

birth: A new cluster rises, but no merge happens.

These six change types cover all the possible changes that could be caused by any single update to the window. The *negative* and *positive changes* are mutually exclusive to each other, meaning the removal may cause *negative changes* only, while insertion may cause *positive changes* only.

7.3. Challenges for maintaining cluster memberships

After a careful examination of the costs of handling each change type, we found that the most expensive operation for incremental cluster membership maintenance lies in the handling of *negative changes*. We conclude our analysis by identifying the following challenge.

Observation 7.1. *We have a dilemma on solving the problem of determining and handling the* negative changes *on the density-based cluster structures. That is to determine the specific cluster members affected by the removal of a data point consumes either a large of amount of memory or CPU resources. In particular, it needs exact neighborships between the removed data points and their neighbors, which are extremely memory-consuming, or large numbers of range query searches, which could be very expensive in terms of CPU consumption.*

A key challenge for discounting the effect of expired data points lies in the detection and handling of the *split* of a cluster. In particular, when the expiration of data points causes a cluster to be *split*, the remaining data points in this split cluster need to be relabeled with different cluster memberships as they then belong to different clusters. In Fig. 7, the transaction from W_0 and W_1 shows an example of a split cluster. The expiration of data point 2 causes the cluster composed of *core points*, data points 6, 8 and 12 in W_0 to be split into two clusters, each containing only one *core point*. The expiration of only very few data points may cause a total break of the existing cluster structures into many small pieces, each may continue to persist as a smaller cluster or even may completely degrade to *noise*. Such *split* detection is non-trivial as elaborated upon below.

Observation 7.2. *Given connection information* (*links*) *among data points, the problem of detecting a split of a density-based cluster can be mapped to the graph-theoretic problem of identifying* "*cut-points*" *in a connected graph* [26]. *The complexity of this problem is known to be* $O(n^2)$,

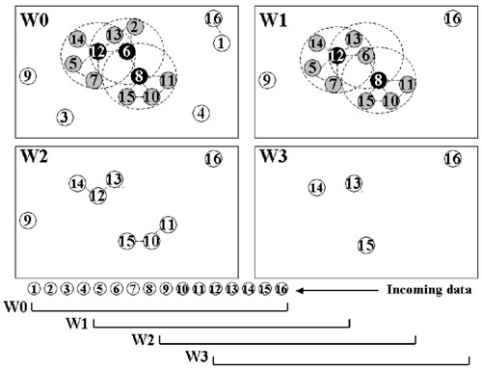

Fig. 7. "Predicted Views" of four successive windows at W_0. The number on each data points indicates its time stamp. The black data points are *core points*. The dashed circle around each *core point* denote its neighborhood, namely any data point in the dashed circle of a *core point* is its neighbor. The grey ones are *edge points* (dashed line) and the white ones are noise points. The edge between any two data points denotes the neighborship between them.

with n the number of vertices in the connected graph and in our case the number of core points in a cluster.

Moreover, our problem is harder than the problem of identifying the "cut-points", because we do not even have the explicit connection information, namely the exact *neighborships* among the existing data points in hand. Without such connection information, we have to again re-run expensive range query searches for every *core point* in the window whenever the window slides. Obviously, this will make Abstract-M no better than Abstract-C and thus defeats the purpose of the Abstract-M design.

7.4. View prediction technique for cluster membership maintenance

We now demonstrate how the "predictability" property (Definition 5) can once again be exploited to address the dilemma described in Observation 7.1. Specifically, by Observation 5.1, given the data points in the current window W_i, we always know the different subsets of them that will participate in each future window, W_{i+1}, W_{i+2} and so on. This will enable us to predetermine the cluster structures that will surely appear in each future window based on the data points in the current window. We call such prediction about the characteristics of future windows "predicted views". Fig. 7 gives an example of the data points falling into the current window W_0. Given these data points in W_0 and the window size $Q.win = 4$ (time units), the "predicted views" of the subsequent windows of W_0 (until all the data points belonging to W_0 expire), namely W_1, W_2 and W_3, are also shown in this figure respectively. Here, the number on each data point indicates its time stamp.

In particular, at time of W_0 (as shown in Fig. 7), there are 16 data points in W_0, namely the data points with time stamps from 1 to 16. At this moment, as we know the window will slide 4 time units in each of the next windows, we know that, among these 16 data points, the data points with time stamps 5–16 will surely be in the next window W_1 (those with time stamps 1–4 will be expired at the time of W_1). For the same reason, the data points with time stamp 9–16 and those with time stamps 13–16 will be in W_2 and W_3 respectively.

With such "predicted views", we can maintain the cluster structures in each future window *independently*. We call this technique "view prediction". This "view prediction" technique is a general principle that can be equally applied to many other pattern types, such as graphs, in streaming window semantics.

For density-based clustering, in particular, we "premark" each of the data points with the "predicted cluster membership" for each future window in its life span, if it belongs to any cluster in the corresponding windows. Then, at each window slide, we can simply update the "predicted views" by adding the new data points to each of them and then handling the potential *positive changes* caused by these additions. More specifically, for each window W_i, we update the "predicted cluster memberships" of each data point if it is involved in any *positive change* in this window caused by the participation of the new data points.

Using the example shown in Fig. 7, at the time of W_1 (as shown in Fig. 8), data points with time stamps 1–4 have already expired and thus been purged, while the new data points with time stamps 17–20 now will join W_1. To handle the impact of the expired data points, we simply need to discard the view of W_0. No computational effort is needed to handle such impact in W_1, W_2 and W_3, as these expired data points are never used to construct these predicted views at the first place. By doing so, all the expirations are predicted and preprocessed. Based on this foundation provided by the view prediction technique, we develop the following lemma.

Lemma 7.1. *By using the "view prediction" technique to incrementally maintain the cluster memberships for density-based clusters, we eliminate the need to discount the effect of expired data points to extracted clusters. Thus we simplify the problem of incremental density-based cluster detection to the much simpler problem of handling the addition of new data points only on such clusters.*

Proof. We pre-handle the expiration of data points by not using them for cluster formation in the windows that they will not participate in. Therefore, no maintenance of the

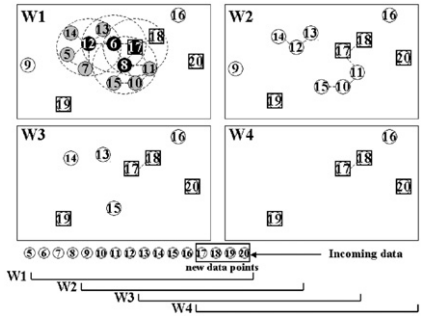

Fig. 8. Updated "Predicted Views" of four successive windows at W_1. The data points with time stamps 1–4 have expired and thus do not appear in W_1 and its subsequential windows. The data points 17–20 are new data points from the input stream falling into W_1.

cluster structures is needed for these windows when those "not-used" objects are purged. □

To handle the impact of the new data points with time stamps 17–20, we first create a new window W_4, which is the last window that these new data points can participate in before they expire. Then, we insert these data points into the predicted views of the W_1, W_2 and W_3. Fig. 8 demonstrates the updated views of W_1, W_2, W_3 and W_4, which are computed when the new data points join W_1. Fortunately, handling the insertion of new data points is much easier than removal. We will discuss the specific maintenance process for each type of *positive change* after the introduction of our proposed data structure in the later part of this work.

7.5. Abstract-M algorithm

Based on the abstract neighborship, we design the Abstract-M algorithm. This algorithm uses both the "view prediction" techniques and abstract *neighborship* (cluster membership) maintenance. By maintaining the cluster memberships of data points in predicted views, Abstract-M avoids the expensive cost for handling *negative changes* on density-based clusters. Also, since the cluster memberships maintained by Abstract-M indeed capture and preserve the cluster structures, Abstract-M no longer needs one extra query search for each *core point* at the output stage as Abstract-C does to re-build the clusters. As a stepping-stone algorithm, the details of Abstract-M are covered in our previous conference version [1] but omitted here.

While a significant step forward, Abstract-M does not completely "cure" the "amnesia" suffered by Abstract-C, as it still requires a certain number of extra range query searches, namely $N_{prmtcore}$ (number of existing data points that are "promoted" as new cores) extra range query searches at each window. This is because, as we analyzed earlier, the newly arrived data points may promote the existing *non-core points* to become *core points*. In such cases, the promoted *core points* need to communicate with their neighbors about their new "roles" and thus update the cluster memberships, such as two clusters merged. However, as Abstract-M only maintains cluster membership for each data point, the promoted *core points* have no direct access to their neighbors and thus each of them needs a range query search to broadcast its new role to its neighbors.

Considering the expensiveness of range query searches and the fact that $N_{new}+N_{prmtcore}$ could be as large as N even when N_{new} is very small, Abstract-M does not make the ideal solution that keeps the number of range query searches minimal (N_{new}) and the memory consumption linear. The reason for this is that the enhanced abstracted *neighborship* maintenance mechanism, namely the cluster memberships, still cannot completely represent the *neighborships* among the data points. The data points marked with cluster memberships still have no knowledge about who their neighbors are.

8. Exact+abstracted *neighborship* based solution (Extra-N)

By carefully analyzing the strengths and weaknesses of prior algorithms, we finally propose an ideal solution achieving the merits in terms of both memory and CPU utilization based on a more capable *neighborship* maintenance mechanism.

