Received September 2, 2018, accepted October 14, 2018, date of publication October 22, 2018,
date of current version November 19, 2018.

Digital Object Identifier 10.1109/ACCESS.2018.2877138

Efficient Data Stream Clustering With Sliding Windows Based on Locality-Sensitive Hashing

JONGHEM YOUN[1], JUNHO SHIM[2], (Senior Member, IEEE), AND SANG-GOO LEE[3], (Member, IEEE)

[1]Voost Inc., Seoul 06232, South Korea
[2]Department of Computer Science, Sookmyung Women's University, Seoul 04310, South Korea
[3]Department of Computer Science and Engineering, Seoul National University, Seoul 08826, South Korea

Corresponding author: Junho Shim (jshim@sookmyung.ac.kr)

This work was supported in part by the National Research Foundation of Korea (NRF) funded by the Ministry of Science, ICT and Future Planning through the Basic Science Research Program under Grant 2017R1E1A1A03070004 and in part by NRF funded by the Ministry of Science, ICT and Future Planning through the PF Class Heterogeneous High Performance Computer Development under Grant NRF-2016M3C4A7952587.

ABSTRACT Data stream clustering over sliding windows generates clusters as the window moves. However, iterative clustering using all data in a window is highly inefficient in terms of memory use and computational load. In this paper, we improve data stream clustering over sliding windows using sliding window aggregation and nearest neighbor search techniques. Our algorithm constructs and maintains temporal group features as a summary of the window using the sliding window aggregation technique. In order to maintain a constant size for the summary, the algorithm reduces the size of the summary by joining the nearest neighbor. We exploit locality-sensitive hashing for rapid nearest neighbor searching. In addition, we also suggest a re-clustering policy that determines whether to append a new summary to pre-existing clusters or to perform clustering on the whole summary. We conduct experiments on real-world and synthetic datasets in order to demonstrate that our algorithm can significantly improve continuous clustering on data streams with sliding windows.

INDEX TERMS Data stream, k-means clustering, locality-sensitive hashing, sliding window.

I. INTRODUCTION

Large-scale data streams are generated from various applications, including social media, news feeds, sensor networks, transportation monitoring, and smart devices. Data streams are massive, rapidly evolving, and continuously created. Data stream clustering is an important task for analyzing or retrieving information from the overall data streams. Since the data streams are too large, data stream clustering algorithms are designed by considering the computational load and memory use. For example, news articles are continuously published every moment throughout the web. In order to track recent hot topics, articles with similar contents can be grouped by clustering, as shown in Fig. 1. As time passes along with the continuous movement of the sliding window, similar news articles can be grouped by clusters. Since we need to perform clustering whenever the window moves, it is indeed important to determine how to effectively perform clustering in this domain.

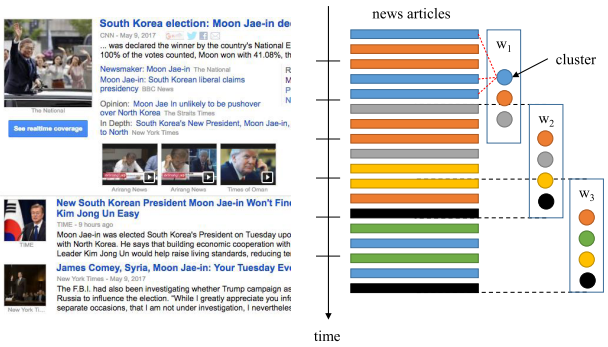

FIGURE 1. Clustering with sliding windows for news articles.

The general procedure for carrying out data stream clustering consists of two steps: grouping and clustering [3], [4], as shown in Fig. 2. The grouping step summarizes the original data into specific data structures, called synopses, in order to reduce memory use. The synopses are then used to grasp the semantics of the original data from the summarization without actually storing the entire data.

In general, the grouping step constructs synopses through particular heuristic methods with linear time complexity. The clustering step performs clustering on the synopses generated

through grouping. Various clustering algorithms, such as k-means [5], [6], k-median [7], [8], DBSCAN [9]–[11], and affinity propagation [12], [13] have been employed to identify the partitions of the synopses. The clustering algorithms that use the synopses are efficient because the synopses are relatively small compared to the entire data. The processing time for the data stream clustering depends primarily on the time required to construct the synopses.

Early studies assumed that clustering is to be performed over entire data streams, and directly applied one-pass clustering algorithms to those data streams [14]. However, data streams evolve continuously over time. In most data stream applications, the most recent tuples are considered to be more decisive and influential. This characteristic caused clustering algorithms with window models to be developed. Window models that are widely used by the algorithms include the landmark window [6], [12], [13], [15] and the damped window [10], [11], [16]. While these two window models are effective in certain applications, they are insufficient for domains requiring the sliding window model.

In the sliding window model, the window contains only tuples with timestamps from a given timestamp in the past up to the current timestamp. As time passes, the window removes the tuples whose timestamps have expired. The exact number of recent tuples is a critical and important factor for implementing the sliding window model, such as for topic extraction in news feeds and real-time traffic monitoring systems.

Clustering with sliding windows should produce results for each window movement, and a seemingly straightforward approach would be to perform repeat clustering. This is impractical and requires a substantial computational cost. Therefore, a clustering algorithm is needed to delete expired data and insert new data to construct the synopses and generate the cluster results.

There are two main problems that must be solved for the algorithm. First, the synopses should support partial deletion for expired data. The synopses require additional data structures for deletion. The data structure should also remain fast with little memory usage. Second, the clusters should be updated without performing clustering. Clusters could not be partially updated because clustering is a holistic function that requires whole tuples to generate results [17]. An approximation method is required to insert and delete data into the clusters.

We present an efficient data stream clustering algorithm with sliding windows. The highlighted features of our algorithm are as follows: First, it constructs fixed-size synopses efficiently based on Locality-Sensitive Hashing [18]. It maintains temporal group features, as synopses, to reduce both space and computation time. The data structure for synopsis allows expired and new data to be handled, so that the algorithm can maintain a precise range of tuples for the window. The LSH is employed to efficiently search for the nearest group feature in the synopses. Previous studies have used a tree-based index structure. Second, the algorithm employs a re-clustering policy for pre-existing clusters in order to avoid clustering each time new data arrives. The clustering operation is expensive because it needs to access all data objects. We measure the difference between the quality of the pre-existing clusters and the one that we could obtain if we reconstructed the clusters. We then perform re-clustering only when the difference is expected to be significantly large. Last, we demonstrate the performance of the proposed algorithm through extensive experimentation as well as a theoretical analysis.

This paper is organized as follows: In Section II, we provide an overview of related studies on data stream clustering algorithms. In Section III, we present background information and the problem statement. In Section IV, we show how coresets for clustering are constructed and maintained. In Section V, we describe an improved algorithm for coreset construction. In Section VI, a clustering methodology is proposed. In Section VII, an analysis of our experimental results is described. Finally, we present our concluding remarks in Section VIII.

II. RELATED WORK

Multiple clustering algorithms for data streams have been developed, and a detailed survey of these algorithms is presented in [19]. Data stream clustering algorithms are categorized into partition-based [3], [6], density-based [10], [16], and message passing-based [12], [13] clustering, and these are developed under the landmark window model [6], [12], [13], damped window model [10], [16], and sliding window model [4], [7], [20], [21]. Table 1 shows the related studies categorized by clustering algorithms and window models. In the following subsections, we present clustering algorithms for each window model, and also explain the pros and cons of each as compared to our algorithm.

A. CLUSTERING WITH LANDMARK WINDOWS

The landmark window model splits data streams into fixed-size, non-overlapping chunks and maintains the tuples that arrive after the landmark; this model is usually used when periodic results are needed (e.g., on a daily or weekly basis). Clustering algorithms for landmark window model take the early studies on data stream clustering, and exploit a method of creating a data summary for clustering. **BIRCH**(Balanced Iterative Reducing and Clustering using Hierarchies) [14] is an initiative algorithm that enables k-means clustering for a large data. In order to summarize a large amount of data, BIRCH first introduces a concept of *clustering feature* (*CF*) which stores a linear sum of the tuples, square sum of the tuples, and the number of the tuples. For a new tuple, a nearest *CF* absorbs the tuple if the radius of the *CF* is smaller than a given threshold θ value. If the radius exceeds that value, a new *CF* based on the tuple is created. Users may adjust the threshold θ appropriately according to the memory capacity. BIRCH runs quickly, but its clustering quality is poor, which has been experimentally shown in [6].

TABLE 1. Clustering algorithms with window models.

Clustering	Window Models		
	Landmark	**Damped**	**Sliding**
Partition-based	BIRCH [14] ScaleKM [22] STREAM [23] LSEARCH [24], [15] CluStream [3] DGClust [25] StreamKM++ [6] Ackerman et al. [26] E2SC [27]		Babcock et al. [7] SWClustering [4] G2CS [28] Braverman et al. [21]
Density-based	Any-OPTICS [29]	DenStream [10] D-Stream [9], [30] MR-Stream [11] SOStream [31] ClusTree [16] DASC [32] DBSTREAM [33] ADStream [34] CEDAS [35]	
Message Passing-based	STRAP [12] IAP [13] SAIC [36]		ID-AP [37]

Since BIRCH, multiple algorithms have been developed to exploit the concept of *CF*. **Scalable k-means** [22] uses *CF* as statistics and suggests selectively retaining some data points that might affect the clustering quality. **STREAM** [23] is a k-median clustering algorithm for data streams with constant-factor approximation in a single pass based on divide-and-conquer strategy. STREAM processes data points in a hierarchical scheme, and provides a theoretical analysis to show how an approximated clustering quality relates to the hierarchy. **LSEARCH** [15], [24] improves STREAM for the k-median problem, by exploiting the concept of local search. The algorithm begins with some initial clusters, then refines those clusters by making local improvements. The authors present how to find a good initial solution that leads to an expected 8-approximation to the optimum. **CluStream** [3] presents the concepts of micro-clusters and pyramidal time frames for clustering data stream. The micro-cluster maintains statistics such as the spatial locality and the temporal features of data. The pyramidal time frame is used to effectively store the snapshots of the micro-clusters. **DGClust** [25] is a monitoring and clustering algorithm for distributed sensor networks. In this algorithm, local sensors do not send the entire data to a central server, but maintain the statistics of data, and the algorithm performs clustering on the statistics. **StreamKM++** [6] is one of the most recently developed algorithms for data stream clustering with landmark windows. It constructs a coreset tree for the synopses based on a *k-means++* [38] seeding procedure. However, the cost of constructing such a tree is relatively high, making it unsuitable for sliding windows.

Recently, algorithms which do not construct summaries have come to be studied. Ackerman and Dasgupta [26] focus on an incremental clustering problem where existing clusters would be modified with the new tuples being inserted. **E2SC** [27] presents an efficient spectral clustering algorithm based on a low rank approximation of the Laplacian matrix. **Any-OPTICS** [29] is a density-based clustering algorithm, and it continuously produces and refines the clusters by using the lower-bound distances which are the approximations of the true distances.

In the massage passing-based clustering algorithms, **STRAP** [12] is a data stream clustering algorithm based on Affinity Propagation (AP) [39]. STRAP combines AP with a statistical test for change detection in data distribution. The clustering is performed whenever a change in data distribution is detected. **IAP** [13] (Incremental Affinity Propagation) proposes two methodologies for incremental clustering: IAP clustering based on K-Medoids (IAPKM) and IAP clustering based on Nearest Neighbor Assignment (IAPNA). **SAIC** [36] proposes a clustering algorithm for dynamic datasets of arbitrary shapes and sizes based on incremental learning.

Some algorithms such as BIRCH have exploited various approaches of constructing the summary for data stream clustering. As long as they are based on the landmark window model, they share a common disadvantage in that all data should be accessed to rebuild the summary when the data is deleted. Our algorithm is different in that it builds the summary in an efficient way so as to support the deletion of expired data.

B. CLUSTERING WITH DAMPED WINDOWS

In the damped window model, also known as the fading window model, the tuples are associated with the weight values that decrease over time. As is the case with the landmark window model, the algorithms with the damped window model are based on the insertion-only model, which assumes

that the tuples would not be removed from the window once they were received, and gives newer tuples higher weight values than older ones. For example, the weight values can decrease exponentially by a decay function $f(t) = 2^{-\lambda t}$ over time t, where $\lambda > 0$. The value of λ adjusts the effect of the past data. If λ increases, the importance of the historical data becomes more reduced.

Algorithms in this category are mainly based on density-based clustering. **DenStream** [10] creates the micro-clusters which are similar to *CF*, and identifies the outlier micro-clusters based on the density of data. It applies a variant of DBSCAN [40] not to the outlier micro-clusters but only to the micro-clusters, which helps the algorithm be robust to the noise data. **D-Stream** [9], [30] maintains the clustering features in the grid units. The decay function is applied to the entire grids at a specific interval, and removes sparse grids as the outliers. The algorithm assumes that the sparse grids do not become dense grids in the future. **MR-Stream** [11] maintains the hierarchically-divided grids within a tree data structure. A node is created with a new data point, and added to a tree, where the weight values are updated beginning from its parent node and iteratively upto the root node. **SOStream** [31] automatically determines a threshold value for the density-based clustering through the use of competitive learning. **ClusTree** [16] maintains the clustering features in an extended index structure based on R-tree. When new data arrives, the algorithm searches for the nearest micro-cluster by traversing the tree of which leaf nodes contain all the micro-clusters.

A disadvantage of a density-based clustering algorithm is that it requires more computation cost than does a partition-based method, thus some scalable versions of density-based clustering have recently been proposed. **DASC** [32] is designed to cluster an incremental large dataset with grid-based settings based on the MapReduce framework. **DBSTREAM** [33] proposes a shared density graph to capture the density between adjacent micro-clusters for efficient reclustering. **ADStream** [34] identifies the initial micro-clusters using the affinity propagation method, then generates the clusters in different time granularities by using a density grid clustering. **CEDAS** [35] proposes the hyper-spherical micro-clusters for high-dimensional data.

The clustering algorithms with damped window model are designed to consider that the influence of past data becomes gradually reduced. However, as is the case with the landmark window model, they should access the entire data when they need to rebuild micro-clusters in order to perform clustering only with unexpired data. This is because some damped window may still contain expired data even though the effect of those data is reduced.

