Data Stream Clustering: A Survey

Article *in* ACM Computing Surveys · March 2014

DOI: 10.1145/2522968.2522981

6 authors, including:

Jonathan de Andrade Silva
Universidade Federal de Mato Grosso do Sul
20 PUBLICATIONS **311** CITATIONS

SEE PROFILE

Elaine Faria
Universidade Federal de Uberlândia (UFU)
10 PUBLICATIONS **265** CITATIONS

SEE PROFILE

Rodrigo C. Barros
Pontifícia Universidade Católica do Rio Grande do Sul
85 PUBLICATIONS **904** CITATIONS

SEE PROFILE

Eduardo R Hruschka
University of São Paulo
110 PUBLICATIONS **2,577** CITATIONS

SEE PROFILE

Data Stream Clustering: A Survey

JONATHAN A. SILVA, ELAINE R. FARIA, RODRIGO C. BARROS,
EDUARDO R. HRUSCHKA, and ANDRE C. P. L. F. DE CARVALHO
University of São Paulo
and
JOÃO GAMA
University of Porto

Data stream mining is an active research area that has recently emerged to discover knowledge from large amounts of continuously generated data. In this context, several data stream clustering algorithms have been proposed to perform unsupervised learning. Nevertheless, data stream clustering imposes several challenges to be addressed, such as dealing with non-stationary, unbounded data that arrive in an online fashion. The intrinsic nature of stream data requires the development of algorithms capable of performing fast and incremental processing of data objects, suitably addressing time and memory limitations. In this paper, we present a survey of data stream clustering algorithms, providing a thorough discussion of the main design components of state-of-the-art algorithms. In addition, this work addresses the temporal aspects involved in data stream clustering, and presents an overview of the usually-employed experimental methodologies. A number of references is provided that describe applications of data stream clustering in different domains, such as network intrusion detection, sensor networks, and stock market analysis. Information regarding software packages and data repositories are also available for helping researchers and practitioners. Finally, some important issues and open questions that can be subject of future research are discussed.

Categories and Subject Descriptors: I.5.3 [**Pattern Recognition**]: Clustering—*Algorithms*

General Terms: Algorithms

Additional Key Words and Phrases: Data stream clustering, online clustering.

1. INTRODUCTION

Recent advances in both hardware and software have allowed large-scale data acquisition. Nevertheless, dealing with massive amounts of data poses a challenge for researchers and practitioners, due to the physical limitations of the current computational resources. For the last decade, we have seen an increasing interest in managing these massive, unbounded sequences of data objects that are continuously generated at rapid rates, the so-called *data streams* [Aggarwal 2007; Gama and Gaber 2007; Gama 2010]. More formally, a data stream $\mathcal{S}$ is a massive sequence

The authors would like to thank Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq), Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), ERDF through the COMPETE Programme, project FCOMP -01-0124-FEDER-022701, and Foundation for Science and Technology (FCT) project KDUDS - Knowledge Discovery from Ubiquitous Data Streams (ref. PTDC/EIA/098355/2008) for funding this research.

of data objects $\mathbf{x}^1, \mathbf{x}^2, ..., \mathbf{x}^N$, *i.e.*, $\mathcal{S} = \{\mathbf{x}^i\}_{i=1}^N$, which is potentially unbounded ($N \to \infty$). Each data object[1] is described by an $n-$dimensional attribute vector $\mathbf{x}^i = [x_j^i]_{j=1}^n$ belonging to an attribute space Ω that can be continuous, categorical, or mixed.

Applications of data streams include mining data generated by sensor networks, meteorological analysis, stock market analysis, and computer network traffic monitoring, just to name a few. These applications involve data sets that are far too large to fit in main memory and are typically stored in a secondary storage device. From this standpoint, and provided that random access is prohibitively expensive [Guha et al. 2003], performing linear scans of the data is the only acceptable access method in terms of computational efficiency.

Extracting potentially useful knowledge from data streams is a challenge *per se*. Most data mining and knowledge discovery techniques assume that there is a finite amount of data generated by an unknown, stationary probability distribution, which can be physically stored and analyzed in multiple steps by a *batch*-mode algorithm. For data stream mining, however, the successful development of algorithms has to take into account the following restrictions:

(1) Data objects arrive continuously;
(2) There is no control over the order in which the data objects should be processed;
(3) The size of a stream is (potentially) unbounded;
(4) Data objects are discarded after they have been processed. In practice, one can store part of the data for a given period of time, using a forgetting mechanism to discard them later;
(5) The unknown data generation process is possibly non-stationary, *i.e.*, its probability distribution may change over time.

The development of effective algorithms for data streams is an effervescent research issue. This paper is particularly focused on algorithms for clustering data streams. Essentially, the clustering problem can be posed as determining a finite set of categories (clusters) that appropriately describe a data set. The rationale behind clustering algorithms is that objects within a cluster are more similar to each other than they are to objects belonging to a different cluster [Fayyad et al. 1996; Arabie and Hubert 1999]. It is worth mentioning that batch-mode clustering algorithms have been both studied and employed as data analysis tools for decades. The literature on the subject is very large — *e.g.* see [Kogan 2007; Gan et al. 2007; Xu and Wunsch 2009] — and out of the scope of this paper.

Clustering data streams requires a process able to continuously cluster objects within memory and time restrictions [Gama 2010]. Bearing these restrictions in mind, algorithms for clustering data streams should ideally fulfill the following requirements [Babcock et al. 2002; Barbará 2002; Tasoulis et al. 2006; Bifet 2010]: (i) provide timely results by performing fast and incremental processing of data objects; (ii) rapidly adapt to changing dynamics of the data, which means algorithms should detect when new clusters may appear, or others disappear; (iii) scale to the number of objects that are continuously arriving; (iv) provide a model rep-

[1]In this paper we adhere to this most commonly used term, but note that the terms element, example, instance, and sample are also used.

resentation that is not only compact, but that also does not grow with the number of objects processed (notice that even a linear growth should not be tolerated); (v) rapidly detect the presence of outliers and act accordingly; and (vi) deal with different data types, *e.g.*, XML trees, DNA sequences, GPS temporal and spatial information. Although these requirements are only partially fulfilled in practice, it is instructive to keep them in mind when designing algorithms for clustering data streams.

The purpose of this paper is to survey state-of-the-art algorithms for clustering data streams. Surveys on this subject have been previously published, such as [Mahdiraji 2009; Kavitha and Punithavalli 2010; Khalilian and Mustapha 2010]. In [Mahdiraji 2009], the authors present a very brief description of only five algorithms. In [Kavitha and Punithavalli 2010], a short description of clustering algorithms for time series data streams is presented. Lastly, in [Khalilian and Mustapha 2010], the authors discussed some challenges and issues in data stream clustering. Differently from previous papers, we offer an extensive review of the literature, as well as comprehensive discussions of the different design components of data stream clustering algorithms. As an additional contribution of our work, we focus on relevant subjects that have not been carefully considered in the literature, namely: i) providing a taxonomy that allows the reader to identify every surveyed work with respect to important aspects in data stream clustering; ii) analyzing the influence of the time element in data stream clustering; iii) analyzing the experimental methodologies usually employed in the literature; and iv) providing a number of references that describe applications of data stream clustering in different domains, such as sensor networks, stock market analysis, and grid computing.

The remainder of this paper is organized as follows. In Section 2, we describe a taxonomy to properly classify the main data stream clustering algorithms. Section 3 presents the components responsible for online management of data streams, namely: data structures (Section 3.1), window models (Section 3.2), and outlier detection mechanisms (Section 3.3). The offline component of data stream clustering algorithms is presented in Section 4. Relevant issues regarding the temporal aspects of data stream clustering are addressed in Section 5. Afterwards, in Section 6, we present an overview of the most usual experimental methodologies employed in the literature. In Section 7, we review practical issues in data stream clustering, such as distinct real-world applications, publicly-available software packages, and data set repositories. Finally, we indicate challenges to be faced and promising future directions for the area in Section 8 and provide the complexity analysis of the main data stream clustering algorithms in the Electronic Appendix.

2. OVERVIEW OF DATA STREAM CLUSTERING

In this section, we provide a taxonomy that allows the reader to identify every surveyed work with respect to important aspects in data stream clustering, namely:

(1) data structure for statistic summary;
(2) window model;
(3) outlier detection mechanism;
(4) number of user-defined parameters;
(5) offline clustering algorithm;

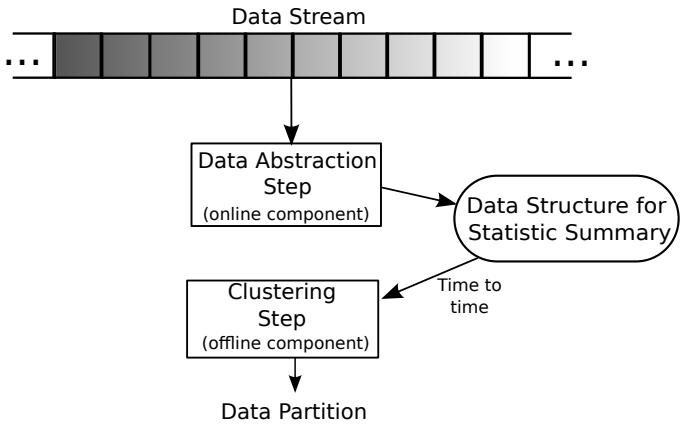

Fig. 1. Object-based data stream clustering framework.

(6) cluster shape;
(7) type of clustering problem.

The application of clustering algorithms to data streams have been concerned with either object-based clustering or attribute-based clustering, with the former being far more common.

Object-based data stream clustering algorithms can be summarized into two main steps [Cao et al. 2006; Aggarwal et al. 2004a; Yang and Zhou 2006; Zhou et al. 2008], namely: data abstraction step (also known as online component) and clustering step (also known as offline component), as illustrated in Figure 1. The online abstraction step summarizes the data stream with the help of particular data structures for dealing with space and memory constraints of stream applications. These data structures summarize the stream in order to preserve the meaning of the original objects without the need of actually storing them. Among the commonly-employed data structures, we highlight the feature vectors, prototype arrays, coreset trees, and data grids (details in Section 3.1).

For summarizing the continuously-arriving stream data and, at the same time, for giving greater importance to up-to-date objects, a popular approach in object-based data stream clustering consists of defining a time window that covers the most recent data. Among the distinct window models that have been used in the literature, we highlight the landmark model, sliding-window model, and damped model — all covered in Section 3.2.

Still regarding the data abstraction step, data stream clustering algorithms should ideally employ outlier detection mechanisms that are able to distinguish between actual outliers and cluster evolution, considering that the data stream distribution may vary over time. Outlier detection mechanisms are covered at Section 3.3.

After performing the data abstraction step, data stream clustering algorithms obtain a data partition via an offline clustering step (offline component). The offline component is used together with a wide variety of inputs (*e.g.,* time horizon, and number of clusters) to provide a quick understanding of the broad clusters in the data stream. Since this component requires the summary statistics as input, it turns out to be very efficient in practice [Aggarwal et al. 2003]. Based on this

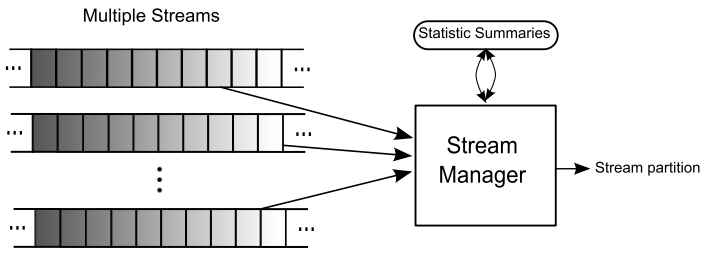

Fig. 2. Attribute-based data stream clustering framework.

assumption, traditional clustering algorithms (like *DBSCAN* [Ester et al. 1996] and *k-means*[2] [MacQueen 1967; Lloyd 1982]) can be used to find a data partition over the summaries, whose size is relatively small compared to the entire data stream. Note that the cluster shape will be directly related to the clustering algorithm being employed. For instance, *k-means* generates hyper-spherical clusters, whereas *DBSCAN* allows the discovery of arbitrarily-shaped clusters. We present the offline clustering step in Section 4.

Even though most data stream clustering algorithms aim at performing object clustering, there are works that perform attribute clustering (also known as variable clustering). Attribute clustering is usually considered a batch offline procedure, in which the common strategy is to employ a traditional clustering algorithm over the transposed data matrix. However, for online processing of data streams, it is not possible to transpose the (possibly infinite) data matrix. Clearly, there is the need of developing attribute clustering algorithms for data streams, whose objective is to find groups of attributes (*e.g.,* data sources like sensors) that behave similarly through time, under the constraints assumed in a data stream scenario. Examples of algorithms that perform attribute clustering are *Online Divisive-Agglomerative Clustering* (*ODAC*) [Rodrigues et al. 2006; 2008] and *DGClust* [Gama et al. 2011]. Figure 2 depicts the general scheme of data stream attribute clustering, in which we have one data stream per attribute and a manager that processes data from the distinct streams. Considering that each attribute constitutes a different stream, attribute clustering may benefit from parallel and distributed systems, which is precisely the case of *DGClust* [Gama et al. 2011].

Table I classifies the 13 most relevant data stream clustering algorithms to date according to the dimensions defined by our taxonomy. They are:

(1) *BIRCH* [Zhang et al. 1997];
(2) *CluStream* [Aggarwal et al. 2003];
(3) *ClusTree* [Kranen et al. 2011];
(4) *D-Stream* [Chen and Tu 2007];
(5) *DenStream* [Cao et al. 2006];
(6) *DGClust* [Gama et al. 2011];
(7) *ODAC* [Rodrigues et al. 2006; 2008];

[2]As observed by Jain [2009], *k*-means has a rich and diverse history as it was independently discovered in different scientific fields by Steinhaus [1956], Lloyd [1982] (who proposed it in 1957 and published it in 1982), Ball and Hall [1965], and MacQueen [1967].

Table I. Analysis of 13 data stream clustering algorithms.