Challenges: To achieve the minimum number of range query searches (N_{new}) at each window, we need to completely avoid re-searching for any *neighborships* that have been identified before. This indicates that we have to give data points direct access to their neighbors whenever communication between them is needed. However, this leads to a dilemma in the design of the *neighborship* maintenance mechanism as explained below.

Observation 8.1. *On one hand, to give data points direct access to their neighbors, we have to preserve all the exact neighborships identified in earlier windows. On the other hand, to keep the memory consumption linear, we cannot afford to store the exact neighborships in the window.*

Accommodating these two conflicting goals within a single *neighborship* maintenance mechanism is the key challenge that we need to address for our algorithm design.

Solution: We now propose a strategy that successfully tackles this problem by achieving optimality in both memory and CPU consumption. We call this the Ex act+abstra cted Neighborship based solution (Extra-N). Extra-N combines the *neighborship* maintenance mechanisms proposed in Exact-N, Abstract-C and Abstract-M into one integrated solution. This solution overcomes the shortcomings of the prior solutions while keeping their respective benefits.

We observe that different types of *neighborship* abstractions are most useful during different stages of a data point's life-span. In particular, we need to maintain the exact *neighborships* for a data point in its "non-core point career", while abstracted *neighborships* will be sufficient for its "core point career". More precisely, Extra-N marks each data point p_i by a cluster membership in each window in which it is predicted to be a *core point*, while it keeps the exact *neighborships* (links) to all p_i's predicted neighbors for the windows where p_i is predicted to be a *noise* or an *edge point*. Such hybrid *neighborship* maintenance mechanism carries sufficient information to produce the density-based clusters. Namely, all *core points* in a window W_i are marked with a cluster membership, and all the *edge points* can quickly figure out their cluster memberships by checking those of the *core points* in their neighbor list. We will next demonstrate that Extra-N employs only the minimum number of range query searches while keeping the memory consumption linear.

Data structure: As mentioned earlier, Extra-N combines the *neighborship* maintenance mechanisms used by all previous three algorithms discussed in this work. In particular, Exact-N inherits the two "life time marks" from Abstract-M, namely *lt_cnt* (life time neighbor counts) and *lt_type* (life time types). In addition, Extra-N introduces a new "life time mark" called "life time hybrid *neighborship*" (*lt_hn*), which stores the "predicted cluster memberships" and the "predicted neighbors" of a data point across different windows in a compact structure. We call the overall data structure composed of *lt_cnt*, *lt_type* and *lt_hn* the Hybrid *neighborship* Mark (*H-Mark)* for a data point.

Fig. 9 depicts the *H-Marks* of the data points in Fig. 7. As shown in Fig. 9, we use the columns named *C*, *T* and *H*

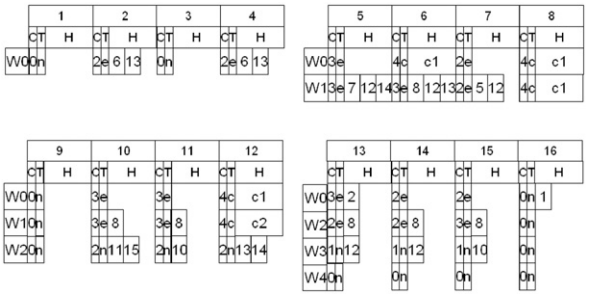

Fig. 9. The H-Marks of the data points at W_0.

to present the *lt_cnt*, *lt_type* and *lt_hn* of each data point respectively. Since *lt_cnt* has been carefully discussed in Section 6 and *lt_type* is easy to understand, here we explain *lt_hn*. For example, at W_0, the *core point*12 is predicted to be *core point* also in W_1. Thus it is marked by cluster memberships in both windows ($p_{12}.lt\_hn[0]=$ "*c*1", $p_{12}.lt\_hn[1]=$"*c*2"). Then, as it is predicted to be a *non-core point*, more precisely, a *noise* point in W_2, we start to keep the predicted neighbors of p_{12} from this window onwards ($p_{12}.lt\_hn[2]=\{p_{13},p_{14}\}$). Since the number of "predicted neighbors" of a data point follows a monotonically decreasing function (discussed earlier in Section 6), the "non core object career" windows of a data point are continuous and right after its "core object career" windows. Here we note that although we maintain the neighbor lists of each data point p_i for all its "non-core point career" windows, the link to each of these neighbors is only physically stored once in *lt_hn*, no matter how many times it appears in $p_i's$ neighborhood in different windows. This means that the number of predicted neighbors each data point p_i keeps track of is equal to the maximum number of predicted neighbors it has among all its "non-core point career" windows. Given the monotonicity property, this is equal to the number of predicted neighbors it has in its first "non-core point career" windows. For example, data point 13 in Fig. 9 has in total 3 predicted neighbors, namely data points 2, 6, and 12, in its first "non-core point career" window W_0. At the same moment, its predicted neighbors in later windows are subsets of these three. For ease of expiration, a predicted neighbor $p_j$of the data point p_i is stored in the specific row of p_i's *H-Mark* corresponding to the last window in which their *neighborship* will hold. The pseudo code of Extra-N is shown in Fig. 8.

Lemma 8.1. *Extra-N has a memory consumption linear in the number of data points in the window.*

Proof. Since the maximum number of predicted neighbors of each "non core point" p_i is less than the constant θ^{cnt} (otherwise p_i would have been classified as *core point*), and we already know that $p_i.lt\_cnt$, *lt_type* and *lt_hn* all have a constant length $\leq C_{ils}$ (defined in Section 6), *H-Mark* of any data point is of a constant size. This proves Lemma 8.1. □

Algorithm: Similar to Abstract-M, at each window slide, Extra-N runs a range query search for each new data point to update the "predicted views" of future windows. However, the hybrid *neighborship* maintenance mechanism brings the key advantage to Extra-N of eliminating any extra range query searches from the update processes. That is when *promotions* happen to the *non core points*, they now have direct access to their neighbors. Thus the *promoted cores* no longer need to run any range query search to re-collect their neighbors.

Lemma 8.2. *Extra-N achieves the minimum number of range query searches needed for detecting density-based clusters at each window.*

Proof. First, since Extra-N inherits the *neighborship* maintenance mechanism from Abstract-M, it needs at most $N_{new}+N_{prmtcore}$ range query searches at each window as Abstract-M does. Second, we know that the $N_{prmtcore}$ range query searches are caused by the handling of *promotions*. Lastly, no range query search is needed when the *promotions* happen in Extra-N. Thus, Extra-N only needs N_{new} queries at each window. This proves Lemma 8.2. □

Conclusion: Finally, we conclude the optimalities in terms of both CPU and memory utilization achieved by Extra-N algorithm in the following theorem.

Theorem 8.2. *For detecting density-based clusters, Extra-N requires the minimum number of range query searches needed by this problem at each window (by Lemma8.2), while keeping the memory consumption linear in the number of data points in the window (by Lemma8.1).*

These properties make Extra-N a very efficient solution for detecting density-based clusters over sliding windows in terms of both CPU and memory resource utilization.

9. Discussion on output models

Although in this work we focus on generating complete pattern detection results for each window, our proposed algorithms can be easily extended to handle other output formats. Different output models may be required by different applications depending on the environmental constraints, such as potentially expensive transmission costs. Generally, we can report pattern detection results in three different models, namely complete output, incremental output, and evolution summary. The techniques to generate the complete output for both neighbor-based pattern types have been carefully discussed in the previous parts of this work. They now represent the basis for the other two output models. The other two output models are needed when the users are not only interested in the patterns existing in each window, but also interested in how the patterns detected in different windows are correlated.

Evolution summary: Instead of outputing exact pattern members, the evolution summary aims to provide summary information of how the patterns have changed from the last window to the current one. It is suitable for the applications where users are monitoring the data streams, while their reactions are needed only if significant changes happened to the patterns. It is also the most transmission-efficient output format, as it only outputs summary information when actual changes in the patterns arise.

To output the evolution summary, we need to define the evolution semantics that are able to capture all possible changes on the patterns types. Such semantics can be defined by comparing the patterns detected from the current and the previous window by our proposed algorithms. For distance-based outlier detection, since each pattern, namely each outlier, corresponds to a single data point, the change semantics are straightforward. We can simply use *birth* of new outliers and *death* of existing outliers to express all the possible changes that may happen between two windows. However, the evolution semantics for density-based clusters are more interesting and complex. Namely, we have to deal with the "many-to-many" relations among the extracted patterns, such as merge and split. More details of pattern evolution detection can be found in our work [1].

Incremental output: The incremental output format aims to provide exact pattern members to the users, while it outputs the incremental differences between the patterns existing in the current and the previous window. It is suitable for situations when the users need the precise patterns, while the transmission costs are considerable and thus should be minimized.

For incremental output, we assume that both the users (client side) and pattern detection system (server side) keep the complete patterns detected from the last window, namely all the outliers and all the cluster members with corresponding cluster ids respectively depending on their targeted patterns. When our system (server side) finishes the pattern detection for the new window, instead of re-outputing and transmitting the complete result, we determine and output the increments between the patterns detected in the two adjacent windows. Such increments will update the patterns stored at the client side and eventually form the patterns in the current window. The incremental output format is closely related to the evolution semantics, especially for the density-based clusters. This is because we may need to re-label the cluster members in the previous windows with different cluster memberships when the cluster structure changes. Such re-labeling mechanism has to be decided by the evolution semantics. We leave the design of the re-labeling and incremental output mechanism as our future work.