C. CLUSTERING WITH SLIDING WINDOWS

In contrast to the landmark or damped window model, only a small number of studies have focused on clustering algorithms with sliding windows.

Babcock *et al.* [7] presented a technique for maintaining the variance and k-median based on an *exponential histogram(EH)* for the sliding window. Zhou *et al.* [4] focus on the problem of tracking the cluster evolution in the sliding window, and developed **SWClustering**, a k-means clustering algorithm based on an extension of EH, *exponential histogram of clustering features(EHCF)*, which combines a temporal attribute with EH. From a theoretical perspective, Braverman *et al.* [21] proposed a merge-and-reduce technique to transform the coreset construction in the insertion-only streaming model to the sliding window model. The algorithms that exploit EH as a synopsis data structure support both insertion and deletion of the window, but they have a disadvantage in that the window contains expired tuples when the window size increases [4], [7].

In their works, EH is defined as a collection of buckets on a set of tuples. Due to the memory limitation, if the number of buckets exceeds a user defined number, the buckets are merged, with each merged bucket holding a number of tuples equal to or double that held in the previous unmerged buckets. For example, let us say that the input tuples are $x_1, x_2, \ldots$ (x_{i+1} is newer than x_i), and the current state of the buckets is $B_1 = \{x_1, x_2\}$, $B_2 = \{x_3\}$, $B3 = \{x_4\}$. As a new tuple x_5 arrives, the old buckets are merged and a new bucket is created with the new tuple, i.e., $B_1 = \{x_1, x_2\}$, $B_2 = \{x_3, x_4\}$, $B3 = \{x_5\}$. When a sliding window moves, buckets whose timestamps have expired are removed. However, there may exist a deviation of timestamps in a bucket. If the size of the sliding window is 4 in the example, it should drop the tuple x_1 from the window. However, the bucket B_1 also contains x_2 which is valid for the window, so it cannot be removed. This case occurs more frequently as the window size increases.

G2CS (Generic 2-phase Continuous Summarization framework) [28] proposes a mechanism for sliding window maintenance, and C-BIRCH which is a data summarization technique based on BIRCH. The problem that G2CS solves is similar to ours. However, unlike our algorithm, G2CS does not limit the size of the summaries and has a higher time complexity with tree-based indexes. In addition, there is an overhead to creating and maintaining unnecessary lattices to deal with various windows queries on the databases. The clustering quality of G2CS is worse than that of the other clustering algorithms because C-BIRCH is based on BIRCH which is sensitive to the arrival order of the data points. **ID-AP** [37] is a semi-supervised clustering algorithm which improves the clustering performance by using both the existing labeled and the unlabeled datasets. ID-AP is designed to handle both increases and decreases of the data, but not to precisely remove all expired tuples of the window.

In this paper, we propose an efficient algorithm for partition-based clustering with sliding windows. The algorithm aims to quickly produce high-quality clustering results. Unlike with other algorithms, the novel data structure and procedures for our algorithm enable clustering on tuples in exact ranges and reduce the computational cost of the operations, including insertion, deletion, searching

and clustering. Our algorithm maintains the temporal group features, as synopses, using a sliding window aggregation to reduce the footprint and the computational load [41]. In the sliding window aggregation, a window is divided into disjoint chunks, and a synopsis of the window is computed by merging the synopses of these chunks. While the BIRCH-alike algorithms use a tree structure to search for the nearest CF, our algorithm exploits Locality-Sensitive Hashing [18]. It reduces the synopses by joining the nearest neighbors, and the LSH can be used to efficiently search for the nearest group feature in the synopses. The hash-based searching has an average time complexity of $O(1)$ in searching for the nearest neighbor. Clustering operations are very expensive, as they access all data objects iteratively, thus our algorithm also avoids unnecessary clustering. The re-clustering policy that our algorithm employs is unique in that it allows appending new input tuples to pre-existing clusters if the quality of the modified clustering results is acceptable.

III. PRELIMINARIES AND PROBLEM STATEMENT

A. DATA STREAMS

A data stream is defined as an infinite sequence of tuples.

$$S = \langle x_1, t_1\rangle, \langle x_2, t_2\rangle, \ldots, \langle x_n, t_n\rangle, \ldots$$

where x_i is a tuple, and t_i is a timestamp. A tuple x_i is represented using a multi-dimensional attribute vector. A tuple of d dimensions is denoted by $x_i = (x_{i,1}, \ldots, x_{i,d})$. A timestamp t_i is a non-negative integer value, and t indicates the current time. For the sake of simplicity, we assume that tuples arrive in chronological order, i.e., for any $i < j$, a tuple $s_i = \langle x_i, t_i\rangle$ arrives earlier than $s_j = \langle x_j, t_j\rangle$. The timestamp value denotes a sequence number in tuple-based window, as well as a particular time instance in time-based window.

B. SLIDING WINDOW

A sliding window contains only tuples whose timestamps are within the range of the current timestamp and the starting timestamp of the window. Formally, the window is defined as a weight function of two variables, the timestamp of tuple t_i and the current timestamp t.

$$w(t - t_i) = \begin{cases} 1, & \text{if } t - t_i \leq R \\ 0, & \text{if } t - t_i > R \end{cases}$$

where R is the window$'$s time range. As time passes, the window removes the tuples whose timestamps have expired. Since the sliding window is specified according to the most general definition, a landmark window or a damped window can also be defined using the sliding window. In continuous queries in DSMS [42], the sliding window is specified by the `RANGE` for the length of the window and `SLIDE` for the movement intervals of the window.

For example, `S [RANGE 1000 TUPLES SLIDE 100 TUPLES]` is a sliding window that contains the most recent 1,000 tuples, and the window is updated upon the arrival of every 100 tuples from data stream S. Once another 100 tuples arrive, the oldest 100 tuples are removed from the window and the new 100 tuples are appended. For the sake of clarity, if we set `SLIDE` to L, the expression "a window slides" or "a window moves" means that the oldest L tuples in the window are deleted and new L tuples are added to the window.

Windows are categorized into tuple-based and time-based sliding windows according to the sliding condition and the time unit. For ease of explanation, we only consider a tuple-based window, but these same methodologies can be just as easily applied to a time-based window as well.

C. K-MEANS CLUSTERING

Let $S \subseteq \mathbb{R}^d$ be a set of tuples in d-dimensional Euclidean space with size $|S| = n$. For any two tuples x_1, x_2, we denote the Euclidean distance between x_1 and x_2 by $dist(x_1, x_2) = \|x_1 - x_2\| = \sqrt{\sum_{i=1}^{d}(x_{1,i} - x_{2,i})^2}$, and the squared Euclidean distance by $dist^2(x_1, x_2) = \|x_1 - x_2\|^2$. For any finite set $C \subset \mathbb{R}^d$, we define $dist^2(x, C) = \min_{c\in C}\|\text{x-c}\|^2$. k-Means clustering is defined as follows.

Definition 1 (k-Means Clustering [43]): For a set $S \subseteq \mathbb{R}^d$, k-means clustering is to find a set $C \subset \mathbb{R}^d$ of k tuples that minimize the $cost(S, C)$, where

$$cost(S, C) = \sum_{x\in S} dist^2(x, C).$$

The objective of k-means clustering is to minimize the sum of the squared distance of all tuples in S to their nearest tuple in C, i.e., $min_{C\in\mathbb{R}^d} cost(S, C)$.

Similarly, for any weight function $w(x)$ for every $x \in S$, the weighted k-means clustering is to find a set $C \subset \mathbb{R}^d$ of k tuples that minimize the $cost_w(S, C)$, where

$$cost_w(S, C) = \sum_{x\in S} w(x) \cdot dist^2(x, C).$$

With sliding window, the cost includes a weight function of the time parameter, i.e., $cost_w(S, C) = \sum_{x\in S} w(t - t_x) \cdot w(x) \cdot dist^2(x, C)$, where t is the current time and t_x is the timestamp of x.

The optimal cost of k-means clustering of S is denoted by

$$cost^k_{OPT}(S) = \min_{C_O\in\mathbb{R}^d, |C_O|=k} cost(S, C_O).$$

Since k-means clustering problem is an NP-hard problem even for $k = 2$ [44], heuristic algorithms have been proposed. One of the classical heuristic algorithms is *Lloyd*$'$s algorithm [5]. The process of the algorithm is described as follows: Given k random initial cluster centers, 1) Each tuple is assigned to the cluster C_i whose center is nearest 2) Each cluster C_i updates its center to reflect the centroid of tuples in the cluster. This process iterates until the cluster centers go unchanged. Note that the algorithm may not converge to a global optimum but only to a local optimum. The quality of clustering depends on the initial cluster centers.

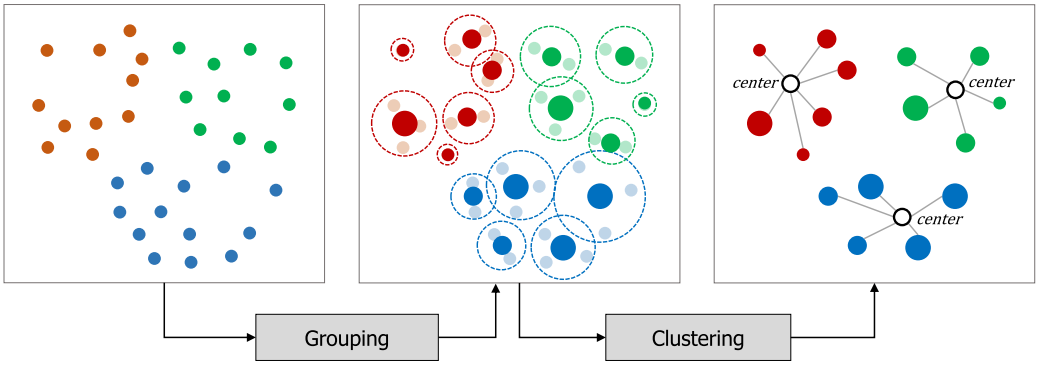

FIGURE 2. Two steps for data stream clustering.

D. CORESET

A coreset for a set S is a small weighted set that approximates S with respect to an optimization problem. The cost of the coreset is an approximation for the cost of an original set S within a factor $(1+\varepsilon)$ for $0 \leq \varepsilon \leq 1$. The cost of the coreset for k-means clustering is computed using weighted k-means clustering, and is an approximation for the cost of the original set S with relative error ε. Then, the coreset for k-means clustering can be defined as follows.

Definition 2 ((k,ε)-Coreset [8]): Let $S \subseteq \mathbb{R}^d$, $k > 0$, and $0 \leq \varepsilon \leq 1$. A weighted set $M \subseteq \mathbb{R}^d$ is (k,ε)-coreset of S for k-means clustering, if for all $C \subset \mathbb{R}^d$, $|C| = k$, we have

$$(1-\varepsilon)\cdot cost(S,C) \leq cost_w(M,C) \leq (1+\varepsilon)\cdot cost(S,C).$$

The approximation algorithm using the coreset is efficient because the algorithm is applied on a small sized coreset rather than on the entire data. The algorithm using the fixed-size coreset is expected to be completed within a certain amount of time. The processing time of the algorithm is primarily dependent on the time it takes to construct the coreset. For example, the coreset construction time is linear to the number of data n, the dimension of data d, and the number k for the k-segmentation problem [45].

However, the coreset with an approximation guarantee cannot be computed for the data streams. Previous approaches have focused on the shown approximation guarantee of the coreset when appropriate centers are given [8], [46], [47]. In practice, these centers are unknown, and the problem of finding optimal centers is NP-hard [44] in the k-means problem. The algorithms to find k centers with $(1+\varepsilon)$-approximation are unknown because the problem is also APX-hard [48]. A state-of-the-art algorithm in the k-means problem is *k-means++*, where the cost is $E[Cost(S,C)] \leq 8(\log k + 2)\cdot cost^k_{OPT}(S)$ [38].

Two common tasks typically performed by data stream clustering algorithms [3], [4], [6], [8] are the 1) grouping task and 2) clustering task, as shown in Fig. 2. The grouping task constructs a *coreset* from the original data using a particular heuristic method. The clustering task applies a specific clustering algorithm in order to the *coreset* to identify clusters of the original data.

We used the more general term *synopses* to refer to summarized data in the explanation in Section I. However, we refer to a small weighted set generated through the grouping task as a *coreset* from this section. Although the approximation factor of the weighted set cannot be proven in the data streams, we consider that the term *coreset* represents the concept more clearly than does the term *synopses*.

E. GROUP FEATURE

Group Feature(GF) is defined as a data structure to store the statistic summaries of a set of tuples contained in the *coreset*. Previous studies have used the term *Clustering Feature(CF)* to refer to statistic summaries [4], [14]. However, the term *Clustering Feature(CF)* is confused with the results made through the clustering task. Therefore, we will use the term *Group Feature(GF)* which is an improved version of the *CF* in our algorithm.

GF consists of a linear sum of the tuples LS, square sum of the tuples SS, the number of the tuples N, and the most recent timestamp of the tuples T. The tuples are in the range of the sliding window. LS and SS are generated using an element-wise summation of the tuples, i.e. for d-dimensional n tuples, $LS = \sum_{i=1}^{n} x_i = \sum_{i=1}^{n}(x_{i,1},\ldots,x_{i,d})$ and $SS = \sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n}((x_{i,1})^2,\ldots,(x_{i,d})^2)$. LS and SS are d-dimensional vectors, and N and T are numeric values. The basic components LS, SS, and N are proposed in [14], and the timestamp component T is added in [4]. In addition, *GF* contains a hash value generated by LSH for our algorithm, which we will explain in a following section.

*GF*s hold incrementality and additivity. Incrementality means that the *GF* is updated by adding a new tuple x_j, while additivity means that two disjoint *GF*s can be merged into a new *GF* by adding their components [14]. These properties enable modifying the coreset in a constant time.