Algorithm	Data Structure	Window Models	Outlier Detection	Number of Parameters
(1) *BIRCH*	feature vector	landmark	density-based	5
(2) *CluStream*	feature vector	landmark	statistical-based	9
(3) *ClusTree*	feature vector	damped	—	3
(4) *D-Stream*	grid	damped	density-based	5
(5) *DenStream*	feature vector	damped	density-based	4
(6) *DGClus*	grid	landmark	—	5
(7) *ODAC*	correlation matrix	landmark	—	3
(8) *Scalable k-means*	feature vector	landmark	—	5
(9) *Single-pass k-means*	feature vector	landmark	—	2
(10) *Stream*	prototype array	landmark	—	3
(11) *Stream LSearch*	prototype array	landmark	—	2
(12) *StreamKM++*	coreset tree	landmark	—	3
(13) *SWClustering*	feature vector	landmark	—	5

Algorithm	Cluster Algorithm	Cluster Shape	Cluster Problem
(1) *BIRCH*	*k-means*	hyper-sphere	object
(2) *CluStream*	*k-means*	hyper-sphere	object
(3) *ClusTree*	*k-means/DBSCAN*	arbitrary	object
(4) *D-Stream*	*DBSCAN*	arbitrary	object
(5) *DenStream*	*DBSCAN*	arbitrary	object
(6) *DGClust*	*k-means*	hyper-sphere	attribute
(7) *ODAC*	hierarchical clustering	hyper-ellipsis	attribute
(8) *Scalable k-means*	*k-means*	hyper-sphere	object
(9) *Single-pass k-means*	*k-means*	hyper-sphere	object
(10) *Stream*	*k-median*	hyper-sphere	object
(11) *Stream LSearch*	*k-median*	hyper-sphere	object
(12) *StreamKM++*	*k-means*	hyper-sphere	object
(13) *SWClustering*	*k-means*	hyper-sphere	object

(8) *Scalable k-means* [Bradley et al. 1998];
(9) *Single-pass k-means* [Farnstrom et al. 2000];
(10) *Stream* [Guha et al. 2000];
(11) *Stream LSearch* [O'Callaghan et al. 2002];
(12) *StreamKM++* [Ackermann et al. 2012];
(13) *SWClustering* [Zhou et al. 2008].

Note that parameters related to distance measures were not included in Table I. A more detailed discussion about the parameters of each clustering algorithm is presented in the next sections.

We notice that most data stream clustering algorithms neglect an important aspect of data stream mining: *change detection*. It is well-known that the data generation of several stream applications is guided by *non-stationary* distributions. This phenomenon, also known as *concept drift*, means that the concept about which data is obtained may shift from time to time, each time after some minimum permanence. The current strategy of most data stream clustering algorithms is to implicitly deal with *non-stationary* distributions through *window models*. An exception is ODAC [Rodrigues et al. 2006; 2008], which explicitly provides *change detection* mechanisms. A discussion on temporal aspects is presented in Section 5.

3. DATA ABSTRACTION STEP

As we have previously seen, most data stream clustering algorithms summarize the data in an abstraction step. In this section, we detail important aspects involved in

data abstraction: (i) data structures; (ii) window models; and (iii) outlier detection mechanisms.

3.1 Data Structures

Developing suitable data structures for storing statistic summaries of data streams is a crucial step for any data stream clustering algorithm, specially due to space-constraints assumptions made in data stream applications. Considering that the entire stream cannot be stored in the main memory, special data structures must be employed for incrementally summarizing the stream. In this section, we present four data structures commonly employed in data abstraction step: (i) feature vector; (ii) prototype array; (iii) coreset trees; and (iv) grids.

3.1.1 *Feature Vector.* The use of a feature vector for summarizing large amounts of data was first introduced in the *BIRCH* algorithm [Zhang et al. 1996]. This vector, named **CF**, from Clustering Feature vector, has three components: N, the number of data objects, LS, the linear sum of the data objects, and SS, the sum of squared the data objects. The structures LS and SS are n-dimensional arrays. These three components allow to compute cluster measures, such as cluster mean (Equation (1)), radius (Equation (2)), and diameter (Equation (3)).

$$centroid = \frac{LS}{N} \tag{1}$$

$$radius = \sqrt{\left(\frac{SS}{N} - \left(\frac{LS}{N}\right)^2\right)} \tag{2}$$

$$diameter = \sqrt{\left(\frac{2N * SS - 2 * LS^2}{N(N-1)}\right)} \tag{3}$$

The **CF** vector presents important incrementality and additivity properties, as described next:

(1) Incrementality: A new object $\mathbf{x}^j$ can be easily inserted into **CF** vector by updating its statistic summaries as follows:

$$LS \leftarrow LS + \mathbf{x}^j$$
$$SS \leftarrow SS + (\mathbf{x}^j)^2$$
$$N \leftarrow N + 1$$

(2) Additivity: Two disjoint vectors $\mathbf{CF}_1$ and $\mathbf{CF}_2$ can be easily merged into $\mathbf{CF}_3$ by summing up their components:

$$N_3 = N_1 + N_2$$
$$LS_3 = LS_1 + LS_2$$
$$SS_3 = SS_1 + SS_2$$

The other data structure employed in *BIRCH* is a height-balanced tree (B+-Tree), named **CF** tree, where each non-leaf node contains at most B entries, having each a **CF** vector and a pointer to a child-node. Similarly, every leaf node contains at most L entries, where each entry is a **CF** vector. Figure 3 depicts the **CF** tree

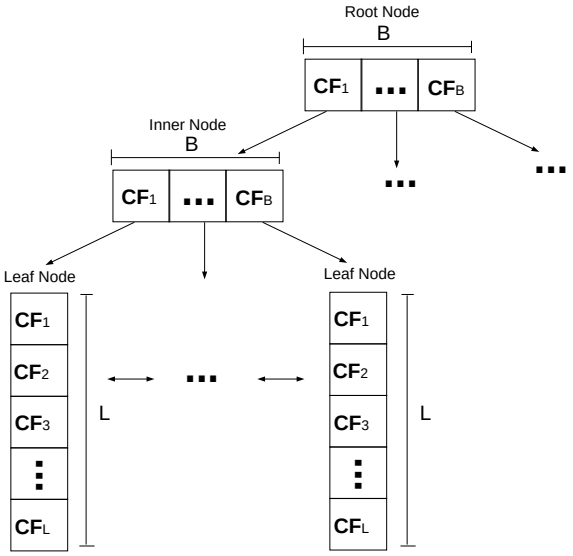

Fig. 3. **CF** tree structure.

structure, where every non-leaf node represents a cluster consisting of sub-clusters (its entries).

In the initial phase of *BIRCH*, the data set is incrementally scanned to build a **CF** tree in-memory. Each leaf node has a maximum diameter (or radius) represented by a user-defined parameter, T. The value of this parameter defines whether a new data object may be absorbed by a **CF** vector. Thus, T determines the size of the tree, where higher values of T lead to smaller trees.

When a new object arrives, it descends the **CF** tree from the root to the leaves by choosing in each non-leaf node its closest **CF** entry (closeness is defined by the Euclidean distance between new objects and the centroids of **CF** entries in non-leaf nodes). In a leaf node, the closest entry is selected and tested to verify whether it can absorb the new object. If so, the **CF** vector is updated, otherwise a new **CF** entry is created — at this point it only contains this particular object. If there is no space for a new **CF** entry in the leaf node (*i.e.*, there are already L entries within that leaf), the leaf is split into two leaves and the farthest pair of **CF** entries is used as seed to the new leaves. Afterwards, it is necessary to update each non-leaf node entry in the path until the root. Updating a tree path is a necessary step for every new insertion made in the **CF** tree. When the value of T is so low that the tree does not fit in memory, *BIRCH* makes use of different heuristics to increase the value of T so that a new **CF** tree that fits in memory can be built.

The **CF** vector from *BIRCH* has been employed by different algorithms. *Scalable k-means* [Bradley et al. 1998], for instance, employs a **CF** vector structure to enable the application of the k-means algorithm in very large data sets. The basic idea is that the objects in the data set are not equally important to the clustering process. From this observation, *Scalable k-means* employs different mechanisms to identify

objects that need to be retained in memory. This algorithm stores data objects in a block (buffer) in the main memory. Through the **CF** vectors, it discards objects that were previously statistically summarized into buffer. The block size is a user-defined parameter. When the block is full, an extended version of k-means is executed over the stored data. This extended version of k-means, named *Extended k-means*, can handle with both single data objects and sufficient statistics of summarized data. Based on the first generated partition, two compression phases are applied to the data objects that are continuously arriving and stored in the buffer.

In the primary compression phase, data objects that are unlikely to change their membership to a cluster in future iterations are discarded. To detect these objects, two strategies are employed, namely: PDC1 and PDC2. PDC1 finds the p% objects that are within the Mahalanobis radius [Bradley et al. 1998] of a cluster and compresses them — p is an input parameter. Only sufficient statistics are stored for these objects and, after computed, they are discarded. Next, PDC2 is applied to the objects that were not compressed by PDC1 (those outside the radius of its closest cluster). Therefore, for every remaining object $\mathbf{x}^j$, PDC2 finds its closest centroid according to the Mahalanobis distance [Maesschalck et al. 2000], say the centroid of cluster $\mathbf{C}_i$; afterwards, the centroid of $\mathbf{C}_i$ is perturbed and moved the farthest away from $\mathbf{x}^j$ (within a pre-calculated confidence interval). In addition, the centroid of the second-closest cluster to $\mathbf{x}^j$ is moved to be as close as possible (within a confidence interval) to it. If $\mathbf{x}^j$ still lies within the radius of cluster $\mathbf{C}_i$, the sufficient statistics of $\mathbf{x}^j$ are stored in the corresponding **CF** vector, and the object is discarded. Otherwise, $\mathbf{x}^j$ is kept to be processed by the secondary compression phase. Note that PDC2 is actually creating a "worst case scenario" by perturbing the cluster means within computed confidence intervals.

The secondary compression phase is applied to those objects that were not discarded in the primary compression. The objective of this phase is to release space in memory to store new objects. The objects that were not discarded are clustered by k-means into a user-defined number of clusters, k_2. Each one of the k_2 clusters is evaluated according to a compactness criterion that verifies whether the variance of a cluster is below a threshold β (input parameter). The statistical summary (**CF** vector) of the clusters that meet this criterion are stored in the buffer, together with the **CF** vectors obtained from the primary compression. The **CF** vectors of those clusters that do not attend to the compactness criterion may be permanently discarded [Bradley et al. 1998].

Farnstrom et al. [2000] present a simplification of *Scalable k-means*, named *Single-pass k-means*. In an attempt to improve computational efficiency, their work does not employ the compression steps of *Scalable k-means*. Instead, in each iteration of the algorithm, all objects in the buffer are discarded after the summary statistics are computed, and only the summary statistics of the k-means clusters are kept in the buffer.

The **CF** vector concept was extended and named *micro-cluster* in the *CluStream* algorithm [Aggarwal et al. 2003]. Each micro-cluster has five components. Three of them (N, LS, and SS) are the regular components of a **CF** vector. The additional components are the sum of the timestamps (LST) and the sum of the squares of the timestamps (SST). The online phase stores q micro-clusters in memory, where q is

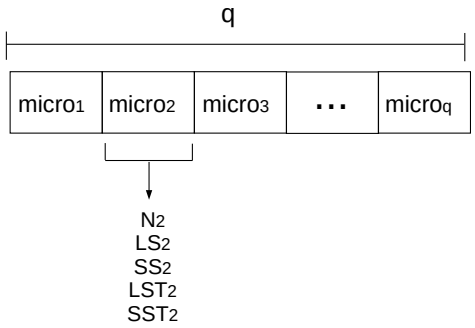

Fig. 4. Micro-cluster structure used in the *CluStream* algorithm [Aggarwal et al. 2003].

an input parameter. Figure 4 shows the CluStream structure. Each micro-cluster has a maximum boundary, which is computed as the standard deviation of the mean distance of the cluster objects to their centroids multiplied by a factor f. For each new object, the closest micro-cluster (according to the Euclidean distance) is selected to absorb it. For deciding whether a cluster should absorb a new object or not, it is verified if the distance between the new object and the closest centroid falls within the maximum boundary. If so, the object is absorbed by the cluster and its summary statistics are updated. If none of the micro-clusters can absorb the object, a new micro-cluster is created. This is accomplished by either deleting the oldest micro-cluster or by merging two micro-clusters. The oldest micro-cluster is deleted if its timestamp is below a given threshold δ (input parameter), which is deemed to be an outlier and therefore removed. Thus, the CluStream algorithm finds the arrival time (known as the *relevance time*) of the $m/(2N_i)^{th}$ percentil of the N_i objects in a micro-cluster i, whose timestamps are assumed to be normally distributed. Otherwise, the two closest micro-clusters are merged, using the additivity property of the **CF** vectors, which takes $O(q^2)$ time. The q micro-clusters are stored in a secondary storage device from time to time, *i.e.*, in time intervals that decrease exponentially — α^l, where α and l are user-defined parameters — the so-called snapshots. These snapshots allow the user to search for clusters in different time horizons, h, through a *pyramidal time window* concept [Aggarwal et al. 2003].

Similar to *CluStream*, the authors in [Zhou et al. 2008] propose the *SWClustering* algorithm, which uses a Temporal **CF** vector (**TCF**). **TCF** holds the three components of the original **CF** vector, plus the timestamp t of its most recent object. *SWClustering* also has a new data structure called **EHCF** (Exponential Histogram of Cluster Feature), which is defined as a collection of **TCF**s. Each **EHCF** is distributed in levels that contain at most $\frac{1}{\phi} + 1$ **TCF**s, where $0 < \phi < 1$ is a user-defined parameter. The number of objects in a given TCF_i is the same or twice as much of the number of objects in TCF_j, for $i > j$. Initially, the first **TCF** contains only one object. The center of **EHCF** is computed as the mean of the LS of all **TCF**s from an **EHCF**. When a new object $\mathbf{x}$ arrives, the nearest **EHCF** is selected (according to the Euclidean distance between the object and the center of **EHCF**). If nearest **EHCF** can absorb $\mathbf{x}$, *i.e.*, its distance to the object $\mathbf{x}$ is below

$R * \beta$, where R is the radius of nearest **EHCF** and β is a threshold radius ($\beta > 0$), then $\mathbf{x}$ is inserted in this **EHCF**. Else, a new **EHCF** is created. However, it is necessary to check if the maximum number of allowed **EHCF**s is reached. If so, the two nearest **EHCF** are merged. Then, the expired records of the **EHCF** are deleted, leaving only the most recent N timestamps.