Table 1
Symbols used in the cost models.

Average number of expired data points	N_{exp}
Average number of new data points	N_{new}
Average number of "core points"	N_{core}
Average number of "promoted core points"	$N_{prmtcore}$
Number of neighbors for a specific data point p_i	N_{p_i}
Average number of data points	N
Average initial life-span for each data point	C_{ils}

10. Cost models and cost analysis

To better analyze and compare the algorithms we discussed in this work, we now design cost models for modeling both the CPU and memory consumption in these continuous neighbor-based pattern detection processes. We establish a CPU cost model capturing the response time of each algorithm to answer the query in each individual window. Such response time includes all the time consumed by the four stages of the neighbor-based pattern detection process (discussed in Section 3), namely the purging, loading, *neighborship* maintenance, and output. The memory cost model is designed to describe the memory space utilized by each algorithm. Such memory utilization includes the memory space for storing both the raw data and also the meta-information in each window.

We first define the symbols used in our cost models in Table 1. These symbols are used to indicate the information for a single window.

10.1. Cost of each algorithm

CPU costs: Now we define the CPU costs of primitive operations in the neighbor-based pattern detection processes in Table 2.

We use the average costs in the estimation of c_{lt_cnt}, c_{mt} and c_{hm}. This is because such cost for a specific data point is decided by its life-span, which indicates the number of "predicted views" that need to be updated. We assume the life-spans of the data points in each window are uniformly distributed from 1 to C_{ils}, thus we use $C_{ils}/2$ to present the average. This assumption holds for all count-based window cases and also for time-based window cases if the input rate is stable. The same estimation is used for memory costs of *lt_cnt*, *mt* and *hm* later.

Given the costs of individual operations, we now design models for the CPU cost of each algorithm. Again,

Table 2
CPU cost of individual operations in our cost model.

CPU cost of purging a data point	c_p
CPU cost of loading a data point into index	c_l
CPU cost of removing/establishing an exact *neighborship* (single-directional)	c_n
CPU cost of updating a integer attribute	c_i
CPU cost of running a range query search	c_{rqs}
CPU cost of examining a data point during the output	c_o
CPU cost of updating the *lt_cnt* of a data point	$c_{lt\_cnt} = \frac{C_{ils}}{2} c_i$
CPU cost of updating the M-Table of a data point	$c_{mt} = \frac{3C_{ils}}{2} c_i$
CPU cost of updating the H-Marks of a data point	$c_{hm} = C_{ils} * c_{int} + \frac{\theta^{cnt}}{2} c_i$

the CPU cost is the sum of the cost for the four stages of purging, loading, *neighborship* maintenance and output. We use the symbols, C_{purge}, C_{load}, C_{nei_main} and C_{output} to indicate the cost of each algorithm for these stages respectively. Also, we use superscripts to denote the cost of a specific algorithm and the objective pattern type along with these symbols. For example, $C_{purge}^{Exact-N(c)}$ indicates the cost of purging for Exact-N to detect density-based clusters (c), while $C_{nei\_main}^{Abs-C(o)}$ indicates the cost of *neighborship* maintenance for Abstract-C to detect distance-based outliers (o). For a given algorithm, if the cost of a certain stage is the same for both of the pattern types (density-based clusters and distance-based outliers), we omit the pattern type part of the superscript and only use the stage name as well as the algorithm name to generalize the cost for this stage. For example, $C_{purge}^{Exact-N}$ indicates the cost of purging for Exact-N to detect either of the two pattern types. Note, algorithms Exact-N, Abstract-C and the naive approach handle both cluster and outlier detection, while Abstract-M and Extra-N support clustering only. The specific costs of each alternative algorithm at each query processing stage are listed in Table 3.

Naive approach: (1) Purge cost: remove all expired data points from the window. (2) Load cost: load all new data points into the index. (3) *Neighborship* maintenance cost: for all data points in the window, run a range query search to form the patterns. (4) Output for clusters: check each data point in the window.

Exact-N algorithm: (1) Purge cost: remove all expired data points. Then for each data point remaining from the window, remove the expired neighbors from its neighbor list. (2) Load cost: load all new data points into the index. (3) *Neighborship* maintenance cost: for each new data point p_i, run a range query search. Then for each of its neighbor p_j found, add p_i and p_j to each others' neighbor list. (4.1) Output for clusters: check each data point in the window and run depth first search on all *core points*. (4.2) Output for outliers: check each data point in the window.

Abstract-C algorithm: (1) Purge cost: remove all expired data points from the window. (2) Load cost: load all new data points into the index. (3) *Neighborship* maintenance cost: for each new data point p_i, run a range query search. Then for each of p_i's neighbor p_j found, update p_i and p_j's *lt_cnt*. (4.1) Output for clusters: check each data point in the window and run one range query search for each *core point* to form clusters. (4.2) Output for outliers: check each data point in the window.

Abstract-M algorithm: (1) Purge cost: remove all expired data points from the window. (2) Load cost: load all new data points into the index. (3) *Neighborship* maintenance cost: for each new data point p_i, run a range query search. Then for each of p_i's neighbor p_j found, update p_i and p_j's *lt_mt*. Also, for each promoted *core point*, run a range query search and update its *lt_mt*. (4) Output for clusters: check each data point in the window.

Extra-N Algorithm: (1) Purge cost: remove all expired data points from the window. (2) Load Cost: load all new data points into the index. (3) *Neighborship* maintenance cost: for each new data point p_i, run a range query search. Then for each of $p_i's$ neighbor p_j found, update p_i and $p_j's$ H-Marks. (4) Output for clusters: check each data point in the window.

Memory costs: Similarly, we define the memory cost of individual data structures in Table 4 before we discuss the memory cost of each algorithm.

Again, we use $C_{ils}/2$ to represent the average length of each data point's life-span, namely the number of "predicted views" that the data point needs to be maintained in. Since *lt_cnt* for each point is simply composed by $C_{ils}/2$ neighbor counts (integers), thus it memory cost is $(C_{ils}/2)m_i$. For *mt*, as each data point needs to maintain three integers, namely a neighbor counter (integer), a type indicator (integer) and a cluster membership (integer), for one "predicted view" its memory cost is $(3C_{ils}/2)m_i$. For *hm* of each data point, it needs to maintain two integers, namely a neighbor counter (integer) and type indicator (integer) for one "predicted view". Also, it needs to maintain a certain numbers of cluster memberships (between 0 and $C_{ils}/2$) and exact neighbors (between 0 and θ^{cnt}). We use half of the maximum numbers

Table 3
CPU costs of alternative algorithms at four stages by our cost model.

C_{purge}^{Naive}	$N_{exp}*(c_p)$
C_{load}^{Naive}	$N_{new}*c_l$
$C_{nei\_main}^{Naive}$	$\sum_{1 \le i \le N_{new}}(c_{rqs})$
C_{output}^{Naive}	$N*c_o$
$C_{purge}^{Exact-N}$	$\sum_{1 \le i \le N_{exp}}(\sum_{1 \le j \le N_{p_i}} c_n + c_p)$
$C_{load}^{Exact-N}$	$N_{new}*c_l$
$C_{nei\_main}^{Exact-N}$	$\sum_{1 \le i \le N_{new}}(c_{rqs} + \sum_{1 \le j \le N_{p_i}} 2*c_n)$
$C_{output}^{Exact-N(c)}$	$N*c_o + \sum_{1 \le i \le N_{core}}(\sum_{1 \le j \le N_{p_i}} c_o)$
$C_{output}^{Exact-N(o)}$	$N*c_o$
C_{purge}^{Abs-C}	$N_{exp}*(c_p)$
C_{load}^{Abs-C}	$N_{new}*c_l$
$C_{nei\_main}^{Abs-C}$	$\sum_{1 \le i \le N_{new}}(c_{rqs} + \sum_{1 \le j \le N_{p_i}} c_{lt\_cnt})$
$C_{output}^{Abs-C(c)}$	$N*c_o + \sum_{1 \le i \le N_{core}} c_{rqs}$
$C_{output}^{Abs-C(o)}$	$N*c_o$
$C_{purge}^{Abs-M(c)}$	$N_{exp}*(c_p)$
$C_{load}^{Abs-M(c)}$	$N_{new}*c_l$
$C_{nei\_main}^{Abs-M(c)}$	$\sum_{1 \le i \le N_{new}}(c_{rqs} + \sum_{1 \le j \le N_{p_i}} c_{mt}) + \sum_{1 \le i \le N_{prmtcore}}(c_{rqs} + c_{mt})$
$C_{output}^{Abs-M(c)}$	$N*c_o$
$C_{purge}^{Extra-N(c)}$	$N_{exp}*(c_p)$
$C_{load}^{Extra-N(c)}$	$N_{new}*c_l$
$C_{nei\_main}^{Extra-N(c)}$	$\sum_{1 \le i \le N_{new}}(c_{rqs} + \sum_{1 \le j \le N_{p_i}} c_{hm}) + \sum_{1 \le i \le N_{prmtcore}}(c_{hm})$
$C_{output}^{Extra-N(c)}$	$N*c_o$

Table 4
Memory costs of individual data structures in our cost model.