Incrementality	**Additivity**
$LS = LS_1 + x_j$	$LS = LS_1 + LS_2$
$SS = SS_1 + (x_j)^2$	$SS = SS_1 + SS_2$
$N = N_1 + 1$	$N = N_1 + N_2$
$T = t_j$	$T = max(T_1, T_2)$

The values for clustering, such as the centroid, can be easily calculated using the components of the GF, i.e., $Centroid = LS/N$. GF is continuously updated as the tuples are input from the data streams. For the sliding window, the expired GFs are removed based on the timestamp T from the window, and new GFs are appended to the window.

F. PROBLEM STATEMENT

Based on the definitions so far, we define the problem statement as follows. Given a stream of tuples(S), user-specified number of clusters (k), window size (RANGE R), sliding interval (SLIDE L), and a coreset (M), the problem of the data stream clustering with a sliding window involves how to generate clusters of tuples in a sliding window while considering both accuracy and runtime.

Our approach to the problem consists of two sub tasks in general: 1) Construction and maintenance of a constant size of coreset (M) with sliding windows (R and L) for data streams (S) 2) Decision of whether to append a new coreset to pre-existing k clusters or perform clustering on a whole coreset to generate new k clusters.

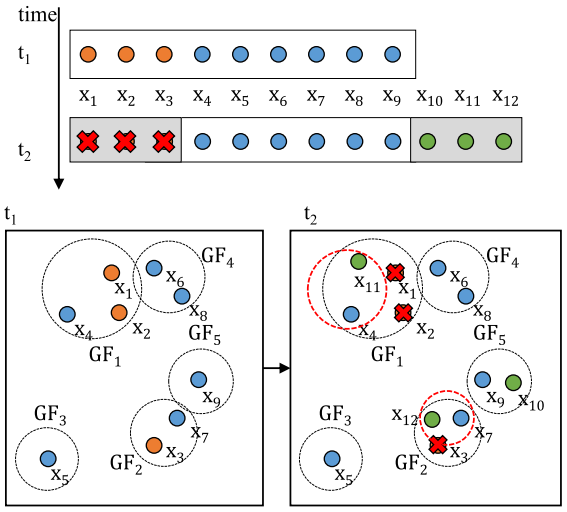

FIGURE 3. A snapshot to cluster data with a sliding window.

Fig. 3 shows a simple example in order to illustrate how we approach the problem. Let us say that the input data streams $S = \langle x_1, t_1\rangle, \langle x_2, t_2\rangle, \ldots$, the number of clusters $k = 3$, the window size $R = 9$, the sliding interval $L = 3$, and the coreset size $|M| = 5$. For the ease of presentation, we simply illustrate a GF with a set of grouped tuples, rather than showing the summary values contained in the GF. First, the grouping step in the algorithm generates 5 GFs as a coreset at time t_1. Let GFs in the first window be $GF_1 = \{x_1, x_2, x_4\}$, $GF_2 = \{x_3, x_7\}$, $GF_3 = \{x_5\}$, $GF_4 = \{x_6, x_8\}$, $GF_5 = \{x_9\}$, the weighted k-means clustering be applied on GFs, and it generate 3 clusters, $C_1 = \{GF_1, GF_4\} = \{x_1, x_2, x_4, x_6, x_8\}$, $C_2 = \{GF_2, GF_5\} = \{x_3, x_7, x_9\}$, and $C_3 = \{GF_3\} = \{x_5\}$. As the sliding window moves by the sliding interval $L = 3$, x_1, x_2, and x_3 are expired at time t_2, and they should be removed from the GFs and clusters, while x_{10}, x_{11}, and x_{12} be appended. Because a GF maintains the statistics of the tuples, GF_1 and GF_2 which include the expired tuples are removed. GF_3, GF_4, and GF_5 contain the valid tuples regarding the slide interval and they remain. Since the coreset size is 5, two new GFs can be generated. GFs for $\{x_4, x_{11}\}$ and $\{x_7, x_{12}\}$ are created. There is no guarantee that the GFs and clusters of the first window at t_1 also minimize the $cost(S, C)$ at t_2 with the new tuples. GFs and clusters for the tuples, $x_4, \ldots, x_{12}$ should be constructed and maintained so as to minimize the $cost(S, C)$ at t_2. Since the new tuples arrive rapidly in a data stream environment, the algorithm needs to run efficiently enough to make it practical, i.e., it should minimize the inefficiency of data re-processing while maximizing the clustering quality.

IV. CORESET CONSTRUCTION BASED ON NEAREST NEIGHBOR SEARCH

A. ALGORITHM FOR CORESET CONSTRUCTION

In this section, we describe a method for constructing and maintaining a coreset with sliding windows. The GF for the sliding window contains statistic summaries of unexpired tuples. The incrementality enables updating the GF with new arrival tuples. However, once a tuple is subsumed in the GF, it cannot be subtracted from the GF unless the value of the tuple is kept separately. Since it is inefficient to keep all tuples in the sliding window, we propose a data structure for the coreset based on pane-based aggregation of the sliding window [41].

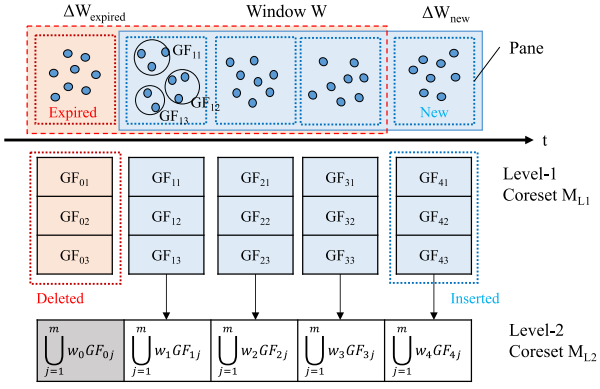

FIGURE 4. Data structure for the coreset in the sliding window.

Fig. 4 shows an overview of the coreset structure. A window is decomposed into panes composed of non-overlapping sets of tuples. Suppose that `RANGE` is R, `SLIDE` is L. The number of panes is $\lceil R/L \rceil$, and each pane represents at most L tuples. For example, a sliding window which is defined by `S [RANGE 120 TUPLES SLIDE 30 TUPLES]` has four panes each of which contains 40 tuples. In `S [RANGE 120 TUPLES SLIDE 29 TUPLES]`, window consists of five panes, where four panes each contain 29 tuples, and the last pane contains four tuples. For the ease of presentation, we only discuss the case in which R is divisible by L.

When the window slides, $\Delta W_{expired}$ is removed and ΔW_{new} is appended. New GFs are generated based on tuples in ΔW_{new}. The detailed process of creating GF is described in Algorithms 1 and 2.

Algorithm 1 ConstructCoreset

Input: A set of tuples B, coreset size m
Output: Coreset P

1: $b_1 \leftarrow$ an initial tuple in B
2: $\theta \leftarrow$ minimum distance between b_1 and random samples D from B
3: **for** each $b \in B$ **do**
4: $\quad GF_p \leftarrow$ nearest GF in P to b
5: $\quad$ **if** $dist(b, GF_p) < \theta$ or $dist(b, GF_p) <$ radius of GF_p **then**
6: $\quad\quad GF_p = GF_p + b$
7: $\quad$ **else**
8: $\quad\quad$ create new GF_b based on b, and $P \leftarrow P \cup \{GF_b\}$
9: $\quad$ **end if**
10: $\quad$ **if** $|P| \geq 2m$ **then**
11: $\quad\quad P \leftarrow$ ReduceCoreset(P, m)
12: $\quad$ **end if**
13: **end for**
14: **return** P

Algorithm 2 ReduceCoreset

Input: Coreset P, reduced coreset size m
Output: Reduced coreset Q

1: $Q \leftarrow \{\}$, and $R \leftarrow P$
2: **for** each $GF_p \in P$ **do**
3: $\quad R \leftarrow R - \{GF_p\}$
4: $\quad GF_q \leftarrow$ nearest GF in Q to GF_p
5: $\quad GF_r \leftarrow$ nearest GF in R to GF_p
6: $\quad$ **if** $dist(GF_p, GF_q) < dist(GF_p, GF_r)$ **then**
7: $\quad\quad GF_q \leftarrow GF_q + GF_p$
8: $\quad\quad Q \leftarrow Q \cup \{GF_q\}$
9: $\quad$ **else**
10: $\quad\quad GF_r \leftarrow GF_r + GF_p$
11: $\quad\quad Q \leftarrow Q \cup \{GF_r\}$
12: $\quad\quad R \leftarrow R - \{GF_r\}$
13: $\quad$ **end if**
14: $\quad$ **if** $|Q| + |R| \leq m$ **then**
15: $\quad\quad Q \leftarrow Q \cup R$
16: $\quad\quad$ **break**
17: $\quad$ **end if**
18: **end for**
19: **return** Q

Algorithm 1 shows a process for creating a coreset P for a set of tuples B. For example, in Fig. 4, B is a set of tuples in ΔW_{new}, and a coreset with $m = 3$ consists of three GFs, denoted by $\{GF_{41}, GF_{42}, GF_{43}\}$. A coreset M of the window is made through the union of the coresets of each pane, and its size $|M|$ is 12. First, the algorithm calculates the minimal distance between b_1 and D, uses it as threshold θ (line 2). The reason for using the minimum instead of the maximum and the average is that the threshold becomes very large when an outlier exists. The GF with large radius contains too many tuples that cannot be split. However, even if a large number of GFs are generated with a small threshold, they are reduced to GFs of an appropriate radius and number through ReduceCoreset. After determining the threshold θ, the tuples whose distances are below θ are grouped into the same GF. If the distance between a tuple and a centroid of GF, $dist(b, GF_p)$ is below θ or the radius of GF, the GF absorbs the tuple (line 5).

GFs are continuously generated with θ. When the number of GFs reaches $2m$, they are reduced to m through ReduceCoreset in Algorithm 2. Since all GFs in coreset P are merged once through ReduceCoreset, the number of coreset becomes $1/2$ the initial size. The algorithm employs a nearest neighbor search. Specifically, target GF_p is added to the closer GF of unprocessed set R and processed set Q. Next, the added GF is included in Q (line 5-13). If the size of the input coreset is $2m$, all GFs should be processed at least once in order to reduce the size to m. If the size of the input coreset is smaller than $2m$, it is not necessary to process all GFs. In this case, when the sum of the number of R and Q becomes the desired size, the process ends.

As shown in Fig. 4, the data structure for the sliding window consists of a level-1 coreset and a level-2 coreset. The level-1 coreset has $\lceil R/L \rceil$ columns, and each column contains m GFs. The GFs are generated from tuples in the new pane through ConstructCoreset, and they are inserted into the last column of the level-1 coreset. The timestamps Ts of the inserted GFs are in $(R - L, R]$. When expired tuples are removed, the first column of the level-1 coreset with timestamps in $[t_1, t_1 + L]$ is truncated, where t_1 is the earliest timestamp. The removal operation for the level-1 coreset has $O(1)$ time complexity if an adequate data structure is utilized such as a linked list queue. The exact number of tuples in the level-1 coreset is R and remains constant.

The level-2 coreset is built through the union of those GFs which are in the same column. Each GF in the level-1 coreset is multiplied with the weights and is merged into the level-2 coreset. The weights can be defined as either a time-decay function or as simply the same values. Let w_i be the weight for GF_i. Then, the level-2 coreset is represented as $GF_i = \bigcup_{j=1}^{m} w_j GF_{ij}$, as shown in Fig. 4. The sliding window updates the level-2 coreset by appending new GFs and truncating the expired GFs in the same way as the level-1 coreset. Clustering is performed on the generated level-2 coreset.

Algorithm 1, ConstructCoreset, takes linear time with respect to the size of input $|B|$. The most time consuming operation of the algorithm is the nearest GF search, and an efficient algorithm to improve the search time is described in Section V.

B. THEORETICAL ANALYSIS OF CORESET CONSTRUCTION

In this section, we investigate the accuracy guarantee of the coreset that is constructed by using ConstructCoreset in Algorithm 1. As we state in Section III-D, the coreset with an approximation guarantee cannot be computed for the data streams. Therefore, we show the upper bound of the k-means cost for our approach with the assumption that the tuples in a given set are unchanged, and that the optimal centers are known. We start by analyzing the cost for $k = 1$.

Lemma 1: Let S be a set of tuples, M be a set of group features generated from S, and C be the optimal cluster centers. For $k = 1$, it holds that $cost(S, C) = cost(M, C)$.

Proof: The center for optimal 1-mean cost is the center of mass, which is calculated by the linear sum of the tuples divided by the number of tuples. If all GFs in M are merged into one GF, the LS of the merged GF is equal to the linear sum of the tuples due to the additivity property. The number of tuples is also preserved in the GF. □

We see that ConstructCoreset does not increase the optimal cost for $k = 1$. For $k \geq 2$, we consider that ReduceCoreset is applied between the GFs in the same cluster, and between GFs in the different clusters separately. For the sake of simplicity, we assume that the threshold $\theta = 0$.

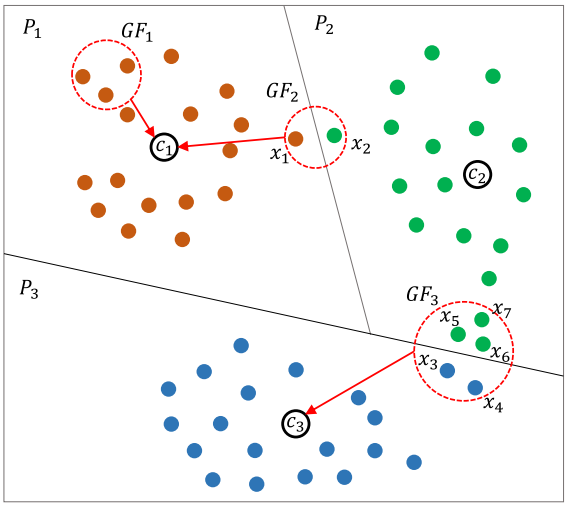

FIGURE 5. Constructing *GF*s by ConstructCoreset.

Let us see an example illustrated in Fig. 5. The nearest center for the tuples in GF_1 is center c_1. The nearest center for GF_1 is also center c_1. In this case, the cost of GF_1 is the optimal 1-mean cost according to Lemma 1. If all GFs are generated only from tuples in the same cluster, the k-means cost calculated from the GFs is optimal.