DenStream [Cao et al. 2006] is a density-based data stream clustering algorithm that also uses a feature vector based on *BIRCH*. In its online phase, two structures — *p-micro-clusters* (potential clusters) and *o-micro-clusters* (a buffer for aiding outlier detection) — are provided to hold all the information needed for clustering the data. Each *p-micro-cluster* structure has an associated weight w that indicates its importance based on temporality (micro-clusters with no recent objects tend to lose importance, *i.e.* their respective weights continuously decrease over time in outdated *p-micro-clusters*). The weight of the micro-cluster, w, at time T is computed according to Equation (4), where $t^1, ..., t^j$ are the timestamps, and the importance of each object decreases according to the fading function in Equation (5), parameterized with λ, a user-defined parameter.

$$w = \sum_{j \in p\text{-}micro\text{-}cluster} f(T - t^j), \tag{4}$$

$$f(t) = 2^{-\lambda t} \tag{5}$$

Two other statistics are stored for each *p-micro-cluster*: the weighted linear sum of objects (WLS) and the weighted sum of squared of objects (WSS), computed according to Equations (6) and (7), respectively. From these equations, it is possible to compute the radius r of each *p-micro-cluster* (Equation (8)) as well as its mean.

$$WLS = \sum_{j \in p\text{-}micro\text{-}cluster} f(T - t^j)\mathbf{x}^j \tag{6}$$

$$WSS = \sum_{j \in p\text{-}micro\text{-}cluster} f(T - t^j){\mathbf{x}^j}^2 \tag{7}$$

$$r = \sqrt{\sum_{j=1}^{n} \left(\frac{WSS^j}{w} - \left(\frac{WLS^j}{w} \right)^2 \right)} \tag{8}$$

Each *o-micro-clusters* structure is defined in a similar way. The timestamp of creation for each *o-micro-cluster*, T_{Ini}, is also stored.

When a new object $\mathbf{x}^j$ arrives, the algorithm tries to insert it into its nearest *p-micro-cluster* by updating the cluster summary statistics. The insertion will be successful if its updated radius is within a pre-defined boundary ϵ (input parameter). Otherwise, the algorithm tries to insert $\mathbf{x}^j$ into its closest *o-micro-cluster* by updating its summary statistics. In this case, the insertion is successful if its updated radius is within ϵ. Moreover, if the updated *o-micro-cluster* weight exceeds $\beta \times \mu$, this *o-micro-cluster* has grown into a potential *p-micro-cluster*. Both β and μ are input parameters. β controls the threshold level, whereas μ is the integer weight

of a given *p-micro-cluster*. If $\mathbf{x}^j$ was not absorbed by its closest *o-micro-cluster*, then a new *o-micro-cluster* is created to absorb $\mathbf{x}^j$.

At delimited time periods T_p (given by Equation(9)), the set of *p-micro-clusters* is checked to verify whether a *p-micro-cluster* should become an *o-micro-cluster*. Similarly, *o-micro-clusters* may become *p-micro-clusters* after the analysis of their corresponding weights. If the parameter values of λ, β, and μ suggested by the authors [Cao et al. 2006] are employed, this clean-up task is performed quite often, *i.e.*, $T_p \leq 4$, leading to a high computational cost.

$$T_p = \frac{1}{\lambda} \log(\frac{\beta\mu}{\beta\mu - 1}) \tag{9}$$

Similar to *Denstream*, the *ClusTree* algorithm [Kranen et al. 2011] also proposes to use a weighted **CF**-vector, which is kept into a hierarchical tree (R-tree family). Two parameters are used to build this tree: the number of entries in a leaf node and the number of entries in non-leaf nodes. *ClusTree* provides strategies for dealing with time constraints for anytime clustering, *i.e.*, the possibility of interrupting the process of inserting new objects in the tree at any moment. This algorithm makes no *apriori* assumption on the size of the clustering model, since its aggregate and split operations adjust the size of the model automatically. The objects that were not inserted due to an interruption are temporarily stored in the buffer of the immediate sub-tree entry. When the sub-tree is accessed again, these objects are taken along as a "hitchhiker", and the operation of object insertion in a leaf node continues. *ClusTree* can also adapt itself to fast and slow streams. In fast streams, *ClusTree* aggregates similar objects in order to do a faster insertion in the tree. In slow streams, the idle time is used to improve the quality of the clustering.

3.1.2 *Prototype Array.* Some data stream clustering algorithms use a simplified summarization structure, hereby named *prototype array* [Domingos and Hulten 2001; Shah et al. 2005]. It is an array of prototypes (e.g., medoids or centroids) that summarizes the data partition.

For instance, *Stream* [Guha et al. 2000] employs an array of prototypes for summarizing the stream by dividing the data stream into chunks of size $m = N^\rho$, $0 < \rho < 1$. Each chunk of m objects is summarized in $2k$ representative objects by using a variant of the k-medoids algorithm [Kaufman and Rousseeuw 1990] known as Facility Location [Charikar and Guha 1999; Meyerson 2001]. The process of compressing the description of the data objects is repeated until an array of m prototypes is obtained. Next, these m prototypes are further compressed (clustered) into $2k$ prototypes and the process continues along the stream (see Figure 5).

Stream LSearch [O'Callaghan et al. 2002] uses a similar summarizing structure. This algorithm assumes that the objects arrive in chunks $X_1, X_2, ..., X_Z$, where each chunk X_i ($i \in [1, Z]$) can be clustered in the memory, thus producing k clusters. At the i^{th} chunk of the stream, the algorithm retains $O(i \times k)$ medoids. However, as $Z \to \infty$, it is not possible to keep the $O(i \times k)$ medoids in memory. Therefore, when the main memory is full, *Stream LSearch* clusters the $O(i \times k)$ medoids and keeps in memory only the k medoids obtained by this process.

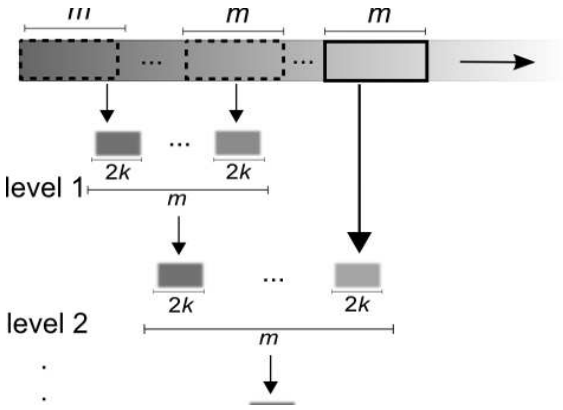

Fig. 5. Overview of *Stream* [Guha et al. 2000], which makes use of a prototype array.

3.1.3 *Coreset Tree.* A significantly different summary data structure for data stream clustering is the **coreset tree** employed in *StreamKM++* [Ackermann et al. 2012]. This structure is a binary tree in which each tree node i contains the following components: a set of objects, E_i; a prototype of E_i, $\mathbf{x}^{p_i}$; the number of objects in E_i, N_i; and the sum of squared distances of the objects in E_i to $\mathbf{x}^{p_i}$, SSE_i. E_i only has to be stored in the leaf nodes of the **coreset tree**, because the objects of an inner node are implicitly defined as the union of the objects of its child nodes.

The **coreset tree** structure is responsible for reducing $2m$ objects to m objects. The construction of this structure is defined as follows. First, the tree has only the root node v, which contains all the $2m$ objects in E_v. The prototype of the root node $\mathbf{x}^{p_v}$ is chosen randomly from E_v and $N_v = |E_v| = 2m$. The computation of SSE_v follows from the definition of $\mathbf{x}^{p_v}$. Afterwards, two child nodes for v are created, namely: v_1 and v_2. To create these nodes, it is necessary to choose an object from E_v with probability proportional to $\frac{Dist(\mathbf{x}^{j_v},\mathbf{x}^{p_v})^2}{SSE_v}, \forall \mathbf{x}^{j_v} \in E_v$, *i.e.*, the object that is farthest away from $\mathbf{x}^{p_v}$ has the highest probability of being selected. We call the selected object $\mathbf{x}^{q_v}$. The next step is to distribute the objects in E_v to E_{v_1} and E_{v_2}, such that:

$$E_{v_1} = \{\mathbf{x}^{i_v} \in E_v | Dist(\mathbf{x}^{i_v}, \mathbf{x}^{p_v}) < Dist(\mathbf{x}^{i_v}, \mathbf{x}^{q_v})\} \quad (10)$$

$$E_{v_2} = E_v \backslash E_{v_1}. \quad (11)$$

Later, the summary statistics of child node v_1 are updated, *i.e.*, $\mathbf{x}^{p_{v_1}} = \mathbf{x}^{p_v}$, $N_{v_1} = |E_{v_1}|$ and SSE_{v_1} follows from the definition of $\mathbf{x}^{p_{v_1}}$. Similary, the summary statistics of child node v_2 are updated, but note that $\mathbf{x}^{p_{v_2}} = \mathbf{x}^{q_v}$. This is the *expansion* step of the tree, which creates two child nodes for a given inner node. When the tree has many leaf nodes, one has to decide which one should be expanded first. For such, it is necessary to start from the root node of the **coreset tree** and descend it by iteratively selecting a child node with probability proportional to $\frac{SSE_{child}}{SSE_{parent}}$, until a leaf node is reached for the expansion step to be re-started. The **coreset tree** expansion stops when the number of leaf nodes is m.

StreamKM++ [Ackermann et al. 2012] is a two-step algorithm, *i.e.*, *merge-and-*

reduce. The reduce step is performed by the **coreset tree**, considering that it reduces $2m$ objects to m objects. The merge step is performed by another data structure, namely the **bucket set**, which is a set of L buckets (also named buffers), where L is an input parameter. Each bucket can store m objects. When a new object arrives, it is stored in the first bucket. If the first bucket is full, all of its data are moved to the second bucket. If the second bucket is full, a merge step is computed, *i.e.*, the m objects in the first bucket are merged with the m objects in the second bucket, resulting in $2m$ objects, which, by their turn, are reduced by the construction of a **coreset tree**, as previously detailed. The resulting m objects are stored in the third bucket, unless it is also full, and then again a new merge-and-reduce step is needed. This procedure is illustrated by the pseudo-code in Figure 6.

Input: New object $\mathbf{x}^j$, bucket set $B = \bigcup_{i=1}^{L} B_i$, size m.
Output: Updated bucket set B.
1: $B_0 \leftarrow B_0 \cup \{\mathbf{x}^j\}$
2: **if** $|B_0| \geq m$ **then**
3: create temporary bucket Q.
4: $Q \leftarrow B_0$
5: $B_0 \leftarrow \emptyset$
6: $i \leftarrow 1$
7: **while** $B_i \neq \emptyset$ **do**
8: $Q \leftarrow coresetReduction(B_i \cup Q)$
9: $B_i \leftarrow \emptyset$
10: $i \leftarrow i + 1$
11: **end while**
12: $B_i \leftarrow Q$
13: $Q \leftarrow \emptyset$
14: **end if**

Fig. 6. Pseudo-code for the insertion of a new object into the **bucket set** [Ackermann et al. 2012]. Function $coresetReduction(A \cup B)$ (line 8) receives $2m$ objects and returns m summarized objects.

3.1.4 *Grids.* Some data stream clustering algorithms perform data summarization through grids [Cao et al. 2006; Park and Lee 2007; Chen and Tu 2007; Gama et al. 2011], *i.e.*, by partitioning the n-dimensional feature space into density grid cells. For instance, *D-Stream* [Chen and Tu 2007] maps each data stream object into a density grid cells. Each object at time t is associated to a density coefficient that decreases over time, as shown in Equation (12), where $\lambda \in (0, 1)$ is a decay factor. The density of a grid cell g at time t, $D(g, t)$, is given by the sum of the adjusted densities of each object that is mapped to g at or before time t ($E(g, t)$), as shown in Equation (13). Each grid cell is represented by a tuple $< t_g, t_m, D, label, status >$, where t_g is the last time the grid cell was updated, t_m is the last time the grid cell was removed from the hash table that holds the valid grid cells, D is the grid cell density at its last update, *label* is the class-label of the grid cell and *status* indicates whether the grid cell is *NORMAL* or *SPORADIC*, as will be explained later.

$$D(\mathbf{x}^j, t) = \lambda^{t-t^j} \tag{12}$$

$$D(g,t) = \sum_{x \in E(g,t)} D(x,t) \tag{13}$$

The grid cells maintenance is performed during the online phase. A grid cell can become sparse if it does not receive new objects for a long time. In contrast, a sparse grid cell can become dense if it receives new objects. At fixed intervals of time (dynamic parameter *gap*), the grid cells are inspected with regard to their status. Considering that the number of grid cells may be large, specially in high-dimensional streams, only the grid cells that are not empty are stored. Additionally, grid cells with few objects are treated as outliers ($status$ = SPORADIC). Sporadic grid cells are periodically removed from the list of valid grid cells. Also, during the online phase, when a new n-dimensional object arrives, it is mapped into its corresponding grid cell g. If g is not in the list of valid grid cells (structured as a hash table), it is inserted in it and its corresponding summary is updated.

