Ph.D. DISSERTATION

A Scalable Clustering Algorithm for High-dimensional Data Streams over Sliding Windows

슬라이딩 윈도우 상의 고차원 데이터 스트림을 위한 클러스터링 알고리즘

BY

Jonghem Youn

AUGUST 2017

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
COLLEGE OF ENGINEERING
SEOUL NATIONAL UNIVERSITY

Abstract

Data stream clustering over sliding windows generates clustering results whenever a window moves. However, iterative clustering using all data in a window is highly inefficient in terms of memory and computation time. In this thesis, we address problem of 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. The technique divides a window into disjoint chunks, computes partial aggregates over each chunk, and merges the partial aggregates to compute overall aggregates. To maintain constant size of the summary, the algorithm reduces the size of summary by joining the nearest neighbor. We exploit Locality-Sensitive Hashing for fast nearest neighbor search. We show that Locality-Sensitive Hashing can serve as an effective method for reducing synopses while minimizing the impact on quality. In addition, we also suggest re-clustering policy, which decides whether to append new summary to pre-existing clusters or to perform clustering on whole summary. Our experiments on real-world and synthetic datasets demonstrate that our algorithm can achieve a significant improvement when performing continuous clustering on data streams with sliding windows.

keywords: clustering, data streams, sliding windows, real-time processing
student number: 2008-20919

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Large-scale data streams are generated from a variety of applications such as social media, news feeds, sensor networks, transportation monitoring, and smart devices. In traditional database management systems (DBMS), data should be stored before processing. This way of processing is suitable for applications where read operations occur more frequently than insert or update operations. However, data streams are massive, rapidly evolving, and infinitely created, thus it is impractical to process a large amount of data streams with accessing multiple times after storing.

Clustering to summarize data streams is an important mining task because the amount of data is too large to analyze or retrieve overall information. Clustering is a technique to group multi-dimensional tuples to maximize intra-cluster similarity and minimize inter-cluster similarity. Clustering on data streams involves not only creating clusters but also tracking the evolution of individual clusters.

For example, in news website, news articles are published on the website. For tracking topics, similar news articles are grouped by clustering on a **window** of a recent news articles. Figure 1.1 shows the example for clustering with sliding windows. Over time, new clusters are created, and old clusters disappear.

As shown in the example, clustering is performed iteratively whenever window changes. The objective is to make clusters based on the statistics over recently observed tuples. Sliding window model is introduced for the objective. Tuples arrive continuously, and each tuple expires after pre-defined time period. The target to perform operations is the set of most recent tuples. Performing clustering once is a computationally intensive task. The main objective of data stream clustering with sliding windows is to perform required operations using limited memory in near real time. Therefore, the clustering algorithm is designed by considering the computation time and memory usage.

As shown in Figure 1.2, the general procedure for data stream clustering consists of two steps: grouping step and clustering step [3, 68]. The grouping step summarizes the original data into specific data structures, called synopses, to reduce memory usage. The synopses are used to grasp the semantics of the original data from the summarization without actually storing the entire data.

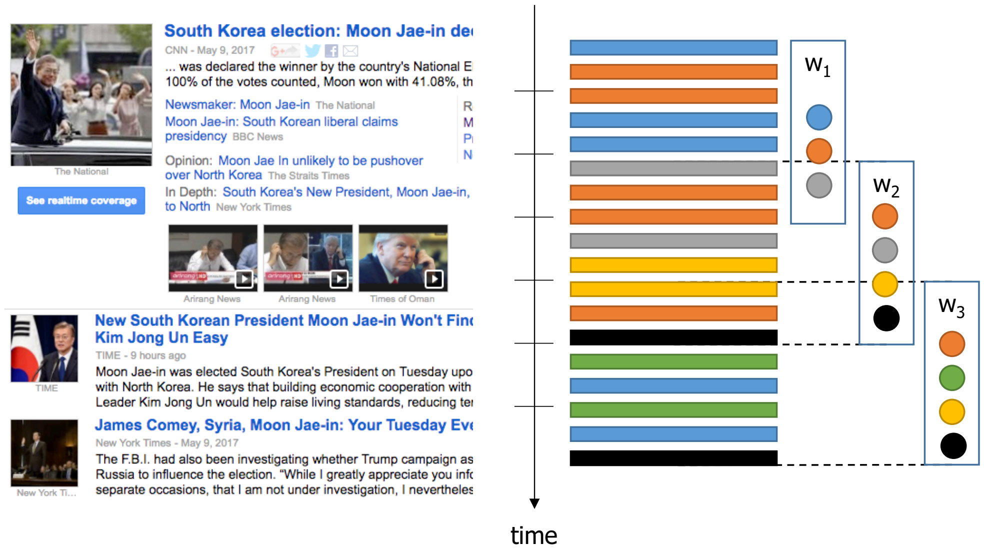

South Korea election: Moon Jae-in de
CNN - May 9, 2017
... was declared the winner by the country's National E
100% of the votes counted, Moon won with 41.08%, th
Newsmaker: Moon Jae-in The National
Moon Jae-in: South Korean liberal claims presidency BBC News
Opinion: Moon Jae In unlikely to be pushover over North Korea The Straits Times
In Depth: South Korea's New President, Moon Jae-in, to North New York Times
The National
See realtime coverage
Arirang News Arirang News Times of Oman
New South Korean President Moon Jae-in Won't Fin Kim Jong Un Easy
TIME - 9 hours ago
Moon Jae-in was elected South Korea's President on Tuesday upo with North Korea. He says that building economic cooperation with Leader Kim Jong Un would help raise living standards, reducing te
TIME
James Comey, Syria, Moon Jae-in: Your Tuesday Eve
New York Times - May 9, 2017
The F.B.I. had also been investigating whether Trump campaign as Russia to influence the election. "While I greatly appreciate you inf separate occasions, that I am not under investigation, I neverthele
New York Ti...

Figure 1.1: Clustering with sliding windows for news articles

Generally, the grouping step constructs synopses through particular heuristic methods in linear time complexity. The clustering step performs clustering on the synopses generated through grouping. Various clustering algorithms, such as k-means [2, 47], k-median [9, 39], DBSCAN [18, 20, 61], and affinity propagation [59,66] are employed for finding 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. If the synopses are of fixed size, the clustering can be performed in constant time. In this case, the processing time of the algorithm is only dependent on the time required for the construction of 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 [65]. 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 [2, 34, 59, 66] and the damped window [18, 42, 61]. The landmark window model splits data streams into fixed-size, non-overlapping chunks and

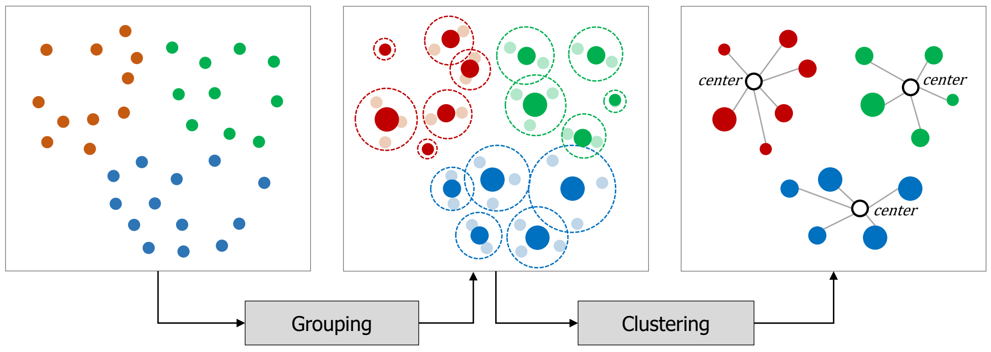


Figure 1.2: Data stream clustering

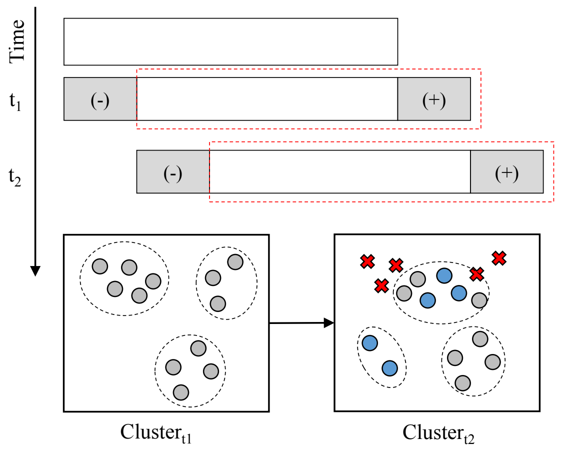


Figure 1.3: Clustering with sliding windows

contains 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). In the damped window model, also known as the fading window model, the tuples are associated with weights that decrease over time. Algorithms with these window models are based on the insertion-only model, which assumes that the tuples that are received only once are not removed from the window at a later time, and give newer tuples higher weight values than older ones.

While these two window models are effective in some data stream applications, they are insufficient for domains requiring the sliding window model. In this model, the window contains only the tuples whose timestamps are from a certain 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 critical and important factors for the applications of the sliding window model include topic extraction in news feeds and real-time traffic monitoring systems.

Extensive research has been conducted on clustering with the landmark or the damped window model, but only a small number of studies exist on clustering with the sliding window model [9, 17, 23, 68]. Clustering with sliding windows should produce results for every window movement, and a seemingly straightforward approach would be to perform repeated clustering. However, this is impractical and requires significant computational costs. Therefore, a clustering algorithm is needed that considers the deletion of expired data and the insertion of new data in both synopses construction and cluster results generation.

Although data stream clustering algorithms were originally designed to allow error tolerance and obtain approximate clustering results, it is important to maintain a precise range of target tuples for tracking clusters in evolving data streams. We consider general and expressive sliding window specifications in the continuous query language [6] of data stream management systems(DSMS) for clustering with accurate sliding window operations. The length of the sliding window is denoted by `RANGE` and the movement intervals are denoted by `SLIDE`.

We present an efficient data stream clustering algorithm with sliding windows. Our algorithm contains methods for fixed-size synopses construction and a re-clustering policy for updating pre-exist clusters. Main contributions of the algorithm are as follows:

1) Our algorithm supports general and precise sliding window operations [6] for data stream clustering. It maintains temporal group features, as synopses, using a sliding window aggregation technique that reduces space and computation time [46]. In 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.

2) Based on Locality-Sensitive Hashing, our algorithm constructs fixed-size synopses efficiently [24]. It reduces the synopses by joining the nearest

neighbors, and LSH can be used to search the nearest group feature in the synopses efficiently. Previous studies have used the tree-based index structure, but the hash-based index structure is suitable for our algorithm because it has an average time complexity of $O(1)$ for searching the nearest neighbor.

3) We propose a re-clustering policy for pre-existing clusters to avoid clustering every time data arrives. Clustering operations are very expensive as they access all the data objects iteratively. We allow appending of new input tuples to pre-existing clusters if the quality of the modified clustering results is acceptable. We also suggest measuring the difference in the quality between the two sets of clusters.

4) We extend the algorithm to density-based clustering and message passing-based clustering. The extended algorithms are applied to the clustering on document datasets, which include real-world and high-dimensional data.

This thesis is organized as follows: In Chapter 2, we give an overview of related studies on data stream clustering algorithms, and present the background information and problem statement. In Chapter 3, we develop a straightforward algorithm GFCS(Group Feature-based Data Stream Clustering with Sliding Windows) based on the sliding window aggregation technique. In Chapter 4, we propose CSCS(Coreset-based Data Stream Clustering with Sliding Windows) which is an improved algorithm of GFCS. In Chapter 5, an analysis of our experimental results for GFCS and CSCS is described. Finally, we present our extended algorithms for clustering on documents, and experimental results in Chapter 6.

Chapter 2

Preliminaries and Related Work

2.1 Data Streams

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

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

where x_i is a tuple, and t_i is a timestamp. A tuple x_i is represented by multi-dimensional attribute vector. A tuple of d dimensions is denoted by $x_i = (x_{i,1}, ..., x_{i,d})$. A timestamp t_i is non-negative integer value, and t indicates current time. For 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 $s_j = \langle x_j, t_j \rangle$. Timestamp value denotes a sequence number in tuple-based window, and a particular time instance in time-based window.

2.2 Window Models

In applications for data streams, a typical requirement is to perform operations over subsets of tuples within certain period of time. Since recent tuples from a stream reflects changes in data distribution, it is important to utilize time component as an essential element for operations. A window contains tuples

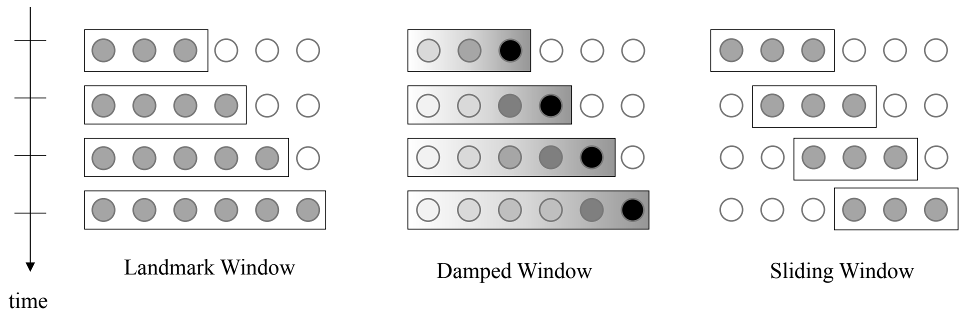


Figure 2.1: Window models

within a time range, and enables the applications to perform various operations on the tuples of the window. In data stream, three kinds of window models are studied [69], landmark windows, damped windows and sliding windows, which are shown in Figure 2.1.

2.2.1 Landmark Window Model

A landmark window contains tuples whose timestamps are from a specific time point called landmark and to the present. It is used for processing over entire tuples in the data stream after a certain time point. Tuples that arrive after the landmark are stored in the window, and processed. When the landmark is updated, all tuples in the previous window is removed, and the newly arrived tuples is stored in the new window. This process continues until the landmark is renewed. Therefore, landmark windows are non-overlapping, fixed-sized, and contiguous time intervals, and tuples in the landmark windows are split by landmarks. Landmark window is also called tumbling window. Almost partition-based data stream clustering algorithms adopted landmark window model.

2.2.2 Damped Window Model

A damped window gives weights to the tuples in the window based on decay function. Higher weight is given to recent tuples compared to older tuples, and the weights of the tuples decrease over time. Damped window model is used for reducing the impact of previous tuples on clustering results. In general, exponential time decay function $w(t) = \exp(-\lambda \Delta t)$ is used, where λ is decay rate, $\lambda > 0$, and Δt is difference between current timestamp and timestamp of the target tuple. Damped window is also referred to as fading window. Almost density-based data stream clustering algorithms adopted damped window model.

2.2.3 Sliding Window Model

A sliding window contains only tuples whose timestamp is within the range of the current timestamp and the start 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.

$$w(t - t_i) = \begin{cases} 1, & \text{if } t - t_i \leq R \\ 0, & \text{if } t - t_i > R \end{cases} \tag{2.1}$$

where R is window's time range. As time passes, the window removes the tuples whose timestamps have expired. Because the sliding window is specified by the most general definition, landmark window and damped window can be defined using the sliding window.

In continuous queries in DSMS [6], sliding window is specified by `RANGE` for the length of the window and `SLIDE` for the movement intervals of the window. Definition of sliding window is shown in Figure 2.2.

For example, `S [RANGE 1000 TUPLES SLIDE 100 TUPLES]` is a sliding window which contains the most recent 1,000 tuples with the window updated

```
<window> ::= <stream_source> [ '[' <window_frame> ']' ]
<window_frame> ::= <unbounded_frame> | <bounded_frame>
<unbounded_frame> ::= <range> UNBOUNDED
<bounded_frame> ::= <range> <time_spec> [ <slide> <time_spec> ]
<time_spec> ::= <time_value> <time_unit>
<time_value> ::= <number>
<time_unit> ::= <tuple> | <datetime>
<stream_source> ::= <identifier>
<range> ::= RANGE
<slide> ::= SLIDE
<tuple> ::= TUPLES
<datetime> ::= YEARS | MONTHS | DAYS | HOURS | MINUTES | SECONDS
```

Figure 2.2: BNF grammar for sliding window definition

upon 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 appended. For the sake of clarity, if we set `SLIDE` to L, an expression ”a window slides” or ”a window moves” means that oldest L tuples in the window are deleted and new L tuples are added to the window.

A Landmark window and a damped window are also defined by the definition. `S [RANGE 7 DAYS SLIDE 7 DAYS]` is a landmark window which updates its statistics every week. A damped window is defined upon a sliding window or a landmark window. A damped window reduces weights of existing tuples in a given window.

Windows are categorized into tuple-based and time-based sliding windows according to sliding condition and time unit. A tuple-based sliding window slides when new tuples arrive. The window contains a fixed number of tuples and also slides by fixed number of tuples. For example, `S [RANGE 1000 TUPLES`

SLIDE 100 TUPLES] is a tuple-based window, and the time unit is the number of tuples. A time-based sliding window slides as time progresses. For example, S [RANGE 20 MINUTES SLIDE 5 MINUTES] contains tuples from the most recent 20 minutes and slides every 5 minutes. Granularity of the window depends on time units which include HOURS, MINUTES, or SECONDS. In the time-based sliding window, the number of tuples within the window is not bounded.

For ease of explanation, we only consider tuple-based windows, but the methodologies can be applied to time-based windows as well.

2.3 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 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 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

$$dist^2(x, C) = \min_{c \in C} \|x - c\|^2.$$

k-Means clustering is defined as follows.

Definition 2.1 (k-Means Clustering)**.** *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*

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

Objective of k-means clustering 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

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

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

Optimal cost of k-means clustering of S is denoted by

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

Because k-means clustering problem is NP-hard even for $k = 2$ [4], heuristic algorithms are proposed. One of the classical heuristic algorithms is *Lloyd*'s algorithm [47]. Process of the algorithm is described as follows: Given k random initial cluster centers,

1. Each tuple x_j is assigned to the cluster C_i whose nearest distance between the cluster center and the tuple.

2. Each cluster C_i updates its center c_i to be the centroid of tuples in the cluster.

The process iterates until the cluster centers have not changed.

Quality of clustering depends on the initial cluster centers, and the algorithm converges to local optimum. The algorithm does not guarantee to converge to global optimum.

2.4 Coreset

A coreset for a set S is a small weighted set that approximates S with respect to an optimization problem. Cost of the coreset is an approximation for 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 with the weighted k-means clustering, and is an approximation for the cost of the original set S with relative error ε.

We denote definition of coreset for k-means clustering.

Definition 2.2 ((k, ε)-coreset [39])**.** *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*

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

Approximation algorithm using coreset is efficient because the algorithm is applied on the small size of coreset instead of entire data. The algorithm using fixed-size coreset is expected to be completed within a certain amount of time. Processing time of the algorithm is primarily dependent on time for construction of the coreset. For example, time for construction of the coreset is linear in the number of data points n, dimensionality d, and the number k for k-segmentation problem [57]. The size of the coreset is $O(dk/\varepsilon^2)$, which is independent on the input size n.

Coreset defined in Definition 2.2 is called strong coreset. The strong coreset is required to guarantee quality of approximation for every set of centers. This condition is not always necessary for designing an approximation algorithm, and weak coreset with more relaxed condition is proposed. The most important property of weak coreset is the possibility to compute the solution with $(1+\varepsilon)$-approximation for the original set using the weak coreset [**?**]. Unlike strong coreset, weak coreset does not need to guarantee the quality of approximation for every set of centers. In k-means clustering, it is reasonable to utilize weighted centroids of a set of tuples as a coreset because they covers original tuples with relative small error.