The distance between tuple x_1 and tuple x_2 in GF_2 is relatively far from the other tuples in their clusters, but close to each other. Since the nearest neighbor of x_1 is x_2, they are combined into GF_2. GF_2 is located on two clusters. GF_2 will be assigned to the center c_1 or center c_2. If GF_2 is assigned to center c_1, x_1 is correct and does not cause an error. However, for x_2, the cost is increased because it is misclassified. Then, the following equations can be established.

$$\|x_1 - c_1\| > \|x_1 - x_2\| \tag{1}$$

$$\|x_2 - c_2\| > \|x_1 - x_2\| \tag{2}$$

$$\|\frac{x_2 + x_1}{2} - c_2\| > \|\frac{x_2 + x_1}{2} - c_1\| \tag{3}$$

Lemma 2: Let x be a tuple in GF, and c_x be the nearest center of x, and c_g is the nearest center of GF. For an arbitrary tuple $z \in GF$,

$$\|x - c_g\|^2 < 2(\|x - c_x\|^2 + \|z - c_g\|^2) \tag{4}$$

Proof: For an arbitrary tuple z in GF_2 and an incorrectly assigned tuple x_2, the following equation is derived by the triangle inequality and Caucy-Schwarz inequality.

$$\begin{aligned}\|x_2 - c_1\|^2 &= \|x_2 - z + z - c_1\|^2 \\ &< 2\|x_2 - z\|^2 + 2\|z - c_1\|^2\end{aligned}$$

Equation (2) implies $\|x_2 - c_2\| > \|x_2 - z\|$. This yields that

$$\begin{aligned}\|x_2 - c_1\|^2 &< 2\|x_2 - c_2\|^2 + 2\|z - c_1\|^2 \\ &\Leftrightarrow \|x - c_g\|^2 < 2(\|x - c_x\|^2 + \|z - c_g\|^2).\end{aligned}$$

□

We now provide the upper bound of a generalized cost considering that GF contains two or more tuples.

Theorem 1: Let S be a set of tuples, M be a set of group features generated from S, C be the optimal cluster centers, and Q be a set of incorrectly assigned tuples. Then, it holds that $cost_w(M, C) < cost(S, C) + 9\sum_{x \in Q}\|x - c_x\|^2$.

Proof: For intuitive presentation, consider the case of GF_3 in Fig. 5. The optimal cost with GF_3 is

$$\begin{aligned}cost(S, C) = &\sum_{x \in P_3 - \{x_3, x_4\}} \|x - c_3\|^2 \\ &+ \sum_{x \in P_2 - \{x_5, x_6, x_7\}} \|x - c_2\|^2 \\ &+ \|x_3 - c_3\|^2 + \|x_4 - c_3\|^2 \\ &+ \|x_5 - c_2\|^2 + \|x_6 - c_2\|^2 \\ &+ \|x_7 - c_2\|^2.\end{aligned}$$

Let us say that the constructed coreset is M and GF_3 is assigned to c_3. The Euclidean distance between GF_3 and c_3 is $\|\frac{x_3+x_4+x_5+x_6+x_7}{5} - c_3\|$. The cost for M is

$$\begin{aligned}cost_w(M, C) = &\sum_{x \in P_3 - \{x_3, x_4\}} \|x - c_3\|^2 \\ &+ \sum_{x \in P_2 - \{x_5, x_6, x_7\}} \|x - c_2\|^2 \\ &+ 5 \cdot \|\frac{x_3 + x_4 + x_5 + x_6 + x_7}{5} - c_3\|^2.\end{aligned} \tag{5}$$

Note that we consider only the unweighted input tuples for the sake of clarity. The coreset construction can be extended to the weighted tuples. The equation for the cost of assigning x_5, x_6, x_7 incorrectly to c_3 instead of c_2 is $\|x_5 - c_3\|^2 + \|x_6 - c_3\|^2 + \|x_7 - c_3\|^2$. Then the sum of this equation and $\|x_3 - c_3\|^2 + \|x_4 - c_3\|^2$ should be greater than $5 \cdot \|\frac{x_3+\cdots+x_7}{5} - c_3\|^2$.

By applying this inequation into the cost difference between M and S, we obtain the following:

$$\begin{aligned} &cost_w(M, C) - cost(S, C) \\ &\quad = (\|x_3 - c_3\|^2 + \|x_4 - c_3\|^2 + \|x_5 - c_3\|^2 \\ &\qquad + \|x_6 - c_3\|^2 + \|x_7 - c_3\|^2) \\ &\qquad - (\|x_3 - c_3\|^2 + \|x_4 - c_3\|^2 + \|x_5 - c_2\|^2 \\ &\qquad + \|x_6 - c_2\|^2 + \|x_7 - c_2\|^2) \end{aligned}$$

By using Lemma 2, we have

$$\begin{aligned} &cost_w(M, C) - cost(S, C) \\ &\quad < \|x_5 - c_2\|^2 + \|x_6 - c_2\|^2 + \|x_7 - c_2\|^2 \\ &\qquad + 2 \cdot \|z - c_3\|^2 + 2 \cdot \|z - c_3\|^2 + 2 \cdot \|z - c_3\|^2. \end{aligned} \tag{6}$$

Tuple z can be an arbitrary tuple in GF_3, and we set z to the center of GF_3, c_{g3}. By Equation (3), we have $\|c_{g3} - c_2\| > \|c_{g3} - c_3\|$. For example, for x_5, it holds that $\|c_{g3} - c_2\| < \|c_{g3} - x_5\| + \|x_5 - c_2\|$ by using the triangle inequality. We also obtain $\|x_5 - c_2\| > \|x_5 - c_{g3}\|$ by Equation (2). Based on these statements, we have

$$\begin{aligned} \|z - c_3\|^2 = \|c_{g3} - c_3\|^2 &< \|c_{g3} - c_2\|^2 \\ &< 2\|x_5 - c_2\|^2 + 2\|x_5 - c_{g3}\|^2 \\ &< 4\|x_5 - c_2\|^2. \end{aligned}$$

Then, the Equation (6) can be restated as follows.

$$\begin{aligned} &cost_w(M, C) - cost(S, C) \\ &\quad < 9\|x_5 - c_2\|^2 + 9\|x_6 - c_2\|^2 + 9\|x_7 - c_2\|^2. \end{aligned}$$

In order to generalize the above equation, let Q be a set of incorrectly assigned tuples, and c_x be the nearest for x in the optimal solution. Then,

$$cost_w(M, C) - cost(S, C) < 9 \sum_{x \in Q} \|x - c_x\|^2$$

□

Theorem 1 means that the cost does not exceed nine times the optimal cost of the incorrectly assigned tuples. As the optimal solution of $cost(S, C)$ cannot be obtained as mentioned in Section III-D, let us say that we obtain a solution by the α-approximation algorithm, i.e., $cost(S, C) = \alpha \cdot cost^k_{OPT}(S)$. To compute the upper bound of the cost, we can assume that all tuples are assigned incorrectly.

$$\begin{aligned} cost_w(M, C) &= \alpha \cdot cost^k_{OPT}(S) + 9 \sum_{x \in S} \|x - c_x\|^2 \\ &\leq (\alpha + 9) cost^k_{OPT}(S) \end{aligned}$$

The cost $(\alpha + 9)cost^k_{OPT}(S)$ may seem relatively high for guaranteeing the accuracy. However, it represents only the worst case cost. If the number of tuples which are assigned incorrectly is small, the cost would be close to the optimal cost of the algorithm.

C. THEORETICAL ANALYSIS OF SLIDING WINDOWS

Theorem 1 is a cost analysis for the case with regard to reducing the tuples from S to M. In this subsection, we provide another aspect of cost analysis of our algorithm, with its regard to the sliding window. Given `RANGE` R and `SLIDE` L, a sliding window contains $b = \lceil R/L \rceil$ panes. A pane includes $|S|/b$ tuples, and the coreset which contains $|M|/b$ GFs is produced. In order to present the accuracy guarantee, we adopt a similar methodology that is used for the k-Median problem [15], [23], where the entire tuples are partitioned first and the summaries of each partition are merged later.

Lemma 3: Let $S_1, \ldots, S_p$ be arbitrary partitions of a set S. Then,

$$\sum_{i=1}^{p} cost(S_i, C) = cost(\bigcup_{i=1}^{p} S_i, C) = cost(S, C).$$

Proof: Because partitions are not overlapped, each tuple belongs to exactly one partition. Therefore, the cost of a partition is equal to the sum of the squared distances of the tuples which the partition contains. □

Lemma 4: Let M be a set of group features which are generated from S, and $S_1, \ldots, S_p$ be arbitrary partitions of a set S. Coresets $M_1, \ldots, M_p$ are generated from $S_1, \ldots, S_p$ respectively. Then,

$$\sum_{i=1}^{p} cost_w(M_i, C) \leq (\alpha + 9) cost^k_{OPT}(S) \tag{7}$$

Proof: From Lemma 3, we know that the cost of a tuple set is the sum of the costs of its partitions. Let C be a set of k centers, and Q_i be a set of the incorrectly assigned tuples in M_i.

$$\begin{aligned} &\sum_{i=1}^{p} cost_w(M_i, C) - cost(S, C) \\ &\quad = \sum_{i=1}^{p} cost_w(M_i, C) - \sum_{i=1}^{p} cost(S_i, C) \\ &\quad = \sum_{i=1}^{p} (cost_w(M_i, C) - cost(S_i, C)) \\ &\quad \leq \sum_{i=1}^{p} 9 \sum_{x \in Q_i} \|x - c_x\|^2 = 9 \sum_{x \in \bigcup_{i=1}^{p} Q_i} \|x - c_x\|^2 \end{aligned}$$

As was the case previously, if we consider the worst case in which all tuples are assigned incorrectly, the following holds,

$$\begin{aligned} \sum_{i=1}^{p} cost_w(M_i, C) &\leq cost(S, C) + 9 \sum_{x \in \bigcup_{i=1}^{p} Q_i} \|x - c_x\|^2 \\ &\leq \alpha \cdot cost^k_{OPT}(S) + 9 \sum_{x \in S} \|x - c_x\|^2 \\ &\leq (\alpha + 9) cost^k_{OPT}(S). \end{aligned}$$

□

Based on Lemma 4, the cost difference between $M_1, \ldots, M_p$ and M is

$$\begin{aligned}
&\sum_{i=1}^{p} cost_w(M_i, C) - cost_w(M, C) \\
&= \sum_{i=1}^{p} cost_w(M_i, C) - \sum_{i=1}^{p} cost(S_i, C) \\
&\quad - (cost_w(M, C) - \sum_{i=1}^{p} cost(S_i, C)) \\
&= \sum_{i=1}^{p} (cost_w(M_i, C) - cost(S_i, C)) \\
&\quad - (cost_w(M, C) - cost(S, C)) \\
&\leq 9 \sum_{x \in \bigcup_{i=1}^{p} Q_i} \|x - c_x\|^2 - 9 \sum_{x \in Q} \|x - c_x\|^2
\end{aligned}$$

Intuitively, there is no relationship between $\bigcup_{i=1}^{p} Q_i$ and Q. However, if both sets contain the same incorrectly assigned tuples, the cost of $M_1, \ldots, M_p$ and the cost of M become equal. Now, we consider a case where the window slides. Let S_i be the set of tuples in the i^{th} pane of the sliding window, the window contains p panes, and $S = \{S_1, \ldots, S_p\}$. As the window slides, S_1 is deleted and S_{p+1} is appended. Let the updated sliding window be $S^* = \{S_2, \ldots, S_{p+1}\}$.

Previously, we computed the cost for optimal centers C under the assumption that the tuples are unchanged. However, because the tuples are appended and deleted, their optimal centers may also be changed. Let a set of optimal centers for S^* be C^*. Let us compare the cost for C with the cost for C^*.

As S_1 is removed from the window and S_{p+1} is added to the window, their corresponding coresets M_1 and M_{p+1} are also removed, and newly created and added, respectively. Since each partition does not overlap with each other by Lemma 3, addition and deletion do not affect the cost of coresets in another partitions.

Theorem 2: Let a data stream S be $S = \{S_1, \ldots, S_p\}$, its updated sliding window be $S^* = \{S_2, \ldots, S_{p+1}\}$, M_i be the generated coreset from S_i, and C and C^* be a set of optimal centers for S and S^* respectively. For the common tuples in S and S^*, it holds that

$$cost_w(\bigcup_{i=2}^{p} M_i, C^*) < 2 \cdot cost_w(\bigcup_{i=1}^{p} M_i, C) + \gamma$$

where γ is constant.

Proof: Let c_x and c_x^* be the nearest center for x in C and C^*, respectively. By triangle inequality, we have

$$\begin{aligned}
\|x - c_x^*\| &< \|x - c_x\| + \|c_x - c_x^*\| \\
&\Leftrightarrow \|x - c_x^*\|^2 < 2\|x - c_x\|^2 + 2\|c_x - c_x^*\|^2
\end{aligned}$$

Let us apply the sum of the above expression for every x in $\bigcup_{i=2}^{p} M_i$, and replace $2\sum_{x \in \bigcup_{i=2}^{p} M_i} \|c_x - c_x^*\|^2$ by γ, as follows.

$$\begin{aligned}
&\sum_{x \in \bigcup_{i=2}^{p} M_i} \|x - c_x^*\|^2 \\
&\quad < 2 \sum_{x \in \bigcup_{i=2}^{p} M_i} \|x - c_x\|^2 \\
&\qquad + 2 \sum_{x \in M_1} \|x - c_x\|^2 + \gamma \\
&\Leftrightarrow \sum_{x \in \bigcup_{i=2}^{p} M_i} \|x - c_x^*\|^2 < 2 \sum_{x \in \bigcup_{i=1}^{p} M_i} \|x - c_x\|^2 + \gamma \\
&\Leftrightarrow \quad cost_w(\bigcup_{i=2}^{p} M_i, C^*) < 2cost_w(\bigcup_{i=1}^{p} M_i, C) + \gamma
\end{aligned}$$

□

Theorem 2 shows that the cost for a sliding window with an expired partition removed is about twice as much as the cost for the previous window. By appending the cost for a new partition, the cost for the updated sliding window holds that

$$\begin{aligned}
&cost_w(\bigcup_{i=2}^{p+1} M_i, C^*) \\
&\qquad < 2cost_w(\bigcup_{i=1}^{p} M_i, C) + cost_w(M_{p+1}, C^*) + \gamma.
\end{aligned}$$

This indicates that the cost for the updated sliding window is increased by the cost for new tuples, $cost_w(M_{p+1}, C^*)$, regardless of the cost of the previous window. The data stream clustering using the partitioning with sliding windows needs processing within a certain upper bound of error. This is why employing a re-clustering policy like ours, as explained in Section VI, is indeed necessary in practice.