DGClust [Rodrigues et al. 2008; Gama et al. 2011] is an algorithm for distributed clustering of sensor data that also employs grid cells for summarizing the stream. It receives data from different sensors — where each sensor produces a univariate data stream. The data are processed locally in each sensor and when there is an update in a local grid cell (state change), this is communicated to the central site. The local site communicates the local state change by sending the number of the grid cell that was updated. The global state is a combination of the local states (grid cells) of each sensor. Each local site i keeps two layers of discretization with p_i and w_i bins, respectively, where $k < w_i < p_i$. The discretization algorithm used for generating the bins for each layer is Partition Incremental Discretization (PID) [Gama and Pinto 2006], which assumes grid cells of equal width. Each time a new value x_i^t is read, the counter of the corresponding bin is incremented in both the first and second layers. The number of bins in the first layer may change, given that the following condition is met: if the value of the counter associated to a bin in the first layer is larger than a user-defined threshold, α, the bin is split into two. The second layer discretizes the p_i bins into w_i bins, *i.e.*, it summarizes the information of the first layer in a higher granularity. The object counter of a bin in the second layer is incremented when the corresponding bin in the first layer is incremented. Next, a communication with the central site is performed to send the update information, so that the global state is updated at each timestamp. If there was a split, all bins of the second layer are sent to the central site, otherwise only the updated bin is sent to the central site.

3.2 Window Models

In most data stream scenarios, more recent information from the stream can reflect the emerging of new trends or changes on the data distribution. This information can be used to explain the evolution of the process under observation. Systems that give equal importance to outdated and recent data do not capture the evolving characteristics of stream data [Chen and Tu 2007]. The so-called moving window techniques have been proposed to partially address this problem [Barbará 2002; Babcock et al. 2003; Gama 2010]. There are three commonly-studied models in data streams [Zhu and Shasha 2002]: i) sliding windows; ii) damped windows; and

iii) landmark windows.

3.2.1 *Sliding Window Model.* In the sliding window model, only the most recent information from the data stream are stored in a data structure whose size can be variable or fixed. This data structure is usually a *first in, first out* (*FIFO*) structure, which considers the objects from the current period of time up to a certain period in the past. The organization and manipulation of objects are based on the principles of queue processing, where the first object added to the queue will be the first one to be removed. In Figure 7, we present an example of the sliding window model.

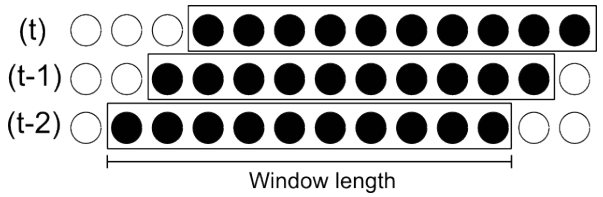

Fig. 7. Sliding window model.

Several data stream clustering algorithms find clusters based on the sliding window model, *e.g.* [Babcock et al. 2003; Zhou et al. 2008; Ren and Ma 2009]. In summary, these algorithms only update the statistic summaries of the objects inserted into the window. The size of the window is set according to the available computational resources.

3.2.2 *Damped Window Model.* Differently from sliding windows, the damped window model, also referred to as time-fading model, considers the most recent information by associating weights to objects from the data stream [Jiang and Gruenwald 2006]. More recent objects receive higher weight than older objects, and the weight of the objects decrease with time. An illustrative example of the damped window model is presented in Figure 8, in which the weight of the objects exponentially decays from black (most recent) to white (expired).

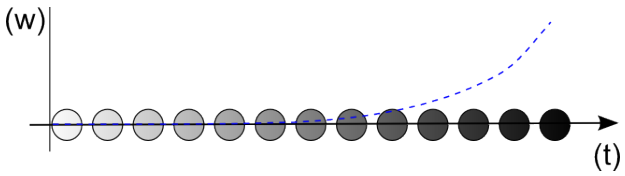

Fig. 8. Damped window model.

This model is usually adopted in density-based clustering algorithms [Cao et al. 2006; Chen and Tu 2007; Isaksson et al. 2012]. These algorithms usually assume an exponential decay function to weight the objects from the stream. In [Cao et al. 2006], *e.g.*, the adopted decay function follows the exponential function given by Equation (5), where the $\lambda > 0$ parameter determines the decay rate and t is the current time. The higher the value of λ, the lower the importance of the past data regarding the most recent data. The *D-Stream* algorithm [Chen and Tu 2007]

assigns a density coefficient for each element that arrives from the stream, whose value decreases with the object's age. This density coefficient is given by λ^{t-t_c}, where t_c is the instant in time that the object arrived from the stream.

3.2.3 *Landmark Window Model.* Processing a stream based on landmark windows requires handling disjoint portions of the streams (chunks), which are separated by landmarks (relevant objects). Landmarks can be defined either in terms of time, (*e.g.*, on daily or weekly basis) or in terms of the number of elements observed since the previous landmark [Metwally et al. 2005]. All objects that arrived after the landmark are kept or summarized into a window of recent data. When a new landmark is reached, all objects kept into the window are removed and the new objects from the current landmark are kept in the window until a new landmark is reached. Figure 9 illustrates an example of landmark window.

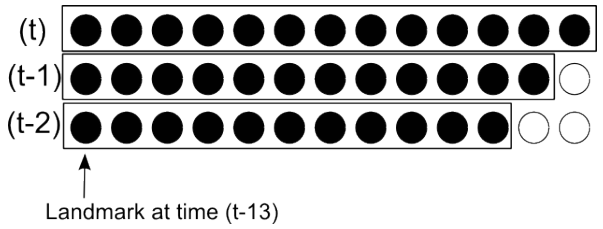

Fig. 9. Landmark window for a time interval of size 13.

Data stream clustering algorithms that are based on the landmark window model include [O'Callaghan et al. 2002; Bradley et al. 1998; Farnstrom et al. 2000; Ackermann et al. 2012; Aggarwal et al. 2003]. In [O'Callaghan et al. 2002], *e.g.*, the *Stream* algorithm adopts a divide-and-conquer strategy based on a landmark window whose landmark is defined at every m number of objects. Note that, in this kind of window model, the relationship between objects from neighboring windows is not considered.

The problem in using any fixed-length window scheme is in finding out the ideal window size to be employed. A small window guarantees that the data stream algorithm will be able to rapidly capture eventual concept drifts. At the same time, in stable phases along the stream, it may affect the performance of the learning algorithm. On the other hand, a large window is desirable in stable phases, though it may not respond rapidly to concept drifts [Gama et al. 2004].

3.3 Outlier Detection Mechanisms

Besides the requirements of being incremental and fast, data stream clustering algorithms should also be able to properly handle outliers throughout the stream [Barbará 2002]. Outliers are objects that deviate from the general behavior of a data model [Han and Kamber 2000], and can occur due to different causes, such as problems in data collection, storage and transmission errors, fraudulent activities or changes in the behavior of the system.

Density-based approaches look for low-density regions in the input space, which may indicate the presence of outliers. For instance, the *BIRCH* algorithm [Zhang

et al. 1996] has an optional phase that scans the **CF** tree and stores leaf entries with low density on a disk. The number of bytes reserved to store outliers on the disk is specified by the user. The **CF** vectors with low density — estimated according to a threshold value — are considered to be outliers. The threshold value is specified by the average size of the **CF** vectors on leaf nodes. Periodically, the algorithm checks whether the **CF** vectors stored on the disk (outlier candidates) can be absorbed by the current **CF** tree (kept in main memory). This monitoring occurs when either the disk runs out of space or the entire stream (assuming a finite one) has been processed. Potentially, this optional phase can be used to monitoring changes in the data distribution when more data are absorbed by the **CF** vectors.

The *DenStream* algorithm [Cao et al. 2006] introduces the notion of outlier-buffer. The online phase of *DenStream* keeps the statistical summaries of the stream by means of *p-micro-clusters*. Every T_p time periods — see Equation (9) — the online phase of *DenStream* checks the *p-micro-clusters* to identify potential outliers, the so-called *o-micro-clusters*. These are described by the tuple $< WLS, WSS, w, T_{Ini} >$, where T_{Ini} is the timestamp of their creation. A *p-micro-cluster* becomes an *o-micro-clusters* if its weight (w) is below the outlier threshold ($w < \beta\mu$), where β and μ are user-defined parameters. Note that keeping all *o-micro-clusters* in memory may become prohibitive after some time. Hence, some *o-micro-clusters* need to be removed. The idea is to keep in the outlier-buffer only the *o-micro-clusters* that may become *p-micro-clusters* (*i.e.*, *o-micro-clusters* whose weight increases over time). In order to safely remove the "real" outliers, at every T_p time period, the weights of the *o-micro-clusters* are checked, and all those whose weight is below the limit are removed. The limit is captured by Equation (14), where T is the current time.

$$\xi(T, T_{Ini}) = \frac{2^{-\lambda(T - T_{Ini} + T_p) - 1}}{2^{-\lambda T_p} - 1} \tag{14}$$

The *D-Stream* algorithm [Chen and Tu 2007] identifies and removes low-density grid cells that are categorized into sporadic grid cells. Such sporadic grid cells (outlier candidates) can occur for two reasons: i) grid cells that have been receiving very few objects; and ii) crowded grid cells that have their densities reduced by means of the decay factor. The goal is to remove sporadic grid cells that occur by the first reason. A hash table stores the list of grid cells, and the algorithm checks periodically which grid cells in the hash table are sporadic. To do so, at every *gap* time interval (Equation (16)), the grid cells whose density is below a given threshold are labeled as sporadic grid cells. If, in the next gap time period, the grid cells are still labeled as sporadic, they are removed from the hash table. The threshold to determine grid cells with low density ($D(g,t) < \pi(t_g, t)$) is calculated according to Equation (15), where t is the current time, t_g is the last update time ($t > t_g$), C_l and C_m are user defined parameters, G is the number of grid cells and $\lambda \in (0,\ 1)$ is a constant called *decay factor*. We note that by employing the parameter values suggested by the authors [Chen and Tu 2007], the grid cells are inspected every $gap \leq 2$ objects, which may be computationally costly.

$$\pi(t_g, t) = \frac{C_l(1 - \lambda^{t-t_g+1})}{G(1-\lambda)} \tag{15}$$

$$gap = \left\lfloor log_\lambda \left(max \left\{ \frac{C_l}{C_m}, \frac{N - C_m}{N - C_l} \right\} \right) \right\rfloor \tag{16}$$

4. OFFLINE CLUSTERING STEP

In this section, we discuss the clustering step, which typically involves the application of a *standard* clustering algorithm to find clusters on the previously generated statistical summaries.

One of the most popular algorithms for data clustering is k-means [MacQueen 1967] due to its simplicity, scalability and empirical success in many real-word applications [Wu et al. 2007]. Not surprisingly, k-means and its variants are widely used in data stream scenarios [Bradley et al. 1998; Farnstrom et al. 2000; O'Callaghan et al. 2002; Aggarwal et al. 2003; Zhou et al. 2008; Ackermann et al. 2012].

A powerful idea in clustering data streams is the use of **CF** vectors [Gama 2010], as previously discussed in Section 3.1. Some k-means variants have been proposed for dealing with **CF** vectors. In [Zhang et al. 1997], *e.g.*, the authors suggest three ways to adapt the k-means algorithm to handle **CF** vectors:

(1) Calculate the centroid of each **CF** vector — $\frac{LS}{N}$ — and consider each centroid as an object to be clustered by k-means.
(2) Do the same as before, but weighting each object (**CF** vector centroid) proportionally to N, so that **CF** vectors with more objects will have a higher influence on the centroid calculation process performed by k-means.
(3) Apply the clustering algorithm directly to the **CF** vectors, since their components keep the sufficient statistics for calculating most of the required distances and quality metrics.

The first and third strategies are commonly used by clustering algorithms based on **CF** vectors. The first strategy is the simplest one to be used in practice, since no further modification of the clustering algorithm is needed. This strategy is suggested by the *ClusTree* algorithm [Kranen et al. 2011] to group the leaf nodes (**CF** entries) in order to produce k clusters. The third strategy requires the modification of the clustering algorithm to properly handle the **CF** vectors as objects. In *CluStream* [Aggarwal et al. 2003], the employed k-means variant uses an adapted version of the second strategy that chooses the initial prototypes with a probability proportional to the number of objects in a micro-cluster (expanded **CF** vector). This variant is presented in Figure 10.

In [Bradley et al. 1998], another k-means variant for dealing with **CF** vectors is presented. This variant, named *Extended k-means*, uses both the **CF** vectors of the data that have been processed and new objects as input to find k clusters. It also considers the **CF** vector as an object weighted by N. This is similar to the idea presented in Figure 10, but it contains an additional step to handle empty clusters. After convergence (step 6), clusters are verified in order to detect empty groups. An empty cluster has its center set to the farthest object from it and the *Extended k-means* algorithm is called again to update the new prototypes [Bradley

Input: Number of clusters k, and set of micro-clusters $Q = \{Q_1, Q_2, ..., Q_q\}$.
Output: Data partition with k clusters.
1: Consider each micro-cluster centroid, $\frac{LS}{N}$, as an object.
2: Initialization: k initial prototypes are sampled with probability proportional to N.
3: **repeat**
4: Partitioning: compute the distance between prototypes and micro-clusters.
5: Updating: the new prototype is defined as the weighted centroid of the objects in a cluster.
6: **until** Prototypes get stabilized.

Fig. 10. k-means variant to handle statistical summaries [Aggarwal et al. 2003].

and Fayyad 1998]. In [Farnstrom et al. 2000], the *Extended k-means* algorithm is also used in the *Single-pass k-means* framework.

Yet another k-means variant to handle statistical summaries is presented in [Ackermann et al. 2012]. This clustering algorithm, named *k-means++*, can be viewed as a seeding procedure for the original k-means algorithm. As detailed in Section 3.1, the authors in [Ackermann et al. 2012] propose a way of summarizing the data stream by extracting a small set of objects, named **coreset** [Bādoiu et al. 2002; Agarwal et al. 2004]. Recall that a **coreset** is a small (weighted) set of objects that approximates the objects from the stream regarding the k-means optimization problem. The algorithm proposed in [Ackermann et al. 2012] extracts the coreset from the stream by means of a merge-and-reduce technique [Har-Peled and Mazumdar 2004] and finds k clusters through the *k-means++* algorithm, described in Figure 11.