Memory cost of a data point	m_p
Memory cost of an exact *neighborship* (single-directional)	m_n
Memory cost of an integer attribute	m_i
Memory cost of the *lt_cnt* of a data point (used by Abstract-C)	$m_{lt\_cnt} = \frac{C_{ils}}{2} m_i$
Memory cost of the M-Table (mt) of a data point (used by Abstract-M)	$m_{mt} = \frac{3C_{ils}}{2} m_i$
Memory cost of the H-Marks (hm) of a data point (used by Extra-N)	$m_{hm} = \frac{5C_{ils}}{4} m_i + \frac{\theta^{cnt}}{2} m_n$

for both of their estimations, which would be the case when the time stamps of the input data are uniformly distributed or queries are using count-based windows. Thus its memory cost is $(5C_{ils}/4)m_i+(\theta^{cnt}/2)m_n$.

The memory costs of each algorithm are listed in Table 5. Here we note that since the three algorithms, namely the Exact-N, the Abstract-C, and the naive solution have the same memory costs when detecting density-based clusters and distance- based outliers, we do not distinguish them in our cost model. We put a (c) after each algorithm designed to detect density-based clusters only. Basically, the memory cost of each algorithm are composed of the utilization for storing the raw data $N*m_p$ and the utilization for storing the corresponding meta-data about each data point.

10.2. Cost analysis

Analysis for density-based clusters algorithms: We first analyze the costs of the algorithms for detecting density-based clusters in Table 6. There are several observations that can be made through our analysis shown in Table 6. (1) There are two major factors affecting the performance of all algorithms, namely $\overline{N}_{(p_i)}$ the average number of neighbors each data point has and N_{new} the average number of new data points in each window (except that the performance of the naive solution depends on N instead of N_{new}). $\overline{N}_{(p_i)}$ has a great influence on the cost for *neighborship* maintenance, as it determines the number of *neighborships* exist in each window. The increase of $\overline{N}_{(p_i)}$ will cause increases in the costs for all algorithms, as the range query searches become more expensive and each data point needs to communicate with more neighbors to update the meta-data. N_{new} is the proven lower bound for the minimum number of range query searches needed at each window for neighbor-based pattern detection (see Section 4). Obviously, the larger N_{new} is, the more range query searches are needed for all algorithms. The only exception is the naive solution that needs N range query searches in all cases.

(2) Only Exact-N and Extra-N guarantee N_{new} the minimal number of range query searches at each window and thus avoid from the most expensive operations to the best level. All the other alternative algorithms may need extra range query searches depending on the characteristics of the input data.

(3) The performance of Exact-N is very sensitive to $\overline{N}_{(p_i)}$, namely its memory overhead becomes quadratic in N when $\overline{N}_{(p_i)}$ approaches N. This makes it work well only when $\overline{N}_{(p_i)} \Downarrow$, meaning $\overline{N}_{(p_i)}$ is very small. All three predictability-based solutions, Abstract-C, Abstract-M, and Extra-N, have linear memory overhead in all cases.

(4) The performance of Abstract-M is largely decided by $N_{prmtcore}$, which may significantly increase the number of range query searches that Abstract-M needs to run at each window. As $N_{prmtcore}$ can potentially be as large as the size of all data points inherited from the previous window $N-N_{new}$, it makes Abstract-M suffer from the risk of having an even worst performance than the Naive solution. The same problem may happen to Abstract-C as well, when N_{core} approaches N.

(5) The cost of the predictability-based solutions drops even lower as the constant C_{ils} decreases, because it decides upon the number of "predicted views" to be stored and updated. Besides C_{ils}, the performance of Extra-N is also affected by another constant θ^{cnt}, which works as the upper bound for the number of exact *neighborships* each data point stores. Although we list these two factors for the completeness of our analysis, they are not the key factors deciding the algorithms' performance. This is because they are unrelated to the number of range query searches needed and are usually very small constants compared with N.

Analysis for distance-Based outlier algorithms: For detecting distance-based outliers in sliding windows, Abstract-C achieves both the minimal number of range query searches and linear memory requirement, while Exact-N suffers from a potentially quadratic memory overhead. The naive solution does not take advantage of incremental computation.

Conclusion: Based on our cost analysis, we conclude that Extra-N and Abstract-C are the best solutions for detecting density-based clusters and distance-based outliers over sliding windows respectively. To validate our claims derived from this analytical evaluation, a thorough experimental study is presented in Section 11.

Table 5
Memory costs of each algorithm in our cost model.

Memory cost of the Naive Solution	$N*(m_p+m_{int})$
Memory cost of Exact-N	$\sum_{1\le i\le N}(m_p+\sum_{1\le j\le N_{p_i}} m_n)$
Memory cost of Abstract-C	$N*(m_p+m_{lt\_cnt})$
Memory cost of Abstract-M(c)	$N*(m_p+m_{mt})$
Memory cost of Extra-N(c)	$N*(m_p+m_{hm})$

11. Experimental study

A thorough experimental study evaluating the continuous neighbor-based pattern detection algorithms is presented in this section. In our experiments, for each algorithm we first utilize synthetic data to observe its scope of applicability for a wide range of parameter settings. Our experiments cover

Table 6
Cost analysis of each algorithm. (↓ means when the specific factor is small. ⇓ means very small. ↑ means large. We use () on a factor if its impact is minor.)

	Exact-N	Abstract-C	Abstract-M	Extra-N	Naive
Num of rqs	N_{new}	$N_{new}+N_{core}$	$N_{new}+N_{prmtcore}$	N_{new}	N
Worst case memory overhead	N^2*m_n	$N*\frac{C_{ils}}{2}*m_{int}$	$N*\frac{3C_{ils}}{2}*m_{int}$	$N*(C_{ils}*m_{int}+\theta^{cnt}*m_n)$	$N*m_i$
Performance factors	$\overline{N}_{(p_i)}$ N_{new}	$\overline{N}_{(p_i)}$ N_{new} N_{core} (C_{ils})	$\overline{N}_{(p_i)}$ N_{new} $N_{prmtcore}$ (C_{ils})	$\overline{N}_{(p_i)}$ N_{new} $(C_{ils}$ $\theta^{cnt})$	$\overline{N}_{(p_i)}N$
More efficient if	$\overline{N}_{(p_i)} \Downarrow N_{new}\downarrow$	$\overline{N}_{(p_i)}\downarrow N_{new}\downarrow N_{core}\downarrow (C_{ils}\downarrow)$	$\overline{N}_{(p_i)}\downarrow N_{new}\downarrow N_{prmtcore}\downarrow (C_{ils}\downarrow)$	$\overline{N}_{(p_i)}\downarrow N_{new}\downarrow (C_{ils}\downarrow)(\theta^{cnt}\downarrow)$	$\overline{N}_{(p_i)}\downarrow N\downarrow (N_{new}\uparrow)$

all major combinations of the important cost factors as identified in our cost analysis (Section 9). Then, for a closer comparison, we zoom into the parameter space and focus on the cases in which their performance showed noticeable differences. Last but not least, to confirm the superiority of our proposed methods in real applications, we also evaluate them along with other alternative methods using real streaming datasets.

11.1. Experiment setup and data sets

All experiments are conducted on a HP Pavilion dv4000 laptop with Intel Centrino 1.6 GHz processor and 1GB memory, which runs Windows XP professional operating system. The algorithms are implemented with VC++ 6.0.

Real datasets: We use two real streaming datasets in our experiments. The first dataset GMTI (Ground Moving Target Indicator) [16] records the real-time information of the moving ground vehicles and helicopters that would be gathered by both ground stations and dedicated aircraft. In particular, the GMTI dataset we use, which is provided by MITRE Corporation, has around 100,000 records regarding the information of vehicles and helicopters (speed ranging from 0 to 200 miles per hour) moving in a certain area. These records are gathered by 24 different data ground stations or aircraft in 6 h. GMTI has 14 dimensions, including the latitude and longitude of each target's position. In our experiment, we detect clusters and outliers based on the targets' position information.

The second real dataset we use is the Stock Trading Traces data (STT) from [23]. This dataset has about one million transaction records throughout the trading hours of a day. Each transaction contains the name of the stock, time of sale, matched price, matched volume, and trading type. We use all the five attributes in our experiments while detecting the clusters based on the matched price, volume and trading type.

Synthetic datasets: For a thorough evaluation of density-based clustering algorithms, we built a synthetic data generator to create a variety of controlled datasets containing clusters with different characteristics, along with noise. For each synthetic dataset with *NumOfDim* dimensions, the data generator first creates *NumOfClu* non-overlapping clusters sized *CluSize*, each following a Gaussian Distribution but with different randomly selected *mean* and *variance* parameter values. Then, it scatters $NumOfClu \times CluSize \times NoiseRate$ random noise data points into the data space. These two steps generate the first portion data of a synthetic dataset. Then the data generator repeats these two steps up to 1 K times to reach the size of the data we desire. During the repetitions, all input parameters (denoted in italics) remain the same except *mean* and *variance* of each cluster, which are randomly varied.

For the evaluations of distance-based outlier detection algorithms, we use the Gauss dataset, which is a synthetically generated time sequence of 35,000 one dimensional observations. It consists of a mixture of three Gaussian distributions with uniform noise. This is the dataset used by the only previous work [5] on detecting distance-based outliers in continuous windows.