V. CORESET CONSTRUCTION BASED ON LOCALITY-SENSITIVE HASHING

Our proposed algorithm is designed to take into account several problems in [4] and [7]: 1) the appropriate threshold should be determined according to dataset, 2) there is no limit to the number of groups, and 3) expired tuples may exist in the synopses. In addition, there are still time-consuming operations that can be improved for practical use.

Assume that we have a coreset $M = \{GF_1, \ldots, GF_m\}$ from a set of input tuples $\{x_1, \ldots, x_i\} \subset B$, where $|M| = m$, $|B| = n$, and $i < n$. For the next input tuple $x_{i+1} \in B$, Algorithm 1 needs to find the nearest neighbor $GF \in M$ of x_{i+1}. Finding the nearest GF is a computationally heavy operation since it scans all GFs in M and computes all distances for each GF. The search operation requires $O(dmn)$ time to obtain a coreset per window slide, which is too slow for a large window or for high-dimensional data. Furthermore, Algorithm 2 computes the distances between all pairs of GFs in the coreset, and it requires $O(dm^2)$ time. Therefore, we propose an improved algorithm based on the concept of Locality-Sensitive Hashing (LSH) [18] in order to reduce the

distance computation. The proposed algorithm also works for the data streams.

The basic concept of the LSH in Euclidean space is to map similar vectors to hash values that have a higher probability of collision than the hash values of dissimilar vectors. In other words, if two vectors are close to each other, the vectors remain close after projection. The hash function $h_{\vec{a},b}(\vec{x}) : \mathcal{R}^d \rightarrow \mathcal{N}$ is a scalar projection that maps a vector $\vec{x}$ to an integer. For example, the hash function can be given as follows: $h_{\vec{a},b}(\vec{x}) = \lfloor(\vec{a} \cdot \vec{x} + b)/w\rfloor$, where $\vec{a}$ is a randomly drawn d-dimensional vector, w is the width of the quantization bin, and b is a random variable in the interval $[0, w)$. The general LSH generates hash values that are computed from hash functions in order to decrease the probability that dissimilar vectors fall into the same quantization bin. We define the global hash function obtained by concatenating the values from multiple hash functions, e.g., when we have three hash functions, the global hash function $g(\vec{x})$ is $(h_{\vec{a_1},b}(\vec{x}), h_{\vec{a_2},b}(\vec{x}), h_{\vec{a_3},b}(\vec{x}))$. Although there is an overhead to generate and compute the hash value, it is an acceptable load if we use the proper number of hash functions.

The main idea of our algorithm is that vectors with a near distance have similar values of LSH. Based on the concept of LSH, if the distance between two vectors x and y, $dist(x, y)$, is less than θ, the distance after projection is also less than θ. Therefore, by setting the width of the quantization bin w of the hash function to θ, each element of the hash table contains vectors whose distances are within θ. The distance computation is not necessary to combine vectors within the distance θ.

Our algorithm utilizes a hash table whose key is a hash value generated by LSH, and each bucket stores their GFs. For data streams, the algorithm processes the input tuple in realtime. The algorithm appends the input tuple to an existing or new GF in the bucket, and it reduces the hash table by merging buckets when the hash table exceeds a certain size.

Algorithm 3 describes the detailed process of creating a coreset based on LSH. First, the algorithm initializes the hash table H and the global hash function g. The number of hash functions is given by the user, and it creates d-dimensional random unit vectors $a_1, \ldots, a_l$. The process of obtaining the threshold value θ is omitted here because the same method is used in Algorithm 1. The global hash function g is defined using the unit vectors and θ. For each tuple b, the algorithm generates a hash value by $g(b)$. If the hash value $g(b)$ does not exist in the hash table, it creates a new GF of b, and stores the GF with the key of $g(b)$ in the table. If the hash value exists, the GF of the key $g(b)$ absorbs b, and the hash table is updated with the GF. When the hash table exceeds the user-specified size, it is reduced through ReduceCoreset2.

We keep the first tuple b of the group feature as an index tuple. For an index tuple b, we denote the corresponding group feature by GF_b. In order to calculate the distance between the new input tuple and the GFs with more precision, the center of the GF should be indexed. However, since the center of the GF continues to change as tuples are appended, it is inefficient to update the key of the hash table every time. We thus use the fixed key of the index tuple with some error tolerance.

Algorithm 3 ConstructCoreset2

Input: A set of tuples B, coreset size m, number of hash functions l
Output: Coreset P

initialize hash table H, and hash function $g(x) = (h_{\vec{a_1},\theta}(\vec{x}), \ldots, h_{\vec{a_l},\theta}(\vec{x}))$
for each $b \in B$ **do**
  if $g(b) \notin H$ **then**
    create new GF_b based on b
    $H[g(b)] = GF_b$
  else
    $GF_b \leftarrow H[g(b)]$, $GF_b = GF_b + p$, and $H[g(b)] = GF_b$
  end if
  if $|H| \geq 2m$ **then**
    $H \leftarrow$ ReduceCoreset2(H, m)
  end if
end for
return GFs in H

The time complexity of Algorithm 3 is $O(dln)$, since the search time of the hash table is $O(1)$ and the generation time of the hash value for a tuple is $O(dl)$. Since $l \ll m$ and $O(dln) < O(dmn)$, Algorithm 3 runs much more quickly than Algorithm 1. In particular, for high-dimensional data, Algorithm 3 can be more efficient because the cost of the distance calculation is much higher than the cost of the retrievals. The algorithm uses at most $2m$ space.

Algorithm 4 ReduceCoreset2

Input: Hash table H, reduced size m
Output: Reduced hash table H

while $|H| < m$ **do**
  $Q \leftarrow$ a set of key pairs with minimal difference in hash value
  for each $(x, y) \in Q$ **do**
    $GF_x \leftarrow H[g(x)]$
    $GF_y \leftarrow H[g(y)]$
    $GF_x = GF_x + GF_y$
    $H[g(x)] = GF_x$
    remove the bucket of $g(y)$ from H
    if $|H| \leq m$ **then**
      break
    end if
  end for
end while
return H

Algorithm 4 shows the process of reducing the hash table. The algorithm requires a list of key pairs ordered by the differences of the keys. The calculation of the differences of

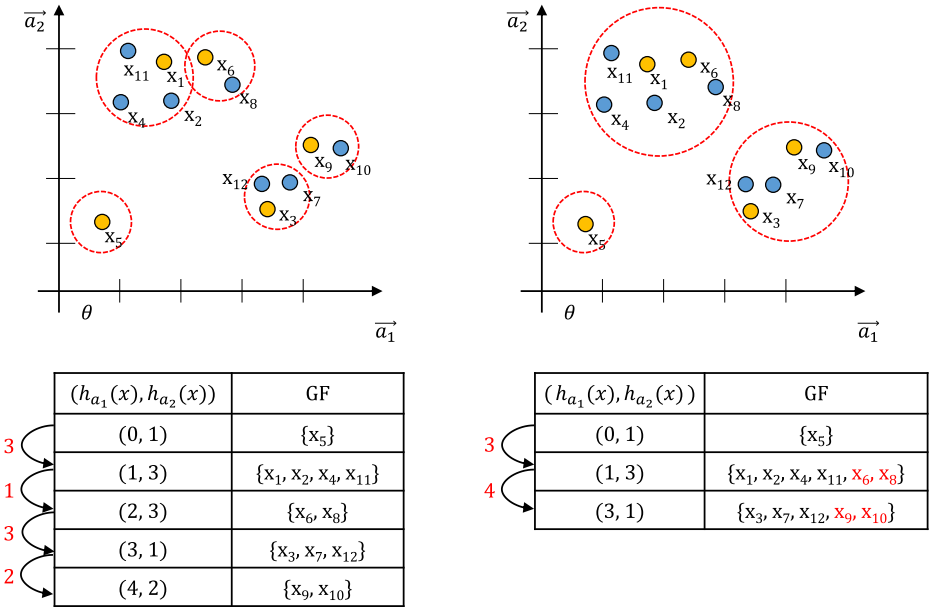

FIGURE 6. Merging buckets in hash table for coreset.

all key pairs is $O(lm^2)$, which is less than the original $O(dm^2)$, but it is still a heavy computational operation. Therefore, we adopt a simple heuristic for linear time complexity.

The heuristic method requires a list of keys ordered by comparing all components of the hash values. A sorted list can be created in the ReduceCoreset2, but we maintain an additional sorted list of keys in ConstructCoreset2, which is updated whenever data arrives. With the sorted list, we compute the difference between adjacent buckets only, not all pairs. We organize the key pairs according to the difference in the computed result, and the key pairs are merged in order of a small difference.

Fig. 6 shows an example of merging buckets. The keys are sorted, and their differences between adjacent buckets are computed. For example, if we calculate the difference by the Manhattan distance, then the difference between the first and second buckets of the left hash table is 3, while the second and third is 1. After calculating the key differences between adjacent buckets, we iterate to merge adjacent two buckets with the smallest difference into one bucket. For example, we may begin to search for any adjacent buckets of one difference, and if found, they are merged. In the figure, the second and third buckets are merged into a bucket that now contains $\{x_1, x_2, x_4, x_{11}, x_6, x_8\}$. Then the fourth and fifth buckets have the next smallest difference, and they are merged to contain $\{x_3, x_7, x_{12}, x_9, x_{10}\}$. After merging, the differences are computed again for newly adjacent buckets. For example, the difference between the second and third buckets of the right hash table is four. If the reduced hash table size is greater than m, this entire process iterates until the desired size is reached.

In our implementation of hash table, we employ a multi-linked list in order to traverse the adjacent buckets with the same difference in order of the smallest value first. Each bucket represents a GF, and it also stores the statistic summaries of the grouped tuples, such as LS, SS, N, and T, as they are explained in Section III-E. In addition, we do not physically remove a bucket when merging it into its adjacent one, but just reset the link pointers of the buckets in order to avoid the time for dynamically changing the hash table. For detailed information on the hash table data structure, interested readers may refer to [1].

VI. RE-CLUSTERING POLICY

We perform weighted k-means clustering using the weighted centroids of the GFs in the coreset. The centroid is computed by LS/N, and N is used as the weight. There is no additional computation cost to computing the centroids. We utilize *Lloyd'* s algorithm for clustering, where the distance between center and GF_x is $w(GF_x)\|c - GF_x\| = N_x\|c - LS_x/N_x\|$.

In order to reduce the total computational cost of clustering, we include a re-clustering policy in our algorithm so as to determine whether if we may append a newly created GFs to pre-existing clusters while maintaining the quality of clusters at a certain degree. For example, let us consider Fig. 3. Suppose that three clusters of the first window were $C_1 = \{GF_1, GF_4\}$, $C_2 = \{GF_3\}$, and $C_3 = \{GF_2, GF_5\}$. If the newly appended tuples, x_{10}, x_{11}, and x_{12} at time t_2, are close enough to the centers of the previous three clusters, then it is more efficient to assign each tuple to one of the closest clusters among the previous C_1, C_2, C_3. The resulting clusters of the second window will be $C_1 = \{x_4, x_6, x_8, x_{11}\}$, $C_2 = \{x_7, x_9, x_{10}, x_{12}\}$, and $C_3 = \{x_5\}$.

Our algorithm exploits GFs for clustering. Upon the arrival of the new tuples, their nearest GFs instead of the centers of the clusters are searched for. The new tuples are assigned

to the nearest GFs and those GFs are updated with the tuples. Because of the incrementality and additivity of GF, the update can be done in a constant time.

An issue faced by a re-clustering policy is how to measure a possible degradation of cluster quality that we would encounter if we did not re-cluster. In other words, clustering should be performed on the whole GFs if the quality is expected to drop below a certain level. We employ the probability distributions of clusters in order to measure the quality. A detailed algorithm for clustering with the LSH-based coreset is similar to that in [2]. The previous algorithm also seeks for a possibility to append newly created GFs to pre-existing clusters to avoid repeated clustering. Unlike the previous algorithm, the algorithm in this paper exploits a reduced coreset which is created by Algorithm 3. We assumed that the input data distribution follows the Gaussian distribution in our previous work. However, since this assumption does not always apply to every dataset, we propose a new probability distribution for measuring the clustering quality.

We use *Kullback-Leibler divergence* [49] to measure the quality degeneration of the original and modified clusters.

$$D_{KL}(p(x)||q(x)) = \sum_{x} p(x) \log \frac{p(x)}{q(x)}.$$

The *KL-divergence* of the probability distributions $p(x)$ and $q(x)$ is a measure of the information gain achieved in the case that $p(x)$ is used instead of $q(x)$.

To compute the *KL-divergence*, we model the quality degradation by the location of the cluster center, which changes by the appended data. If the appended data fits the cluster well, the cluster center will not change significantly. If the appended data is very different from the data that already exists in the cluster, a new cluster center computed by including the appended data will also move substantially from its original location.

Usually, the quality of the clusters is measured by the sum of squared distance, which is presented in Definition 1. However, it is not suitable as a quality measure because the optimal centers are unknown, and the range of values varies depending on the data. Instead, we define the probability mass function for the probability distribution of a cluster as the ratio of the squared distance from the center to a tuple. The base concept is introduced in *k-means++* [38], and we modify it for the re-clustering policy. Specifically, the function for x in cluster P is defined as

$$p(x) = \frac{\|x - c_x\|^2}{\sum_{y \in P} \|y - c_x\|^2}.$$

Then, the probability distribution for the changed cluster $q(x)$ is computed using the distance from the new center. Based on the definition, the error of the modified clusters is obtained by averaging the *KL-divergence* of each cluster. Note that we maintain only the GFs, so the weighted distance using GF is used.