Input: Number of clusters k and coreset M.
Output: Data partition.
1: Choose an initial center c_1 uniformly at random from M.
2: **for** i=2 to k **do**
3: Let $d(\mathbf{x}^j)$ be the shortest distance from an object $\mathbf{x}^j \in M$ to its closest center already chosen $\{c_1, ..., c_{i-1}\}$
4: Choose the next center $c_i = \mathbf{x}^t \in M$ with probability $d(\mathbf{x}^t)^2 / \sum_{\mathbf{x}^j \in M} d(\mathbf{x}^j)^2$.
5: **end for**
6: Proceed with the standard k-means algorithm.

Fig. 11. k-means++ algorithm [Arthur and Vassilvitskii 2007].

The *LSearch* algorithm [O'Callaghan et al. 2002] uses the concept of *facility location* [Meyerson 2001] to find a solution to the k-medoids optimization problem. The main idea is to find the facility location (cluster medoid) that represents objects from the stream by minimizing a cost function. Each facility (medoid) has an associated cost to be opened on a given location and a service cost to attend demanding objects. The cost function is the sum of the associated costs to open facilities and the service costs. Thus, it is a combination of the sum of squared errors (SSE) and a cost to insert a medoid within a partition, providing more flexibility to find the number of clusters. Nevertheless, the user still needs to provide an initial estimate of k before running the algorithm. *LSearch* searches for a data partition with the number of clusters between $[k, 2k]$. Ackermann et al. [2012] observe that *LSearch* does not always find the pre-specified k and that usually the difference lies within a 20% margin from the value of k chosen in advance.

Besides k-means, density-based clustering algorithms, like *DBSCAN* [Ester et al. 1996], are also used in the offline clustering step. In [Cao et al. 2006], the authors present the *DenStream* algorithm, which uses a feature vector approach for summarizing the data and a *DBSCAN* variant for performing data clustering. This variant receives as input the *p-micro-clusters* (feature vectors) and two parameters — ϵ and μ, previously presented in Section 3.1 — to partition the data. Each *p-micro-cluster* structure is seen as a virtual object with center equals to $\frac{LS}{WA_i}$, where WA_i is the weighting area of objects in a given neighborhood. Even though the user does not need to explicitly specify the number of clusters, the definition of ϵ and μ has a strong influence on the resulting data partition.

As seen in Section 3.1, another paradigm for clustering data streams partitions the data space into discretized grid cells. Typically, these algorithms create a grid data structure by dividing the data space into grid cells followed by the use of a standard clusterer to cluster these cells. As an example, the offline component of *D-Stream* [Chen and Tu 2007] adopts an agglomerative clustering strategy to group grid cells. The algorithm starts by assigning each dense cell to a cluster. Afterwards, an iterative procedure merges two dense cells that are strongly correlated into a single cluster. This procedure is repeated until no changes in the partition can be performed. A parameter whose value is defined by the user determines if two grid cells are strongly correlated.

Based on adaptive grid cells, the Distributed Grid Clustering algorithm (*DGClust*) [Rodrigues et al. 2008; Gama et al. 2011] is an online 2-layer distributed clustering algorithm for sensor data. *DGClust* reduces data dimensionality by monitoring and clustering only frequent states. As previously mentioned, the *DGClust* algorithm is composed of local and central sites, and each sensor is related to a univariate stream whose values are monitored in a local site. The goal of the central site is to find k clusters and keep the data partition continuously updated. In order to reduce its computational complexity, *DGClust* keeps only the top-m ($m > k$) most frequent global states. The central object of each of the most frequent global states will be used in the final clustering. As soon as the central site finds the top-m set of states, a simple partitioning algorithm can be applied to the most frequent states, to minimize the cluster radius (or equivalently the cluster diameter). The Furthest Point algorithm [Gonzalez 1985] is used for this task. It selects an initial object as the seed to the first cluster and iteratively selects the next object as the center cluster if its distance to the remaining clusters is maximized. In order to dynamically adjust the data partition, the algorithm operates in one of two possible conditions: *converged* or *non-converged*. If the system is operating in the *non-converged* condition, when a new state $s(t)$ is reached, it updates the centers of the clusters according to the top-m states. If the system has already converged and the current state has effectively become a part of the top-m states, the system updates the centers and changes its status to *non-converged*.

5. TEMPORAL ASPECTS

Besides the time and space constraints that distinguish batch-mode clustering from data stream clustering, the influence of time is a very important issue when clustering data streams. Indeed, there are several works in the literature that stress

the importance of considering the time element when designing a data stream clustering algorithm. We highlight the following temporal aspects one should consider when designing a new algorithm: (i) time-aware clustering; (ii) outlier-evolution dilemma; and (iii) cluster tracking. We detail them next.

5.1 Time-aware Clustering

The inherent time element in data streams should be properly exploited by data stream clustering algorithms. For instance, these algorithms should be able to implicitly or explicitly consider the influence of time during the clustering process (time-aware clustering). Current data stream clustering algorithms perform time-aware clustering by either assigning different levels of importance to objects (considering that recent data is more relevant than old data) or by modeling the behavior of the arriving data in such a way that objects can be clustered regarding different temporal patterns instead of a traditional spatial-based approach.

In the first case, the clustering process is affected by the *age* of objects, which is explicitly modeled by a decay function [Aggarwal et al. 2003; Cao et al. 2006; Chen and Tu 2007; Kranen et al. 2011], as previously mentioned in Sections 3.2 and 3.3. For the second case, a typical example is the *Temporal Structure Learning for Clustering Massive Data Stream in Real Time* (*TRACDS*) algorithm [Hahsler and Dunham 2011]. It is essentially a generalization of the *Extensible Markov Model (EMM)* algorithm [Dunham et al. 2004; Hahsler and Dunham 2010] for data stream scenarios. In *TRACDS*, each cluster (or micro-cluster) is represented by a state of a Markov Chain (MC) [Markov 1971; Bhat and Miller 2002], and the transitions represent the relationship between clusters. With the MC model, TRACDS can model the behavior of the objects that continuously arrive through state-change probabilities, in which the time element is implicitly considered through the different temporal patterns of the sequences of objects.

5.2 Outlier-Evolution Dilemma

As previously seen in Section 3.3, outlier detection mechanisms can be modeled with the help of decay functions [Zhang et al. 1997; Aggarwal et al. 2003; Cao et al. 2006; Chen and Tu 2007]. These functions evaluate the relevance of clusters according to their age, assuming that clusters that are seldom updated should be deemed as outliers. Nevertheless, we observe that there is a thin line between outlier detection and cluster evolution, and correctly distinguishing between them may be an application-dependent procedure. In certain applications, objects deemed as outliers may actually be the indication of a new emerging cluster. For instance, Barbará [2002] cites an example of a weather data application, in which sufficient outliers indicate a new trend that needs to be represented by new clusters.

There are cases in which outliers actually redefine the boundaries of existing clusters. An example is a data stream of spotted cases of an illness, in which outliers indicate the spread of the epidemics over larger geographical areas [Barbará 2002]. Finally, there are scenarios in which objects deemed as outliers are indeed noise produced by uncalibrated sensors or improper environmental influence during data collection. An uncalibrated sensor may give the false impression that a new emerging cluster is arising when there is actually a large amount of spurious objects that should not be considered during clustering.

For the cases in which there are changes in the data probability distribution (*e.g.*, real-time surveillance systems, telecommunication systems, sensor networks, and other dynamic environments), the data stream clustering algorithm should ideally be able to detect these changes and adapt the clusters accordingly. Aggarwal [2003] proposes a framework towards efficient change detection that enables the diagnostic of fast and multidimensional data streams. It allows visualizing and determining trends in the evolution of the stream, according to a concept called *velocity density estimation*. This mechanism creates temporal and spatial velocity profiles, which in turn can be used to predict different types of data evolution. Even though this framework is not meant particularly for clustering applications, it can help users to understand the nature of the stream, and perhaps give new insight to researchers for developing change detection mechanisms for data stream clustering algorithms.

5.3 Cluster Tracking

The exploration of the stream over different time windows can provide the users a better comprehension about the dynamic behavior of the clusters. Hence, data stream clustering algorithms must provide to the user a way to examine clusters occurring in different granularities of time (*e.g.,* daily, monthly, yearly). Statistics summary structures like **CF** vectors are a powerful tool to help in the cluster exploration due to its additivity and subtractive properties.

In addition, clusters upon the data of many real applications are affected by changes the underlying data suffers with time. Whereas many studies have been devoted to adapting clusters to the evolved data, we believe it is necessary to encompass tracing and understanding of cluster evolution itself, as a means of gaining insights on the data and supporting strategic decisions. In other words, it is necessary to provide insights about the nature of cluster change: is a cluster disappearing or are its members migrating to other clusters? Does a new emerging cluster reflect a new profile of objects (novelty detection) or does it rather consist of old objects whose characteristics have evolved?

Following the necessity of tracking and understanding cluster evolution, *MONIC* [Spiliopoulou et al. 2006] is an algorithm that was proposed for modeling and tracking clustering transitions. These transitions can be internal, related to each cluster, or external, related to the clustering process as a whole. *MONIC* categorizes internal transitions into three types: i) changes in compactness; ii) changes in size; and iii) changes in location. For external transitions, five outcomes are possible: i) the cluster survives; ii) the cluster is split into multiple clusters; iii) the cluster is absorbed by another cluster; iv) the cluster disappears; and v) a new cluster emerges. The transition tracking mechanism is based on the degree of overlapping between two clusters. Overlapping is defined as the number of common objects weighted by the age of the objects.

Another algorithm that performs cluster tracking is *MEC* [Oliveira and Gama 2010; 2012], which traces the evolution of clusters over time through the identification of the temporal relationship among them. It aims at identifying hidden behavioral patterns and developing knowledge about the evolution of the phenomena. Unlike *MONIC*, *MEC* employs different metrics to detect changes and provide techniques to visualize the cluster evolution. A bipartite graph structure is used to visualize the clusters evolution and to formalize the definition of transition. This

structure is used to compute the conditional probabilities for every pair of possible connections between nodes of a bipartite graph (clusters) obtained at consecutive time points.

6. ASSESSING CLUSTER STRUCTURES

An issue that arises when one proposes a new data stream clustering algorithm is how to properly assess its effectiveness. Determining suitable criteria to validate new and even existing algorithms is particularly important.

The literature on data clustering is very large, and due to the inherent subjectivity of the clustering task, several methodologies and clustering validity measures have been proposed in the past decades. Works on data stream clustering usually adopt well-known evaluation criteria. For instance, the most commonly employed criteria to evaluate the quality of stream data partitions are the sum of squared errors (SSE) and the so-called *purity*. The former is an internal validity criterion, whereas the latter is an external validity criterion, in which the true labels (groups) of the data are available and are compared with the data partition obtained by a clustering algorithm.

The SSE criterion evaluates the compactness of clusters. The lower the SSE value, the more compact the clusters of the resulting partition are. The SSE criterion can be formally described by Equation (17), where $\mathbf{c}_i$ is the centroid of cluster C_i. SSE decreases monotonically as the number of clusters increase. Hence, it cannot be used to estimate the optimal value of k, because it tends to find the trivial solution, namely: N singletons.

$$\sum_{i=1}^{K} \sum_{\mathbf{x}_j \in C_i} \|\mathbf{x}_j - \mathbf{c}_i\|^2 \tag{17}$$

The purity is related to the entropy concept. In order to compute the purity criterion, each cluster is assigned to its majority class, as described in Equation (18), where v_j is the number of objects in cluster C_j from class i. The purity is the sum of v_j over all clusters, as captured by Equation (19).

$$v_j = \frac{1}{N_j} \operatorname*{argmax}_i (N_j^i) \tag{18}$$

$$Purity = \sum_{j=1}^{k} \frac{N_j}{N} v_j \tag{19}$$

Note that criteria like SSE and purity are usually employed in a sliding window model, which means the clustering partition is obtained with data within the sliding window. However, if the algorithm does not use the sliding window model (*e.g.*, if it employs the concept of representative objects), evaluating a partition created with the most recent objects of the stream may not be a good idea, considering that representative objects summarize both past and present information.

Another important issue to be addressed in an experimental methodology is how to validate partitions generated with non-stationary data. For instance, one needs to verify how the partition has evolved since the last time it was generated.

In this sense, it is not enough to evaluate the quality of the generated partition (spatial criterion), but it is also necessary to evaluate the changes that occur in the partition over time (temporal criterion). Even though the quality of the partition may indicate that changes occurred in the data distribution — *e.g.,* degradation of quality due to a new emerging cluster —, it is not possible to clearly state what is causing the quality degradation. Hence, there is a need of combining spatial and temporal criteria to properly evaluate the quality of partitions and their behavior over the course of the stream.

There are few efforts towards developing more sophisticated evaluation measures for data streams. In [Kremer et al. 2011], the authors propose an external criterion for evaluating clustering algorithms, named CMM (Clustering Mapping Measure), which takes into account the age of objects. The CMM measure is a combination of penalties for each one of the following faults:

(1) Missed objects — clusters that are constantly moving may eventually "lose" objects, and thus CMM penalizes for these missed objects;
(2) Misplaced objects — clusters may eventually overlap over the course of the stream, and thus CMM penalizes for misplaced objects;
(3) Noise inclusion — CMM penalizes for noisy objects being inserted into existing clusters.

The CMM measure can reflect errors related to emerging, splitting, or moving clusters, which are situations inherent to the streaming context. Nevertheless, note that CMM is an external criterion, and thus requires a "gold standard" partition, which is not available in many practical applications.

7. DATA STREAM CLUSTERING IN PRACTICE

Our discussion so far has concentrated on techniques for data stream clustering and analysis of existing algorithms. All these are in vain unless data stream clustering is useful in practice. In this section, we address the applicability of data stream clustering, tabulating some relevant examples of its use in diverse real-world applications. We also briefly discuss existing software packages and data set repositories for helping practitioners and researchers in designing their experiments.

7.1 Applications

Data stream mining is motivated by emerging applications involving massive data sets. Examples of these data include [Guha et al. 2003]: customer click streams, telephone records, large sets of Web pages, multimedia data, financial transactions, and observational science data. Even though there are several interesting and relevant applications for data stream clustering (see Table II), most of the studies still propose evaluating algorithms on synthetic data.