11.2. Experimental methodologies

We run all experiments using both synthetic and real data for 10 K windows. For the experiments that involve data sets larger than the sizes of the real datasets, we append multiple rounds of the original data varied by setting random differences on all attributes, until the data stream reaches the desired size.

We measure two key metrics for stream processing algorithms, namely response time and memory footprint. Those two metrics are also evaluated by our cost models in Section 10. In particular, we measure the average response time (referred as CPU time henceforth) it takes to answer a pattern detection query at each window. This response time includes the time consumed by all four stages of pattern detection at each window. The response time is averaged over all windows in each experiment. The memory footprint indicates the peak memory space consumed by an algorithm is record over all the windows.

11.3. Evaluations for density-based cluster detection methods

Overall evaluation: To compare the performance of all five algorithms discussed in this work, namely Exact-N, Abstract-C, Abstract-M, Extra-N and the naive solution, we conduct a comprehensive experiment with a wide range of the synthetic data created by our generator. These experiments cover all important combinations of the two major cost factors identified in our cost analysis (Section 10), namely $\overline{N}_{(p_i)}$ and N_{new}.

To avoid the performance fluctuations caused by different base sizes, namely different number of data points in the window, we use count-based windows (equal in concept to time-based windows with uniform data rates). Thus, N_{new} is equal to the slide size $Q.slide$, and $\overline{N}_{(p_i)}$ is controlled by adjusting two input parameters of the data generator. More specifically, we can increase $\overline{N}_{(p_i)}$ by expanding the size of each cluster *CluSize* while decreasing the *variance* of its Gaussian Distribution (Fig. 10).

To cover all the major combinations of these two factors, we vary $\overline{N}_{(p_i)}$ from 1% to 50%, and $Q.Slide$ from 10% to 100%, both in terms of the percentage to window size $Q.win$ and both with 7 different settings. In particular, the seven different $\overline{N}_{(p_i)}$ settings represent the data from "very sparse" ($\overline{N_{(p_i)}} = 1\%$), "medium dense" ($\overline{N_{(p_i)}} = 20\%$) and finally to "very dense" ($\overline{N_{(p_i)}} = 50\%$). The 7 different $Q.Slide$ settings, which are 10% to 50% with 10% increments plus 80% and 100%, cover all the increments from "mostly remaining" ($Q.Slide = 10\%$), "half-half" ($Q.Slide = 50\%$), "mostly new" ($Q.Slide = 80\%$) and finally to "all new" ($Q.Slide = 100\%$). We measure the CPU time (shown in Fig. 11) as well as the memory footprint (shown in Fig. 12) of the five algorithms for all $7 \times 7 = 49$ combinations. Other settings of this experiment include window size $Q.win = 5$ K, $\theta^{range} = 0.003$ and $\theta^{cnt} = Q.win \times 5\% = 250$.

From Figs. 11 (CPU) and 12 (memory), we observe that Abstract-M and Extra-N clearly outperform the other three algorithms, namely the Exact-N, Abstract-C and the naive solution, in most of the test cases. Besides the naive solution which does not take advantage of incremental computation,

```
Extra-N (θ^range, θ^cnt)
1  For each Window Slide
2      For each expired data point p_exp // Purge
3          purge p_exp;
4      For each new data point p_new // Load
5          InitializeHMark (p_new)
6          load p_new into index
7      For each new data point p_new // neighborship Maintenance
8          Neighbors = RangeQuerySearch(p_new, θ^range)
9          UpdateHMark (p_new, Neighbors)
10     OutputPatterns(PatternType); // Output

InitializeHMark (p)
1  length := ⌈(p.T − Window.T_start) / Window.Slide⌉ ;
2  set the lengthes of p.lt_cnt, lt_type and lt_hn to length;
3  For n:=1 to length do
4      p.lt_cnt[i] := 0;
5      p.lt_type[i] := "n";
6      p.lt_hn[i] := "empty";

UpdateHMark (p, Neighbors)
1  For i:=1 to Len(p.lt_hn)
2    For j:=1 to Len(Neighbors)
3      If Len(Neighbors[j].lt_hn) < i
4        remove Neighbors[j] from Neighbors
5      Else If Neighbors[j] is NOT New
6        Neighbors[j].lt_cnt[i] + + ;
8        add p to Neighbors[j].lt_hn if not added ;
9        add Neighbors[j] to p.lt_hn if not added ;
9        If Neighbors[j].lt_cnt[i] ≥ θ^cnt
10         Mark(Neighbors[j], i);
11   p.lt_cnt[i] := Len(Neighbors);
12   If p.lt_cnt[i] ≥ θ^cnt
13     Mark(p,i);

Mark(p,i)
1  p.lt_type[i] := "c";
2  tempH = "empty";
3  For each p's predicted neighbor p_j;
4      If p_j.lt_type[i] = "c" AND tempH ≠ p_j.lt_hn[i]
5          equalize tempH with p_j.lt_hn[i] ;
6      tempH := p_j.lt_hn[i];
7  If tempH = unmarked
8      tempH := ClusterId[i];
9      ClusterId[i] + +;
10 For each p's predicted neighbor p_j;
11     If p_j.lt_type[i] = "n";
12         p_j.lt_type[i] := "e" ;
12         p_j.lt_hn[i] := tempH;
13 remove all the pointers in p.lt_hn[i] (if any);
14 p.lt_hn[i] := tempH;

OutputPatterns(Density-Based Clusters)
1  For each data point p_i in the window
2      If p_i.lt_type[1] ≠ "n"
3          output(p_i);
4      remove first elements on p_i.lt_cnt, p_i.lt_type and p_i.lt_hn;
```

Fig. 10. Extra-N algorithm.

the other two incremental algorithms Exact-N and Abstract-C suffer from a huge consumption of either memory space or CPU time in many cases.

In particular, as shown in Fig. 12, the memory consumption of Exact-N is at least 80% higher than the naive solution as the latter can be considered as having no memory overhead. More importantly, this 80% gap only happens when the data is very sparse ($\overline{N_{(p_i)}} = 1\%$ of $Q.win$). It increases to more than 4000 percent when $\overline{N}_{(p_i)}$ reaches 50% of the window size, indicating that the data stream contains very dense sub-regions. Such results agree with our earlier analysis, considering the large number of links each data point has to store. This experiment confirms that Exact-N is not an efficient algorithm in terms of memory consumption. In addition, the CPU time it uses in all 49 cases is on average 25 present higher than that used by Extra-N. This is calculated by summing the difference percentage in all 49 cases and dividing it by 49. This fact eliminates it from the set of plausible candidates even in terms of CPU-efficiency.

Abstract-C, an incremental algorithm which does not maintain the exact neighborships, has good memory efficiency. However, since the time efficiency of Abstract-C is highly sensitive to the number of *core points* N_{core} in the window, the performance of Abstract-C is largely decided by the interrelationship between two variables, namely $\overline{N}_{(p_i)}$ and θ^{cnt}. More specifically, when $\overline{N}_{(p_i)}$ is far below θ^{cnt} and thus there exist very few or no *core points* in the window, Abstract-C is an efficient algorithms in terms of both CPU and memory. As shown in Fig. 11, it is even faster than Abstract-M and Extra-N for $\overline{N_{(p_i)}} = 1\%$ cases in terms of CPU time. However, as $\overline{N}_{(p_i)}$ increases and eventually surpasses θ^{cnt}, indicating that more and more data points become *core points*, the time efficiency of Abstract-C drops dramatically. In many cases shown in Fig. 11, it is not only much slower than our proposed algorithms, Abstract-M and Extra-N, but also slower even than the naive solution. This experiment illustrates that Abstract-C is very inconsistent in terms of CPU time and it performs well only if N_{core} is very low. Given the limited scope of applicability, Abstract-C is not an attractive solution in general.

Our proposed algorithms, namely Abstract-M and Extra-N, take advantage of incremental computations while successfully avoiding the huge overhead on both

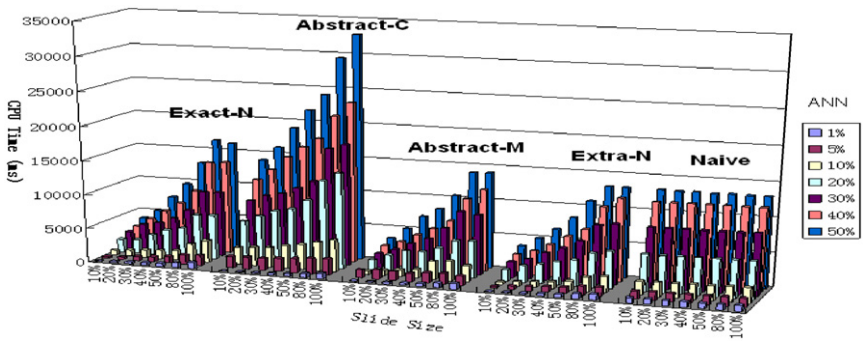

Fig. 11. Comparison of CPU performances of five algorithms.