Error bound ε is required for clustering in order to adjust how much the algorithm tolerates the error. The value is determined experimentally. Lower values allow for more frequent clustering. While performing clustering, it is easy to generate and maintain the cluster statistics. GF maintains the distances from the original cluster centers and the distances from the modified cluster centers. When a cluster is modified, the statistics are updated in $O(km)$ time, where k is the number of clusters, and m is the number of GFs.

VII. EXPERIMENT

A. EXPERIMENTAL SETUP

We evaluated the efficiency and scalability of our clustering algorithm using synthetic and real-world datasets.

GFCS(Group Features based Clustering with Sliding windows) is a basic approach which was presented in [2]. *GFCS* also maintains panes for the window and creates GFs based on a predefined threshold. *GFCS* exploits the simplified LSH in order to find the nearest GF, but there is no reduction step in *GFCS*. *CSCS*(CoreSet based Clustering with Sliding windows) is the improved algorithm based on LSH presented in this paper. We compared our algorithms with recent data stream clustering algorithms, *SWClustering* [4], *StreamKM++* [6], *ClusTree* [16], and *G2CS* [28], each of which is explained in Section II. We also measured the performance of a basic k-means clustering on raw tuples, which is implemented by *Lloyd*'s algorithm [5].

All algorithms were implemented by Java. *StreamKM++* and *ClusTree* were implemented based on the MOA framework [50]. We implemented *GFCS*, *CSCS*, *SWClustering*, *G2CS* and *Lloyd*'s algorithm from scratch in Java. For proper comparison, G2CS is implemented except for the lattice generation. We executed all experiments with 64-Bit OpenJDK 1.8.0_91 on Intel i7-3820 3.60GHz CPU and 32GB main memory using Linux 4.4.0-43 kernel. The maximum Java heap size (-Xmx option) is set to 8GB.

TABLE 2. Real-world datasets.

Dataset	Size	Dimension
kddcup99	4,898,431	34
covtype	581,012	54
tower	4,915,200	3
census1990	2,458,285	68

TABLE 3. Synthetic datasets.

Dataset	Size	Dimension
syn1k30d40	2,000,000	40
syn1k30d80	2,000,000	80
syn1k30d160	2,000,000	160
syn1k30d200	2,000,000	200

Table 2 and Table 3 show real-world and synthetic datasets for the experiments. For synthetic data, we generate data which follow the Gaussian distribution and have

30 clusters with dimensions of 40, 80, 160, and 200, respectively. For example, synlk30d40 contains data with 40 dimensions. Kddcup99 contains network data streams to detect network intrusion. Kddcup99 contains logs of TCP connection of the network at MIT Lincoln Labs for 2 weeks. This dataset is used to evaluate the clustering algorithms in [3]. Covtype contains cartographic data from the Roosevelt National Forest of northern Colorado. Tower consists of RGB values of an image file. Census1990 contains personal records sampled from the 1990 U.S. census data. Covtype and census1990 are used in *StreamKM++*. The datasets of kddup99,[1] covtype,[2] and census1999[3] come from the UCI Machine Learning Repository [51]. Tower dataset[4] is from [52]

In order to evaluate the efficiency and scalability, we measure the total processing time and processing time for each sliding window. In order to evaluate the quality of the clusters, we measure the sum of squared distance (SSQ) of the clusters, which is the k-means cost. SSQ is defined as $\sum \|s_i - c_i\|^2$, which means the sum of squared distance between each tuple and their nearest cluster center. The lower SSQ value indicates a better quality of clusters. In the sliding windows, the algorithm produces multiple results as the window moves. Therefore, the quality is evaluated by averaging the SSQ of the results. Note that there are other quality measures such as purity, normalized mutual information, or rand index. However, they can be precisely measured when each tuple is class-labeled. Furthermore we use the k-means as a baseline algorithm to compare the performance of different algorithms, and SSQ itself as a single quality measure works fine. Other works such as *SWClustering* have also taken this approach in their experiments.

B. EXPERIMENTAL RESULTS

For *CSCS*, we set the number of hash functions $l = 15$, and the coreset size $m = 10000$, error bound $\varepsilon = 0.2$, if not mentioned explicitly. For *GFCS*, we set threshold θ as $\theta_{kddcup99} = 60$, $\theta_{census1990} = 9$, $\theta_{covtype} = 75$, and $\theta_{tower} = 5$.

For the parameters of the competitive algorithms, a coreset size of *StreamKM++* was set to $200k$, and and the maximal height of the tree of *ClusTree* is set to 10. For *SWClustering*, the values of threshold θ is equal to *GFCS*, and $\epsilon = 0.05$ which limits the number of expired records.

The clustering quality and speed are in a trade-off with the values. We tested several parameters and error bounds, and selected the values that generated the best clustering quality. The parameters of the competitive algorithms are based on values showing good quality for the same datasets by the authors. Since the other algorithms, except for *SWClustering*, do not support sliding operation, we repeatedly ran clustering algorithms on tuples that are within the range of the window.

[1] http://archive.ics.uci.edu/ml/datasets/kdd+cup+1999+data
[2] https://archive.ics.uci.edu/ml/datasets/covertype
[3] https://archive.ics.uci.edu/ml/datasets/US+Census+Data+%281990%29
[4] http://homepages.uni-paderborn.de/frahling/instances/Tower.txt

Fig. 7 shows the clustering quality of the algorithms for different values of k with real-world datasets. We set RANGE = 100,000 and SLIDE = 10,000. The basic *k-means* in the experiment shows the best quality because it performs clustering directly on the input tuples, and not on any summarization of the tuples. Overall, we observe that our algorithm *CSCS* loses some accuracy, but is comparable to other algorithms such as *StreamKM++* and *ClusTree*. For the kddcup99 dataset, *CSCS* shows the best quality except *k-means*, but G2CS outperforms our previous algorithm *GFCS* as well as other competitors. For census1990 and tower datasets, *StreamKM++* shows the good quality. This seems that the algorithm works well for finding the good initial centers of k-means. However, finding the good initial centers is a very time-consuming task. Fig. 8 shows that *StreamKM++* takes the longest time in census1990 dataset.

In terms of the processing time, our algorithm shows the best performance. It also scales up the best as the window size increases. In Fig. 8, we measure the average processing time of each algorithm for the different window sizes, where the size value is given by RANGE. We set $k = 30$, and SLIDE = 10,000. We tested only datasets with sufficient quantities and large dimensions, which are kddcup99 and census1990. We also used synthetic datasets to test larger dimensions.

As the window size increases, the processing times of the other algorithms increase as well. For both datasets and all window sizes, *CSCS* shows the fastest average processing time. On average, it runs on the kddcup99 dataset 2501%, 507%, 1576%, 860%, 1208% faster than *SWClustering*, *ClusTree*, *StreamKM++*, *G2CS*, and *k-means*, respectively (Fig. 8a). The improvement obviously increases with the size of the window. It also exhibits the similar improvement on the average processing time on the census1990 dataset (Fig. 8b). On average, it runs 920%, 641%, 2840%, 1343%, 1487% faster than *SWClustering*, *ClusTree*, *StreamKM++*, *G2CS*, and *k-means*, respectively.

GFCS, our previous work, follows the next performance in all cases. Both algorithms scale up well as the window sizes increase, in that the processing time they require for bigger size window does not increase as much as the others do. In the meanwhile, *ClusTree* is the third fastest algorithm, while its processing time increases more rapidly than *CSCS* or *GFCS* as the window size increases.

Fig. 9 shows the processing times at each timestamp when the window slides. We set $k = 30$, RANGE = 100,000, SLIDE = 10,000, and the number of tuples = 200,000. Our algorithms of *CSCS* and *GFCS* are both stable and fast. On average, *CSCS* runs on the kddcup99 dataset 2208%, 540%, 1700%, 619%, and 1187% faster than *SWClustering*, *ClusTree*, *StreamKM++*, *G2CS*, and *k-means*, respectively. *CSCS* also runs on average 152% faster than *GFCS*. *ClusTree* shows the third best performance, and *SWClustering* shows the lowest performance (Fig. 9a and 9c). Since *k-means* is a randomized algorithm, the processing time fluctuates with the data distribution. However, *k-means* shows the proper performance, meaning that the

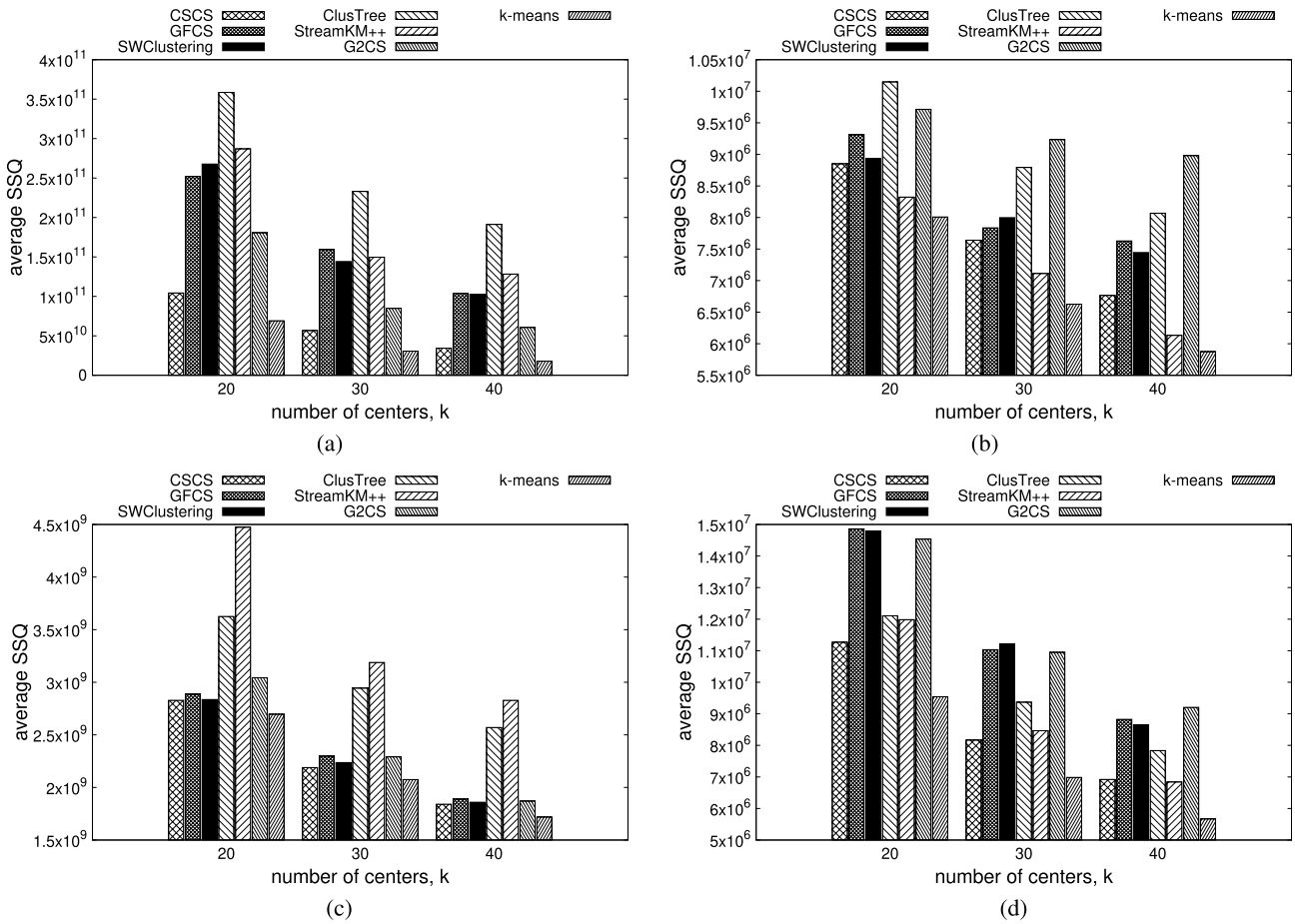

FIGURE 7. Clustering quality comparison with real-world datasets. (a) kddcup99. (b) census1990. (c) covtype. (d) tower.

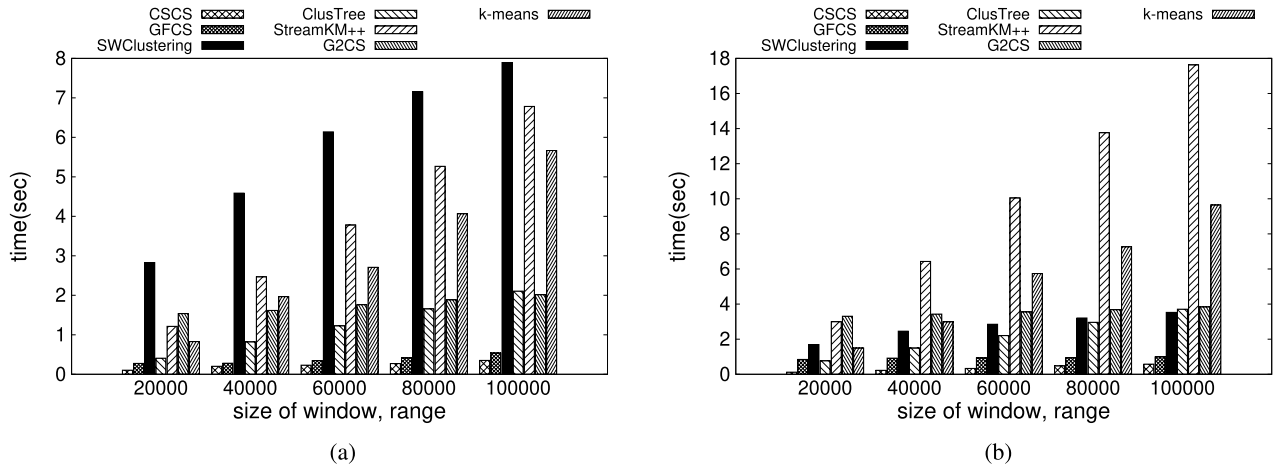

FIGURE 8. Processing time of a window with specific size. (a) kddcup99. (b) census1990.

cost of constructing additional synopses of the streaming algorithms is quite high. Fig. 9b and 9d shows only the results of *GFCS* and *CSCS*. Their processing times can vary depending on the characteristics of the dataset. For example, *CSCS* outperforms *GFCS* on a dense dataset census1990 consistently and much more than it does on a less dense dataset kddcup99.