The most notable exception is the public network intrusion data set, known as KDD-CUP '09 [Tavallaee et al. 2009], available at the UCI repository [Frank and Asuncion 2010]. This data set has two weeks of raw TCP dump data for a local area network and simulates an environment with occasional attacks. It is used in several experimental studies in data mining, both for classification and clustering. Due to its large size, it has also been consistently used to assess data stream clustering algorithms (*e.g.*, [Aggarwal et al. 2003; Aggarwal and Yu 2008; Aggarwal 2010]).

Table II. Application areas of data stream clustering.

Application area	References
Bearing prognostics	[Serir et al. 2012]
Charitable donation (KDD '98)	[Aggarwal et al. 2003], [Cao et al. 2006],[Gao et al. 2010]
Forest cover	[Aggarwal et al. 2004b; Tasoulis et al. 2006] [Aggarwal and Yu 2008; Lühr and Lazarescu 2009]
Grid computing	[Zhang et al. 2009; Wang and Wei 2010]
Network intrusion detection (KDD '99)	[O'Callaghan et al. 2002; Cao et al. 2006; Guha et al. 2003] [Aggarwal et al. 2004b; Tasoulis et al. 2006] [Csernel et al. 2006; Aggarwal et al. 2003] [Chen and Tu 2007; Aggarwal and Yu 2008] [Wan et al. 2008; Lühr and Lazarescu 2009] [Zhu et al. 2010; Ackermann et al. 2012] [Wang and Wei 2010; Aggarwal 2010; Li and Tan 2011]
Sensor networks	[Rodrigues et al. 2006; 2008; Gaber et al. 2010] [Silva et al. 2011; Gama et al. 2011]
Stock market analysis	[Kontaki et al. 2008]
Synthetic data	[Zhang et al. 1997; Ong et al. 2004; Chen and Tu 2007] [Guha et al. 2003; Aggarwal et al. 2004b; Dang et al. 2009] [Wan et al. 2008; Aggarwal et al. 2003; Kontaki et al. 2008] [Serir et al. 2012; Cho et al. 2006; Aggarwal and Yu 2006] [O'Callaghan et al. 2002; Cao et al. 2006; Zhu et al. 2010] [Al Aghbari et al. 2012; Lühr and Lazarescu 2009]
Text data	[Aggarwal and Yu 2006; Liu et al. 2008]
VOIP data	[Aggarwal 2010]
Water distribution networks	[Qiong Li and Xie 2011]

Note that some evaluation measures may not be suitable for this data set. For instance, the purity measure should not be employed for evaluating the cluster structures found within KDD-CUP '09 data set because the majority of its objects belong to the same class, resulting in large (and perhaps misleading) values of purity.

Synthetic data sets are usually preferred because testing hypotheses like noise-robustness, and scaling for high-dimensionality are easier to perform with synthetic data. Examples of artificially generated data sets are: (i) data generated by varying Gaussian distributions [Aggarwal et al. 2003; 2004b; Wan et al. 2008; Dang et al. 2009]; (ii) data generated by the IBM synthetic data generator [Ong et al. 2004]; (iii) data simulating a taxi location tracking application [Cho et al. 2006]; and (iv) data sets formed by arbitrarily-shaped clusters, like those presented in Figure 12.

In [Serir et al. 2012], the authors propose to group data streams from bearing prognostics. They employ a platform developed within the Department of Automatic Control and Micro-Mechatronic Systems of the FEMTO-ST institute to generate data concerning the test and validation of bearing prognostics approaches. The platform is able to characterize both the ball bearing functioning and its degradation along its whole operational life (until fault/failure). Vibration and temperature measurements of the rolling bearing during its functioning mode are collected by different sensors.

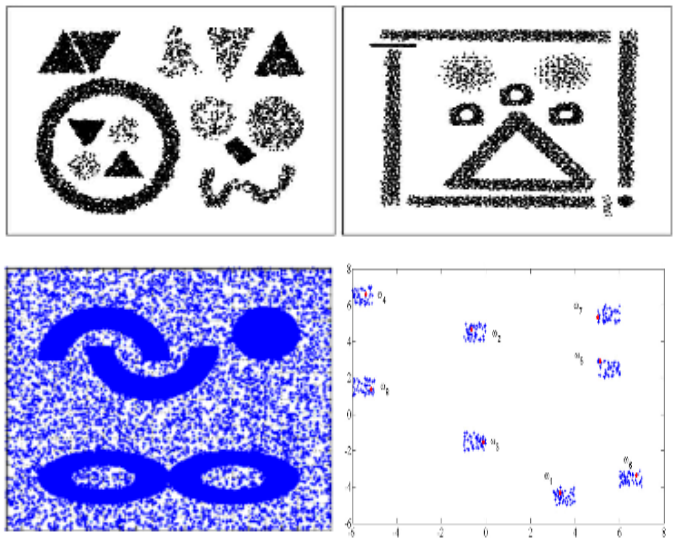

Fig. 12. Arbitrarily-shaped synthetic data sets — adapted from [Lühr and Lazarescu 2009; Chen and Tu 2007; Serir et al. 2012].

A data set commonly exploited by the data stream clustering research community is the charitable donation data set (KDD-CUP '98) [Aggarwal et al. 2003; Cao et al. 2006; Gao et al. 2010], which contains records of information about people who have made charitable donations in response to direct mailing requests. In this kind of application, one possible clustering application is grouping donors that show similar donation behavior.

Yet another commonly exploited data set in data stream clustering is the forest cover type data set [Aggarwal et al. 2004b; Tasoulis et al. 2006; Aggarwal and Yu 2008; Lühr and Lazarescu 2009]. It can be obtained from the UCI machine learning repository website[3]. This data set contains a total of 581,012 observations with 54 attributes (10 quantitative, 4 binary values for wilderness areas and 40 binary soil type). Each observation is labeled as one of seven forest cover types.

In [Zhang et al. 2009; Wang and Wei 2010], the authors propose clustering data streams from real-time grid monitoring. In order to diagnose the EGEE grid (Enabling Grid for E-SciencE[4]), they exploited the gLite reports on the lifecycle of the jobs and on the behavior of the middleware components for providing the summarized information of grid running status.

Clustering data streams collected by sensor networks [Rodrigues et al. 2006; 2008; Silva et al. 2011; Gama et al. 2011] is another typical application. Sensor networks may be responsible, *e.g.*, for measuring electric power consumption in a given city. Electricity distribution companies usually set their management operators on SCADA/DMS products (Supervisory Control and Data Acquisition / Distribution Management Systems). In this context, data is collected from a set

[3] `http://www.ics.uci.edu/~mlearn`.

[4] `http://www.eu-egee.org/`, the largest grid infrastructure in the world.

of sensors distributed all around the network. Sensors can send information at different time scales, speed, and granularity. Data continuously flow eventually at high-speed, in a dynamic and time-changing environment. Clustering of the time series generated by each sensor is one of the learning tasks required in this scenario, considering that it allows the identification of consumption profiles, and the identification of urban, rural, and industrial consumers. Clustering this kind of information can help to understand patterns of electrical demand over different periods of the day.

In [Kontaki et al. 2008], the authors evaluate their method in a stock prices data set. This data set has 500 time series, with a maximum length of 3,000 objects, collected at `http://finance.yahoo.com`. Clustering stock prices may provide insights on the evolution of stocks over time, and may help deciding when it is the right time for buying or selling stocks.

Clustering documents is also a relevant application area. In [Aggarwal and Yu 2006], the authors utilize a number of documents obtained from a 1996 scan of the *Yahoo!* taxonomy, and a stream was synthetically produced from this scan by creating an order that matched a depth-first traversal of the *Yahoo!* hierarchy. Considering that web pages at a given node in the hierarchy are crawled at once, the web pages are also contiguous by their particular class, as defined by the *Yahoo!* labels. In [Liu et al. 2008], a corpus of 20,000 documents [Zhang et al. 2006] is employed for evaluating the proposed algorithm. Each document was randomly assigned a timestamp, and three different text data streams with different document sequences were created. Clustering text data streams is a useful tool with many applications, such as news group filtering, text crawling, document organization and topic detection.

In [Aggarwal 2010], a VOIP system that generates network packets in compressed G729 format is used. Each network packet contains a snapshot of the voice signal at a 10-ms interval. Each record contains 15 attributes corresponding to several characteristics of the speech, vocal tract model, pitch and excitation. The data set contains voice packets from six speakers and the clustering objective would be grouping packets from the same speaker together.

Finally, in [Qiong Li and Xie 2011], the authors monitor water distribution networks. Considering that evaluating the drinking water quality is a typical large-scale real-time monitoring application, the authors performed experiments with two distribution networks of different scales. The first network is a real water distribution system with 129 nodes [Ostfeld et al. 2008]. The second network, with 920 nodes, comes from the Centre for Water Systems at the University of Exeter[5].

7.2 Data Repositories

We highlight the following public data repositories that may be of interest for researchers and practitioners of data stream clustering:

(1) UCI Knowledge Discovery in Databases Archive — online repository of large data sets which encompasses a wide variety of data types, analysis tasks, and application areas. Available at `http://kdd.ics.uci.edu/`.

[5]Center for Water System at University of Exeter, `http://centres.exeter.ac.uk/cws`.

(2) KDD Cup Center — annual Data Mining and Knowledge Discovery competition organized by ACM Special Interest Group on Knowledge Discovery and Data Mining. Available at `http://www.sigkdd.org/kddcup/`.
(3) UCR Time-Series Datasets — maintained by Eamonn Keogh, University California at Riverside, US. Available at `http://www.cs.ucr.edu/~eamonn/time_series_data`.

7.3 Software Packages

As we have presented in this paper, several data stream clustering algorithms were proposed in the specialized literature. We believe it would be useful if the research community joined forces to develop an unified software environment for implementing new algorithms and evaluation tools for data stream clustering. Recent efforts towards this objective include the following publicly available software packages:

(1) MOA [Bifet et al. 2010] — java-based software package that contains state-of-the-art algorithms and measures for both data stream classification and clustering. It also embodies several evaluation criteria and visualization tools. Available at `http://moa.cs.waikato.ac.nz`.
(2) Rapid-Miner [Mierswa et al. 2006] — java-based data mining software package that contains a plugin for data stream processing. Available at `http://rapid-i.com/`.
(3) VFML [Hulten and Domingos 2003] — C-based software package for mining high-speed data streams and very large data sets. Available at `http://www.cs.washington.edu/dm/vfml/`.

8. CHALLENGES AND FUTURE DIRECTION

Probably the greatest challenge in data stream clustering is building algorithms without introducing *ad-hoc* critical parameters, such as: i) the expected number of clusters or the expected density of clusters; ii) the window length, whose size controls the trade-off between quality and efficiency; and iii) the fading factor of clusters or objects, which gives more importance to the most recent objects. To address (i), there are a few recent studies that propose methods to automatically estimate the number of clusters in k-means based stream clustering algorithms [Andrade and Hruschka 2011; Faria et al. 2012]. Algorithms that assume a fixed number of clusters generate partitions that do not adapt over time, which is specially problematic when dealing with non-stationary distributions.

Another challenge that should be handled by data stream clustering algorithms is the ability of properly dealing with outliers, and also of detecting changes in the data distribution. The dynamic nature of evolving data streams, where new clusters often emerge while old clusters fade out, imposes difficulties for outlier detection. In general, new algorithms should provide mechanisms to distinguish between seeds of new clusters and outliers. Regarding the challenge of dealing with *non-stationary* distributions, the current — and naive — strategy employed by most available algorithms is to implicitly deal with them through *window models*. Even though more robust *change detection* mechanisms have been implemented in generic frameworks, we believe future data stream clustering algorithms should explicitly provide mechanisms for performing *change detection*.

Dealing with different data types imposes another challenge in data stream clustering. Different data types such as categorical and ordinal values are present within several application domains. In addition, complex data structures like DNA data and XML patterns are largely available, thus a more careful attention should be given to algorithms capable of dealing with different data types. **For instance, algorithms such as those described in [Charikar and Guha 1999; Meyerson 2001; Guha et al. 2003; Charikar et al. 2003; Guha 2009] can deal with complex data, since that the distance between objects is a metric (not just Euclidean distance).**

Considering that the number of mobile applications grows every year, as well as the volume of data generated by these devices, we believe that clustering data streams produced by mobile devices will constitute an interesting application in years to come.

Another interesting future application of data stream clustering is social network analysis. The activities of social network members can be regarded as a data stream, and a clustering algorithm can be used to show similarities among members, and how these similar profiles (clusters) evolve over time. Social network stream clustering may support services such as intelligent advertisement and custom-made content. Finally, applications involving real-time distributed systems should also deserve particular attention from upcoming data stream clustering algorithms.

Bearing in mind that clustering data streams is a relevant and challenging task, we believe that much effort should be addressed to developing more sophisticate evaluation criteria, high-quality benchmark data, and a sound methodology for reliable experimental comparison of new data stream clustering algorithms.

ELECTRONIC APPENDIX

The electronic appendix for this article can be accessed in the ACM Digital Library by visiting the following URL: `http://www.acm.org/pubs/citations/journals/jacm/20YY-V-N/p1-`.

ACKNOWLEDGMENTS

The authors would like to express their appreciation to the anonymous referees of the original manuscript for their constructive comments.

REFERENCES

Ackermann, M. R., Märtens, M., Raupach, C., Swierkot, K., Lammersen, C., and Sohler, C. 2012. Streamkm++: A clustering algorithm for data streams. *ACM Journal of Experimental Algorithmics 17,* 1.

Agarwal, P. K., Har-Peled, S., and Varadarajan, K. R. 2004. Approximating extent measures of points. *Journal of the ACM 51,* 4, 606–635.

Aggarwal, C. 2003. A Framework for Diagnosing Changes in Evolving Data Streams. In *ACM SIGMOD Conference*. 575–586.

Aggarwal, C. 2007. *Data Streams – Models and Algorithms*. Springer.

Aggarwal, C., Han, J., Wang, J., and Yu, P. 2004a. A framework for projected clustering of high dimensional data streams. In *Proceedings of the Thirtieth International Conference on Very Large Data Dases*. 852–863.