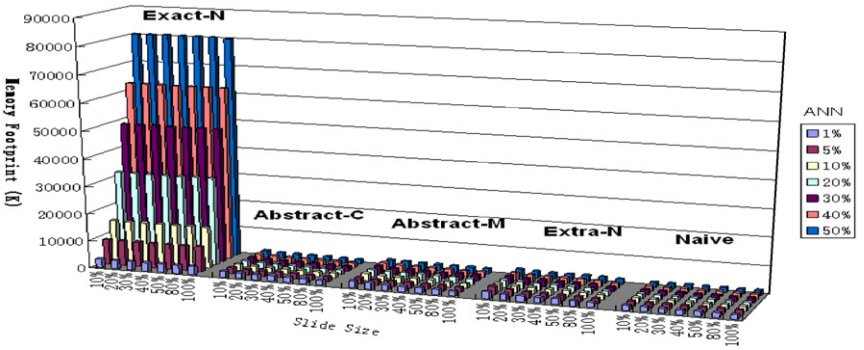

Fig. 12. Comparison of memory performances of five algorithms.

memory and CPU. Compared with the naive solution, both Abstract-M and Extra-N need more memory space as they need to maintain a certain amount of meta-information for future windows. However, such overhead is much smaller than that of Exact-N and in fact it is always being kept at very acceptable levels. In particular, for Abstract-M the memory consumption in all 49 cases is on average 33% higher than that of the naive solution. This is calculated in the same way as we compared the CPU time of Exact-N and Extra-N earlier. For Extra-N, this number becomes 36%, which is slightly higher but still quite modest. These facts confirm that our proposed algorithms, Abstract-M and Extra-N, have very good and consistent memory-efficiency.

The negligible CPU overhead of our proposed algorithms is also confirmed by this experiment. As shown in Fig. 11, Abstract-M and Extra-N saved CPU time substantially compared to the naive solution in all the cases where $Q.Slide \leq 50\% \times Q.win$. Even in the cases when $Q.Slide$ is very close (80%) or even equal to $Q.win$ (typically the limit of the incremental algorithms), Abstract-M and Extra-N exhibit comparable performances with those of the naive solution. Actually, both Abstract-M and Extra-N can be taken as variances of the naive solution when the windows are non-overlapping, because they only detect the patterns based on the "view" of the current window and no "predicted view" would be generated nor maintained. This indicates that our proposed algorithms have very small CPU overhead in all cases and thus are viable candidates for a system's only implementation, regardless of the input data and queries.

Abstract-M versus Extra-N: We first discuss the similar performances of Abstract-M and Extra-N shown in many of our above test cases, which on first sight does not appear as one would have expected in our cost analysis. The main reason for this is that the number of *promoted core points* $N_{prmtcore}$ stayed small in many cases and thus did not affect the performance of Abstract-M. Actually, we found that $N_{prmtcore}$ tends to be small, unless a large number of data points, which have a "boundary" (close to θ^{cnt}) number of neighbors, exists. However, such situations are not found to be frequent in our experiments for both the synthetic and the real data.

Although Abstract-M and Extra-N work equally well in many of our test cases, they do behave quite differently

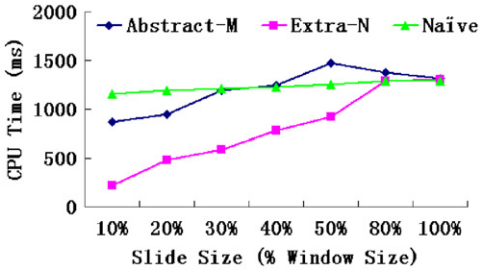

Fig. 13. Comparison on CPU time of Abstract-M and MPS in $\overline{N_{(p_i)}} = 5\%$ cases.

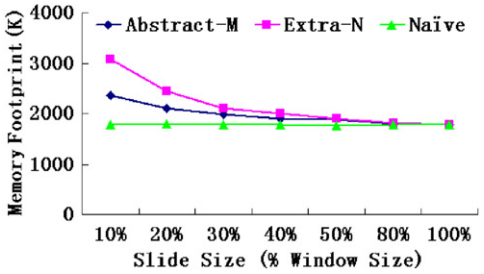

Fig. 14. Comparison on memory usage of Abstract-M and MPS in $\overline{N_{(p_i)}} = 5\%$ cases.

when $N_{prmtcore}$ turns to be an non-ignorable factor. To better understand their performance in these special cases, we zoom into the cases with $\overline{N_{(p_i)}} = 5\%$ in our comprehensive experiment (Figs. 11 and 12). Figs. 13 and 14 show the zoomed in subparts of the same experiment results discussed in the earlier part of this subsection.

In the cases shown in Figs. 13 and 14, Abstract-M tends to use more CPU time while Extra-N consumes more memory space. This is as expected because of the existence of a large number of *promoted core points* in each window. In particular, since in the $\overline{N_{(p_i)}} = 5\%$ cases, the number of neighbors each data point has is quite close to the population threshold, $\theta^{cnt} = 5\%$ of the window size, many *core points* may be demoted to become *edge points* or even *noise points* after losing some of their neighbors as the window slides. For the same reason, the *non-core points* have a good chance to be promoted to become

promoted core points after gaining some new neighbors at the window slide. Corresponding to our analysis in Sections 7 and 8, each *promoted core point* charges an extra range query search from Abstract-M, while it charges Extra-N for the memory space to store the links to its neighbors in its "*non core point* career" before its promotion. On the one hand, Extra-N guarantees the minimum number of range query searches and thus time efficiency in all cases, but it may consume more memory, especially when $\overline{N}_{(p_i)}$ and θ^{cnt} are both large and close to each other. In the other hand, Abstract-M never stores the exact *neighborships* and thus is more memory-efficient. However, it usually takes extra range query searches and thus consumes more CPU time. Such preferences of Abstract-M and Extra-N for CPU time versus memory space utilization can be observed in most of the test cases, although they are most apparent in the cases we zoomed into. Thus, in general, a system can choose to implement Abstract-M when the memory space is its key limit, while implementing Extra-N if CPU time is its major resource concern.

Scalability analysis: We now look at the scalability in terms of the base size, meaning that how many data points the algorithms can cluster at each window. So, in this experiment, we test count-based windows sized from 10 K to 50 K with a fixed slide size 5 K. Other settings of this experiment are equal to those from the previous comprehensive one, except that we fixed $\overline{N}_{(p_i)}$ at 1 K.

As shown in Figs. 15 and 16 both our algorithms, Abstract-M and Extra-N exhibit very good scalability in window sizes in terms of both CPU and memory, while others failed in either or both of them. In particular, both Abstract-M and Extra-N only need 5 s to cluster 50 K data points at each window given 5 K new data points. On the other words, both algorithms can comfortably handle a data rate of 1 K per second with a 50 K window. Also, the memory usage of both algorithms increases very modestly with the growth of the window size.

Second, we investigate the effect of dimensionality on the performance of our algorithms. As shown in Fig. 17, the CPU time of both our proposed algorithms, especially Extra-N, increases only modestly with the number of dimensions. This demonstrates that our algorithms have an even better than linear scalability in the dimensionality. This is because the number of dimensions will only affect the CPU time needed for the range query searches but has no impact on the *neighborship* maintenance costs. As we have already largely reduced the number of range query searches needed in these two algorithms, and even achieved the minimal for Extra-N (see our cost analysis in Section 10), they both are expected to have excellent scalability in the number of dimensions. Our experimental results confirmed this.

Evaluation with real data and queries: We first evaluate the performance of all five competitors with the GMTI data, which is a representative for moving object monitoring applications. We varied the slide size *Q.Slide* from 10% to 100% of the *Q.win*. Given these query parameters, we find there are 6–11 clusters in the window at different time horizons, and $\overline{N}_{(p_i)}$ in the windows varies from 9% to 11% of the number data points in the windows.

As depicted in Figs. 18 and 19, Extra-N has the best time efficiency compared with all other methods. The memory usage of Extra-N is on average 16% higher than the naive solution in the five cases. This memory overhead is a little bit higher than that of Abstract-M (11% higher than the naive solution) but still very acceptable.

For the STT data, by using the query parameters learned from our pre-analysis of the data, the number of clusters

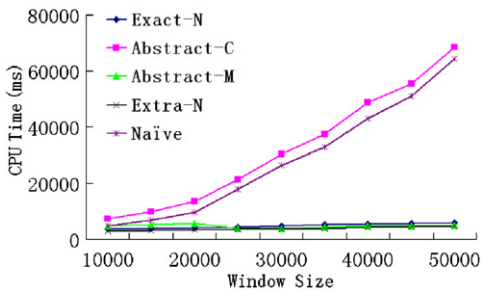

Fig. 15. Comparison of CPU scalability on base (Window) size.

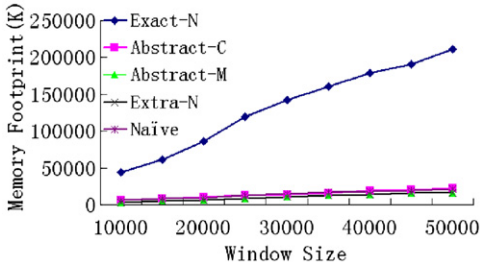

Fig. 16. Comparison of memory scalability on base (Window) size.

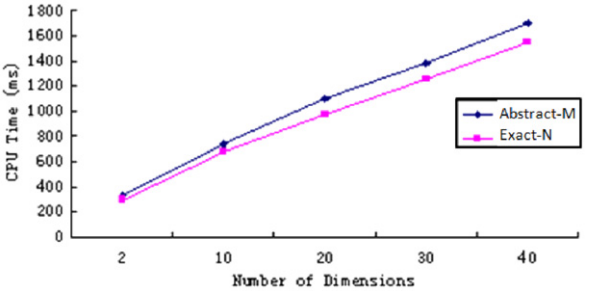

Fig. 17. Comparison of CPU scalability on dimensionality.