In order to investigate the impact of the coreset size and the number of hash functions, we conducted experiments on synthetic datasets. Fig. 10 and 11 show the clustering

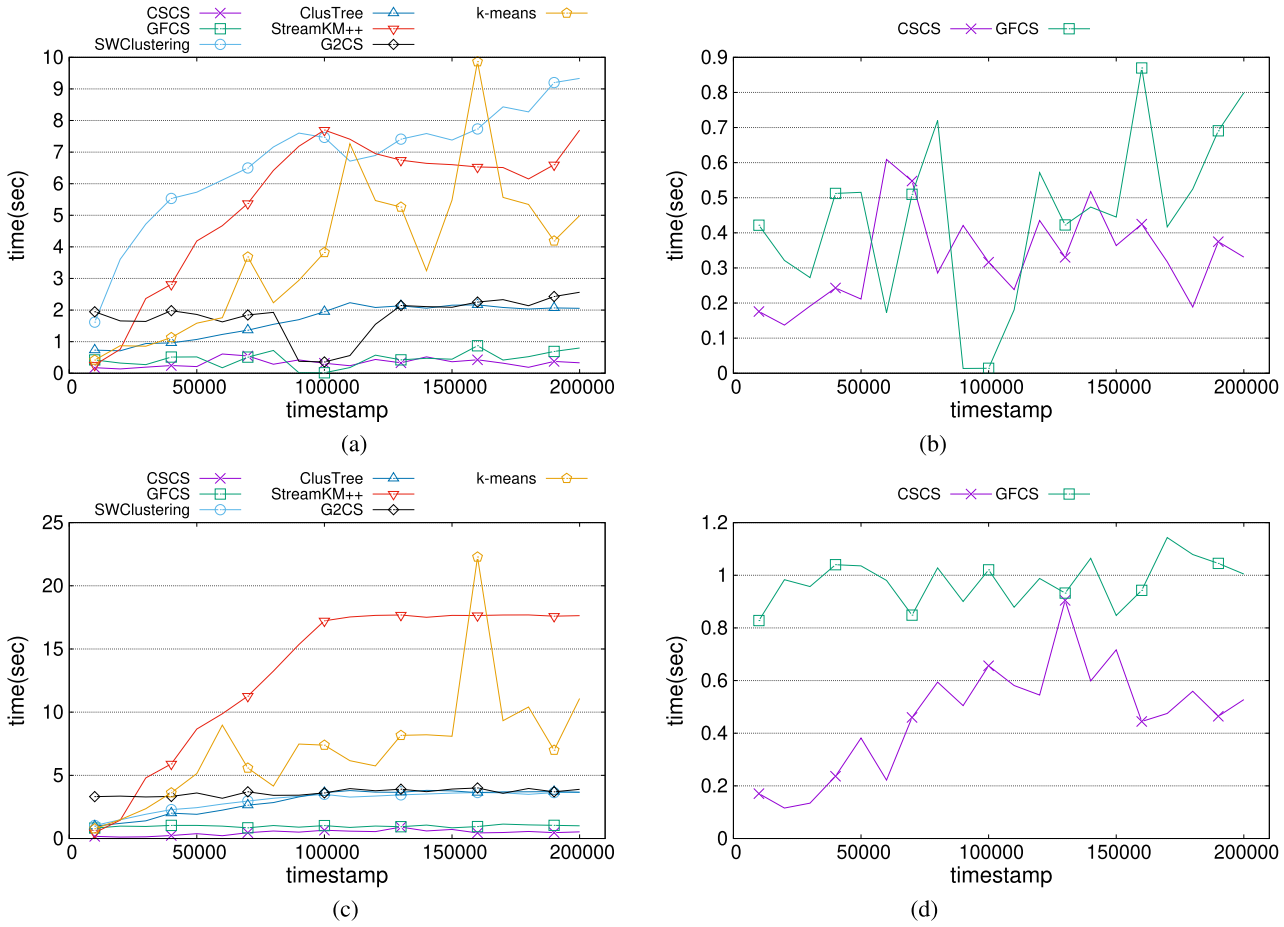

FIGURE 9. Processing time at each timestamp. (a) kddcup99. (b) kddcup99 (CSCS & GFCS). (c) census1990. (d) census1990 (CSCS & GFCS).

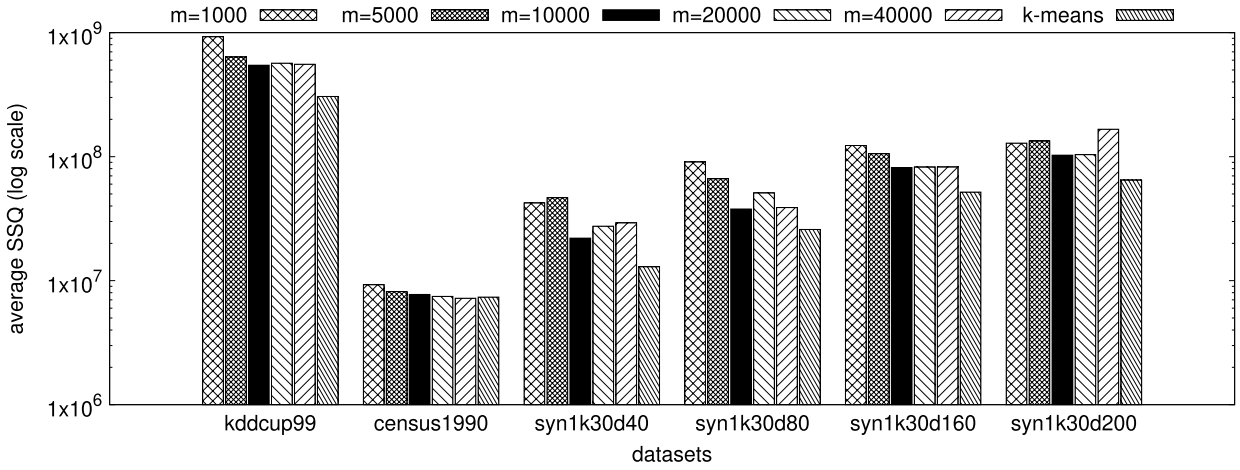

FIGURE 10. Clustering quality by coreset size.

quality and execution time as the coreset size changes. We set RANGE = 100,000 and SLIDE = 10,000, and measured the performance for the datasets by changing the size from 1000 to 40000. Therefore, k-means clustering was performed with a coreset of sizes from 1/100 to 40/100 of the original tuples. The average SSQ is converted to a logarithmic scale so as to present multiple results together in one figure, and the results of kddcup99 are divided by 100 for the same reason.

In terms of the clustering quality, *CSCS* shows good performance with a coreset of 1/10 size, and the performance with a coreset of 1/100 size was not

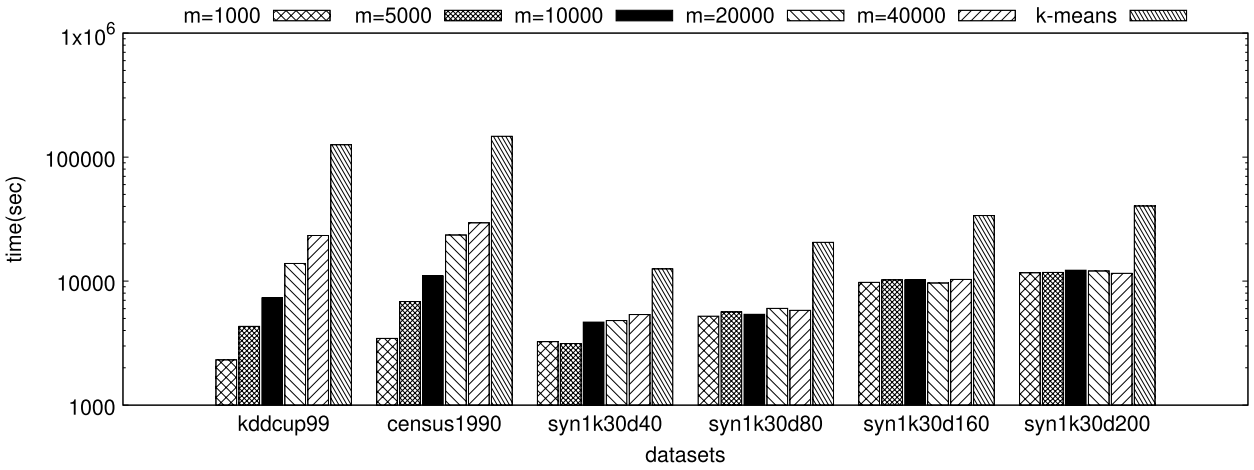

FIGURE 11. Processing time by coreset size.

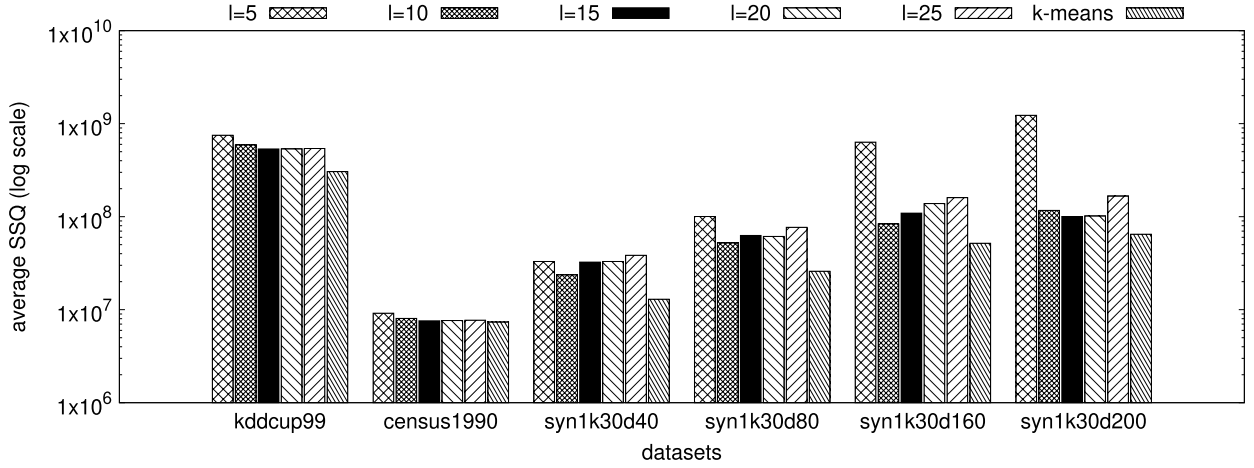

FIGURE 12. Clustering quality by the number of hash functions.

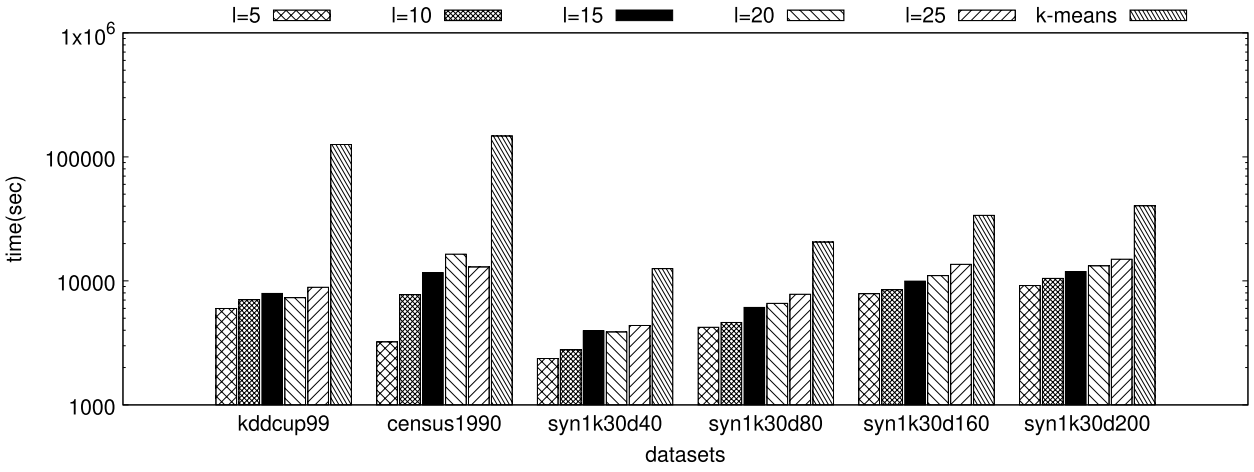

FIGURE 13. Processing time by the number of hash functions.

significantly lowered. The results of the synthetic datasets also show a similar pattern. The reason for the observed good performance in the mid-size coreset is the effect of removing the outliers. In a large-sized coreset, the number of noise *GF*s increases because the reduction process occurs less frequently.

In terms of time, a large size coreset takes more time to process. For synthetic datasets, *CSCS* is the fastest in a small dimension and small coreset. The processing time increases as the values of the parameters increase. In particular, for high-dimensional data, the size of the coreset does not affect the processing time.

Lastly, we also investigated the effect of the number of hash functions on the performance. We changed the number of hash functions from five to 25 and measured the quality and processing time of the algorithm. The results are shown in Fig. 12 and 13. For real-world datasets, the clustering quality is improved as the number of hash functions increases, but the improvement seems less effective as the number grows more than 15. For synthetic datasets, the quality is best at 10 hash functions. It also shows that a very small number of hash functions such as five is not suitable for high-dimensional data. At the same time, the processing time increases consistently as the number of hash functions increases, due to their being more calculations of hash values and distances.

According to the results, the factors which affect the processing time are the dimension of the data and the number of hash functions. On the other hand, we find that increasing the size of coreset does not affect the processing time. This is because the time complexity of the construction and the reduction of coreset is linear to the input size. Based on experimental results, we conclude that our algorithm shows good performance in terms of the quality and processing time. In particular, our algorithm is very effective for high-dimensional data.