However, coreset with approximation guarantee cannot be computed for data streams. Previous approaches focused on showing approximation guarantee of coreset when appropriate centers are given [30, 38, 39]. In practice, the centers are unknown, and problem for finding optimal centers is NP-hard [4] in k-means problem. The algorithms for finding k centers with $(1+\varepsilon)$-approximation is not known because the problem is also APX-hard [8]. The state-of-the-art algorithm in k-means problem is *k-means++*, where the cost is $E[Cost(S,C)] \leq 8(\log k + 2) \cdot cost^k_{OPT}(S)$ [7].

We describe our approach to construct a small weighted set of tuples in next section, and we only show upper bound of cost of our approach with assumption that data is not appended and removed, and the optimal solution is known.

There are two common tasks that are typically performed by data stream clustering algorithms [2, 3, 39, 68]: 1)grouping task and 2)clustering task as shown in Figure 1.2. The grouping task constructs a *coreset* from original data by particular heuristic method. The clustering task apply specific clustering algorithm to the *coreset* for finding clusters of the original data.

We used more general term *synopses* to refer to summarized data for explanation in Chapter 1. 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 proved in data streams, we consider that the term *coreset* represents the concept more clearly than *synopses.*

2.5 Group Features

Group Feature(GF) is defined as a data structure for storing the statistic summaries of a set of tuples which are contained in *coreset.* Previous studies used the term *Clustering Feature(CF)* to refer to the statistic summaries [65, 68]. However, the term *Clustering Feature(CF)* is confused with the results made through the clustering task, and we also modify contents of CF for our algorithm. Therefore, we will use the term *Group Feature(GF)*.

GF consists of the linear sum of tuples LS, the square sum of the tuples SS, the number of tuples N, and the most recent timestamp of the tuples T. Tuples are in the range of sliding window. LS and SS are generated by pairwise summation of tuples, i.e. for d-dimensional n tuples, $LS = \sum_{i=1}^{n} x_i = \sum_{i=1}^{n}(x_{i,1}, ..., x_{i,d})$ and $SS = \sum_{i=1}^{n} x_i^2 = \sum_{i=1}^{n}((x_{i,1})^2, ..., (x_{i,d})^2)$. LS and SS are d-dimensional vectors, and N and T are numeric values. Basic components LS, SS, and N are proposed by [65], and the timestamp component T is added by [68]. In addition, GF includes hash value which is generated by LSH for our algorithm. We will explain it in a following section.

GFs have incrementality and additivity properties. Incrementality means that the GF is updated by adding a new tuple x_j, while additivity means that two disjoint GFs can be merged into a new GF by adding their components [65]. These properties enable to modify the coreset in a constant time.

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

Values for clustering such as centroid can be calculated easily by using components of the GF, i.e., $Centroid = LS/N$.

GFs are continuously updated as tuples are input from the data streams. If the existing GF can subsume the input tuple, the tuple is added to the GF, otherwise a new GF is created using the tuple. sliding The clustering algorithm are applied on the GFs. Moreover, the expired GFs are removed, and the new GFs are appended for the sliding windows.

Table 2.1: Partition-based clustering algorithms with window models

Clustering	Window Models		
	Landmark	**Damped**	**Sliding**
Partition-based	BIRCH [65]		Babcock et al. [9]
	ScaleKM [16]		SWClustering [68]
	STREAM [35]		G2CS [10]
	LSEARCH [34, 52]		
	CluStream [3]		
	DGClust [32]		
	StreamKM++ [2]		
	Ackerman et al. [1]		
	E2SC [44]		
	TADPole [11]		

2.6 Related Work

Clustering algorithms for data streams have been extensively studied, and a detailed survey of these algorithms is presented in [58]. Data stream clustering algorithms are categorized into partition-based [2, 3], density-based [18, 42], and message passing-based [59, 66] clustering, and are developed under landmark window model [2, 59, 66], damped window model [18, 42], and sliding window model [9, 17, 23, 68]. The majority of these algorithms adopt the landmark window model, while density-based algorithms are designed according to the damped window model. Table 2.1, 2.2, and 2.3 show the related studies categorized by clustering algorithms and window models.

We describe clustering algorithms in detail for each window model. Most algorithms using landmark window model utilize partition-based clustering algorithms such as k-means and k-median.

Table 2.2: Density-based clustering algorithms with window models

Clustering Method	Window Models		
	Landmark	**Damped**	**Sliding**
Density-based	Any-OPTICS [49]	DenStream [18]	
		D-Stream [20, 60]	
		MR-Stream [61]	
		SOStream [41]	
		ClusTree [42]	
		DASC [12]	
		DBSTREAM [37]	
		ADStream [25]	
		CEDAS [40]	

Table 2.3: Other clustering algorithms with window models

Clustering Method	Window Models		
	Landmark	**Damped**	**Sliding**
Message	STRAP [66]		ID-AP [63]
Passing-based	IAP [59]		
	SAIC [67]		
Hierarchical	ODAC [55, 56]		
Model-based			SWEM [23]

2.6.1 Clustering with Landmark Windows

BIRCH(Balanced Iterative Reducing and Clustering using Hierarchies) [65] is initiative algorithm that enables k-means clustering for large data. To summarize large data, BIRCH first presented a concept of *clustering feature* (CF) which stores the number of tuples, linear sum of tuples, and square sum of tuples. BIRCH constructs a data structure called CF-tree based on B+-tree, and stores CFs into nodes. Leaf nodes in the tree contain clustering features, and inner nodes contain merged clustering features of child nodes. CF in an inner node is used as an index to find adjacent CF of input tuple. When new tuple is appended to the tree, the tree is traversed from root node to leaf node, and compares with inner nodes to find the closest CF of the tuple. Euclidean distance between center of CF and the tuple is used as similarity. When the nearest CF is found at leaf node, it determines whether CF absorbs new tuple. Maximum radius of CF in the leaf node should be smaller than threshold θ which is given by user. If radius of CF which absorbs new tuple does not exceed θ, new tuple is appended to CF. If it exceeds, new CF based on the tuple is created and appended to leaf node. Therefore, threshold θ determines the size of the tree. If θ is small, the large number of CFs are included in the tree. When the number of CFs is greater than user-specific value, leaf node is divided into two, and the farthest CFs is used as key of each node. In addition, values in root node and inner nodes affected by appended tuple should be updated. User adjusts threshold θ appropriately according to memory capacity. For generating clustering results, BIRCH performs general clustering algorithm using CFs in leaf nodes, or uses internal nodes as a result of hierarchical clustering. BIRCH is very fast, but clustering quality is very poor, which is experimentally shown in [2]. For sliding window, BIRCH is not appropriate because insertion and deletion of CFs which are divided by timestamp occurs more frequently in the tree.

Scalable k-means [16] uses CF of BIRCH to apply k-means clustering to large data. The algorithm selectively remains data points according to the importance that affects clustering quality, and stores them in buffer of memory.

The size of buffer is specified by user, and clustering is performed with data in buffer. Using clustering results, newly input data points are classified into three types: retained set(RS), discard set(DS), compression set (CS). Retained set is kept in the buffer, discard set is removed after updating statistics, and compression set is summarized in statistics by compression process. The algorithm keeps CFs as statistics. Compression process consists of two steps: primary data-compression and secondary data-compression.

Primary data compression discards data points that will not change cluster assignments. Points whose Mahalanobis distance between points and a center of cluster is less than threshold are found, and a certain percentage of points remained. They are used as statistics that represent the cluster. For points with a distance above the threshold, the process found the closest cluster in Mahalanobis distance. Using the points, a new cluster center is calculated. If distance between points and new center is less than threshold, statistics of the points is considered to be sufficient, and are deleted.

Secondary data-compression makes sub-clusters using points that could not be erased by the primary data compression. Purpose of this step is to free memory space to store new data points. The compression finds the dense regions which are not processed by the primary data compression, and performs k-means clustering on the points in the region. CFs of the sub-clusters are included in buffer if the sub-clusters pass filter that evaluates density criterion.

Single-pass k-means [27] performs k-means clustering in one scan of dataset without data compression techniques. An algorithm is a simplification of *scalable k-means* [16] algorithm. Authors conclude that *scalable k-means* is several times slower than standard k-means algorithm due to overhead of data compression techniques. The algorithm performs k-means clustering on data points in buffer, and discards all points except statistics of clusters. They consider that clustering results maintain sufficient statistics. After clustering, points are filled in buffer again, and clustering is performed until convergence. Clustering is performed on data including points in the buffer and weighted means of previous clusters. This process is repeated until all data is read.

STREAM [35] is a constant-factor approximation k-median clustering algorithm for data streams in a single pass based on divide-and-conquer strategy. The algorithm divide data in to l disjoint groups, and finds k centers for each group by k-median clustering. Each center is weighted by the number of points which are assigned to a cluster. The algorithm performs k-median clustering again to find k centers from $O(lk)$ weighted centers. Authors prove that the proposed algorithm is α-approximation if re-clustering is performed in a constant number of times.

For data streams, data points are processed in hierarchical scheme. Approximate clustering algorithm is applied on first m input points to reduce them to $2k$ points. Reduced points are weighed by the number of points assigned to them. The reduction process is repeated until original data points are reduced to $m^2/(2k)$. Clustering is applied again on m medians in first level, and $2k$ medians in second level are generated. Final clustering results are generated using intermediate medians. The algorithm requires $O(nk)$ time, $O(^{\epsilon})$ space, amortized update of $O(k\ polylog(n))$ for $\epsilon < 1$. They also prove that approximate clustering quality depends on the number of levels.

LSEARCH [34,52] is improved algorithm of *STREAM* for k-median problem, which is based on concept of local search. Local search means that the algorithm creates initial clusters, and refines them by making local improvements. Initial solution is n-approximation to facility location on data points with facility cost f. The algorithm reorder points after finding an initial solution. For each reordered point, it computes *gain* of a point, and if the *gain* is greater than zero, performs allowed reassignments and closures. Specifically, new cluster is created with probability d/f, where d is distance between current point and nearest existing cluster center, and f is facility cost. Otherwise, the point is appended to best exiting cluster. Authors also proposed an algorithm to find a good initial solution which obtains an expected 8-approximation to optimum.

CluStream [3] presents concepts of micro-clusters and pyramidal time frames for data stream clustering. Micro-cluster maintains statistics including spatial locality and temporal features of data. CF in a micro-cluster contains sum of timestamps and sum of squares of timestamps in addition to N, LS, and SS. Pyramidal time frame is used to effectively store snapshots of micro-clusters. The algorithm updates micro-clusters in online processing, and performs clustering on micro-clusters to create macro-clusters in offline processing.

Specifically, the algorithm creates initial q micro-clusters by standard k-means clustering in offline processing. Statistics of micro-clusters are updated with input data points. The maximum radius of each micro-cluster is computed based on root-mean-square deviation of distances between data points and center of an assigned cluster. For new input data points, the algorithm find the closest micro-cluster, and calculate distance between them. If the distance is less than the maximum radius, the micro-cluster absorbs the data point, and updates CF. Otherwise, new micro-cluster based on the data point is created. When timestamp of the micro-cluster is less than user-specific time threshold, the micro-cluster is deleted.

A snapshot of the micro-cluster is stored to produce clustering results for a specific time range. However, since it is infeasible to store all snapshots, the algorithm presents a concept of pyramidal time frame which divide time ranges at different levels of granularity depending upon recency. In the pyramidal time frame, snapshots are stored with different time ranges according to their recency. For example, if snapshots within 10 minutes from current time are stored every 1 second, then snapshots before 10 minutes is stored every 5 minutes, and snapshots before 1 hour is stored every hour. Let $\log_a(T)$ determines the time period, where T is elapsed timestamp from current time, and $a \geq 1$. To store 1 year snapshots, about 100 units of memory space are required, which is affordable. For various time-range queries, existing snapshots are combined to create corresponding micro-clusters. Using micro-clusters, the algorithm creates macro-clusters. Similar to other algorithms, the clustering algorithm is applied on pseudo-points that are weighted centroids of micro-clusters. CluStream also presents analysis of evolving clusters based on stored snapshots. For a given time range, the algorithm creates snapshots of macro-clusters at start and end times, which can be used to track changes of clusters.

DGClust [32] is a monitoring and clustering algorithm for distributed sensor networks. General clustering algorithm collects data into a central server and processes them. However, this methodology increases burden on a central server, such as communication costs and large storage space. This algorithm addresses this problem. Each local sensor receives data from a source and generates an infinite univariate data streams. Local sensors do not send entire data to the central server, but maintains statistics of data. Data of each sensor is processed locally, and incrementally discretized into a univariate adaptive grid. For incremental discretization, Partition Incremental Discretization (PiD) algorithm is used. The algorithm consists of two layers. First layer creates grid of fixed size. If frequency of a grid is greater than user-defined threshold, the grid is split in second layer. Otherwise, grids are merged. The central server maintains global states of network. Local sensor transmits its state change to the central server whenever data changes. Global clustering results are generated using data in the central server.

StreamKM++ [2] produce coreset of data stream using coreset tree and *k-means++* [7] seeding procedure. k-Means++ seeding procedure is a methodology for selecting a good initial centers because k-means clustering quality is strongly depends on initial centers. Each points has a probability of being selected as a center, which is ratio of the sum of squared distances from pre-selected centers and squared distance from nearest center. Cost of k-means++is $E[Cost(S,C)] \leq$

$8(\ln k + 2)cost^k_{OPT}(S)$.

StreamKM++ chooses whether data point is included in a coreset or not based on probability, and constructs a sample with m data points. Data points which are not included in the coreset are assigned to their nearest data points in coreset, and each data point in coreset is weighted by the number of assigned points. Unlike k-means++, m data points are selected from input data instead of k. In this thesis, it is shown that sufficient quality can be obtained with $m = 200k$.

Coreset tree is a binary tree to speed up coreset construction, where each node contains sample point of a coreset. Root node is a single cluster which includes all data points. Child nodes are non-overlapping sub-clusters of a cluster in a parent node. Coreset tree construction starts with creating a single cluster containing whole points. Cluster is divided into two sub-clusters for child nodes. Points of a sub-cluster are far from points of the other sub-cluster in sibling node. Partitioning are repeated until the number of cluster is reached to user-specific number. k-Means++ seeding is also used to create the tree. The algorithm choose a cluster in a leaf nodes at random with a probability based on k-means cost. New sample point is selected from clustering results according to k-means++ seeding procedure. Cluster is split into two sub-clusters based on sample point of the cluster and the new sample point, and two child nodes are appended to the selected leaf node.

To apply coreset construction and clustering to data streams, StreamKM++ exploit merge-and-reduce technique. The algorithm maintains a certain number of buckets, and each bucket contains a coreset. For new input data points, new bucket is created, and stores a coreset which is constructed using data points. If newly created bucket is full, the bucket merges with existing bucket. The algorithm performs k-means++ clustering on union of coresets in the buckets.

ODAC [55, 56](Online Divisive-Agglomerative Clustering) build hierarchy of clusters based on top-down strategy. The algorithm uses correlation-based dissimilarity measure, and agglomerative clustering for concept drift detection of clusters. In a tree, leaf nodes contains clustering results which are not overlapped. Pearson's correlation coefficient is used as similarity measure, and rooted normalized one-minus-correlation is used as dissimilarity measure for comparing clusters. Splitting and agglomerative criteria are based on radius of existing clusters. The algorithm maintains radius of cluster for statistics. If the radius of the cluster reaches a predefined threshold, the algorithm splits the cluster, and assigns data points to new clusters. This hierarchical data structure enables to handle concept drift of clusters. If the radius values of child clusters is greater than radius of parent node, it is considered that current state does not reflect structure of data. Therefore, the algorithm re-aggregates child nodes of parent node, and starts split process.

STRAP [66] is data stream clustering algorithm based on Affinity Propagation(AP) [31]. STRAP combines AP with statistical test for change detection of data distribution. The clustering is performed whenever a change in data distribution is detected. First, the algorithm generates initial exemplars, which are similar to centers in k-means clustering, by standard AP clustering. As data point arrives, distances between the point and exemplars are computed to find the nearest exemplar. If the distance from the nearest exemplar is too large, the point is not included in clusters, and stored separately as an outlier. For testing a change of data distribution, statistical test, Page-Hinkley is used. If the change of data distribution is detected or the number of outliers exceed storage size, weighted affinity propagation is perform on union of exemplars and outliers.

IAP [59](Incremental Affinity Propagation) propose two methodologies for incremental clustering which are IAP clustering based on K-Medoids (IAPKM) and IAP clustering based on Nearest Neighbor Assignment (IAPNA). In IAPKM, AP clustering is performed to find good initial exemplars, and k-medoid clustering is applied to modify clustering results as data arrive. It is known that quality of clusters by k-medoid clustering is highly dependent on initial exemplars. IAPKM uses AP clustering to handle this problem. In IAPNA, the algorithm finds the nearest data point in existing clusters for a new input data point. Responsibility and availability of new data point are specified based on values of the nearest data point. Authors extends responsibility matrix and availability matrix based on updated similarity matrix. It is based on assumption that if two points are similar, they have similar relationships with other points. After assigning values to a new data point, AP clustering is performed till convergence.

2.6.2 Clustering with Damped Windows

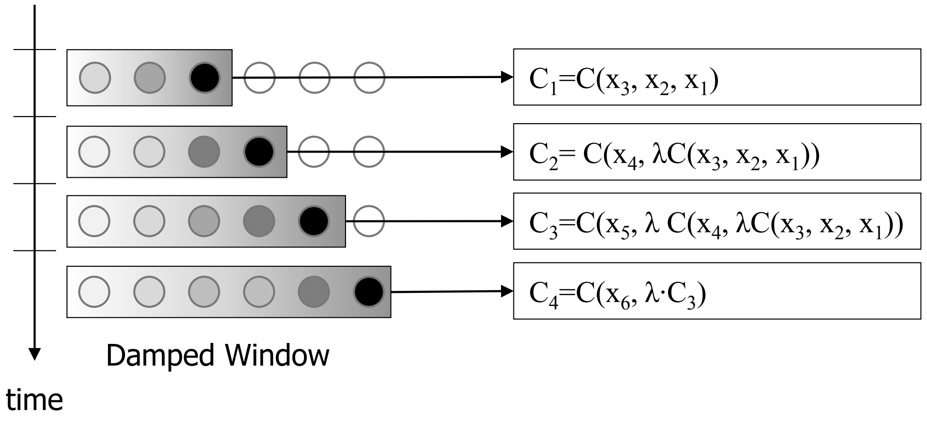
DenStream [18] is a density-based clustering algorithm for evolving data streams, which is robust to noise. The algorithm creates micro-clusters, and classifies into core-micro-clusters(c-micro-cluster), potential core-micro-clusters(p-micro-cluster), and outlier micro-clusters(o-micro-cluster) based on density of data. c-micro-cluster is defined as a group of close points with timestamps, which is a dense micro-cluster for clustering with arbitrary shape. P-micro-cluster and o-micro-cluster are stored in a separate memory space, outlier-buffer, and used in offline phase. The algorithm consists of online phase and offline phase. In online phase, micro-clusters are maintained with new input data points. In offline phase, modified DBSCAN is performed on c-micro-clusters and p-micro-clusters for generating clustering results.