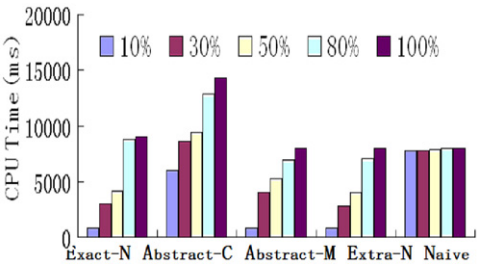

Fig. 18. Comparison on CPU time with GMTI data.

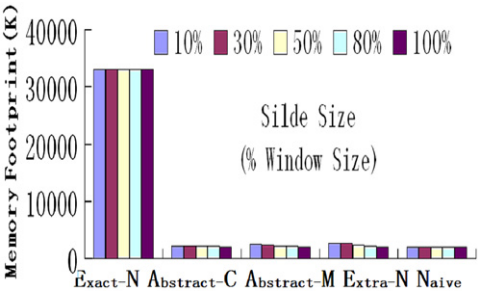

Fig. 19. Comparison on memory usage with GMTI data.

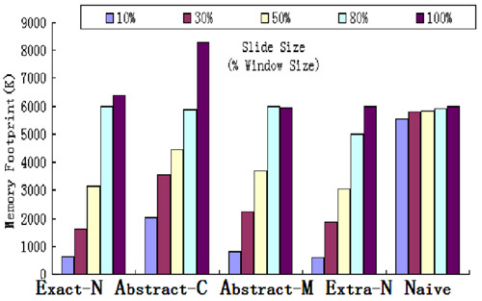

Fig. 20. Comparison on CPU time with STT data.

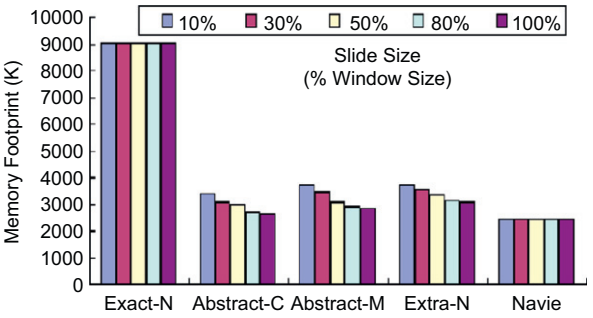

Fig. 21. Comparison on memory usage with STT data.

existing in the windows ranges from 17 to 26, and the number $\overline{N}_{(p_i)}$ in the windows varies from 6% to 9% to the number data points in the windows. Similar behaviors can be observed using the STT data (Figs. 20 and 21).

Generally, our experiments on real data confirm that our proposed algorithms Abstract-M and Extra-N outperform all other alternative methods and thus are the preferred solutions for density-based cluster detection in sliding windows.

11.4. Evaluation of distance-based outlier detection methods

We compare the performance of our outlier algorithm Abstract-C with two alternatives, namely the naive solution and the exact-STORM [5], which is the only previous work in the literature we are aware of that detects distance-based outliers in sliding windows. Exact-STORM is designed to incrementally detect distance-based outliers over count-based windows. Similar to Abstract-C, exact-STORM requires one range query search for every new data point in each window. However, it uses a different *neighborship* maintenance mechanism. In particular, the exact-STORM algorithm requires every data point p_i in the window to maintain two data structures. The first one, called $p_i.nn\_before$, is a list containing the identifiers of the most recent preceding neighbors of p_i. $p_i.nn\_before$ is similar with the "neighbor list" we use in Exact-N that gives p_i direct access to its neighbors. However, it has two special characteristics. First, $p_i.nn\_before$ only stores the preceding neighbors of p_i, whose arrival and expiration are earlier than those of p_i. Second, for "count- based" based windows, the length of $p_i.nn\_before$ has an upper bound $k = N \times \theta^{range}$, which equals the number of neighbors p_i needs to be a "safe inlier". This is because the number of data points in the count-based window is fixed. So, if a data point already has $k = N \times \theta^{range}$ neighbors, it cannot be an outlier in the current and future windows until any of them expire. The second data structure $p_i.count\_after$ is a counter of the number of succeeding neighbors of p_i. The succeeding neighbors denote the neighbors of p_i whose arrival and expiration are later than that of p_i. At each window slide, exact-STORM runs one range query search for every new data point, and updates *nn_before* and *count_after* for each of them and their neighbors. At the output stage, exact-STORM outputs the outliers based on the information in each data point's *nn_before* and *count_after*.

In count-based windows, since exact-STORM achieves both the minimum number of range query searches and also the linear memory consumption (it stores at most k neighbors for each data point), it is equivalent to our proposed algorithm Abstract-C, while using different *neighborship* maintenance mechanisms. However, being designed specifically for the count-based window scenario, exact-STORM would tend to perform badly in the time-based window scenario. This is because the "safe inlier" property, which it relies on to limit the length of *nn_before* in the count-based scenario, no longer holds for time-based windows. In particular, for time-based windows, since the number of data points in the window is not fixed, the number of neighbors a data point needs to be an "inlier" may change as well. So, no matter how many neighbors a data point already has, it can never be viewed as a "safe inlier" in future windows and has to keep the "identifiers" of all its "preceding" neighbors. So in the time-based window, exact-STORM would suffer from the same problem as Exact-N does, namely the huge number of exact *neighborships* (links) that must be stored.

In our experiments, we compare the performance of exact-STORM and Abstract-C in both count- and time-based window scenarios using the Gauss Dataset. In both scenarios, we strictly follow the implementation of exact-STORM presented in [5], except for breaking the upper bound on the length of *nn_before* as required in the time-based window scenario.

As shown in Figs. 22 and 23, for count-based windows, exact-STORM and Abstract-C perform equivalently well and clearly outperform the naive solution in terms of CPU time.

However, as shown in Figs. 24 and 25, Abstract-C clearly outperforms both the naive solution and exact-STORM in time-based window scenarios. This is because the naive solution does not take any advantage of incremental

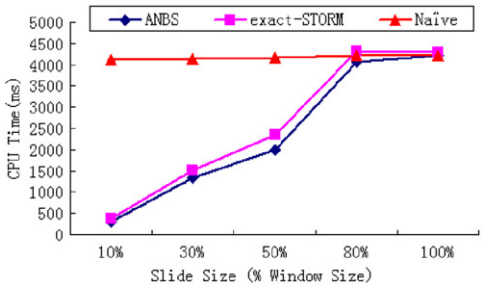

Fig. 22. Comparison on CPU time for count-based window scenario.

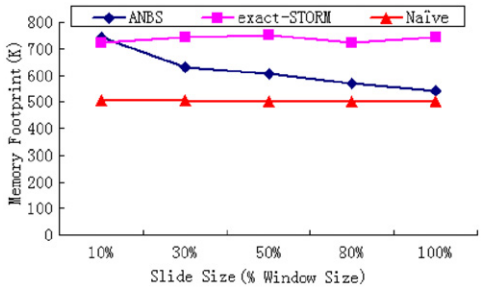

Fig. 23. Comparison on memory usage for count-based window scenario.

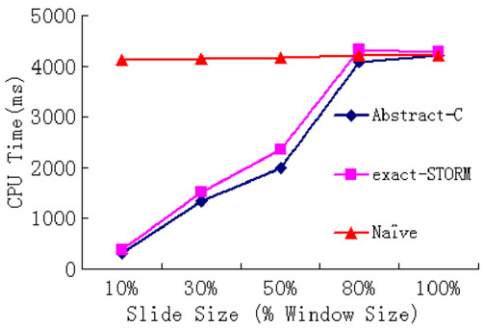

Fig. 24. Comparison on CPU time for time-based window scenario.

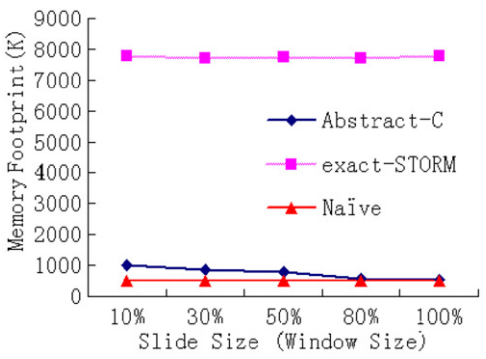

Fig. 25. Comparison on memory usage for time-based window scenario.

computation and Exact-STORM suffers from the huge memory overhead caused by storing the neighbors in time-based windows. In contrast, Abstract-C does not have either of these two problems and thus shows a much better performance.

12. Related work

Traditionally, the pattern detection techniques are designed for static environments, where large amounts of data are being collected. Well-known clustering algorithms for static data include [32,22,21,18,6]. Previous works on detecting outliers include [27,29,11,24,25]. In this literature, both clusters and outliers can either be global patterns [32,22,21] that are defined by global characteristics of all data or local patterns [27,29] that are defined by the characteristics of a subset of the data. In this work, our target pattern types are density-based clusters [18,17] and distance-based outliers [24,25]. Both are popular pattern types defined by local neighborhood properties.