Aggarwal, C. and Yu, P. 2006. A framework for clustering massive text and categorical data streams. *Proceedings of the Sixth SIAM International Conference on Data Mining*, 479.

Aggarwal, C. and Yu, P. 2008. A framework for clustering uncertain data streams. In *IEEE 24th International Conference on Data Engineering (ICDE 2008)*. 150–159.

Aggarwal, C. C. 2010. A segment-based framework for modeling and mining data streams. *Knowledge and Information Systems 30,* 1 (Nov.), 1–29.

Aggarwal, C. C., Han, J., Wang, J., and Yu, P. S. 2003. A framework for clustering evolving data streams. In *Proceedings of the 29th Conference on Very Large Data Bases*. 81–92.

Aggarwal, C. C., Han, J., Wang, J., and Yu, P. S. 2004b. A framework for projected clustering of high dimensional data streams. In *Proceedings of the VLDB*. Vol. 30. 852–863.

Al Aghbari, Z., Kamel, I., and Awad, T. 2012. On clustering large number of data streams. *Intelligent Data Analysis 16,* 1, 69–91.

Amini, A., Wah, T. Y., Saybani, M. R., Aghabozorgi, S. R., and Yazdi, S. 2011. A study of density-grid based clustering algorithms on data streams. In *Eighth International Conference on Fuzzy Systems and Knowledge Discovery*. IEEE press, 1652–1656.

Andrade, J. S. and Hruschka, E. R. 2011. Extending k-means-based algorithms for evolving data streams with variable number of clusters. In *Fourth International Conference on Machine Learning and Applications - ICMLA'11*. Vol. 2. 14–19.

Arabie, P. and Hubert, L. J. 1999. *An Overview of Combinatorial Data Analysis*. World Scientific Publishing, Chapter 1.

Arthur, D. and Vassilvitskii, S. 2006. How slow is the k-means method? In *Proceedings of the twenty-second annual symposium on Computational geometry*. SCG '06. ACM, New York, NY, USA, 144–153.

Arthur, D. and Vassilvitskii, S. 2007. k-means++: the advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*. 1027–1035.

Babcock, B., Babu, S., Datar, M., Motwani, R., and Widom, J. 2002. Models and issues in data stream systems. In *PODS '02: Proceedings of the 21th ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM Press, 1–16.

Babcock, B., Datar, M., Motwani, R., and O'Callaghan, L. 2003. Maintaining variance and k-medians over data stream windows. In *Proceedings of the twenty-second ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 234–243.

Bādoiu, M., Har-Peled, S., and Indyk, P. 2002. Approximate clustering via core-sets. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. STOC '02. ACM Press, 250–257.

Ball, G. H. and Hall, D. J. 1965. ISODATA. A novel method of data analysis and pattern classification. Tech. rep., Menlo Park: Stanford Research Institute.

Barbará, D. 2002. Requirements for clustering data streams. *SIGKDD Explorations (Special Issue on Online, Interactive, and Anytime Data Mining) 3*, 23–27.

Bentley, J. L. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM 18,* 9, 509–517.

Bentley, J. L. and Saxe, J. B. 1980. Decomposable searching problems i: Static-to-dynamic transformation. *Journal of Algorithms 1,* 4, 301–358.

Bhat, U. N. and Miller, G. K. 2002. *Elements of Applied Stochastic Processes*, 3rd ed. John Wiley & Sons, Inc., New Jersey.

Bifet, A. 2010. *Adaptive Stream Mining: Pattern Learning and Mining from Evolving Data Streams*. IOS Press.

Bifet, A., Holmes, G., Kirkby, R., and Pfahringer, B. 2010. Moa: Massive online analysis. *Journal of Machine Learning Research 11*, 1601–1604.

Bradley, P. S. and Fayyad, U. M. 1998. Refining initial points for k-means clustering. In *Proceedings of the Fifteenth International Conference on Machine Learning*. ICML'98. Morgan Kaufmann Publishers Inc., 91–99.

Bradley, P. S., Fayyad, U. M., and Reina, C. 1998. Scaling clustering algorithms to large databases. In *Proceedings of Knowledge Discovery and Data Mining*. AAAI Press, 9–15.

Cao, F., Ester, M., Qian, W., and Zhou, A. 2006. Density-based clustering over an evolving data stream with noise. In *Proceedings of the Sixth SIAM International Conference on Data Mining*. SIAM, 328–339.

Charikar, M. and Guha, S. 1999. Improved combinatorial algorithms for the facility location and k-median problems. In *40th Annual Symposium on Foundations of Computer Science*. IEEE Computer Society, 378–388.

Charikar, M., O'Callaghan, L., and Panigrahy, R. 2003. Better streaming algorithms for clustering problems. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing*. ACM, 30–39.

Chen, Y. and Tu, L. 2007. Density-based clustering for real-time stream data. In *KDD '07: Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM Press, 133–142.

Cho, K., Jo, S., Jang, H., Kim, S., and Song, J. 2006. DCF: An Efficient Data Stream Clustering Framework for Streaming Applications. *Database and Expert Systems Applications*, 114–122.

Csernel, B., Clerot, F., and Hébrail, G. 2006. Datastream clustering over tilted windows through sampling. In *Knowledge Discovery from Data Streams Workshop (ECML/PKDD)*.

Dang, X. H., Lee, V. C. S., Ng, W. K., Ciptadi, A., and Ong, K.-L. 2009. An EM-based algorithm for clustering data streams in sliding windows. In *Proceedings of the 14th International Conference on Database Systems for Advanced Applications*. Lecture Notes in Computer Science. Springer, 230–235.

Domingos, P. and Hulten, G. 2001. A general method for scaling up machine learning algorithms and its application to clustering. In *Proceedings of the 8th International Conference on Machine Learning*. Morgan Kaufmann, 106–113.

Dunham, M. H., Meng, Y., and Huang, J. 2004. Extensible markov model. In *Fourth IEEE International Conference on Data Mining*. ICDM '04. 371–374.

Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *2nd International Conference on Knowledge Discovery and Data Mining*. 226–231.

Faria, E. R., Barros, R. C., Carvalho, A. C. P. L. F., and Gama, J. 2012. Improving the offline clustering stage of data stream algorithms in scenarios with variable number of clusters. In *27th ACM Symposium On Applied Computing - SAC'12*. 572–573.

Farnstrom, F., Lewis, J., and Elkan, C. 2000. Scalability for clustering algorithms revisited. *SIGKDD Exploration*, 51–57.

Fayyad, U. M., Piatetsky-Shapiro, G., and Smyth, P. 1996. From data mining to knowledge discovery: an overview. In *Advances in knowledge discovery and data mining*. American Association for Artificial Intelligence, Menlo Park, CA, USA.

Frank, A. and Asuncion, A. 2010. UCI machine learning repository.

Gaber, M., Vatsavai, R., Omitaomu, O., Gama, J., Chawla, N., and Ganguly, A. 2010. *Knowledge Discovery from Sensor Data*. Springer.

Gama, J. 2010. *Knowledge Discovery from Data Streams*. Chapman Hall/CRC.

Gama, J. and Gaber, M. M. 2007. *Learning from Data Streams: Processing Techniques in Sensor Networks*. Springer.

Gama, J., Medas, P., Castillo, G., and Rodrigues, P. P. 2004. Learning with Drift Detection. In *Proceedings of the 17th Brazilian Symposium on Artificial Intelligence (SBIA 2004)*. Vol. 3171. 286–295.

Gama, J. and Pinto, C. 2006. Discretization from data streams: applications to histograms and data mining. In *ACM symposium on Applied computing (SAC '06)*. 662–667.

Gama, J., Rodrigues, P. P., and Lopes, L. 2011. Clustering distributed sensor data streams using local processing and reduced communication. *Intelligent Data Analysis 15,* 1, 3–28.

Gan, G., Ma, C., and Wu, J. 2007. *Data Clustering: Theory, Algorithms, and Applications (ASA-SIAM Series on Statistics and Applied Probability)*. SIAM.

Gao, M.-M., Liu, J.-Z., and Gao, X.-X. 2010. Application of Compound Gaussian Mixture Model clustering in the data stream. In *International Conference on Computer Application and System Modeling (ICCASM)*.

Gonzalez, T. F. 1985. Clustering to minimize the maximum intercluster distance. *Theorical Computer Science 38*, 293–306.

Guha, Meyerson, Mishra, Motwani, and O'Callaghan. 2003. Clustering data streams: Theory and practice. *IEEE Transactions on Knowledge and Data Engineering 15*, 515–528.

Guha, S. 2009. Tight results for clustering and summarizing data streams. In *Proceedings of the 12th International Conference on Database Theory*. ICDT '09. ACM, New York, NY, USA, 268–275.

Guha, S., Mishra, N., Motwani, R., and O'Callaghan, L. 2000. Clustering data streams. In *IEEE Symposium on Foundations of Computer Science*. IEEE Computer Society, 359–366.

Hahsler, M. and Dunham, M. H. 2010. rEMM: Extensible markov model for data stream clustering in R. *Journal of Statistical Software 35,* 5, 1–31.

Hahsler, M. and Dunham, M. H. 2011. Temporal Structure Learning for Clustering Massive Data Streams in Real-Time. In *SIAM Conference on Data Mining*. SIAM / Omnipress, 664–675.

Han, J. and Kamber, M. 2000. *Data Mining: Concepts and Techniques*. Morgan Kaufmann.

Har-Peled, S. and Mazumdar, S. 2004. On coresets for k-means and k-median clustering. In *36th Annual ACM symposium on Theory of computing*. 291–300.

Hulten, G. and Domingos, P. 2003. VFML – a toolkit for mining high-speed time-changing data streams. Tech. rep., University of Washington.

Isaksson, C., Dunham, M. H., and Hahsler, M. 2012. Sostream: Self organizing density-based clustering over data stream. Lecture Notes in Computer Science, vol. 7376. Springer, 264–278.

Jain, A. K. 2009. Data clustering: 50 years beyond k-means. *Pattern Recognition Letters 31*, 651–666.

Jiang, N. and Gruenwald, L. 2006. Research issues in data stream association rule mining. *SIGMOD Record 35,* 1, 14–19.

Kaufman, L. and Rousseeuw, P. 1990. *Finding Groups in Data An Introduction to Cluster Analysis*. Wiley Interscience.

Kavitha, V. and Punithavalli, M. 2010. Clustering time series data stream - a literature survey. *International Journal of Computer Science and Information Security 8,* 1, 289–294.

Khalilian, M. and Mustapha, N. 2010. Data Stream Clustering: Challenges and Issues. In *Proceedings of International Multi Conference of Engineers and Computer Scientists*. 566–569.

Kogan, J. 2007. *Introduction to Clustering Large and High-Dimensional Data*. Cambridge University Press.

Kontaki, M., Papadopoulos, A., and Manolopoulos, Y. 2008. Continuous trend-based clustering in data streams. *Data Warehousing and Knowledge Discovery*, 251–262.

Kranen, P., Assent, I., Baldauf, C., and Seidl, T. 2011. The clustree: indexing micro-clusters for anytime stream mining. *Knowledge and Information Systems 29,* 2, 249–272.

Kremer, H., Kranen, P., Jansen, T., Seidl, T., Bifet, A., Holmes, G., and Pfahringer, B. 2011. An effective evaluation measure for clustering on evolving data streams. In *Proceedings of the 17th ACM SIGKDD international conference on Knowledge discovery and data mining*. KDD '11. ACM, New York, NY, USA, 868–876.

Li, Y. and Tan, B. H. 2011. Data Stream Clustering Algorithm Based on Affinity Propagation and Density. *Advanced Materials Research 267*, 444–449.

Liu, Y.-B., Cai, J., Yin, J., and Fu, A. 2008. Clustering text data streams. *Journal of Computer Science and Technology 23,* 1, 112–128.

Lloyd, S. 1982. Least squares quantization in pcm. *IEEE Transactions on Information Theory 28,* 2, 129–137.

Lühr, S. and Lazarescu, M. 2009. Incremental clustering of dynamic data streams using connectivity based representative points. *Data Knowledge Engineering 68*, 1–27.

MacQueen, J. B. 1967. Some Methods for Classification and Analysis of MultiVariate Observations. In *5th Berkeley Symposium on Mathematical Statistics and Probability*, L. M. L. Cam and J. Neyman, Eds. Vol. 1. 281–297.

Maesschalck, R., Jouan-Rimbaud, D., and Massart, D. 2000. The mahalanobis distance. *Chemometrics and Intelligent Laboratory Systems 50*, 1 – 18.

Mahdiraji, A. R. 2009. Clustering data stream: A survey of algorithms. *International Journal of Knowledge-Based and Intelligent Engineering Systems*, 39–44.

Markov, A. 1971. Extension of the Limit Theorems of Probability Theory to a Sum of Variables Connected in a Chain. In *Dynamic Probabilistic Systems (Volume I: Markov Models)*, R. Howard, Ed. John Wiley & Sons, Inc., Chapter Appendix B, 552–577.

Metwally, A., Agrawal, D., and El Abbadi, A. 2005. Duplicate detection in click streams. In *Proceedings of the 14th international conference on World Wide Web*. ACM, 12–21.

Meyerson, A. 2001. Online facility location. *Foundations of Computer Science, Annual IEEE Symposium on*, 426–431.

Mierswa, I., Wurst, M., Klinkenberg, R., Scholz, M., and Euler, T. 2006. Yale: Rapid prototyping for complex data mining tasks. In *KDD '06: Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*, L. Ungar, M. Craven, D. Gunopulos, and T. Eliassi-Rad, Eds. ACM, New York, NY, USA, 935–940.

O'Callaghan, L., Mishra, N., Meyerson, A., Guha, S., and Motwani, R. 2002. Streaming-data algorithms for high-quality clustering. In *18th International Conference on Data Engineering*. 685–694.

Oliveira, M. and Gama, J. 2010. MEC –Monitoring Clusters' Transitions. In *Proceedings of the Fifth Starting AI Researchers' Symposium*. IOS Press, 212–224.