DenStream performs clustering based on damped window model. In damped window model, weight of data point decreases exponentially by decay function $f(t) = 2^{-\lambda t}$, where $\lambda > 0$. Value of λ adjust effect of past data. If λ is high value, importance of historical data is reduced.

C-micro-cluster is defined as time-weighted (w, c, r), where w is weight, c is center, and r is radius. Since c-micro-cluster should be dense, $w > \mu$ and $r < \epsilon$, where μ and ϵ are user-specific threshold values. P-micro-cluster contains time-weighted components of CF. Since p-micro-cluster is less dense than c-micro-cluster, weight is greater than $\beta\mu$, where $0 < \beta < 1$ is user-defined parameter. O-micro-cluster contains time-weighted components of CF with timestamp which is creation time. Weight of the o-micro-cluster is $w < \beta\mu$, which is below than the threshold. If new data points are appended, and weight exceeds the threshold, the o-micro-cluster is changed to p-micro-cluster.

In online phase, the algorithm generate initial clusters by using DBSCAN. These cluster are classified as p-micro-clusters. For new input data point, it is merged into either the nearest p-micro-cluster or o-micro-cluster if Euclidean distance between data point and center of a micro-cluster is below than the threshold. Otherwise, new o-micro-cluster is created based on the data point, and it is stored in outlier buffer. The algorithm tests periodically whether the o-micro-cluster is outlier. If weight of the o-micro-cluster is lower limit of weight, the o-micro-cluster deleted from the outlier buffer. This process removed o-micro-clusters which will not change into p-micro-clusters in the future. In offline phase, clustering results are generated. For clustering results, a variant of DBSCAN is applied on p-micro-clusters. Concept of density-connectivity in DBSCAN is modified for micro-clusters, which are based on definitions of directly density-reachable, density-reachable, and density-connected. Although the algorithm generates clusters of arbitrary shape effectively, there is no significant improvement in terms of memory space and execution time compared to typical DBSCAN.

D-Stream [20, 60] is a density-based clustering algorithm based on grids. The algorithm maintains clustering features in grid units. Data point is assigned to corresponding grid, and CF of the grid is updated. Dense grid and sparse grid



C_1=C(x_3, x_2, x_1)
C_2= C(x_4, λC(x_3, x_2, x_1))
C_3=C(x_5, λ C(x_4, λC(x_3, x_2, x_1))
C_4=C(x_6, λ·C_3)
Damped Window
time

Figure 2.3: Clustering with damped window

is determined by user-specific threshold which is similar to minpts in DBSCAN. Because the algorithm is also based on damped window model, decay coefficient $f(x) = \lambda^{t-t_x}$ is defined as decay function to reduce influence of past data points. The decay function is applied to entire grids at a specific interval. The algorithm propose a strategy for determining appropriate interval. If interval is too large, change of clusters cannot be precisely detected. If interval is too small, the same clustering results are generated too frequently, which increases computational cost. D-Stream set interval to smaller value between minimum time required for a dense grid to change to a sparse grid and the minimum time required for a sparse grid to change to a dense grid. To remove outliers, the algorithm defines sporadic grid which is a very sparse grid. Sporadic grids are removed in each time interval because the algorithm assumes that the sporadic grids do not change into dense grids in the future. D-Stream is improved in [60], which merges two dense grids if correlation measure between the grids is greater than threshold.

MR-Stream [61] maintains hierarchically divided grids with tree data structure. A node is created with a new data point, and added to a tree, where weights are updated from parent node to root node. The algorithm traverses the tree for detecting sporadic grids. Sporadic grids whose density values are lower than threshold are removed as outliers. With the tree, clustering is performed on grids in a user-specific height.

SOStream [41](Self Organizing Density-Based Clustering Over Data Stream) automatically determines threshold for density-based clustering based on competitive learning. In competitive learning, technique is to find a winner according to certain criteria, and to make winner affect its neighborhood. For new data point, the winner cluster is selected by Euclidean distance between cluster center and data point. New data points is appended to the winner cluster or new cluster according to the distance. The algorithm finds overlap clusters with the winner cluster, and the clusters whose distance is less than threshold are merged into the winner cluster. There is no offline phase in the algorithm. In online phase, the algorithm produces clustering results in a manner similar to merging procedure for overlap clusters.

ClusTree [42] maintains clustering features by extended index structures based on R-tree. The algorithm finds the nearest micro-cluster which is stored i leaf node to insert a data point by searching the tree from root node to leaf nodes. If micro-cluster can absorb the data point, CF of the micro-cluster is updated incrementally with the data point; otherwise, new micro-cluster is created. Inner node contains summary of its subtree. CF of the inner node is aggregation of CFs of descendant nodes. Based on R-tree definition, an inner node contains a maximum of M CFs, and a leaf node contains a maximum of L CFs, where M and L are user-defined parameters.

Unlike ordinary insertion operation in R-tree, ClusTree introduces a concept of hitchhiker. Each inner node has additional buffer space. If there is a need to interrupt insertion operation while traversing down to leaf node, CF is stored temporarily in a buffer. A buffer is in parent node which have descendant nodes to be traversed for CF later. When there is another insertion operation that passes through this node, the operation finds leaf nodes for CF stored in the buffer in addition to its CF. If the two CFs need to be assigned to different leaf nodes, the leaf node for the previously stored CF is searched first. New CF is stored in a temporary buffer of a branching node, and processed in later operation. This has advantage that input operation is completed within a constant time even for fast data streams. Offline clustering algorithms, such as k-means or density based clustering, are performed on the micro-clusters. Concepts of buffer and hitchhiker allows the algorithm to interrupt a running operation, and it enables to perform clustering operation at any time. This algorithm also adopts damped window model, and uses exponential time-dependent decay function $w(\Delta t) = \beta^{-\lambda \Delta t}$.

2.6.3 Clustering with Sliding Windows

In contrast to landmark or damped window, only a small number of studies focus on clustering algorithms with sliding windows [9, 10, 17, 23, 68]. Dang et al. propose a Gaussian mixture models based clustering algorithm for sliding window [23]. They exploit Expectation Maximization technique, and develop splitting and merging operations to remove expired tuple. Babcock et al. present a technique of maintaining variance and k-median based on *exponential histogram(EH)* for sliding window [9]. Zhou et al. focus on problem of tracking evolution of clusters in sliding window, developing SWClustering, a k-means clustering algorithm based on an extension of EH, *exponential histogram of clustering features(EHCF)* which combines temporal attribute with EH [68]. In theory community, Braverman et al. propose a merge-and-reduce based technique to transform coreset construction in insertion-only streaming model to sliding window model [17].

Specifically, algorithms which exploit EH as synopsis data structure support

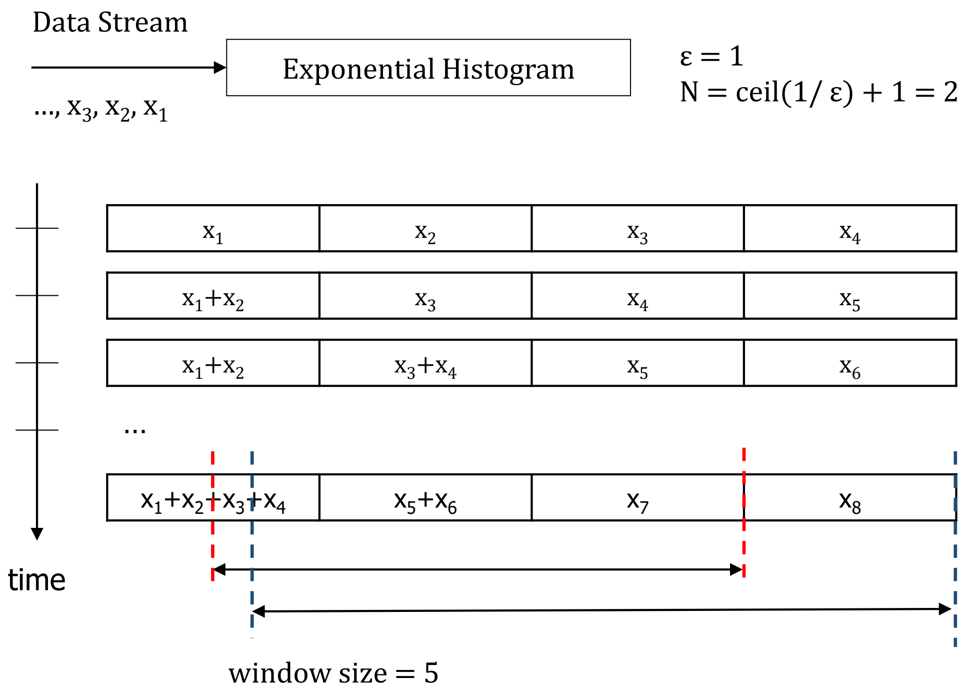


Figure 2.4: Exponential histogram maintenance

insertion and deletion [9, 68]. EH is defined as a collection of buckets on a set of tuples, and generates $(\frac{k}{2}+1)((\log\frac{2N}{k}+1)+1)$ for $k=\lceil\frac{1}{\epsilon}\rceil$. Only synopses of tuples in each bucket is stored by appropriate bucket, with synopsis containing both clustering features and the most recent timestamp of tuples in the bucket. Because of memory limitations, if the number of buckets exceeds user defined number, buckets are merged, with each merged bucket holding a number of tuples equal to or double that held in previous unmerged buckets. For example, we assume that input tuples are $x_1, x_2, \ldots$ (x_2 newer than x_1), and state of buckets is $B_1 = \{x_1, x_2\}$, $B_2 = \{x_3\}$, $B3 = \{x_4\}$. As new tuples arrive, old buckets are merged and a new bucket is created with new tuples, i.e., $B_1 = \{x_1, x_2\}$, $B_2 = \{x_3, x_4\}$, $B3 = \{x_5\}$. When a sliding window moves, buckets whose timestamp have expired are removed. However, a small deviation occurs in timestamp. If the size of sliding window is 4 in example, it should drop 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 window size increases. Therefore, one of objectives of our algorithm is clustering on accurate ranges of tuples.

G2CS [10](Generic 2-phase Continuous Summarization framework) proposes a mechanism for sliding window maintenance, and C-BIRCH which is a data summarization technique based on BIRCH. G2CS generates Partial Grouped Summary (PGS) which is a summary of data in each partial window. Lattices based on PGSs are created and are updated by Sliding Binary Merge method to maintain summaries of whole window efficiently. C-BIRCH is a clustering algorithm to build micro-clusters of PGS. Micro-clusters are used as a summary. C-BIRCH generates CF-tree from data points in PGS in the same way as BIRCH. CF-tree additionally includes temporal features. As in BIRCH, leaf nodes of CF-tree are used as micro-clusters. The problem that G2CS solves is similar to ours. However, unlike our algorithm, G2CS does not limit the size of summaries and has higher time complexity with tree-based indexes. In addition, there is an overhead of creating and maintaining unnecessary lattices to deal with various window queries on databases. Clustering quality of G2CS is worse than other clustering algorithms because C-BIRCH is based on BIRCH which is sensitive to arrival order of data points.

2.7 Problem Statement

Based on the definitions, we define problem statement: Given a stream of tuples(S), user-specified number of clusters(k), window size(RANGE R), sliding interval(SLIDE L), and number of group(m). The purpose of data stream clustering with sliding window is to generate clusters of tuples in a sliding window while considering both accuracy and runtime.

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

Our approach consists of two main steps in general: 1) Construction and maintenance of a constant number of coreset over sliding windows 2) Decision on whether to append new coreset to pre-existing clusters or perform clustering on whole coreset according to the difference between probability distributions of the original and updated clusters.

For ease of reference, Table 2.4 summarizes notations used in the thesis. Throughout this thesis, we use tuples and data points interchangeably. Based on the notations, we formally describe our data stream clustering algorithms, GFCS and CSCS. Table 2.5 shows parameters to be given by user. Criteria for setting the parameters are discussed in detail in subsequence chapters.

Recently, a number of studies related to data stream clustering have been conducted. In partition-based clustering, there are methods for incremental or evolutionary clustering [1, 11, 44]. Density-based clustering methods adopt damped window model Mai:2016:AnyOPTICS [12,25,37,40], but Any-OPTICS [49] is based on landmark window model. In message passing-based clustering, ID-AP [63] is proposed for image clustering, and SAIC [67] performs clustering on data stream of chunks using incremental learning.

As previous related studies, these methods fetch data points one by one from data streams, and update clustering results each time. Our algorithms work the same as these methods if a sliding window is defined by `S[RANGE UNBOUNDED]`. However, it is inefficient to perform clustering in this way. Operation of removing data points from clusters should be done every time. Our algorithms split

Table 2.4: Notations used in the GFCS and CSCS

Notation	Meaning
S	a set of tuples in a data stream
s_i	i^{th} tuple in a data stream
x_i	data point of s_i which is d dimensional vector
t_i	timestamp of tuple s_i
d	the number of dimensions of data points in S
C	a set of centers of clusters. $\lvert C \rvert = k$
$cost(S, C)$	cost function of k-means given C in S
$w(x)$	weight function for x
GF	group feature which is synopses for clustering
LS	the linear sum of tuples in GF
SS	the squared sum of tuples in GF
N	the number of tuples in GF
T	the most recent timestamp of tuples in GF
M	a set of group features generated from S
W	a sliding window
$h(x)$	local hash function for LSH in CSCS
$g(x)$	global hash function for LSH in CSCS.

data streams into chunks, and perform clustering using aggregation of chunks for efficiency. This method is a kind of mini-batch technique, and is widely used in various fields.

Table 2.5: Parameters given by user

Parameters	Meaning
R	the length of the window W used in `RANGE`
L	the movement intervals of the window W used in `SLIDE`
k	the number of clusters for k-means clustering
θ	distance threshold to group near data points
m	coreset size. the number of GFs in a coreset for window W
l	the number of local hash functions

Chapter 3

GFCS: Group Feature-based Data Stream Clustering with Sliding Windows

3.1 2-Level Coresets Construction

Group Feature(GF) contains summaries of tuples in sliding window. If synopses maintain only GF of tuples in sliding window, it is possible to update GF with new arrival tuples because of incrementality. However, it is impossible to update GF with expired tuples because synopses have not preserve values of expired tuples to subtract from the GF. To retain the GF for sliding window, the values to subtract from the GF should be kept. Because it is inefficient to keep all tuples in sliding window, we propose a data structure for synopses, pane-based GF (PGF) and window-based GF (WGF).

Figure 3.1 shows an overview of the coresets structure. A window is decomposed into panes which are non-overlapping sets of tuples. Assume 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, sliding windows, as defined by `S [RANGE 1000 TUPLES SLIDE 100 TUPLES]` have ten panes, with each pane containing a summary of 100 tuples. In `S [RANGE 1000 TUPLES SLIDE 99 TUPLES]`, the

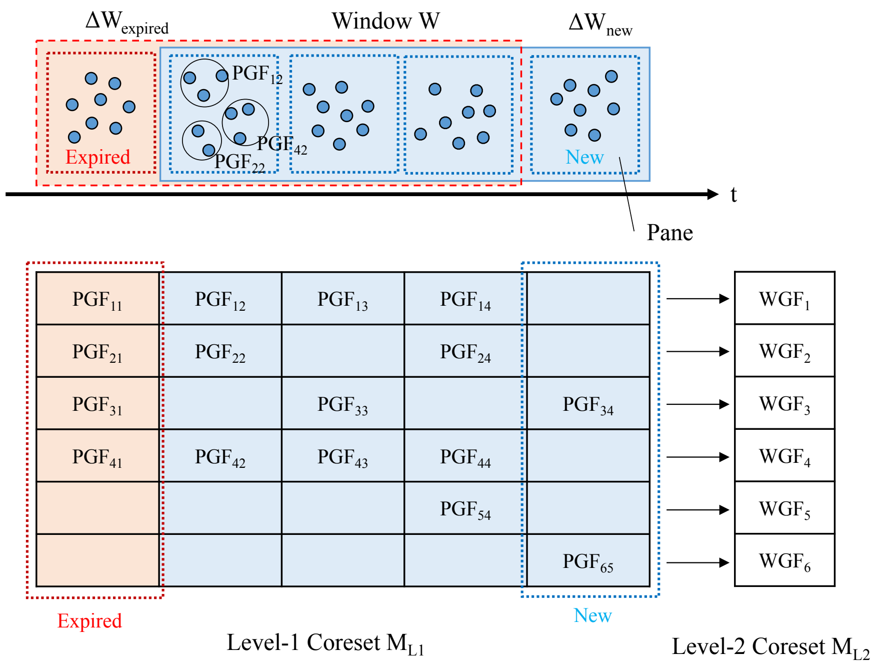


Figure 3.1: Overview of 2-Level coreset structure

window consists of 11 panes, where ten panes each contain 99 tuples, and one pane contains 10 tuples. For ease of presentation, we only discuss the case that R is divisible by L.

When the window slides, $\Delta W_{expired}$ is removed and ΔW_{new} is appended. New PGFs are generated based on tuples in ΔW_{new}. Components of the PGF are the same as for GF, which includes LS, SS, N, and T. The detailed process of creating PGF is described in Algorithm 1. Given threshold θ, tuples whose distances are below θ are grouped into the same PGF. If the distance between a tuple and centroid of PGF, $dist(b, p)$ is below θ or radius of PGF, the PGF absorbs the tuple. Generating operation has $O(L \times m)$ time complexity, where m is the number of generated PGFs.

As shown in Figure 3.1, coresets consist of level-1 coreset and level-2 coreset. Level-1 coreset is a 2-dimensional array of $\lceil R/L \rceil$ width. Each row in level-1

Algorithm 1 CreatePaneGF

```
    Input: A set of tuples $B$, and threshold $\theta$
    Output: A set of $PGF$s
 1: create empty set $P$
 2: for each $b \in B$ do
 3:     if $P$ is empty then
 4:         create new $PGF$ $p$ based on $b$
 5:         $P \leftarrow P \cup \{p\}$
 6:     else
 7:         $p \leftarrow$ nearest $PGF$ in $P$ to $b$
 8:         if $dist(b, p) < \theta$ or $dist(b, p) <$ radius of $p$ then
 9:             update $p$ by adding $b$
10:         else
11:             create new $PGF$ $p$ based on $b$
12:             $P \leftarrow P \cup \{p\}$
13:         end if
14:     end if
15: end for
16: return $P$
```

coreset contains *PGF*s whose distances are close. Generated *PGF*s by Algorithm 1 are inserted into the last column of level-1 coreset. Timestamps T of the inserted *PGF*s are in $(R-L, R]$. When the expired tuples are removed, the first column of level-1 coreset whose timestamps are in $[t_1, t_1+L]$ is truncated, where t_1 is the earliest timestamp. Removing operation for level-1 coreset has $O(1)$ time complexity if an adequate data structure is adopted such as a linked list queue. The exact number of tuples is R in the level-1 coreset, and remains constant.