More recently, pattern detection on streaming data began to be studied. The earlier clustering algorithms applied to data streams [20,19] are global clustering algorithms adapted from the static k-means algorithm. They treat the data stream clustering problem as a continuous version of static data clustering. They treat objects with different time horizons (recentness) equally and thus do not reflect the temporal features of data streams. Later, [3] presented a framework for clustering streaming data using a two stage process. At the first stage, the online component works on the streaming data to summarize it into micro-clusters. At the second stage, an offline component clusters the micro-clusters formed earlier to form final clustering results using a static k-means algorithm. In this framework, a subtraction function is used to discount the effect of the earlier data on the clustering results. Several extensions have been made to this work, focusing respectively on clustering distributed data streams [9], multiple data streams [14], and parallel data streams [10]. None of these works deals with the arbitrarily shaped local clusters, nor do they support sliding window semantics. Ref. [8] is the only work we are aware of that discusses the clustering problem with sliding windows. However, it again is a global clustering algorithm maintaining approximated cluster centers only.

Ref. [17] presented techniques to incrementally update density-based clusters in data warehouse environments. Since all optimizations in this work are designed for a single update (a single deletion or insertion) to the data warehouse, this fits well for the relatively stable (data warehouse) environments but is not scalable for sustainable environments. Refs. [13,12] also studied the problem of detecting density-based clusters over streaming data. However, Refs. [13,12] do not identify individual members in the clusters as required by the application scenarios described earlier in the introduction. Also, to capture the dynamicity of the evolving data, they both use decaying factors derived from the "age" information of the objects. These decaying factors put lighter weights on older objects during the clustering processes. This approach emphasizes the recent stream portion more compared to the older data, still it does not enforce the discounting the impact of expired data from the pattern detection results. So, they are not suitable for applications requiring sliding window scenarios discussed in this work.

The problem of detecting outliers in streams has been studied in [31,2,30,5,28,15,4]. Among these works, Refs. [31,2,28,15,4] work with outliers with different definitions

from ours. Thus, these techniques cannot be applied to detect distance-based outlier as targeted by our method. Ref. [30] study the detection of distance- and MEDF-based outliers in hierarchically structured sensor networks. Also, the outlier detection is based on approximated data distributions. So, their work has a different goal from us. Most similar to our work, Ref. [5] introduced an algorithm to detect the distance-based outliers within sliding windows. However, this work only deals with count-based windows, where the number of objects in the window is a priori known and fixed. Both our analytical and experimental studies reveal that this method is not suitable for answering queries with time-based windows, where each window may have different numbers of objects.

13. Conclusions

In this work, we study the problem of detecting neighbor-based patterns for sliding windows over streaming data. The major difficulty for incremental detection of the neighbor-based patterns exists in handling the impact of expired objects. To solve this, we propose the "view prediction" technique, which elegantly handles the *negative changes* on patterns caused by objects' expiration with very limited CPU and memory overheads. Based on this technique, we design our proposed algorithm, Abstract-C, for distance-based outlier detection in sliding windows. Second, we propose a hybrid *neighborship* maintenance mechanism, which allows us to preserve progressive cluster structures with only linear memory consumption. The combination of these two techniques leads to our proposed algorithm, Extra-N, for density-based cluster detection over sliding windows. To compare our proposed algorithms with other alternatives, we build cost models to measure the algorithms' performance in terms of CPU and memory efficiency. Based on the cost models, we identify the key performance factors for each alternative algorithm and analyze the situations in which each of them would have good versus poor performance. Our experimental studies using both synthetic and real streaming data confirm the clear superiority of our proposed algorithm

The future work directions of this work include: (1) studying other output formats for neighbor-based patterns in streaming window semantics as we discussed in Section 9, (2) designing efficient pattern mining algorithm for other pattern types in streaming data, such as graphs and association rules, (3) building advanced user interfaces which allow users to interactively explore the streaming patterns.

Our experimental studies using both synthetic and real streaming data confirm the clear superiority of our proposed algorithms to all other alternatives.

References

[1] Details omitted due to double-blind reviewing.
[2] C.C. Aggarwal, On abnormality detection in spuriously populated data streams, in: SDM, 2005.
[3] C.C. Aggarwal, J. Han, J. Wang, P.S. Yu, A framework for clustering evolving data streams, in: VLDB, 2003, pp. 81–92.
[4] C.C. Aggarwal, Y. Zhao, P.S. Yu, Outlier detection in graph streams, in: ICDE, 2011, pp. 399–409.
[5] F. Angiulli, F. Fassetti, Detecting distance-based outliers in streams of data, in: CIKM, 2007, pp. 811–820.
[6] M. Ankerst, M.M. Breunig, H.-P. Kriegel, J. Sander, Optics: ordering points to identify the clustering structure, in: SIGMOD, 1999, pp. 49–60.
[7] A. Arasu, S. Babu, J. Widom, The CQL continuous query language: semantic foundations and query execution, The International Journal on Very Large Data Bases 15 (2) (2006) 121–142.
[8] B. Babcock, M. Datar, R. Motwani, L. O'Callaghan, Maintaining variance and k-medians over data stream windows, in: PODS, 2003, pp. 234–243.
[9] S. Bandyopadhyay, C. Giannella, U. Maulik, H. Kargupta, K. Liu, S. Datta, Clustering distributed data streams in peer-to-peer environments, Information Science 176 (14) (2006) 1952–1985.
[10] J. Beringer, E. Hüllermeier, Online clustering of parallel data streams, Data & Knowledge Engineering 58 (2) (2006) 180–204.
[11] M.M. Breunig, H.-P. Kriegel, R.T. Ng, J. Sander, Lof: identifying density-based local outliers, SIGMOD Record 29 (2) (2000) 93–104.
[12] F. Cao, M. Ester, W. Qian, A. Zhou, Density-based clustering over an evolving data stream with noise, in: SDM, 2006.
[13] Y. Chen, L. Tu, Density-based clustering for real-time stream data, in: KDD, 2007, pp. 133–142.
[14] B.-R. Dai, J.-W. Huang, M.-Y. Yeh, M.-S. Chen, Adaptive clustering for multiple evolving streams, IEEE Transactions on Knowledge and Data Engineering 18 (9) (2006) 1166–1180.
[15] M. Elahi, K. Li, W. Nisar, X. Lv, H. Wang, Detection of local outlier over dynamic data streams using efficient partitioning method, in: CSIE, vol. 4, 2009, pp. 76–81.
[16] J.N. Entzminger, C.A. Fowler, W.J. Kenneally, Jointstars and GMTI: past, present and future, IEEE Transactions on Aerospace and Electronic Systems 35 (2) (1999) 748–762.
[17] M. Ester, H. Kriegel, J. Sander, M. Wimmer, X. Xu, Incremental clustering for mining in a data warehousing environment, in: VLDB, 1998, pp. 323–333.
[18] M. Ester, H.-P. Kriegel, J. Sander, X. Xu, A density-based algorithm for discovering clusters in large spatial databases with noise, in: KDD, 1996, pp. 226–231.
[19] S. Guha, A. Meyerson, N. Mishra, R. Motwani, L. O'Callaghan, Clustering data streams: theory and practice, IEEE Transactions on Knowledge and Data Engineering 15 (3) (2003) 515–528.
[20] S. Guha, N. Mishra, R. Motwani, L. O'Callaghan, Clustering data streams, in: FOCS, 2000, pp. 359–366.
[21] S. Guha, R. Rastogi, K. Shim, Cure: an efficient clustering algorithm for large databases, SIGMOD Record 27 (2) (1998) 73–84.
[22] J.A. Hartigan, M.A. Wong, A k-means clustering algorithm, Applied Statistics 28 (1) (1979).
[23] I. INETATS, Stock Trade Traces 〈http://www.inetats.com/〉 (July 2007).
[24] E.M. Knorr, R.T. Ng, Algorithms for mining distance-based outliers in large datasets, in: VLDB, 1998, pp. 392–403.
[25] E.M. Knorr, R.T. Ng, Finding intensional knowledge of distance-based outliers, in: M.P. Atkinson, M.E. Orlowska, P. Valduriez, S.B. Zdonik, M.L. Brodie (Eds.), Proceedings of 25th International Conference on Very Large Data Bases, VLDB'99, September 7–10, 1999, Edinburgh, Scotland, UK, Morgan Kaufmann, 1999, pp. 211–222.
[26] J. Munkres, Topology, Prentice Hall, 2000.
[27] K. Ord, Outliers in statistical data, International Journal of Forecasting 12 (1) (1996) 175–176.
[28] D. Pokrajac, A. Lazarevic, L.J. Latecki, Incremental local outlier detection for data streams, in: CIDM, 2007, pp. 504–515.
[29] I. Ruts, P.J. Rousseeuw, Computing depth contours of bivariate point clouds, Computational Statistics & Data Analysis 23 (1) (1996) 153–168.
[30] S. Subramaniam, T. Palpanas, D. Papadopoulos, V. Kalogeraki, D. Gunopulos, Online outlier detection in sensor data using non-parametric models, in: VLDB, 2006, pp. 187–198.
[31] K. Yamanishi, J. ichi Takeuchi, G.J. Williams, P. Milne, On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms, in: KDD, 2000, pp. 320–324.
[32] T. Zhang, R. Ramakrishnan, M. Livny, Birch: an efficient data clustering method for very large databases, SIGMOD Record 25 (2) (1996) 103–114.