VIII. CONCLUSION

In this paper, we developed an efficient algorithm for data stream clustering over sliding windows. In the grouping step, we presented an aggregation technique for the sliding window model, which divides the window into disjoint chunks and generates the overall coreset by merging partial coresets. LSH is utilized for efficient coreset construction. In the clustering step, the algorithm performs clustering on group features in the coreset. The re-clustering policy of modifying clusters was proposed in order to avoid unnecessary clustering. Our approach has an advantage over recent algorithms in that it performs clustering on entire data streams because it provides the functionality of tracking changes in the data streams by generating snapshots of the clusters using less computational power.

Our algorithm focuses on the k-means problem, but k-means involves several problems. Specifically, a fixed number of clusters should be specified before clustering, and it is difficult to generate clusters for arbitrary shapes. In order to overcome these problems, various clustering algorithms were developed, including density-based clustering and message passing based clustering. However, few studies have examined data streams with sliding windows. In future research efforts, we will extend our technique to other clustering algorithms.

ACKNOWLEDGMENT

A preliminary version of this paper was presented at the 22nd DASFAA Conference, Suzhou, China, in 2017 [2]. This paper is based on the first author's dissertation [1] at Seoul National University. This paper is a fully-rewritten extended version that features LSH-based coreset construction, mathematical formalization and theoretical analysis of the algorithm, as well as extensive experiments with new datasets.

REFERENCES

[1] J. Youn, "A scalable clustering algorithm for high-dimensional data streams over sliding windows," Ph.D. dissertation, Dept. Comput. Sci., Seoul Nat. Univ., Seoul, South Korea, 2017.

[2] J. Youn, J. Choi, J. Shim, and S.-G. Lee, "Partition-based clustering with sliding windows for data streams," in *Proc. 22nd Int. Conf. Database Syst. Adv. Appl.* Cham, Switzerland: Springer, 2017, pp. 289–303.

[3] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu, "A framework for clustering evolving data streams," in *Proc. 29th Int. Conf. Very Large Data Bases*, 2003, pp. 81–92.

[4] A. Zhou, F. Cao, W. Qian, and C. Jin, "Tracking clusters in evolving data streams over sliding windows," *Knowl. Inf. Syst.*, vol. 15, no. 2, pp. 181–214, 2008.

[5] S. Lloyd, "Least squares quantization in PCM," *IEEE Trans. Inf. Theory*, vol. IT-28, no. 2, pp. 129–137, Mar. 1982.

[6] M. R. Ackermann, M. Märtens, C. Raupach, K. Swierkot, C. Lammersen, and C. Sohler, "Streamkm++: A clustering algorithm for data streams," *J. Exp. Algorithmics*, vol. 17, Jul. 2012, Art no. 2.4.

[7] B. Babcock, M. Datar, R. Motwani, and L. O'Callaghan, "Maintaining variance and k-medians over data stream windows," in *Proc. 22nd ACM SIGMOD-SIGACT-SIGART Symp. Princ. Database Syst.*, 2003, pp. 234–243.

[8] S. Har-Peled and S. Mazumdar, "On coresets for k-means and k-median clustering," in *Proc. 36th Annu. ACM Symp. Theory Comput.*, 2004, pp. 291–300.

[9] Y. Chen and L. Tu, "Density-based clustering for real-time stream data," in *Proc. 13th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2007, pp. 133–142.

[10] F. Cao, M. Ester, W. Qian, and A. Zhou, "Density-based clustering over an evolving data stream with noise," in *Proc. SIAM Conf. Data Mining*, 2006, pp. 328–339.

[11] L. Wan, W. K. Ng, X. H. Dang, P. S. Yu, and K. Zhang, "Density-based clustering of data streams at multiple resolutions," *ACM Trans. Knowl. Discovery Data*, vol. 3, no. 3, 2009, Art. no. 14.

[12] X. Zhang, C. Furtlehner, C. Germain-Renaud, and M. Sebag, "Data stream clustering with affinity propagation," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 7, pp. 1644–1656, Jul. 2014.

[13] L. Sun and C. Guo, "Incremental affinity propagation clustering based on message passing," *IEEE Trans. Knowl. Data Eng.*, vol. 26, no. 11, pp. 2731–2744, Nov. 2014.

[14] T. Zhang, R. Ramakrishnan, and M. Livny, "BIRCH: An efficient data clustering method for very large databases," *ACM SIGMOD Rec.*, vol. 25, pp. 103–114, Jun. 1996.

[15] S. Guha, A. Meyerson, N. Mishra, R. Motwani, and L. O'Callaghan, "Clustering data streams: Theory and practice," *IEEE Trans. Knowl. Data Eng.*, vol. 15, no. 3, pp. 515–528, Jun. 2003.

[16] P. Kranen, I. Assent, C. Baldauf, and T. Seidl, "The ClusTree: Indexing micro-clusters for anytime stream mining," *Knowl. Inf. Syst.*, vol. 29, no. 2, pp. 249–272, 2011.

[17] J. Gray *et al.*, "Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals," *Data Mining Knowl. Discovery*, vol. 1, no. 1, pp. 29–53, 1997.

[18] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni, "Locality-sensitive hashing scheme based on p-stable distributions," in *Proc. 20th Annu. Symp. Comput. Geometry*, 2004, pp. 253–262.

[19] J. A. Silva, E. R. Faria, R. C. Barros, E. R. Hruschka, A. C. De Carvalho, and J. Gama, "Data stream clustering: A survey," *ACM Comput. Surv.*, vol. 46, no. 1, p. 13, 2013.

[20] X. H. Dang, V. Lee, W. K. Ng, A. Ciptadi, and K. L. Ong, *An EM-Based Algorithm for Clustering Data Streams Sliding Windows*. Berlin, Germany: Springer, 2009, pp. 230–235.

[21] V. Braverman, H. Lang, K. Levin, and M. Monemizadeh, "Clustering problems on sliding windows," in *Proc. 27th Annu. ACM-SIAM Symp. Discrete Algorithms*, 2016, pp. 1374–1390.

[22] P. S. Bradley, U. Fayyad, and C. Reina, "Scaling clustering algorithms to large databases," in *Proc. 4th Int. Conf. Knowl. Discovery Data Mining*. New York, NY, USA: AAAI Press, 1998, pp. 9–15.

[23] S. Guha, N. Mishra, R. Motwani, and L. O'Callaghan, "Clustering data streams," in *Proc. 41st Annu. Symp. Found. Comput. Sci.*, Nov. 2000, pp. 359–366.
[24] L. O'Callaghan, N. Mishra, A. Meyerson, S. Guha, and R. Motwani, "Streaming-data algorithms for high-quality clustering," in *Proc. 18th Int. Conf. Data Eng.*, Mar. 2002, pp. 685–694.
[25] J. Gama, P. P. Rodrigues, and L. Lopes, "Clustering distributed sensor data streams using local processing and reduced communication," *Intell. Data Anal.*, vol. 15, no. 1, pp. 3–28, 2011.
[26] M. Ackerman and S. Dasgupta, "Incremental clustering: The case for extra clusters," in *Proc. Adv. Neural Inf. Process. Syst.* Red Hook, NY, USA: Curran Associates, 2014, pp. 307–315.
[27] R. Langone, M. V. Barel, and J. A. K. Suykens, "Efficient evolutionary spectral clustering," *Pattern Recognit. Lett.*, vol. 84, pp. 78–84, Dec. 2016.
[28] S. Badiozamany, K. Orsborn, and T. Risch, "Framework for real-time clustering over sliding windows," in *Proc. 28th Int. Conf. Sci. Stat. Database Manage.*, 2016, Art. no. 19.
[29] S. T. Mai, I. Assent, and A. Le, "Anytime OPTICS: An efficient approach for hierarchical density-based clustering," in *Proc. Int. Conf. Database Syst. Adv. Appl.*, 2016, pp. 164–179.
[30] L. Tu and Y. Chen, "Stream data clustering based on grid density and attraction," *ACM Trans. Knowl. Discovery Data*, vol. 3, no. 3, 2009, Art. no. 12.
[31] C. Isaksson, M. H. Dunham, and M. Hahsler, "SOStream: Self organizing density-based clustering over data stream," in *Proc. Int. Workshop Mach. Learn. Data Mining Pattern Recognit.*, 2012, pp. 264–278.
[32] V. Bhatnagar, S. Kaur, R. Saxena, and D. Khanna, "DASC: Data aware algorithm for scalable clustering," *Knowl. Inf. Syst.*, vol. 50, no. 3, pp. 851–881, 2017.
[33] M. Hahsler and M. Bolaños, "Clustering data streams based on shared density between micro-clusters," *IEEE Trans. Knowl. Data Eng.*, vol. 28, no. 6, pp. 1449–1461, Jun. 2016.
[34] S. Ding, J. Zhang, H. Jia, and J. Qian, "An adaptive density data stream clustering algorithm," *Cognit. Comput.*, vol. 8, no. 1, pp. 30–38, 2016.
[35] R. Hyde, P. Angelov, and A. R. MacKenzie, "Fully online clustering of evolving data streams into arbitrarily shaped clusters," *Inf. Sci.*, vols. 382–383, pp. 96–114, Mar. 2017.
[36] L. Zheng, H. Huo, Y. Guo, and T. Fang, "Supervised adaptive incremental clustering for data stream of chunks," *Neurocomputing*, vol. 219, pp. 502–517, Jan. 2017.
[37] C. Yang, L. Bruzzone, R. Guan, L. Lu, and Y. Liang, "Incremental and decremental affinity propagation for semisupervised clustering in multispectral images," *IEEE Trans. Geosci. Remote Sens.*, vol. 51, no. 3, pp. 1666–1679, Mar. 2013.
[38] D. Arthur and S. Vassilvitskii, "K-means++: The advantages of careful seeding," in *Proc. 18th Annu. ACM-SIAM Symp. Discrete Algorithms*. Philadelphia, PA, USA: SIAM, 2007, pp. 1027–1035.
[39] B. J. Frey and D. Dueck, "Clustering by passing messages between data points," *Science*, vol. 315, no. 5814, pp. 972–976, Feb. 2007.
[40] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A density-based algorithm for discovering clusters in large spatial databases with noise," in *Proc. 2nd Int. Conf. Knowl. Discovery Data Mining*. Portland, Oregon: AAAI Press, 1996, pp. 226–231.
[41] J. Li, D. Maier, K. Tufte, V. Papadimos, and P. A. Tucker, "No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams," *ACM SIGMOD Rec.*, vol. 34, no. 1, pp. 39–44, 2005.
[42] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: Semantic foundations and query execution," *VLDB J.*, vol. 15, no. 2, pp. 121–142, 2006.
[43] J. MacQueen, "Some methods for classification and analysis of multivariate observations," in *Proc. 5th Berkeley Symp. Math. Statist. Probab.*, 1967, pp. 281–297.
[44] D. Aloise, A. Deshpande, P. Hansen, and P. Popat, "NP-hardness of Euclidean sum-of-squares clustering," *Mach. Learn.*, vol. 75, no. 2, pp. 245–248, 2009.
[45] G. Rosman, M. Volkov, D. Feldman, J. W. Fisher, III, and D. Rus, "Coresets for k-segmentation of streaming data," in *Proc. Adv. Neural Inf. Process. Syst.*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Red Hook, NY, USA: Curran Associates, Inc., 2014, pp. 559–567.
[46] G. Frahling and C. Sohler, "Coresets in dynamic geometric data streams," in *Proc. 37th Annu. ACM Symp. Theory Comput.*, 2005, pp. 209–217.
[47] S. Har-Peled and A. Kushal, "Smaller coresets for k-median and k-means clustering," in *Proc. 21st Annu. Symp. Comput. Geometry*, 2005, pp. 126–134.
[48] P. Awasthi, M. Charikar, R. Krishnaswamy, and A. K. Sinop, "The hardness of approximation of Euclidean k-means," in *Proc. 31st Int. Symp. Comput. Geometry (SoCG)*, in Leibniz International Proceedings in Informatics, vol. 34. S. Dagstuhl, ed. Dagstuhl, Germany: Leibniz Center for Informatics, 2015, pp. 754–767.
[49] S. Kullback and R. A. Leibler, "On information and sufficiency," *Ann. Math. Statist.*, vol. 22, no. 1, pp. 79–86, 1951.
[50] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "MOA: Massive online analysis," *J. Mach. Learn. Res.*, vol. 11, pp. 1601–1604, May 2010.
[51] D. Dheeru and E. K. Taniskidou. (2017). *UCI Machine Learning Repository*. [Online]. Available: http://archive.ics.uci.edu/ml
[52] G. Frahling and C. Sohler, "A fast k-means implementation using coresets," in *Proc. 22nd Annu. Symp. Comput. Geometry*, 2006, pp. 135–143.

JONGHEM YOUN received the B.S. and Ph.D. degrees in computer science and engineering from Seoul National University, South Korea, in 2008 and 2017, respectively. He is currently an Analytics Lead with Voost Inc. His research interests include recommendation systems, natural language processing, deep learning, and blockchain technology.

JUNHO SHIM (M'14–SM'17) received the B.S. and M.S. degrees from Seoul National University, South Korea, in 1990 and 1994, respectively, and the Ph.D. degree in computer science from Northwestern University, USA, in 1998. He was with Computer Associates International. He was an Assistant Professor with Drexel University, USA. He is currently a Professor with the Department of Computer Science, Sookmyung Women's University, South Korea. He has authored over 60 refereed papers at journals and conferences. His research interests include big data, database systems, e-commerce technology, and the Web. He is a Senior Member of the IEEE Computer Society. He served as a Committee Member for internationally renowned conferences including ICDE2015, MDM2016, WWW2014, and SIGMOD2016.

SANG-GOO LEE received the B.S. degree in computer science and statistics from Seoul National University, South Korea, in 1985, and the M.S. and Ph.D. degrees in computer science from Northwestern University, Evanston, IL, USA, in 1987 and 1990, respectively. Since 2014, he has been an Associate Director of the Big Data Institute, Seoul National University, where he is currently a Professor with the Department of Computer Science and Engineering. His research interests include database systems, big data technology, natural language processing, technology-driven learning, and recommendation systems.