Window-based GFs *WGF*s in the level-2 coreset are built by summing up

*PGF*s which are in the same row. *WGF* is equal to the GF of tuples in the row due to the additivity property in Equation (2.5). *WGF* is represented as

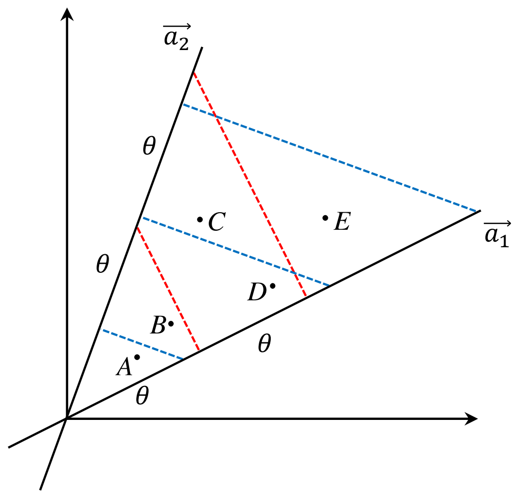
$$WGF_i = \sum_{j=1}^{\lceil R/L \rceil} PGF_{ij}$$

The algorithm updates *WGF*s by adding new *PGF*s and subtracting expired *PGF*s when the window slides. Because level-1 coreset contains *PGF*s based on panes, we specify *PGF*s to be removed and quantify the values of expired tuples. The additivity property also guarantees correct *WGF*s for subtraction.

Algorithm 2 describes a procedure for updating synopses. The algorithm is performed through batch processing for performance. CreatePaneGF in line 8 creates *PGF*s of recent tuples whose timestamps are in $(R - L, R]$ from data streams. The process for removing expired tuples is presented in line 1-6. $dist(p, WGF_i)$ computes distance between centroids of *PGF* and *WGF*. $WGF_i + p$ in line 12 means that it updates components in WGF_i by adding components in PGF p, and timestamp T is replaced by timestamp t of p because p is newer than WGF_i.

3.2 2-Level Coresets Maintenance

Updating M_{L1} and M_{L2} involves linear time complexity. The most time-consuming parts in the Algorithm 2 are CreatePaneGF and finding the nearest *WGF* to a *PGF*. CreatePaneGF can be executed within a reasonable time by adjusting the size of the `SLIDE` L. However, finding the nearest *WGF* is a computationally heavy operation since it scans all M_{L2} and computes all distances for each *PGF*. The operation is well known as the nearest neighbor search problem. The searching operation is executed $N_{L2} \times N_P$ times per the window slides, where N_{L2} is the number of *WGF*s in M_{L2}, and N_P is the number of *PGF*s in P. To avoid unnecessary computation, we utilize a data structure based on Locality-Sensitive Hashing (LSH) [24] for indexing *WGF*s.



$(h_{a_1}(x), h_{a_2}(x))$	CF
(0, 0)	{A}
(0, 1)	{B}
(1, 1)	{D}
(1, 2)	{C}
(2, 2)	{E}

Figure 3.2: Searching nearest WGF with LSH

The basic concept of LSH is to map similar vectors to hash values which have higher probability of collision than hash values of dissimilar vectors. In other words, if two vectors are close to each other, after projection the vectors remain close. Hash function $h_{\vec{a},b}(\vec{x}) : \mathcal{R}^d \rightarrow \mathcal{N}$ is a scalar projection which maps a vector $\vec{x}$ to an integer. The hash function is given by $h_{\vec{a},b}(\vec{x}) = \lfloor(\vec{a} \cdot \vec{x} + b)/w\rfloor$, 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 $[0, w)$.

General LSH generates a hash table whose hash keys are computed from hash functions to decrease the probability that dissimilar vectors fall into the same quantization bin. A hash key is obtained by concatenating values from the hash functions, e.g., when we have 2 hash functions, key $g(\vec{x})$ is $(h_{\vec{a1},b}(\vec{x})$, $h_{\vec{a2},b}(\vec{x}))$.

However, in data streams, the hash table need to be updated continuously as tuples are inserted and deleted. Updating the hash table and computing hash key carry with it high computational costs. Therefore, we maintain an adequate number of hash functions to update and compute distances from a target tuple to neighbors which are found in the hash table.

Figure 3.2 shows an example of the hash table for finding the nearest *WGF*. We utilize the assumption that if distance x between two vectors A and B 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 bucket of hash tables contains vectors whose distances are within θ. To find close vectors of target vector A, the operation first takes a bucket with the same hash value $g(A)$, and if there is no other elements except itself, it searches for buckets with adjacent hash key values from $g(A) - (1, ..., 1)$ to $g(A) + (1, ..., 1)$. Then the operation computes the real distances to the found vectors. The operation need to takes the elements in $g(A) \pm (1, ..., 1)$ adjacent buckets because the target vector A can be located near the border of the $g(A)$ bucket.

As the number of hash functions increases, the number of adjacent hash keys that need to be searched increases exponentially. For m hash functions, the operation searches for 3^m keys. To prevent this, we use a heuristic that the operation access only buckets with key values that differ by 1 in each component of $g(A)$, which are $2m$. For example, in Figure 3.2, for $g(B) = (0, 1)$, the operation takes elements of $(-1, 1), (0, 0), (1, 1), (0, 2)$ keys.

In Algorithm 2, target vector is the centroid of *PGF*, and vectors in hash tables are centroids of *WGF*s.

3.3 Clustering on 2-Level Coresets

In this section, we present the clustering algorithm with sliding windows for data streams, which performs clustering based on 2-level coresets. In order to reduce the total computation cost of clustering, we add a modification step to the algorithm, which appends new coreset to pre-existing clusters based on the probability distributions of those clusters. The detailed process is presented in Algorithm 3. The algorithm is executed as the window slides.

First, the algorithm uses k-means clustering to produce the clusters based on their features in M_{L2}. Clustering based on GF has been widely studied. The basic and most commonly used methodology is to consider the centroid of *WGF* as a tuple, and perform clustering on them. Clustering on centroid with weight is also commonly used, where weight is the number of tuples N in *WGF*.

If clusters already exist, the algorithm detects new and changed *WGF*s, and assigns each of them to its nearest cluster. This produces approximate clusters. However, it is much faster than performing clustering again.

To preserve clustering quality and decide to perform clustering again, we measure quality degeneration of original and modified clusters by *Kullback – Leibler divergence* (*KL-divergence*) [43]. *KL-divergence* of probability distributions $p(x)$ and $q(x)$ is a measure of information gain achieved if $p(x)$ is used instead of $q(x)$. *KL-divergence* is defined as

$$D_{KL}(p(x)||q(x)) = \sum_x p(x) \log \frac{p(x)}{q(x)}. \tag{3.1}$$

For continuous probability distributions $p(x)$ and (qx), *KL-divergence* is defined as

$$D_{KL}(p(x)||q(x)) = \int_{-\infty}^{\infty} p(x) \log \frac{p(x)}{q(x)}. \tag{3.2}$$

When $p(x)$ and $q(x)$ follow the Gaussian distribution, probability density functions are $p(x) = \mathcal{N}(\sigma_p, \mu_p^2)$ and $q(x) = \mathcal{N}(\sigma_q, \mu_q^2)$. *KL-divergence* is calculated from mean and deviation. The following equations are derived in the Gaussian distribution.

$$D_{KL}(p(x)||q(x)) = \int p(x) \log p(x) dx - \int p(x) \log q(x) dx$$

We have that

$$\int p(x) \log p(x) dx = -\frac{1}{2}(1 + \log 2\pi\sigma_p^2)$$

$$
\begin{aligned}
-\int p(x)\log q(x)dx &= -\int p(x)\log \frac{1}{(2\pi\sigma_q^2)^{(1/2)}} e^{-\frac{(x-\mu_q)^2}{2\sigma_q^2}} dx \\
&= \frac{1}{2}\log(2\pi\sigma_q^2) - \int p(x) - \frac{(x-\mu_q)^2}{2\sigma_q^2} dx \\
&= \frac{1}{2}\log(2\pi\sigma_q^2) + \frac{\int p(x)x^2dx - \int p(x)2x\mu dx + \int p(x)\mu^2 dx}{2\sigma_q^2} \\
&= \frac{1}{2}\log(2\pi\sigma_q^2) + \frac{E(x^2) - 2E(x)\mu_q + \mu_q^2}{2\sigma_q^2}
\end{aligned}
$$

Variance is computed as $var(x) = E(x^2) - E(x)^2$, and it holds that

$$
\begin{aligned}
-\int p(x)\log q(x)dx &= \frac{1}{2}\log(2\pi\sigma^2) + \frac{\sigma_p^2 + \mu_p^2 - 2\mu_p\mu_q + \mu_q^2}{2\sigma_q^2} \\
&= \frac{1}{2}\log(2\pi\sigma_q^2) + \frac{\sigma_p^2 + (\mu_p - \mu_q)^2}{2\sigma_q^2}
\end{aligned}
$$

Based on the equations, we have that

$$
\begin{aligned}
D_{KL}(p(x)||q(x)) &= \int p(x)\log p(x)dx - \int p(x)\log q(x) \\
&= -\frac{1}{2}(1 + \log 2\pi\sigma_p^2) + \frac{1}{2}\log(2\pi\sigma_q^2) + \frac{\sigma_p^2 + (\mu_p - \mu_q)^2}{2\sigma_q^2} \qquad (3.3) \\
&= \log\frac{\sigma_q}{\sigma_p} + \frac{\sigma_p^2 + (\mu_p - \mu_q)^2}{2\sigma_q^2} - \frac{1}{2}
\end{aligned}
$$

When the data distribution does not follow the Gaussian distribution, the algorithm needs to select the probability distribution that best fits to a dataset first. Selecting the distribution determines how well the candidate distributions fit to the dataset using the specific goodness of fit tests such as Kolmogorov-Smirnov test.

We assume that distances between a centroid of *WGF* and a center of the assigned cluster follow the Gaussian distribution. k-Means clustering is designed to works well and generates high-quality clusters for the data which follow Gaussian distribution. Therefore, it is a reasonable assumption that data streams and clusters follow the Gaussian distribution. While performing clustering, cluster statistics information is easy to generate and store. We maintain sum of distances, sum of squared distances, and the number of *WGF*s with the cluster, and averages and deviations can be calculated from these. When a cluster is modified, the statistics information can be updated in a constant time because it has additivity property.

In Algorithm 3, the error bound e adjusts how much the algorithm tolerates the error. If the e is small, the algorithm performs clustering frequently. The appropriate value of e depends on datasets, and is tested experimentally. According to our experimental results, this step does not seriously decrease clustering quality.

Algorithm 2 UPDATECORESETS

```
    Input: Stream $S$, threshold $\theta$, range $R$, slide $L$, level-1 synopses $M_{L1}$, and level-2 synopses $M_{L2}$
    Output: updated $M_{L1}$ and $M_{L2}$
 1: if exist expired tuples for $R$ in $M_{L1}$ then
 2:     $E \leftarrow$ expired PGFs in $M_{L1}$
 3:     subtract $E$ from $M_{L2}$
 4:     truncate the column of $E$ in $M_{L1}$
 5:     append new empty column in $M_{L1}$
 6: end if
 7: $B \leftarrow$ recent tuples of $(R - L, R]$ in $S$
 8: $P \leftarrow$ CREATEPANEGF$(B, \theta)$
 9: for each $p \in P$ do
10:     $WGF_i \leftarrow$ nearest $WGF$ to $p$ found by LSH of $M_{L2}$
11:     if $dist(p, WGF_i) < \theta$ or $dist(p, WGF_i) <$ radius of $WGF_i$ then
12:         $WGF_i \leftarrow WGF_i + p$
13:         append $p$ at $i^{th}$ row and last column in $M_{L1}$
14:     else
15:         create new $WGF$ based on $p$
16:         append new $WGF$ to $M_{L2}$
17:         append $p$ at new row and last column in $M_{L1}$
18:     end if
19:     update LSH of $M_{L2}$
20: end for
21: return $M_{L1}$, $M_{L2}$
```

Algorithm 3 Clustering

```
Input: Level-2 coresets $M_{L2}$, the number of clusters $k$, and error bound $e$
Output: Clusters $C$
1: $C_p \leftarrow$ pre-exist clusters
2: if $C_p$ is empty then
3:     $C \leftarrow$ clusters which are created by k-means clustering using $M_{L2}$
4: else
5:     $W \leftarrow$ new and changed $WGF$s in $M_{L2}$
6:     $C \leftarrow$ clusters which are modified by assigning each $w \in W$ to its nearest cluster of $C_p$
7:     $\alpha \leftarrow$ KL-divergence between $C_p$ and $C$
8:     if $\alpha > e$ then
9:         $C \leftarrow$ clusters which are created by k-means clustering using $M_{L2}$
10:    end if
11: end if
12: return $C$
```

Chapter 4

CSCS: Coreset-based Data Stream Clustering with Sliding Windows

In this chapter, we present data stream clustering algorithm that improves GFCS. Our algorithm GFCS works well for data streams with sliding windows, but has several drawbacks. GFCS needs predefined threshold before clustering, and the threshold could not be modified even if the data evolves considerably over time. If the threshold is set to smaller than the appropriate value, data reduction is not performed well so that a large number of GFs are generated. In addition, GFCS reduces the cost of distance calculation by efficient finding the close tuples to be calculated. However, the cost of single distance calculation is high for high-dimensional data.

We address these problems by constructing Locality-Sensitive Hashing (LSH) based coreset, and propose theoretical analysis on coreset and sliding windows. In addition, we improve *re-clustering policy*, which assumes the Gaussian distribution in GFCS, in order to cope with other distribution.

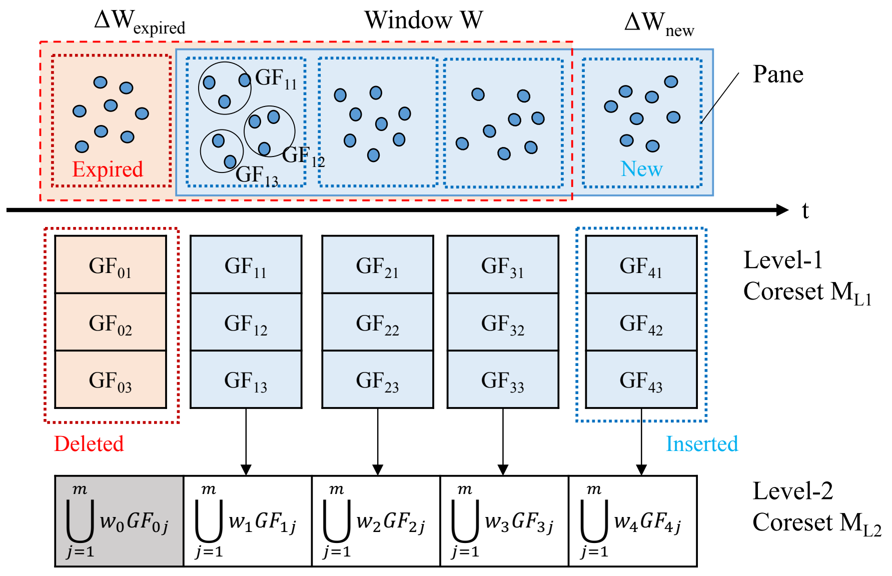


Figure 4.1: Coreset maintenance in the sliding window

4.1 Coreset Construction based on Nearest Neighbor Search

4.1.1 Algorithm for Coreset Construction

In this section, we describe a method for constructing and maintaining coreset with sliding windows. As described in GFCS, it is impossible to keep all values of tuples to be subtracted from a window. To maintain the values efficiently, we utilize 2-level coresets which is proposed in GFCS. The 2-level coresets is based on pane-based aggregation technique for sliding window [46].

Overview of 2-level coresets for CSCS is shown in Figure 4.1. A window is split into panes. Level-1 contains respective summaries for each pane. There are the same number of GFs in each summary. Level-2 coreset is created by combining GFs in level-1 coreset. Like GFCS, the window is updated in pane units. When the window slides, GFs for $\Delta W_{expired}$ are removed, and GFs for ΔW_{new} are appended to level-1 and level-2 coresets. GF includes LS, SS,

N, and T. Detailed process of creating GF is described in Algorithm 4 and Algorithm 5.

Algorithm 4 shows process of creating a coreset of size m for a set of tuples B. First, the algorithm selects initial tuple(b_1) and random samples of a certain size(M). The size of the samples depends on dataset, but it does not significantly affect overall performance. In our experiment setup, we used $10k$ or 5% of the tuples in the pane. After the selection, the algorithm calculates the distance between b_1 and M, uses the minimum value as threshold θ (line 4). The reason for using the minimum instead of the maximum and the average is that the threshold becomes very large when an outlier exists. GF with large radius contains too many tuples which cannot be split. However, even if a large number of GFs are generated with a small threshold, they are reduced to GFs of appropriate radius and numbers through ReduceCoreset. After determining the threshold θ, tuples whose distances are below θ are grouped into the same GF. If distance between a tuple and centroid of GF, $dist(b, GF_p)$ is below θ or radius of GF, the GF absorbs the tuple (line 10).

GFs are continuously generated with θ. When the number of GFs reaches $2m$, they are reduced to m through ReduceCoreset in Algorithm 5. Since all GFs in coreset P are merged once through ReduceCoreset, the number of coreset is 1/2 of initial size. The algorithm is based on nearest neighbor search. Specifically, target GF p is added to the closer GF of unprocessed set R and processed set Q. After that, added GF is included in Q (line 8-13). If the size of input coreset is $2m$, all GFs should be processed at least once to reduce the size to m. If the size of 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 processing ends.

As shown in Figure 4.1, data structure for sliding window consists of level-1 coreset and level-2 coreset. Level-1 coreset have $\lceil R/L \rceil$ columns, and each

Algorithm 4 ConstructCoreset

```
    Input: A set of tuples $B$, coreset size $m$
    Output: Coreset $P$
 1: create empty set $P$
 2: $b_1 \leftarrow$ an initial tuple in $B$
 3: $M \leftarrow$ random samples of size $|B|/20$ from $B$
 4: $\theta \leftarrow \min_{m \in M} dist(b_1, m)$
 5: for each $b \in B$ do
 6:     if $P$ is empty then
 7:         create new $GF_b$ based on $b$, and $P \leftarrow P \cup \{GF_b\}$
 8:     else
 9:         $GF_p \leftarrow$ nearest $GF$ in $P$ to $b$
10:         if $dist(b, GF_p) < \theta$ or $dist(b, GF_p) <$ radius of $GF_p$ then
11:             $GF_p = GF_p + b$
12:         else
13:             create new $GF_b$ based on $b$, and $P \leftarrow P \cup \{GF_b\}$
14:         end if
15:     end if
16:     if $|P| \geq 2m$ then
17:         $P \leftarrow$ ReduceCoreset($P$, $m$)
18:     end if
19: end for
20: if $|P| \geq m$ then
21:     $P \leftarrow$ ReduceCoreset($P$, $m$)
22: end if
23: return $P$
```

column contains m GFs. GFs are generated from tuples in new pane through ConstructCoreset, and they are inserted into the last column of level-1

Algorithm 5 ReduceCoreset

```
    Input: Coreset $P$, reduced coreset size $m$
    Output: Reduced coreset $Q$
 1: create empty set $Q$
 2: $R \leftarrow P$
 3: for each $GF_p \in P$ do
 4:     if $GF_p \notin R$ then
 5:         continue
 6:     end if
 7:     $R \leftarrow R - \{GF_p\}$
 8:     $GF_q \leftarrow$ nearest $GF$ in $Q$ to $GF_p$, and $GF_r \leftarrow$ nearest $GF$ in $R$ to $GF_p$
 9:     if $dist(GF_p, GF_q) < dist(GF_p, GF_r)$ then
10:         $GF_q \leftarrow GF_q + GF_p$, and $Q \leftarrow Q \cup \{GF_q\}$
11:     else
12:         $GF_r \leftarrow GF_r + GF_p$, $Q \leftarrow Q \cup \{GF_r\}$, and $R \leftarrow R - \{GF_r\}$
13:     end if
14:     if $|Q| + |R| \leq m$ then
15:         $Q \leftarrow Q \cup R$
16:         break
17:     end if
18: end for
19: return $Q$
```

coreset. Timestamps Ts of the inserted GFs are in $(R - L, R]$. When expired tuples are removed, first column of level-1 coreset whose timestamps are in $[t_1, t_1 + L]$ is truncated, where t_1 is earliest timestamp. Removing operation for level-1 coreset has $O(1)$ time complexity if an adequate data structure is utilized such as linked list queue. The exact number of tuples in the level-1 coreset is R, and remains constant.