Oliveira, M. D. B. and Gama, J. 2012. A framework to monitor clusters evolution applied to economy and finance problems. *Intell. Data Anal. 16,* 1, 93–111.

Ong, K., Li, W., Ng, W., and Lim, E.-P. 2004. SCLOPE: an algorithm for clustering data streams of categorical attributes. *Data Warehousing and Knowledge Discovery*, 209–218.

Ostfeld, A., Uber, J., Salomons, E., Berry, J., Hart, W., Phillips, C., Watson, J., Dorini, G., Jonkergouw, P., and Kapelan, Z. 2008. The battle of the water sensor networks (BWSN): A design challenge for engineers and algorithms. *Journal of Water Resources Planning and Management 134*, 556.

Park, N. H. and Lee, W. S. 2007. Cell trees: An adaptive synopsis structure for clustering multi-dimensional on-line data streams. *Data and Knowledge Engineering 63,* 2, 528–549.

Qiong Li, X. M. S. T. and Xie, S. 2011. Continuously Identifying Representatives Out of Massive Streams. In *Advanced Data Mining and Applications*. Springer, 1–14.

Ren, J. and Ma, R. 2009. Density-based data streams clustering over sliding windows. In *Sixth International Conference on Fuzzy Systems and Knowledge Discovery*. Vol. 5. 248 –252.

Rodrigues, P., Gama, J., and Pedroso, J. 2006. ODAC: Hierarchical clustering of time series data streams. In *Proceedings of the Sixth SIAM International Conference on Data Mining*. 499–503.

Rodrigues, P., Gama, J., and Pedroso, J. 2008. Hierarchical clustering of time-series data streams. *Knowledge and Data Engineering, IEEE Transactions on 20,* 5 (may), 615 –627.

Serir, L., Ramasso, E., and Zerhouni, N. 2012. Evidential evolving Gustafson–Kessel algorithm for online data streams partitioning using belief function theory. *International Journal of Approximate Reasoning In press.*, 1–22.

Shah, R., Krishnaswamy, S., and Gaber, M. M. 2005. Resource-aware very fast k-means for ubiquitous data stream mining. In *2nd International Workshop on Knowledge Discovery in Data Streams, 16th European Conference on Machine Learning (ECML'05)*.

Silva, A., Chiky, R., and Hébrail, G. 2011. A clustering approach for sampling data streams in sensor networks. *Knowledge and Information Systems*.

Spiliopoulou, M., Ntoutsi, I., Theodoridis, Y., and Schult, R. 2006. Monic: modeling and monitoring cluster transitions. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. KDD '06. ACM, 706–711.

Steinhaus, H. 1956. Sur la division des corp materiels en parties. *Bull. Acad. Polon. Sci 1*, 801–804.

Tasoulis, D., Adams, N., and Hand, D. 2006. Unsupervised clustering in streaming data. *IEEE International Workshop on Mining Evolving and Streaming Data. Sixth IEEE International Conference on Data Mining (ICDM 2006)*, 638–642.

Tavallaee, M., Bagheri, E., Lu, W., and Ghorbani, A. A. 2009. A detailed analysis of the kdd cup 99 data set. In *2nd IEEE International Conference on Computational Intelligence for Security and Defense Applications*. 53–58.

Vattani, A. 2009. k-means requires exponentially many iterations even in the plane. In *Proceedings of the 25th annual symposium on Computational geometry*. SCG '09. ACM, New York, NY, USA, 324–332.

Wan, R., Yan, X., and Su, X. 2008. A weighted fuzzy clustering algorithm for data stream. In *ISECS International Colloquium on Computing, Communication, Control, and Management*. 360–364.

Wang, X. Z. and Wei. 2010. Self-adaptive Change Detection in Streaming Data with Non-stationary Distribution. In *Advanced Data Mining and Applications*. Springer, 1–12.

Wu, X., Kumar, V., Ross Quinlan, J., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G. J., Ng, A., Liu, B., Yu, P. S., Zhou, Z.-H., Steinbach, M., Hand, D. J., and Steinberg, D. 2007. Top 10 algorithms in data mining. *Knowledge and Information Systems 14*, 1–37.

Xu, R. and Wunsch, D. 2009. *Clustering (IEEE Press Series on Computational Intelligence)*. Wiley-IEEE Press.

Yang, C. and Zhou, J. 2006. HClustream: A novel approach for clustering evolving heterogeneous data stream. In *Sixth IEEE International Conference on Data Mining*. IEEE Press, 682–688.

Zhang, T., Ramakrishnan, R., and Livny, M. 1996. BIRCH: An Efficient Data Clustering Method for Very Large Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM Press, 103–114.

Zhang, T., Ramakrishnan, R., and Livny, M. 1997. BIRCH: A new data clustering algorithm and its applications. *Data Mining and Knowledge Discovery 1,* 2, 141–182.

Zhang, X., Sebag, M., and Germain-Renaud, C. 2009. Multi-scale Real-Time Grid Monitoring with Job Stream Mining. In *9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGRID '09)*. 420–427.

Zhang, X., Zhou, X., and Hu, X. 2006. Semantic smoothing for model-based document clustering. In *Sixth International Conference on Data Mining (ICDM '06)*. IEEE, 1193–1198.

Zhou, A., Cao, F., Qian, W., and Jin, C. 2008. Tracking clusters in evolving data streams over sliding windows. *Knowledge and Information Systems 15,* 2, 181–214.

Zhu, H., Wang, Y., and Yu, Z. 2010. Clustering of Evolving Data Stream with Multiple Adaptive Sliding Window. In *International Conference on Data Storage and Data Engineering (DSDE)*. 95–100.

Zhu, Y. and Shasha, D. 2002. StatStream: statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th International Conference on Very Large Data Bases*. VLDB Endowment, 358–369.

This document is the online-only appendix to:

Data Stream Clustering: A Survey

JONATHAN A. SILVA, ELAINE R. FARIA, RODRIGO C. BARROS,
EDUARDO R. HRUSCHKA, and ANDRE C. P. L. F. DE CARVALHO
University of São Paulo
and
JOÃO GAMA
University of Porto

Journal of the ACM, Vol. V, No. N, Month 20YY, Pages 1–App–3.

In this appendix, we analyze the computational complexity of the data stream clustering algorithms regarding their processing time, *i.e.*, our focus will be on time complexity.

We start by analyzing the time complexity of the pioneering *BIRCH* algorithm [Zhang et al. 1996; 1997]. It employs a B+ tree to store the summarized statistics. The cost for (re)inserting an object in a B+ tree is $O(n \times B \times H)$, where B is the number of entries in an inner node and H is the maximum height of the tree. Recall that n is the data dimensionality.

In the *ClusTree* algorithm, the process of inserting a new object in a R-Tree takes $O(log(q))$, where q is the number of **CF**s. The merge process requires $O(M^2)$, where M is the number of entries in a leaf node. The clustering result can be obtained by applying any clustering algorithm, such as k-means or *DBSCAN*, over the **CF**s stored in the leaf nodes.

As previously seen, *Scalable k-means* [Bradley et al. 1998] adopts the divide-and-conquer strategy for processing chunks of the stream. Each chunk holds m n-dimensional objects that are clustered by k-means into k clusters during the primary compression step, taking $O(m \times n \times k \times v)$ time, where v is the number of k-means iterations. There are papers that provide theoretical upper bounds on its running time [Vattani 2009; Arthur and Vassilvitskii 2006], which can be exponential even for low-dimensional data. However, we note that, in practice, usually the number of iterations is fixed a priori or a convergence criterion — such as those based on the difference between centroids in two consecutive iterations — is adopted. Either way, it is reasonable to assume that the number of iterations is usually much less than the number of objects.

In order to identify data objects that will be discarded, the *Scalable k-means* algorithm uses a version of the Mahalanobis distance [Maesschalck et al. 2000] — known as normalized Euclidean distance, which allows simplifying the covariance

matrix inversion — is employed, taking $O(m \times n)$ time. Finding the discard radius for all k clusters takes $O(m \times \log m)$ if sorting is used. The total time complexity for the first method of primary compression is $O(m(n+\log m))$, whereas the second method of primary compression (cluster means perturbation) takes $O(m \times k \times n)$. In the secondary compression step, all objects that did not meet the requirements for being clustered in the first step (the worst case is m objects) are grouped into $k_2 > k$ clusters, thus taking $O(m \times n \times k_2 \times v)$. The k_2 clusters are checked again to verify if their covariances are bounded by the threshold β. The clusters that do not meet this criterion are merged with the k clusters obtained in the first compression step by using a hierarchical agglomerative clustering, whose time complexity is $O(k_2^2 \times n)$. The algorithm total time complexity is therefore $O(m \times n \times k \times v) + O(m(n+\log m)) + O(m \times k \times n) + O(m \times n \times k_2 \times v) + O(k_2^2 \times n)$. *Single-pass k-means* [Farnstrom et al. 2000], which is a simplified version of the *Scalable k-means* algorithm with no compression steps, takes $O(m \times k \times n)$.

For the online phase of *CluStream* [Aggarwal et al. 2003], the cost of creating micro-clusters is $O(q \times N_{first} \times n \times v)$, where q is the number of micro-clusters and N_{first} is the number of objects used to create the initial micro-clusters. Next, for each new data object that arrives, updating micro-clusters requires three steps: i) find the closest micro-cluster, which takes $O(q \times n)$ time; ii) possibly discarding the oldest micro-cluster, also taking $O(q)$ time; and iii) possibly merging the closest micro-clusters, which takes $O(q^2 \times n)$. The offline step runs the traditional k-means algorithm over the micro-clusters, which takes $O(q \times n \times k \times v)$ time.

Similary to *CluStream*, the *SWClustering* algorithm has time complexity of $O(q \times n)$ to insert a new object into the nearest **EHCF**, where q is the number of **EHCF**s. In addition, it takes $O(q^2 \times n)$ to merge the two nearest **EHCF**s, $O(q)$ to update the **EHCF** containing expired objects, and $O(q \times n \times k \times v)$ to cluster the **EHCF**s using the k-means algorithm. Unlike *CluStream*, *SWClustering* does not require the creation of an initial summary structure.

DenStream [Cao et al. 2006] is a density-based algorithm that is also executed in two phases. **In the online phase, the cost for creating micro-clusters through DBSCAN [Ester et al. 1996] is $O(N_{first}^2)$[6].** For each new object of the stream, one of the q *p-micro-clusters* is updated according to the following three steps: i) finding the closest *p-micro-cluster*, which takes $O(q \times n)$; (ii) possibly finding the closest *o-micro-cluster*, which also takes $O(q \times n)$; and iii) possibly verifying if an *o-micro-cluster* is a potential *p-micro-cluster*, which once again takes $O(q \times n)$. Hence, the cost for updating micro-clusters for each new object is $O(q \times n)$. The periodical analysis of outdated *p-micro-clusters* also takes $O(q \times n)$, whereas the offline phase that employs the DBSCAN algorithm over the q *p-micro-clusters* takes $O(q \times \log q)$.

accelerating index structure,

StreamKM++ [Ackermann et al. 2012] maintains a summary of the stream using the merge-and-reduce technique [Har-Peled and Mazumdar 2004; Agarwal et al. 2004; Bentley and Saxe 1980]. Basically, it maintains $\log \frac{N}{m}$ buckets in main memory, where each bucket i stores only m objects. Each bucket represents $2^i \times m$

[6]Accelerating index structure such as KD-trees [Bentley 1975] can be used to reduce the time complexity.

objects from stream. In [Ackermann et al. 2012], the merge-and-reduce technique implements a **coreset tree** that organizes the data in a binary tree (reduce step). The time complexity for a single operation of reduce (from $2m$ to m objects) is $O(m^2 \times n)$ (or $O(m \times n \times \log m)$ to a balanced coreset tree). **In order to obtain a coreset tree with m objects for the union of all buckets, the merge-and-reduce of all buckets is executed in $O(m^2 \times n \times \log \frac{N}{m})$. After having processed the whole input stream, the $k-means++$ algorithm find k clusters on the m points (obtained from the merge-and-reduce of all buckets) with time complexity $O(m \times k \times n \times v)$.**

Stream [Guha et al. 2000] runs the*Facility Location* algorithm [Charikar and Guha 1999; Meyerson 2001] over chunks with B objects to find $2k$ clusters, which takes $O(B \times n \times k)$ to execute. Similarly, *Stream LSearch* [O'Callaghan et al. 2002] also assumes that a chunk of B objects will be clustered into $2k$ clusters, though this time the clustering algorithm employed is *LSearch*. First, *LSearch* creates an initial solution with k' clusters, where the value of k' depends on the properties of the data set but is usually small [O'Callaghan et al. 2002], which takes $O(B \times n \times k')$. Next, a binary search is performed to partition the B objects into k clusters, which takes $O(B \times n \times k \times \log k)$. Hence, the overall time complexity of *Stream LSearch* is $O(B \times n \times k' + B \times n \times k \times \log k)$.

DGClust [Rodrigues et al. 2008; Gama et al. 2011] operates both in local and central sites. Considering the time complexity of a local site, after the bins for the two layers of discretization have been created, the cost of inserting a new object is $O(\log p_i)$, where p_i is the number of bins of the first layer, *i.e.*, it is the cost of searching the proper bin for inserting the object. Since each local site operates in a parallel fashion, the time complexity for the local sites is $O(\log p)$, where $p = \max_i(p_i)$ and $i \in 1, 2, .., n$. Recall that n is the number of dimensions of the stream, and thus the number of local sites for *DGClust*. The central site keeps the top-m most frequent global states. Searching a state in the list of states takes $O(m)$. Clustering the top-m most frequent global states takes $O(m \times n \times k)$.

D-Stream [Chen and Tu 2007] is a grid-based stream clustering algorithm. In the worst-case scenario, at each iteration of *D-Stream* there are p^n grid cells in memory, where p is the number of partitions in each dimension. However, as already noted in [Chen and Tu 2007; Amini et al. 2011], although in theory the number of possible grid cells grows exponentially with n, empty or infrequent grid cells can be discarded, under the assumption that the data space is sparse.