Level-2 coreset is built by unioning GFs which are in the same column. Each GF in level-1 coreset is multiplied with weights and merged into level-2 coreset. The weights can be defined as either time-decay function or simply same values. w_i is the weights for GF_i. Level-2 coreset is represented as

$$GF_i = \bigcup_{j=1}^{m} w_j GF_{ij}$$

Sliding window updates level-2 coreset by appending new GFs and truncating expired GFs in the same way as level-1 coreset. Clustering is performed on generated level-2 coreset.

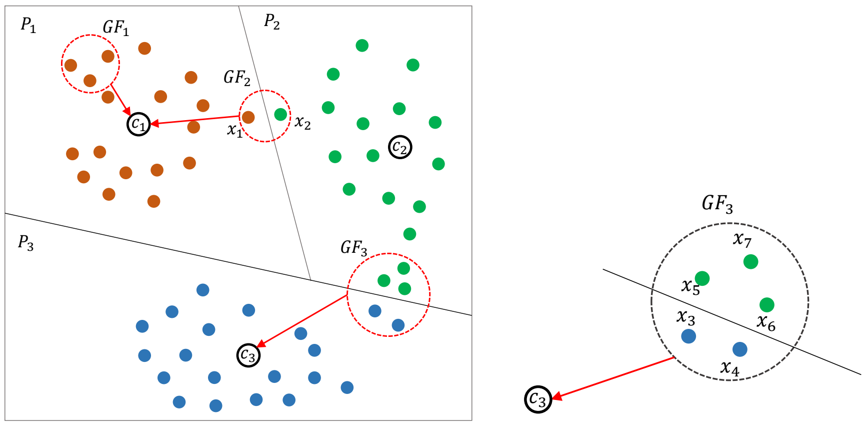
Algorithm 4, ConstructCoreset, takes linear time with respect to the size of input $|B|$. The most time consuming operation of the algorithm is nearest GF search, and an efficient algorithm to improve search time will be described in a following section.

4.1.2 Theoretical Analysis of Coreset Construction

In this section, we investigate accuracy guarantee of coreset which is constructed by ConstructCoreset in Algorithm 4. As we have stated in Section 2.4, coreset with approximation guarantee cannot be computed for data streams. Therefore, we show the upper bound of k-means cost for our approach with assumption that tuples are not inserted / deleted / moved and optimal centers are known. We start by analyzing cost for $k = 1$.

Lemma 4.1. *Let S be a set of tuples, and M be a set of group features generated from S, and C is the optimal cluster centers. For $k = 1$, it holds that*

$$cost(S, C) = cost(M, C)$$



(a) (b)

Figure 4.2: Constructing GFs by CONSTRUCTCORESET

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

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

In Figure 4.2a, nearest tuples are grouped into the GF_1. Nearest center for all tuples in GF_1 is center c_1. Therefore, nearest center for GF_1 is also center c_1. In this case, cost of GF_1 is optimal 1-mean cost according to Lemma 4.1. If all GFs are generated only from tuples in the same cluster, k-means cost calculated from the GFs is optimal.

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 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.

Based on the assumption, following equations are established.

$$\|x_1 - c_1\| > \|x_1 - x_2\| \tag{4.1}$$

$$\|x_2 - c_2\| > \|x_1 - x_2\| \tag{4.2}$$

$$\|\frac{x_2 + x_1}{2} - c_2\| > \|\frac{x_2 + x_1}{2} - c_1\| \tag{4.3}$$

Lemma 4.2. *Let x be a tuple in GF, and c_x be nearest center of x, and c_g is nearest center of GF. For arbitrary tuple $z \in GF$,*

$$\|x - c_g\|^2 < 2(\|x - c_x\|^2 + \|z - c_g\|^2) \tag{4.4}$$

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

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

Equation (4.2) implies $\|x_2 - c_2\| > \|x_2 - z\|$. This yields that

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

□

We show generalized cost considering GF which contains two or more tuples as shown in Figure 4.2b.

Theorem 4.3. *Let S be a set of tuples, and M be a set of group features generated from S, and C is the optimal cluster centers, and Q is a set of incorrect assigned tuples. It holds that*

$$cost_w(M, C) < cost(S, C) + 9 \sum_{x \in Q} \|x - c_x\|^2 \tag{4.5}$$

Proof. For intuitive presentation, we start by cost of GF_3 in Figure 4.2b. Optimal cost with GF_3 is

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

Let constructed coreset is M, and GF_3 is assigned to c_3. Euclidean distance between GF_3 and c_3 is $\|\frac{x_3+x_4+x_5+x_6+x_7}{5} - c_3\|$. Cost for M is

$$\begin{aligned} cost_w(M, C) = &\sum_{x \in P_3 - \{x_3, x_4\}} \|x - c_3\|^2 + \sum_{x \in P_2 - \{x_5, x_6, x_7\}} \|x - c_2\|^2 \\ &+ 5 \cdot \|\frac{x_3 + x_4 + x_5 + x_6 + x_7}{5} - c_3\|^2. \end{aligned} \tag{4.6}$$

Note that we consider only unweighted input tuples for clear explanation. Coreset construction can be naturally extended to weighted tuples.

Equation for cost of assigning x_5, x_6, x_7 incorrectly to c_3 instead of c_2 is $\|x_5 - c_3\|^2 + \|x_6 - c_3\|^2 + \|x_7 - c_3\|^2$. This equation with cost $\|x_3 - c_3\|^2 + \|x_4 - c_3\|^2$ is greater than $5 \cdot \|\frac{x_3 + \cdots + x_7}{5} - c_3\|^2$ in Equation (4.6). Therefore, the equation can be used as the cost of GF_3 for upper bound, and we analyze the cost with the equation.

We define cost of GF_3 is

$$\begin{aligned} cost_w(GF_3, C) = &\|x_3 - c_3\|^2 + \|x_4 - c_3\|^2 \\ &+ \|x_5 - c_3\|^2 + \|x_6 - c_3\|^2 + \|x_7 - c_3\|^2. \end{aligned}$$

Cost difference between M and C is

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

By using Lemma 4.2, we have

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

Tuple z can be arbitrary tuple in GF_3, and we set z to the center of GF_3, c_{g3}. By Equation (4.3), 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 triangle inequality. We also get $\|x_5 - c_2\| > \|x_5 - c_{g3}\|$ by Equation (4.2). Based on these equations, we have

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

Cost difference is that

$$cost_w(M,C) - cost(S,C) < 9\|x_5 - c_2\|^2 + 9\|x_6 - c_2\|^2 + 9\|x_8 - c_2\|^2.$$

For generalizing above equation, let Q be a set of incorrect assigned tuples, and c_x be the nearest for x in optimal solution. Then,

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

□

Theorem 4.3 means that the cost does not exceed 9 times of the optimal cost of incorrect assigned tuples. If GF spans more than three clusters, it can be divided into GFs containing only tuples of two clusters.

However, optimal solution of $cost(S, C)$ cannot be obtained as mentioned in Section 2.4. If the solution is obtained by α-approximation algorithm, we have

$$cost(S, C) = \alpha \cdot cost^k_{OPT}(S) \tag{4.7}$$

For computing upper bound of the cost, we 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 $(\alpha + 9)cost^k_{OPT}(S)$ is relatively high for accuracy guarantee, but may reach in worst case. If the number of tuples which are assigned incorrectly is small, the cost would be close to the optimal cost of the algorithm.

4.1.3 Theoretical Analysis of Sliding Windows

Theorem 4.3 is cost of reducing tuples from S to M. We now analyze cost for sliding window. Given `RANGE` R and `SLIDE` L, sliding window contains $b = \lceil R/L \rceil$ panes. A pane includes $|S|/b$ tuples, and coreset which contains $|M|/b$ GFs is produced. To present accuracy guarantee, we adopt a methodology which merge each summary after partitioning entire tuples for k-Median problem [34, 35].

Lemma 4.4. *Let $S_1, ..., S_p$ 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, cost of a partition is equal to the sum of squared distance of the tuples which the partition contains. □

Lemma 4.5. *Let M be a set of group features which are generated from S, and $S_1, ..., S_p$ be arbitrary partitions of a set S. Coresets $M_1, ..., M_p$ are generated from $S_1, ..., S_p$ respectively. Then,*

$$\sum_{i=1}^{p} cost_w(M_i, C) \leq (\alpha + 9) cost_{OPT}^{k}(S) \tag{4.8}$$

Proof. From Lemma 4.4, we have fact that cost of a tuple set is the sum of costs of its partitions. Let C be a set of k centers, Q_i be wrong assigned tuples in M_i.

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

As before, considering the worst case that all tuples are assigned incorrectly, following equation holds that

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

□

Based on Lemma 4.5, the cost difference between $M_1, ..., M_p$ 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 incorrect assigned tuples, cost of $M_1, ..., M_p$ and cost of M become equal.

Now, we consider a case where window slides. Let tuples in sliding window with panes be $S = \{S_1, ..., S_p\}$. As the window slides, S_1 is deleted and S_{p+1} is appended. Let updated sliding window be $S^* = \{S_2, ..., S_{p+1}\}$.

Previously, we compute cost for optimal centers C with the assumption that the tuples are not changed. However, because the tuples are appended and deleted, optimal centers are also changed. Let a set of optimal centers for S^* be C^*. We compare the costs from the centers C to the centers C^*.

As S_1 is removed from the window and S_{p+1} is added to the window, coreset M_1 is also removed, and coreset M_{p+1} is newly created and added. Since each partition does not overlap with each other by Lemma 4.4, addition and deletion do not affect cost of coresets in another partitions.

Theorem 4.6. *Let data stream be $S = \{S_1, ..., S_p\}$, and updated sliding window be $S^* = \{S_2, ..., S_{p+1}\}$, and M_i be a generated coreset from S_i, and C and C^* be optimal centers of S and S^* respectively. For the common tuples in S and*

S^*, it holds that

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

, 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

$$\|x - c_x^*\| < \|x - c_x\| + \|c_x - c_x^*\|$$

$$\|x - c_x^*\|^2 < 2\|x - c_x\|^2 + 2\|c_x - c_x^*\|^2$$

This expression is summed for every x in $\bigcup_{i=2}^{p+1} M_i$. For presentation, we replace $2\sum_{x\in\bigcup_{i=2}^{p} M_i} \|c_x - c_x^*\|^2$ by γ.

$$\begin{aligned}
\sum_{x\in\bigcup_{i=2}^{p} M_i} \|x - c_x^*\|^2 &< 2\sum_{x\in\bigcup_{i=2}^{p} M_i} \|x - c_x\|^2 + 2\sum_{x\in\bigcup_{i=2}^{p} M_i} \|c_x - c_x^*\|^2 \\
&= 2\sum_{x\in\bigcup_{i=2}^{p} M_i} \|x - c_x\|^2 + \gamma \\
&< 2\sum_{x\in\bigcup_{i=2}^{p} M_i} \|x - c_x\|^2 + 2\sum_{x\in M_1} \|x - c_x\|^2 + \gamma
\end{aligned}$$

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

□

Theorem 4.6 shows that cost for window after removing expired partition is about twice as much as cost for previous window. By appending cost for new partition, cost for updated window is

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

This indicates that the cost for the updated window is increased by the cost for new tuples, $cost_w(M_{p+1}, C^*)$, regardless of the cost of previous windows. Therefore, the data stream clustering using the partitioning and the sliding windows can be processed within a certain upper bound of error.

Consider the cost when we use the old centers C instead of new centers C^* without clustering. First, it holds that $cost_w(\bigcup_{i=2}^{p+1} M_i, C^*) < cost_w(\bigcup_{i=2}^{p+1} M_i, C)$ because C^* is optimal solution. By the triangle inequality, it also holds that $\|x - c_x\|^2 < 2\|x - c_x^*\|^2 + 2\|c_x - c_x^*\|^2$. Similar as above, we have

$$
\begin{aligned}
\sum_{x \in \bigcup_{i=2}^{p+1} M_i} \|x - c_x\|^2 &< 2 \sum_{x \in \bigcup_{i=2}^{p+1} M_i} \|x - c_x^*\|^2 + 2 \sum_{x \in \bigcup_{i=2}^{p+1} M_i} \|c_x - c_x^*\|^2 \\
\Leftrightarrow \quad cost_w(\bigcup_{i=2}^{p+1} M_i, C) &< 2cost_w(\bigcup_{i=2}^{p+1} M_i, C^*) + 2 \sum_{x \in M_{p+1}} \|c_x - c_x^*\|^2 + \gamma \\
&< 4cost_w(\bigcup_{i=1}^{p} M_i, C) + 2cost_w(M_{p+1}, C^*) + 2\gamma \\
&\quad + 2 \sum_{x \in M_{p+1}} \|c_x - c_x^*\|^2 + \gamma \\
&< 4cost_w(\bigcup_{i=1}^{p} M_i, C) + 2cost_w(M_{p+1}, C^*) + \delta
\end{aligned}
$$

This equation means that the cost for C does not exceed twice the cost for C^*. However, comparing error bounds, it is better to perform clustering because it guarantees lower cost.

4.2 Coreset Construction based on Locality-Sensitive Hashing

Our proposed algorithm is designed to take into account several problems in [9, 68]: 1) appropriate threshold should be determined according to dataset 2)there is no limit to the number of groups 3) expired tuples may exist in synopses. In addition, there are still time-consuming operations to be improved for practical use. Assume that we have a coreset $M = \{GF_1, ..., GF_m\}$ from a set of input tuples $\{x_1, ..., x_i\} \subset B$, where $|M| = m$, $|B| = n$, and $i < n$. For next input tuple $x_{i+1} \in B$, Algorithm 4 need to find nearest neighbor $GF \in M$ of x_{i+1}. Finding the nearest GF is a computationally heavy operation since it scans all M and computes all distances for each GF. Search operation requires $O(dmn)$ time to obtain a coreset per window slides, which is too slow for large window and high-dimensional data. Furthermore, Algorithm 5 computes distances between all pairs of GFs in the coreset, and it requires $O(dm^2)$ time. Therefore, we propose an improved algorithm based on concept of Locality-Sensitive Hashing (LSH) [24] for reducing distance computation. Proposed algorithm also works for the data streams.

Basic concept of LSH in Euclidean space is to map similar vectors to hash values which have higher probability of collision than hash values of dissimilar vectors. In other words, if two vectors are close to each other, after projection the vectors remain close. Hash function $h_{\vec{a},b}(\vec{x}) : \mathcal{R}^d \rightarrow \mathcal{N}$ is a scalar projection which maps a vector $\vec{x}$ to an integer. The hash function is given by $h_{\vec{a},b}(\vec{x}) = \lfloor(\vec{a} \cdot \vec{x} + b)/w\rfloor$, where $\vec{a}$ is a randomly drawn d-dimensional vector, w is width of the quantization bin, and b is a random variable in interval $[0, w)$.

General LSH generates hash values which are computed from hash functions to decrease the probability that dissimilar vectors fall into the same quantization bin. We define global hash function which is obtained by concatenating values from multiple hash functions, e.g., when we have 3 hash functions, 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 hash value, it is acceptable load if we use the proper number of hash functions.

Main idea of our algorithm is that vectors with near distance have similar values of LSH. Based on the concept of LSH, if 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 width of the quantization bin w of the hash function to θ, each element of hash table contains vectors whose distances are within θ. Distance computation is not necessary to combine vectors within the distance θ.

Note that vectors of the same hash value are not always within a certain distance. In particular, if hash function h is (r_1, r_2, p_1, p_2)-sensitive with respect to $dist(\cdot, \cdot)$, then it has the following properties:

$$\text{If } dist(x, y) \leq r_1, \text{ then } Pr[h(x) = h(y)] \geq p_1.$$

$$\text{If } dist(x, y) > r_2, \text{ then } Pr[h(x) = h(y)] \leq p_2.$$

For Euclidean distance, if distance of two vectors is that $dist(x_1, x_2) < \theta/2$, then the probability that hash values of the vectors are equal is that $Pr[h(x_1) = h(x_2)] > 1/2$. If $dist(x_1, x_2) > 2\theta$, then $Pr[h(x_1) = h(x_2)] < 1/3$. This yields $(\theta/2, 2\theta, 1/2, 1/3)$-sensitive. Probability of mapping similar vectors to the same hash value is proportional to the number of hash functions. As the number of hash functions increases, the error probability of finding neighbors incorrectly is reduced. If we have l hash functions, the probability is $(1 - p_2^l)$.

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

Algorithm 6 describes detailed process of creating a coreset based on LSH. First, the algorithm initialize hash table H and global hash function g. The number of hash functions is given by user, and it creates d-dimensional random unit vector $a_1, ..., a_l$. Process of obtaining the threshold value θ is omitted because the same method is used in Algorithm 4. Global hash function g is defined using the unit vectors and the θ. For each tuple b, the algorithm

Algorithm 6 CONSTRUCTCORESETTABLE

```
    Input: A set of tuples $B$, coreset size $m$, number of hash functions $l$
    Output: Coreset $P$
 1: initialize hash table $H$, and hash function $g(x) = (h_{\vec{a_1},\theta}(\vec{x}), ..., h_{\vec{a_l},\theta}(\vec{x}))$
 2: for each $b \in B$ do
 3:     if $g(b) \notin H$ then
 4:         create new $GF_b$ based on $b$, and $H[g(b)] = GF_b$
 5:     else
 6:         $GF_b \leftarrow H[g(b)]$
 7:         $GF_b = GF_b + p$, and $H[g(b)] = GF_b$
 8:     end if
 9:     if $|H| \geq 2m$ then
10:         $H \leftarrow$ REDUCECORESETTABLE($H$, $m$)
11:     end if
12: end for
13: if $|H| \geq m$ then
14:     $H \leftarrow$ REDUCECORESETTABLE($H$, $m$)
15: end if
16: return $GF$s in $H$
```

generate hash value by $g(b)$. If the hash value $g(b)$ does not exist in the hash table, it creates 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 REDUCECORESETTABLE. We keep 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. To calculate distance between new input tuple and GFs more precisely, the center of the GF should be indexed. However, since center of GF continues to be changed as tuples are appended, it is inefficient to update

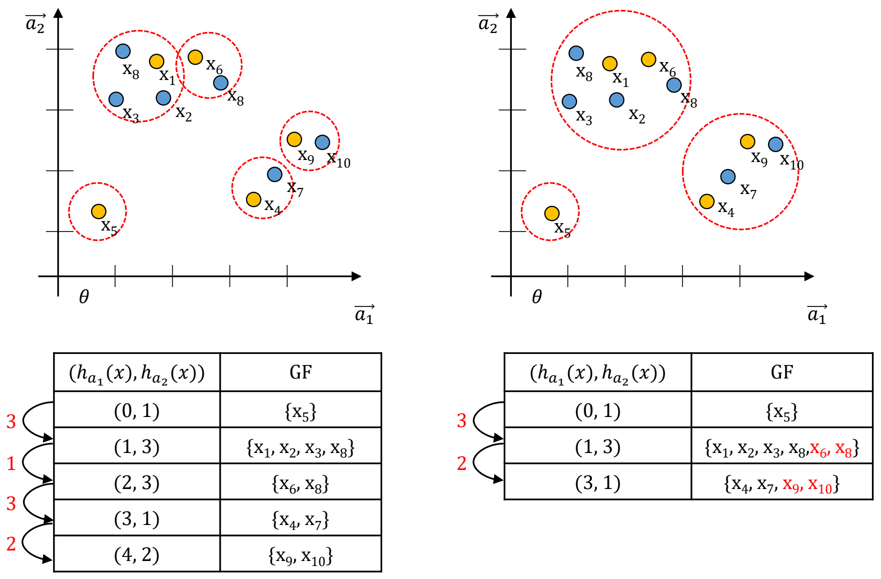
Algorithm 7 ReduceCoresetTable

```
    Input: Hash table $H$, reduced size $m$
    Output: Reduced hash table $H$
 1: while $|H| < m$ do
 2:     $Q \leftarrow$ a set of key pairs with minimal difference in hash value
 3:     for each $(x, y) \in Q$ do
 4:         $GF_x \leftarrow H[g(x)]$, and $GF_y \leftarrow H[g(y)]$
 5:         $GF_x = GF_x + GF_y$
 6:         $H[g(x)] = GF_x$, and remove the bucket of $g(y)$ from $H$
 7:         if $|H| \leq m$ then
 8:             break
 9:         end if
10:     end for
11: end while
12: return $H$
```

key of hash table every time. We use the fixed key of the index tuple with some error tolerance.

The algorithm does not perform operations to find nearest neighbor and calculate distances. The time complexity is linear, which is $O(dln)$, where search time of the hash table is $O(1)$, generation time of the hash value is $O(dl)$, and time of ReduceCoresetTable is ignored. Since $l \ll m$ and $O(dln) < O(dmn)$, Algorithm 6 is done faster than Algorithm 4. In particular, for high-dimensional data, Algorithm 6 is efficient because the cost of distance calculation is much higher than cost of retrievals. The algorithm uses at most $2m$ space.

Algorithm 7 shows process of reducing hash table. The algorithm requires a list of key pairs ordered by the difference of keys. Calculation of differences of all key pairs is $O(lm^2)$, which is less than the original $O(dm^2)$, but it is still



$\vec{a_2}$
$\vec{a_1}$
θ
x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10}
3
1
3
2
3
2

$(h_{a_1}(x), h_{a_2}(x))$	GF
(0, 1)	$\{x_5\}$
(1, 3)	$\{x_1, x_2, x_3, x_8\}$
(2, 3)	$\{x_6, x_8\}$
(3, 1)	$\{x_4, x_7\}$
(4, 2)	$\{x_9, x_{10}\}$

$(h_{a_1}(x), h_{a_2}(x))$	GF
(0, 1)	$\{x_5\}$
(1, 3)	$\{x_1, x_2, x_3, x_8, x_6, x_8\}$
(3, 1)	$\{x_4, x_7, x_9, x_{10}\}$

Figure 4.3: Merging buckets in hash table for coreset

heavy computation operation. Therefore, we adopt a simple heuristic for linear time complexity.

Heuristic method requires a list of keys which is ordered by comparing all components of the hash values. Sorted list can be created in the REDUCECORESETTABLE, but we maintain additional sorted list of keys in CONSTRUCTCORESETTABLE, which is updated whenever data arrive. With the sorted list, we compute difference between only adjacent buckets, not all pairs. We organized the key pairs with the difference in the computed result, and the key pairs are merged in order of small difference.

Figure 4.3 shows example of merging buckets. The keys are sorted, and the difference is computed. First, the pairs of 1 difference are merged. After merging, differences are computed again for newly adjacent buckets. Because desired size is not reached, the pairs of 2 difference are merged. In Algorithm 6, since REDUCECORESETTABLE is called when the size is $2m$, the merging operation is performed m times. In addition, GF maintains pointers to their adjacent GFs. This is additional component in GF to reduce storing cost. The pointer component also has additive property.

4.3 Re-clustering Policy

We perform weighted k-means clustering using weighted centroids of GFs in coreset. Centroid is computed by LS/N, and N is used as weight. There is no additional computation cost for computing 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 total computation cost of clustering, we add a re-clustering step to the algorithm, which appends newly created GFs to pre-existing clusters based on the probability distributions of those clusters. Detailed process is presented in Algorithm 8. The algorithm is executed when window slides.

Algorithm 8 presents a clustering algorithm with LSH-based coreset. The algorithm is similar to that of GFCS in Section 3.3. Algorithm 8 accepts reduced coreset as input which is created by Algorithm 4 instead of WGF. The algorithm also appends newly created GFs to pre-existing clusters to avoid repeated clustering. We use *Kullback-Leibler divergence*to measure quality degeneration of original and modified clusters. *Kullback-Leibler divergence* is presented in Equation (3.1).

For computing *KL-divergence*, we model quality degradation by the location of cluster center, which is changed by the appended data. If appended data fits the cluster well, the cluster center will not be changed significantly. If appended data is very difference from data which already exist in the cluster, new cluster center which computed including appended data will also move a lot from its original location.

In GFCS, we assume that data distribution does not follow the Gaussian distribution. However, since this assumption does not apply to all datasets, we

Algorithm 8 CLUSTERING

```
    Input: Coreset $P$, the number of clusters $k$, and error bound $\varepsilon$
    Output: Clusters $C$
 1: $C_p \leftarrow$ pre-exist clusters
 2: if $C_p$ is empty then
 3:     $C \leftarrow$ clusters which are created by k-means clustering using $P$
 4: else
 5:     $M \leftarrow$ new $GF$s in $P$
 6:     $C \leftarrow$ clusters which are modified by assigning each $x \in M$ to its nearest cluster of $C_p$
 7:     $\alpha \leftarrow$ KL-divergence between $C_p$ and $C$
 8:     if $\alpha > \varepsilon$ then
 9:         $C \leftarrow$ clusters which are created by k-means clustering using $P$
10:     end if
11: end if
12: return $C$
```

propose a new probability distribution to measure clustering quality.

Usually, quality of clusters is measured by the sum of squared distance which is presented in Definition 2.1. However, it is not suitable as a quality measurement because the optimal centers are unknown, and range of values varies depending on data. We define probability mass function for probability distribution of a cluster as ratio of squared distance from the center to a tuple. Base concept is introduced in *k-means++* [7], and we modify it for the renovation policy. The function for x in cluster P is defined as

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

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

Error bound ε is required for clustering to adjusts how much the algorithm tolerates the error. The value is determined experimentally. Lower values allow more frequent clustering.

Chapter 5

Empirical Evaluation of Data Stream Clustering with Sliding Windows

5.1 Experimental Setup

We evaluated efficiency and scalability of our clustering algorithms, GFCS and CSCS, on synthetic and real-world datasets. GFCS(Group Features based Clustering with Sliding windows) is basic approach which is presented in our previous paper [64]. GFCS also maintains panes for window, and creates GFs based on predefined threshold. GFCS exploits simplified LSH for finding nearest GF, but there is no reduction step in GFCS. CSCS(CoreSet based Clustering with Sliding windows) is an improved algorithm based on LSH. We compared our algorithms with recent data stream clustering clustering algorithms, *SWClustering* [68], *StreamKM++* [2], *ClusTree* [42], and *G2CS* [10]. We also measured performance of basic k-means clustering on raw tuples, which is implemented by *Lloyd*'s algorithm [47].

All algorithms were implemented by Java. *StreamKM++* and *ClusTree* were implements based on MOA framework [13]. We implemented GFCS, CSCS, *SWClustering*, *G2CS* and *Lloyd*'s algorithm from scratch in Java. For proper

Table 5.1: Real-world Datasets

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

Table 5.2: Synthetic Datasets

Dataset	**Size**	**Dim.**
syn1k30d40	2,000,000	40
syn1k30d80	2,000,000	80
syn1k30d160	2,000,000	160
syn1k30d200	2,000,000	200

comparison, G2CS is implemented except for 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. Maximum Java heap size (-Xmx option) is set to 8GB.

Table 5.1 and Table 5.2 show the real-world and synthetic datasets for experiments. We generate data which follow the Gaussian distribution and have 30 clusters with dimensions of 40, 80, 160, and 200. Kddcup99 is network data streams to detect network intrusion. Kddcup99 contains logs of TCP connection of network at MIT Lincoln Labs of 2 weeks. This dataset is used to evaluate 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++*.

To evaluate efficiency and scalability, we measure total processing time and processing time of each sliding. To evaluate quality of 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 better quality of clusters. In sliding windows, the algorithm produces multiple results as the window moves. Therefore, the quality is evaluated by averaging SSQ of the results.

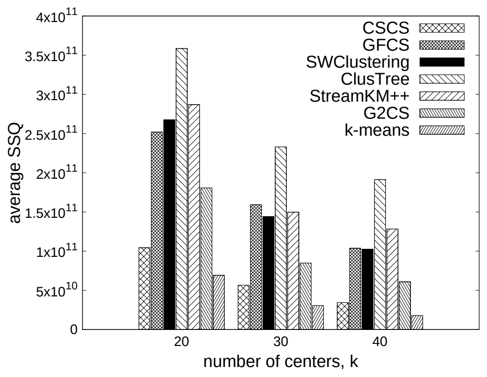
5.2 Experimental Results

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

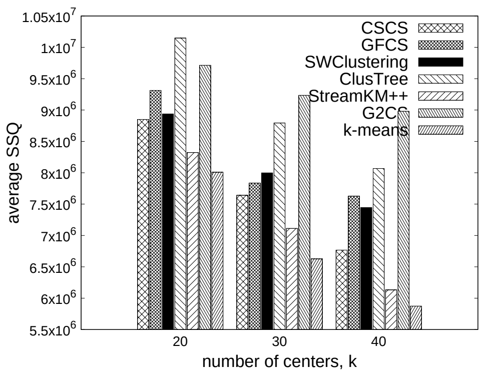
Clustering quality and speed are in trade-off relationship with the values. We tested several parameters and error bounds, and select the values to generate best clustering quality. The parameters of competitive algorithms are based on values showing good quality for the same datasets by the authors.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

Figure 5.1 shows clustering quality of the algorithms for different values of k with real-world datasets. We fix `RANGE` = 100,000 and `SLIDE` = 10,000. We observe that our algorithm CSCS loses some accuracy, but it is comparable to other algorithms such as *StreamKM++* and *ClusTree*. Basic k-means in the experiment shows the best quality because it performs clustering on whole tuples in the window without any summarization. As described in Section 2.6, *SWClustering* contained expired tuples in the synopses. *SWClustering* contained synopses of 155,485 tuples, not 100,000 at 200,000 timestamp for census1990 at $k = 40$.

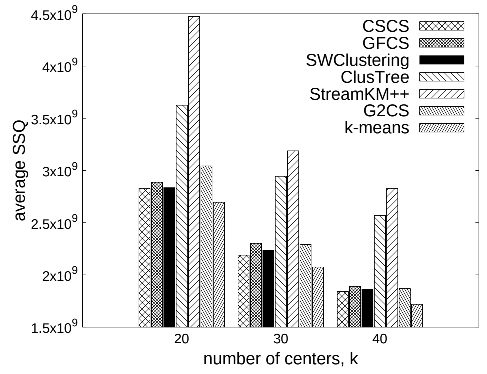
In terms of processing time, our algorithm shows better scalability than the others. In Figure 5.2, we measure the average processing time to process



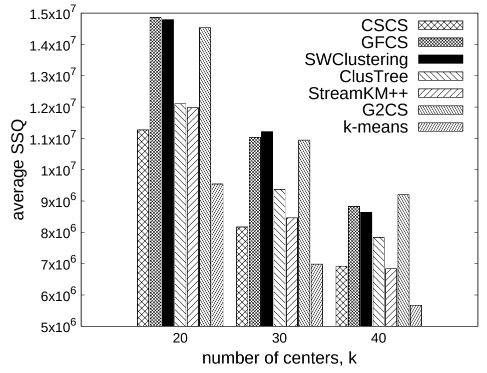
(a) kddcup99



(b) census1990



(c) covtype

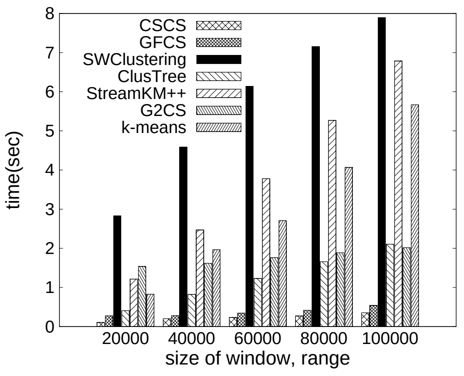


(d) tower

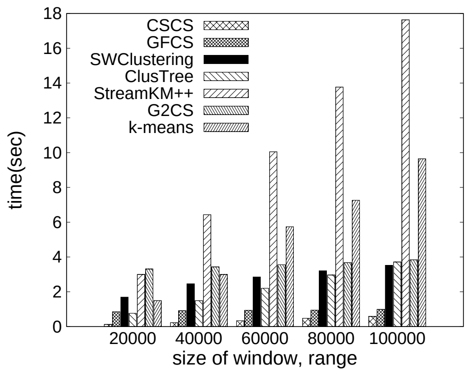
Figure 5.1: Clustering quality comparison with real-world datasets

tuples in the window of a given `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 for testing larger dimensions. As the size of the window increases, processing time of other algorithms also increases. However, increase in the processing time of CSCS is much less than others.

Figure 5.3 shows processing times at each timestamp when window slides. We set $k = 30$, `RANGE` = 100,000, `SLIDE` = 10,000, and the number of tuples = 200,000. Our algorithms, GFCS and CSCS are stable and fastest. *ClusTree* shows the second best performance, and *SWClustering* shows lowest perfor-



(a) kddcup99



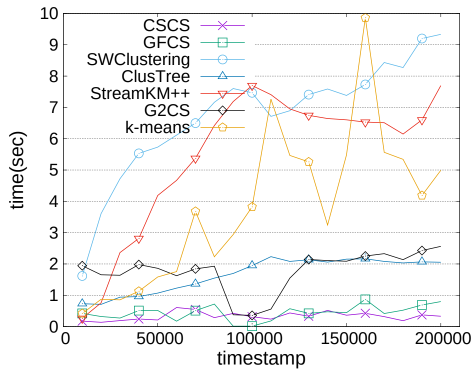
(b) census1990

Figure 5.2: Processing time of a window with specific size

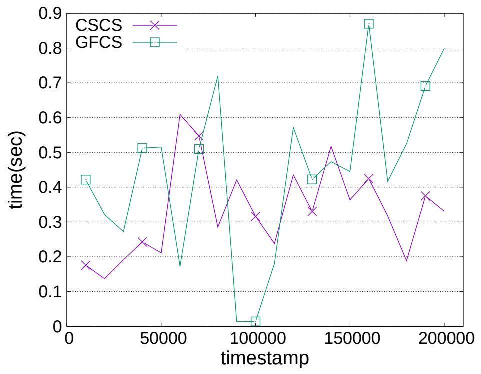
mance. Because k-means is a randomized algorithm, the processing time is fluctuating with data distribution. However, k-means shows the proper performance. This means that cost for constructing additional synopses of streaming algorithms is quite high. Figures 5.3b and 5.3d shows results of GFCS and CSCS only. Processing time varies depending on the characteristics of the dataset. For dense dataset census1990, CSCS is consistently fast.

To investigate impact of coreset size and the number of hash functions, we conducted experiments on the synthetic datasets. Figures 5.4 and 5.5 show clustering quality and execution time as coreset size is changed. We set `RANGE` = 100,000 and `SLIDE` = 10,000, and measured the performance for 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 original tuples. Average SSQ is converted to logarithmic scale to present multiple results together in one figure, and results of kddcup99 is divided by 100 for the same reason.

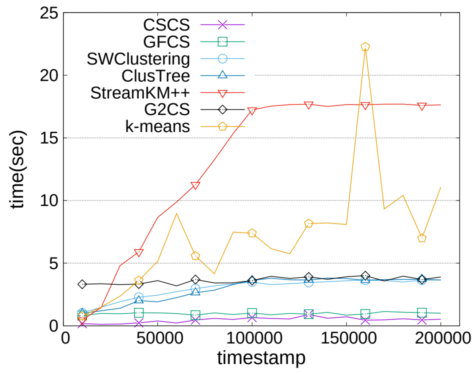
In terms of clustering quality, CSCS shows good performance with the coreset of 1/10 size, and the performance of the coreset of 1/100 size was not significantly lowered. Results of synthetic datasets also show similar tendencies. Reason for good performance of mid-size coreset is due to effect of removing



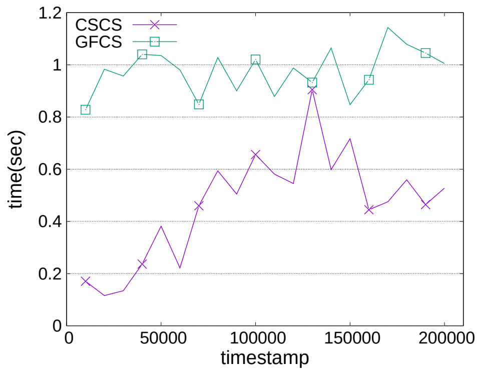
(a) kddcup99



(b) kddcup99 (CSCS & GFCS)



(c) census1990



(d) census1990 (CSCS & GFCS)

Figure 5.3: Processing time at each timestamp

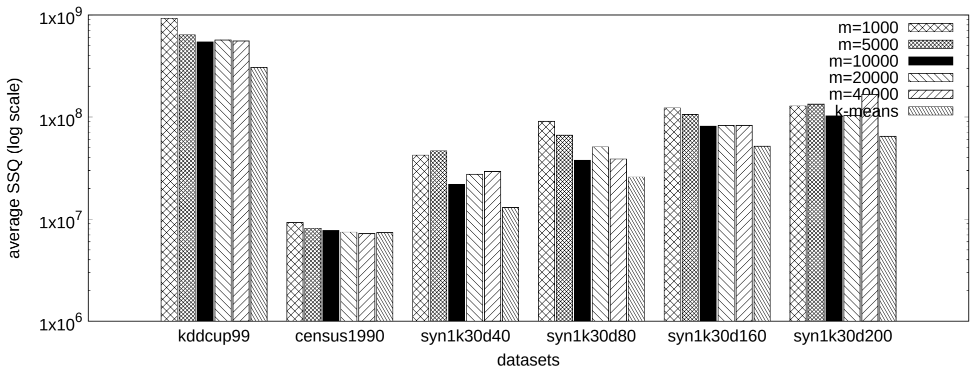


Figure 5.4: Clustering quality by coreset size

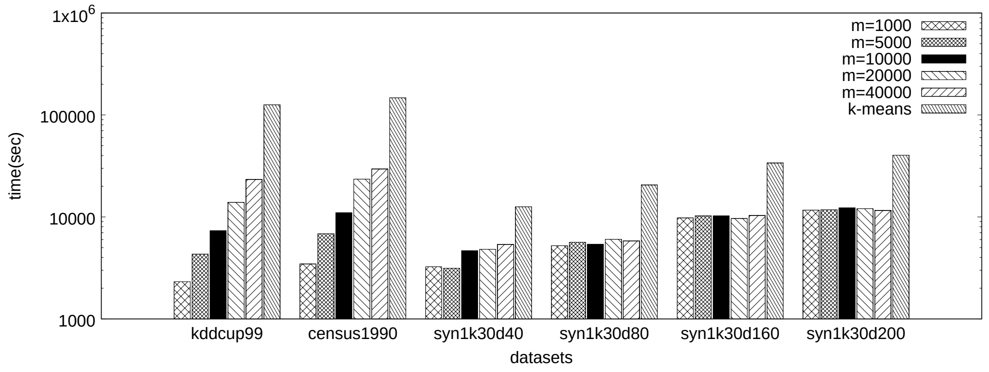


Figure 5.5: Processing time by coreset size

outliers. In large-size coreset, the number of noise GFs increases because the reduction occurs less frequently.

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

We changed the number of hash functions from 5 to 25, and measured quality and time of the algorithm. Results are shown in Figures 5.6 and 5.7. For real-world datasets, the clustering quality is improved as the number of hash functions increases, but it is less effective at more than 15. For synthetic datasets, the quality is best at 10 hash functions. It also shows that very small number of hash functions is not suitable for high-dimensional data. Processing time increases consistently as the number of hash functions increases due to calculation of hash keys and distances.

In Figure 5.8, we redraw the previous results by the dimensions. According to the results, factors which affect the processing time are dimension of the data and the number of hash functions. On the other hand, we find that increase in the size of coreset does not affect processing time. This is because the time complexity of the construction and the reduction of coreset is linear to the input

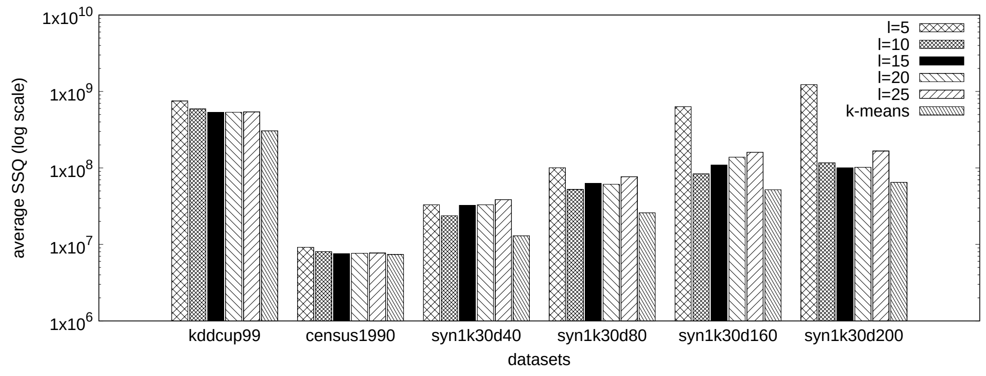


Figure 5.6: Clustering quality by the number of hash functions

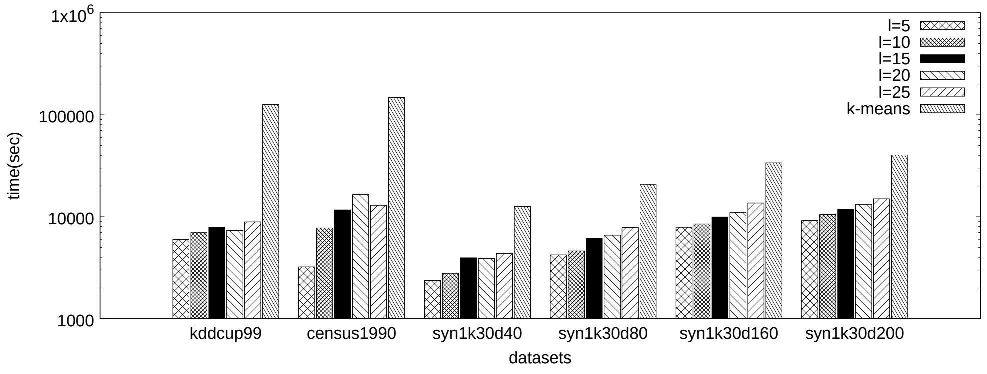
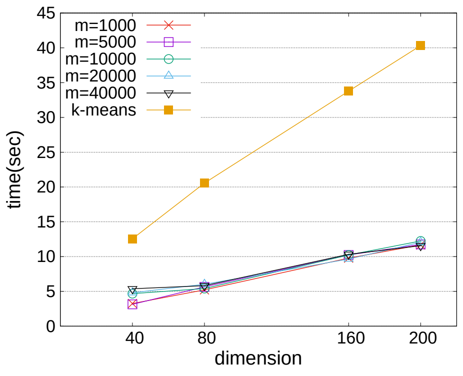
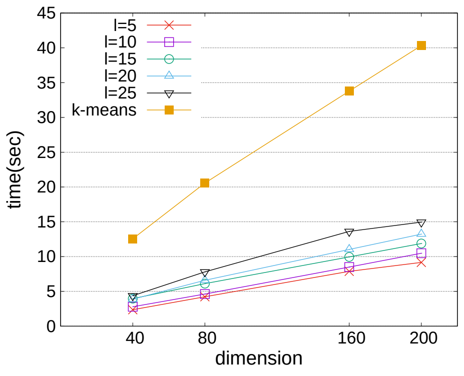


Figure 5.7: Processing time by the number of hash functions



(a) Coreset size



(b) The number of hash functions

Figure 5.8: Processing time by dimension

size.

Based on the experimental results, we conclude that our algorithm presents good performance in terms of quality and processing time. Especially, our algorithm is very effective for high-dimensional data. We provides raw data of the results in the appendix.

Chapter 6

Application: Documents Clustering

News documents are one of the most actively shared information through social media. News documents reflects issues and trends in real time, and are analyzed in many studies and applications. Since the news has the characteristic of timeliness, various analysis tasks should be updated frequently. However, a news document is composed of several sentences, and the length of the document is long, thus it is impossible to analyze by reading them individually. Therefore, in news analysis, clustering is used to group documents of similar theme or specific categories to extract representative document by reducing the number of documents. In this chapter, we will consider how to apply clustering with sliding windows to document clustering.

6.1 Vector Representation of Documents

For clustering documents, the document is expressed as a vector in vector space model. Bag-of-words model is a converting method which is straightforward and commonly used in text mining applications such as information retrieval. Bag-of-words model expresses a document as a vector whose components are words that the document contains. Suppose that a set of documents has n distinct words, $w_1, ..., w_n$. A document is represented by n-dimensional vector. For example, there are documents doc_1 = *"time flies like an arrow"* and document doc_2 = *"fruit flies like a banana"*. All distinct words are time, flies, like, an, arrow, fruit, a, banana, and they are mapped sequentially to each component of the vector. Document doc_1 is represented by $(1, 1, 1, 1, 1, 0, 0, 0)$, and document doc_2 is represented by $(0, 1, 1, 0, 0, 1, 1, 1)$. Sequence of words is ignored. Components of the vector are weighted according to applications. Word existence, tf(term frequency), and tf-idf(inverse document frequency) are used in weight scheme. Because the number of words in a document is much smaller than the number of words in a set of documents, the vector of the document is a sparse vector.

Bag-of-words model is simple and effective, but may not be appropriate for some applications. In particular, there is a problem that it is difficult to grasp latent semantics of a document which are not represented directly. To address this problem, probabilistic topic model such as Latent Dirichlet allocation (LDA) is developed. Latent Dirichlet Allocation (LDA) [14] is a Bayesian probabilistic model of documents, and models a document as a mixture over topics. In LDA, a document is represented by k-dimensional vector whose components are probabilities for each topic of the document.

Recently, neural network-based word embedding is widely used to represent text as vectors. Word embedding is a language model where words or sentences are mapped to vectors of real numbers. It is also referred to as distributed representation or vector representation. We briefly review word embedding first, and explain the method to convert document to vector in next section.

6.1.1 Distributed Representation of Words

Word2vec [50] is the most common technique currently in use for representing vector of a word. In the paper, Continuous Bag-of-Words Model(CBOW) and

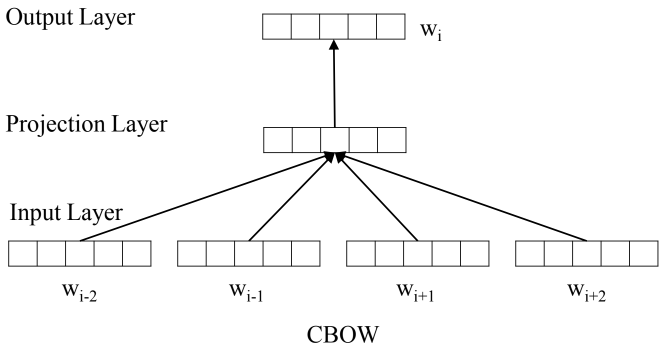


Figure 6.1: Architecture of CBOW model

Continuous Skip-gram Model(skip-gram) are proposed for learning distributed representations of words from corpus.

Continuous Bag-of-Words Model is a neural network architecture for predicting a word given its preceding and following words. Architecture is probabilistic feedforward neural network, and consists of input, projection, and output layers. Given a sequence of words, $W = [w_{i-2}, w_{i-1},\ w_i, w_{i+1}, w_{i+2}]$, the input is each vector of $[w_{i-2}, w_{i-1},\ w_{i+1}, w_{i+2}]$, and the output is vector representation of w_i. Vectors are randomly initialized before the training process. In training, the model read a certain number of words from corpus, and updates the output vector by aggregating input vectors. The number of words depends on the the setting of the architecture, and determined by the user.

Specifically, objective of CBOW is to maximize the conditional probability [45],

$$\frac{1}{M}\sum_{i=1}^{M} \log p(w_i|w_{i-k}, ..., w_{i+k}).$$

Softmax function is used for the classifier,

$$p(w_i|w_{i-k}, ..., w_{i+k}) = \frac{e^{y_{w_i}}}{\sum_j e^{y_j}}$$

where y_i is the probability of w_i in the output layer. y is computed by

$$y = b + Uh(w_{i-k}, ..., w_{i+k}; W)$$

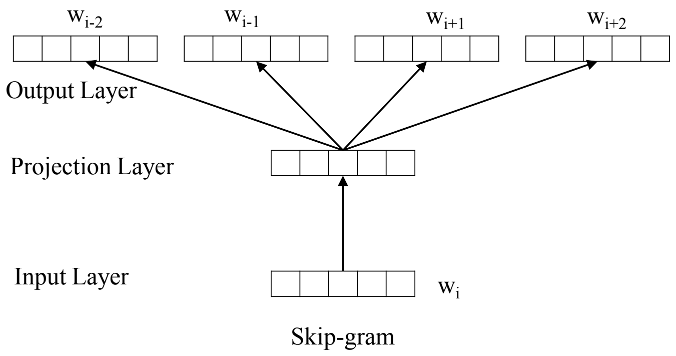


Figure 6.2: Architecture of skip-gram model

where U, b are parameters of softmax function, and h is aggregation of W such as concatenation or average of word vectors.

Continuous Skip-gram Model is similar to CBOW, but input and output layers in the architecture are reversed. Objective of the skip-gram is to predict the preceding and following words given the word. The objective function of the skip-gram is

$$\frac{1}{M}\sum_{i=1}^{M}\sum_{-k<j<k,j\neq 0}\log p(w_{i+k}|w_i).$$

Input is w_i, and output is $[w_{i-2}, w_{i-1}, w_{i+1}, w_{i+2}]$. However, because of computational complexity, the model randomly samples output words. The actual number of words of output layer is randomly chosen between 1 and maximum window size for each training sample. The maximum window size is also determined by the user. Word with the maximum distance is observed with a probability of *1/k*. In general, stochastic gradient descent and backpropagation are used for training.

6.1.2 Distributed Representation of Documents

Paragraph Vector [45] model is unsupervised model for distributed representations for a set of words such as sentence and document. The model is proposed

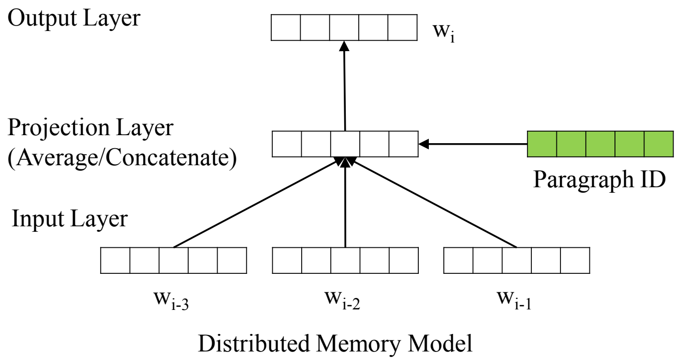


Figure 6.3: Distributed Memory Model of Paragraph Vector (PV-DM)

to convert variable length of texts to vector. The unit that a paragraph vector represents can be the paragraph itself, or it can be sentence, or a document. The Paragraph Vector model has some advantages over bag-of-words model. First, since word embedding is utilized, paragraph vector is generated considering semantic of a word. Second, paragraph vector which is smaller dimension than n-gram vector preserves word sequence information through concatenating word vectors.

Figure 6.3 shows architecture for Distributed Memory Model of Paragraph Vector (PV-DM). The architecture is originated from the CBOW in word2vec. Similar to the CBOW, objective of the model is to predict the following word of the concatenated vector in given context. Paragraph vectors are involved to predict next word given sampled words from paragraph. Each of paragraphs is represented by a unique vector. Input vector is created by averaging or concatenating the paragraph vector and word vectors, and is used for training. Learning processing and equation for Paragraph Vector model are the same as CBOW, but the only difference is that paragraph vector is added to the input vector. The model is trained through the stochastic gradient descent and backpropagation.

Distributed Bag of Words Model of Paragraph Vector (PV-DBOW) which is similar to Skip-gram model in word2vec is also proposed. PV-DBOW uses paragraph vector only as input vector, and predicts the word vectors which are randomly sampled from paragraph. As a result, paragraph vectors are generated by merging two vectors which are obtained by each model.

FastText is a software library for efficient learning of word representations and sentence classification which is developed by Facebook. The library adopts the subword based word representation model [15]. Subword model for morphological word representations is based on the skip-gram model, where each word is divided into morphemes. In the model, a set of character n-grams of a word is used to associate with context vectors instead of a word vector. Because of space and time, the algorithm keeps n-grams with a length between 3 and 6. N-grams also includes special characters which indicate the beginning and end of the word. Subword model contains more vocabulary due to n-grams, which enables to learn reliable representation for rare words.

We used news documents in Korean for experiment. News documents are converted into distributed representations through Paragraph Vector model and subword model.

6.2 Extension to Other Clustering Algorithms

We have developed data stream clustering algorithm focusing on k-means problem in previous chapters. However, k-means based data stream clustering involves several problems. First, user needs to specify a fixed number of clusters before clustering, and it is difficult to change during clustering process. In data streams, the number of clusters is often changed as data is inserted and deleted. The proper number of clusters whenever the clustering is performed is often not known in advanced. Second, it is difficult to generate clusters for arbitrary shapes because k-means clustering is designed to work well for sphere shaped clusters. In particular, data distribution and shape in high-dimensional data streams changed frequently, and could not be specified. Even in low-dimensional data such as geographic information, regions with similar geographic features could be formed an arbitrary shape. Third, k-means clustering is extremely sensitive to outliers. Data streams often include some random noise due to various reasons such as sensor failure and network trouble. These outliers brings unexpected effects to final clusters. Specifically, clustering results have many false negatives.

In order to overcome the problems of the k-means clustering, various clustering algorithms have been developed. We described related studies in previous chapter. To address the problems in data streams with sliding windows, we apply other clustering algorithms to coreset which is constructed by a method in CSCS. We utilize density-based clustering and affinity clustering which create the arbitrary number of clusters. In next section, we briefly explain each original algorithm, and propose a method to utilize the algorithms.

6.2.1 Density-based Clustering

DBSCAN(Density-Based Spatial Clustering of Applications with Noise) [26] is a density-based clustering algorithm for finding arbitrary shaped clusters with considering noise. DBSCAN requires two parameters, *eps*(ε) and *MinPts*. *eps* defines neighborhood between two data points given distance measure. *MinPts* defines the number of minimal points that forms a dense region which is not outlier within *eps* radius. Data points in a dense region are assigned to same cluster.

DBSCAN is defined with several concepts. ε-neighborhood is defined as data points whose distances are within ε. A core object is a data point whose the number of neighborhood within ε is more than MinPts. A data point p is directly density-reachable from core object q if p is ε-neighborhood of q. A data point p is density-reachable from the data point q if there is a chain of data points if there is a path $p_1, ..., p_n$, $p_1 = p$ and $p_n = q$ where each p_{i+1} is directly density-reachable from p_i given eps and MinPts. A data point p is density-connected to a data point q if there is an data point r such that both p and q are density-reachable from r given eps and MinPts. Based on the definitions, a cluster is a set of density-connected data points. Outliers are all data points which are not density-reachable from any other point.

The algorithm reads a starting data point p in dataset, and finds all neighbors of data point p within ε distance. If the number of neighbors is greater than MinPts, a p is classified as a core object. An empty cluster is created, and data point p and its neighbors are assigned to a cluster. Then, the algorithm expands the cluster with finding other core objects iteratively. The algorithm is terminated when all data points are visited.

We modified DBSCAN for data streams with sliding windows. Algorithm 9 is pseudo code for density-based clustering with coreset. Our algorithm generates fixed-size coreset based on LSH before clustering, then DensityClustering is called. In the algorithm, we modify the process of finding neighbors. Unlike DBSCAN, we define neighbors that GFs are within ε weighted distance or are overlapped. Because GF is a pseudo point which have radius, we also define ε-neighborhood between GF_1 and GF_2 with respect to ε distance if $dist(c_1, c_2) < r_1 + r_2 + \varepsilon$, where c_1 is center of GF_1, c_2 is center of GF_2, r_1 is radius of GF_1, and r_2 is radius of GF_2. A core object is determined by comparing the sum of number of tuples in GF with MinPts.

6.2.2 Message Passing-based Clustering

Affinity Propagation(AP) is message passing-based clustering algorithm. AP finds exemplars which are identified by passing messages on bipartite graph. Each data point is assigned to its nearest exemplar. Affinity is consists of two measures, responsibility and availability, and the algorithm find exemplars by

Algorithm 9 DensityClustering

```
    Input: Coreset $P$, eps $\varepsilon$, and MinPts $m$
    Output: Clusters $C$
 1: function DensityClustering($P$, $\varepsilon$, $m$)
 2:     for each $p \in P$ do
 3:         if $p$ is visited then
 4:             continue
 5:         else
 6:             mark $p$ as visited
 7:             $NB \leftarrow GF$s within $\varepsilon$ distance from $p \cup$ overlapped $GF$s of $p$
 8:             if $|NB| < m$ then
 9:                 mark $p$ as noise
10:             else
11:                 create new cluster $c$
12:                 ExpandCluster($p$, $NB$, $c$, $\varepsilon$, $m$)
13:             end if
14:         end if
15:     end for
16:     return $C$
17: end function
```

updating these values on bipartite graph [31].

The algorithm needs similarity matrix s of data points. For Euclidean distance, negative squared distance is used as similarity measure.

$$s(i, j) = -\|x_i - x_j\|^2$$

Values in diagonal of similarity matrix are preferences which adjust the number of exemplars, and is set to the same value for all data points. AP generates small number of exemplars with low value, and many exemplars with high value.

Algorithm 10 ExpandCluster

```
1: function ExpandCluster($p$, $NB$, $c$, $\varepsilon$, $m$)
2:     $c \leftarrow c \cup \{p\}$
3:     for each $q \in NB$ do
4:         if $q$ is not visited then
5:             mark $q$ as visited
6:             $NB2 \leftarrow GF$s within $\varepsilon$ distance from $q$ $\cup$ overlapped $GF$s of $q$
7:             if $|NB2| \geq m$ then
8:                 $NB \leftarrow NB \cup NB2$
9:             end if
10:            if $q$ are not assigned to existing clusters then
11:                $c \leftarrow c \cup \{q\}$
12:            end if
13:        end if
14:    end for
15: end function
```

Responsibility quantifies how much x_j is suited as a exemplar for x_i than other candidate exemplars for x_i. Responsibility matrix is defined as follow:

$$r(i,j) = s(i,j) - \max_{k,k \neq j} \{a(i,k) + s(i,k)\}$$

Availability represents measure how appropriate it would be for x_i to select x_j as its exemplar. Availability matrix is computed as follow:

$$a(i,j) = \min \left\{ 0, r(j,j) + \sum_{k,k \notin \{i,j\}} max\{0, r(k,j)\} \right\}$$

Responsibility matrix and availability matrix update till convergence. Clustering results are generated by

$$C = \arg\max_j \{a(i,j) + r(i,j)\}$$

For data streams, we reduce original tuples into a coreset, then apply AP clustering on GF in the coreset. We simply modify similarity measure as follow:

$$s(i,j) = -w(GF_i)w(GF_j)\|c_i - c_j\|^2$$

, where c_i are centers of GF_i, and $w(GF_i)$ is weight of GF_i. The number of data points in the GF is used as weight. Computation of responsibility and availability is the same as original AP.

6.3 Evaluation

6.3.1 Experimental Setup

We evaluated accuracy and scalability of our clustering algorithms, Den-CS and AP-CS, on real-word datasets. Den-CS is density-based clustering with coreset, and AP-CS is affinity propagation clsutering with coreset. We compared our algorithms with original clustering algorithms, DBSCAN and AP, on raw tuples without coreset construction. In addition, we compare density-based data stream clustering algorithms which are DenStream [18], D-Stream [60], and DBSTREAM [37]. We implemented Den-CS, AP-CS, DBSCAN, and AP from scratch in Java. DenStream was implemented based on MOA framework [13] in Java. We executed experiments written in Java 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. Maximum Java heap size (-Xmx option) is set to 8GB. We use R package `stream` [36] for D-Stream and DBSTREAM. Experiments for D-Stream and DBSTREAM were executed with R 3.4.0 on the same machine.

We used news documents in Korean for a dataset. The dataset contains 731,846 documents, and documents are classified into 17 categories. Each news document is converted into distributed representations using Paragraph Vector model and Subword model. **knews-pv** is dataset converted by Paragraph

Vector model with 400 dimension, and **knews-ft** is dataset converted by Subword model using **FastText** library with 300 dimension. We also used covtype dataset, which has 7 categories with 54 dimension.

We evaluated purity using category information for measuring clustering quality. Purity is defined as

$$purity(C, G) = \frac{1}{N} \sum_{k} \max_{j} |c_k, g_j| \tag{6.1}$$

where C is a set of clusters, and G is a set of categories. Purity computed as the average of the number of majority classes in each cluster.

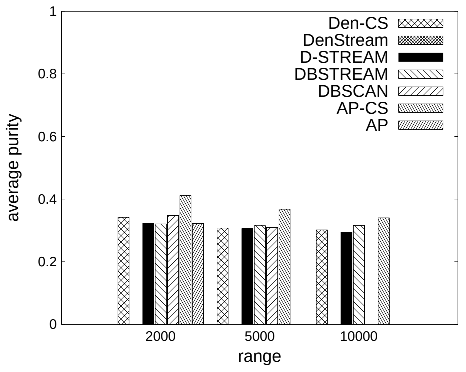
To evaluate scalability, we measure total processing time and the processing time of each sliding. In sliding windows, the algorithm produces multiple results as the window moves. Therefore, the quality is evaluated by averaging purity of the results.

6.3.2 Experimental Results

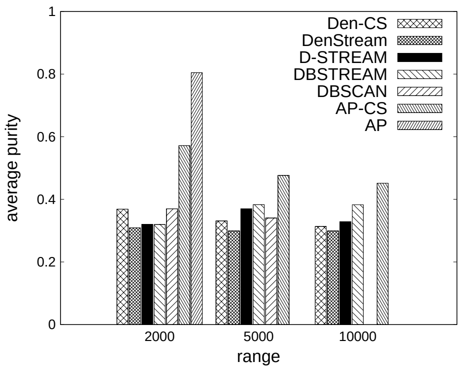
For AP-CS and Den-CS, we set number of hash functions $l = 15$, and coreset size $m = 500$, if we do not mention explicitly.

Figure 6.4 shows the purity of the algorithms for different window size with real-world datasets. In terms of purity, original AP clustering with subword model shows the best performance, but AP could not produce the results because it is very slow when `RANGE` > 2000. AP clustering is not suitable for real-time processing. We observe that our algorithms Den-CS and AP-CS drop the purity slightly. In particular, our density-based clustering outperforms original algorithm in covtype dataset.

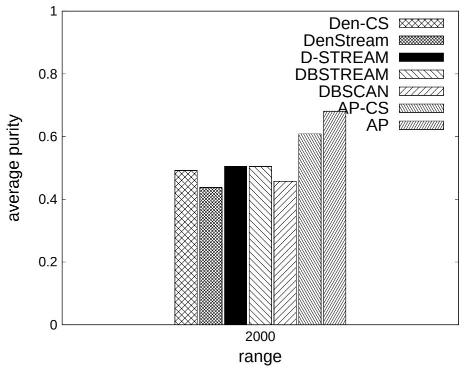
In terms of processing time, our algorithm shows better scalability than the others. In Figure 6.5, we measure the average processing time to process tuples in the window of a given `RANGE`. We fixed coreset size $= 500$, and `SLIDE` $= 500$. As the size of the window increases, the processing time of other algorithms



(a) knews-pv



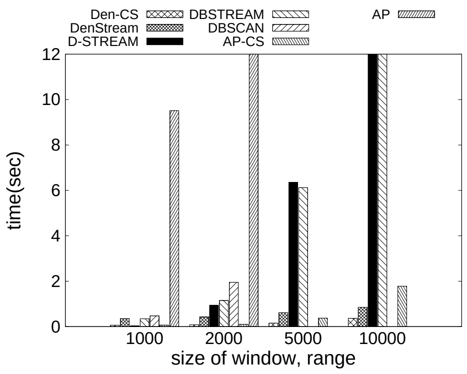
(b) knews-ft



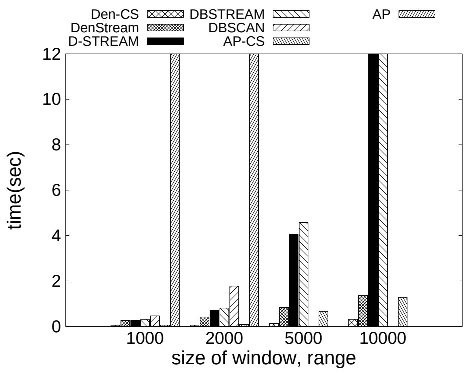
(c) covtype

Figure 6.4: Purity comparison with real-world datasets

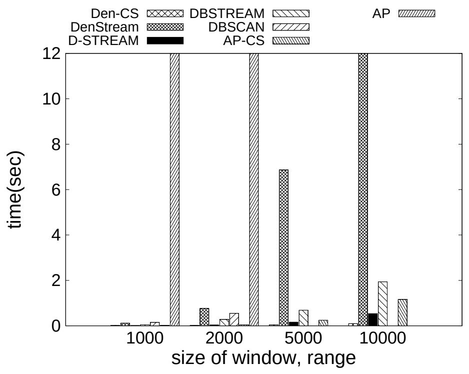
also increases exponentially. However, our algorithms using coreset increases linearly. We cut and plotted the results of AP clustering because of scale, and the raw data is presented in Appendix. Figure 6.6 shows results except for slow algorithms. As the size of window increases, processing time of our algorithms grows at a small rate than D-STREAM and DBSTREAM. Processing times of D-STREAM and DBSTREAM increase exponentially with respect to the size of window. Results of D-STREAM and DBSTREAM are considered to be slow because they are implemented in R. However, even if we do not take into account values of execution time, rate of increase in execution time of Den-CS and AP-CS is lower than D-STREAM and DBSTREAM. Therefore, Den-CS



(a) knews-pv



(b) knews-ft



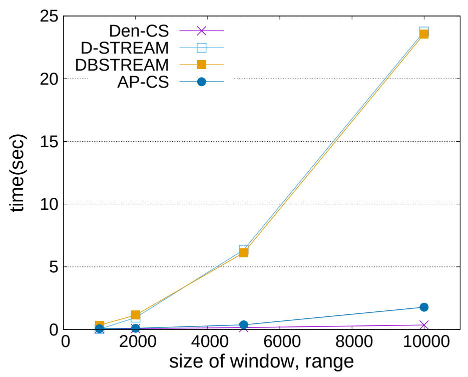
(c) covtype

Figure 6.5: Processing time of a window with specific size

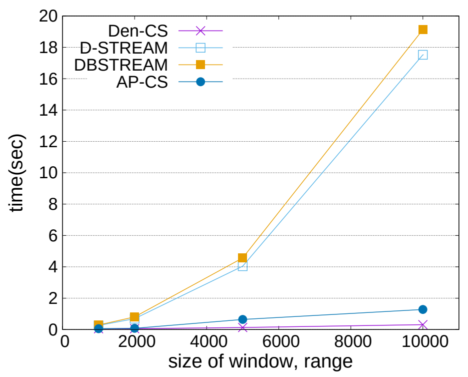
and AP-CS are more scalable than other algorithms.

Figure 6.7 shows the processing times at each timestamp when the window slides. We set `RANGE` = 10,000 and `SLIDE` = 500, and the number of tuples = 20,000. Den-CS is stable and fast, but results of AP-CS are fluctuated. This fluctuatation occurs because AP clustering repeats until the affinity converges. However, since the data with the 10000 tuples can be processed within 1-3 seconds, it is possible to handle data streams in real-time.

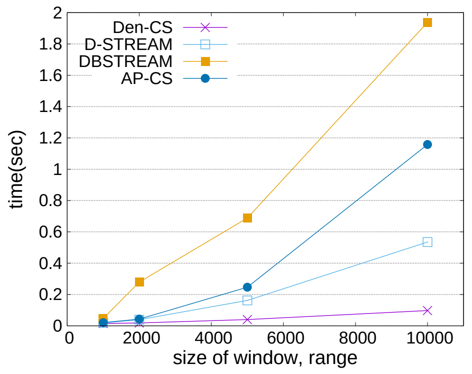
In conclusion, AP clustering is appropriate for document dataset, and methodology using coreset is a way to greatly increase the size of the data which can be processed.



(a) knews-pv



(b) knews-ft



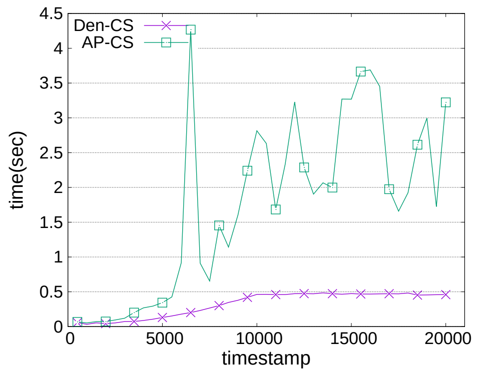
(c) covtype

Figure 6.6: Processing time by window size

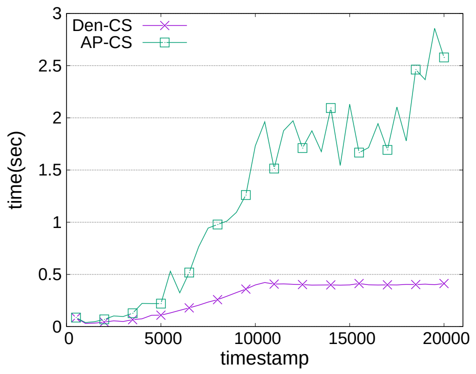
6.4 Discussion

In this chapter, we discussed clustering algorithms focused on document streams. Data stream clustering is utilized in various types of data streams in addition to document streams. For example, there are video, traffic, and sensors in types of data streams. We briefly review recent studies of applications using various types of data streams.

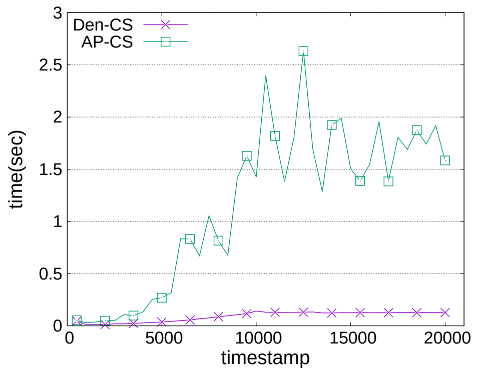
Video data is composed of several consecutive frames. A frame is regarded as a data point, and video is a data stream in which frames are continuously generated. Video of 30 frames per second is data stream in which 30 data



(a) knews-pv



(b) knews-ft



(c) covtype

Figure 6.7: Processing time at each timestamp of Den-CS and AP-CS

points are input into system every second. Data stream clustering on video data is studied for scalability in retrieval problem [19]. Although video data is continuous and has many redundant frames, new objects may appear in a frame at any moment. When clustering is used to reduce duplication or to find new objects, it is important to perform clustering continuously on video data.

Trajectory data is a data stream that tracks various moving objects and generates their path periodically and continuously. Clustering on these data is used to report behavior of moving objects over a period of time. Through clustering for each time window, it is possible to find similar moving objects or detect similar path patterns in real time. However, since the number of moving objects is too large, it is difficult to perform clustering continuously. Silva et al. [21,22] propose an incremental density-based clustering algorithm to maintains sub-trajectory clusters of trajectory data streams. The algorithm tracks moving objects with sensing appearance of new objects and disappearance of existing objects. The authors define macro-groups which are representative groups of moving objects in each time window to discover patterns efficiently.

Data of Internet of Things (IoT) are produced from multiple sources such as sensor networks, smart devices, and home appliances. Because IoT devices are highly heterogeneous, data are also represented in a variety of formats from numeric to text. Single data stream usually follow a specific data distribution. However, in IoT applications, it is necessary to handle data streams of various formats and varying data distributions. Puschmann et al. [54] propose an adaptable clustering algorithm for dynamic IoT data streams. The algorithm tracks evolving clusters based on data distribution and clustering quality measurement. To deal with unknown data, the algorithm finds the right number of clusters based on symbolic aggregate approximation (SAX) algorithm, and detects concept drifts of clusters based on properties of stochastic convergence. If concept drift is detected, the algorithm performs clustering with different setting.

Social media is one of the largest data streams, and have a variety of data formats which are text, tags, metadata, video, and images. There are many studies for clustering of social media, such as emergency management [53], community analysis [5,29,51] and topic detection [28,48,62]. For example, Pohl et al. [53] propose a framework to identify sub-events and produce situational reports based on online indexing and online clustering. The algorithm maintains term frequency-inverse document frequency(tf-idf) and skewness of data using incremental model. Growing Gaussian Mixture Models (2G2M) algorithm is utilized in clustering.

Chapter 7

Conclusion

An efficient algorithm for data stream clustering over sliding windows is proposed in this thesis. In the grouping step, we presented the aggregation technique for the sliding window model, which divides the window into disjoint chunks, and generates overall coreset by merging partial coresets. Locality-Sensitive Hashing is utilized for efficient coreset construction. In clustering step, the algorithm performs clustering on the group features in the coreset. Re-clustering policy was proposed to avoid unnecessary clustering. Our approach has an advantage over recent algorithms that perform clustering on entire data streams as it provides the functionality of tracking the changes in the data streams by generating a snapshot of every cluster using less computational power. We also present theoretical analysis of coreset construction and sliding window model. In addition, we extend our algorithm to density-based clustering and message passing-based clustering, which are DBSCAN and Affinity Propagation respectively. Extensive experiments demonstrate the effectiveness and efficiency of the proposed algorithms.

Discussion. CSCS is an improved algorithm based on GFCS that allows efficient operation with limited memory and high-dimensional data streams. In GFCS, we introduced data stream clustering algorithm based on sliding window aggregation technique. Main contribution of GFCS is a lean data structure of 2-level coreset to support deletion operations of sliding windows. GFCS adopts a straightforward but naive grouping method using distance threshold for clustering. Based on this method, CSCS improves clustering more efficiently. CSCS determines distance threshold adaptively, and maintains a fixed-size coreset through reduction process.

Both GFCS and CSCS make approximate clusters, but qualities of clusters is comparable to results of original k-means algorithm. Our algorithms are faster than conventional data stream clustering algorithms, and require less memory. Experimental results of conventional algorithms should be analyzed carefully because they are slower than k-means clustering algorithm. Although the conventional algorithms use less memory, cost of constructing a data structure for summarization in sliding windows seems to be a load.

CSCS combines advantages of sliding window aggregation with LSH approach, and computes solution within short running time by using hash-based compact data structure. Main tunable parameters of CSCS are the number of hash functions and limit of coreset size. Our algorithm makes process of determining appropriate parameters for new dataset easier because it takes less time to perform a single clustering for testing.

Required parameters for GFCS and CSCS are shown in Table 2.5. Parameters that are common to both algorithms are the number of clusters(k), the length of a window(R), and the movement intervals of a window(L). Appropriate k value is determined by dataset. Density-based clustering is one of the ways to solve problem of determining k. Various methods have been proposed, but a method that a user determines an arbitrary k according to purpose of analysis is still widely used. R value specifies a period of data streams included in clustering results. For example, it is set to 1 week or 24 hours. To process data streams of long period, general clustering algorithm requires a large memory, and takes a long execution time. CSCS can handle large data streams by maintaining a certain size coreset. L value is a size that a sliding window is updated, and is designed to process data streams in batch. Results are updated more frequently at small value, but the amount of memory required also decreases.

GFCS needs to specify distance threshold θ. θ is a trade-off parameter of execution time and accuracy. The value is determined so that the number of generated GFs is a manageable size. Specifically, if θ is too small, the size of summary is not much smaller than the size of raw data. In this case, execution time of clustering may be slower than original k-means algorithm. In contrast, specifying a large value of θ is more likely to cause errors because it group too many data points. In CSCS, both m and l are trade-off parameters of execution time and accuracy. In order to determine m, various values are experimentally tested. Generally the value m is determined by extending from k to multiples thereof. Although StreamKM++ uses about $200k$, our experiment shows a good solution even with m value less than $200k$. Likewise, l value is experimentally determined by increasing gradually from a small value based on execution time. In our experiments, when dataset of 200 dimensions were extremely reduced, performance and execution time did not deteriorate significantly at a small value between 10 and 20. It can be seen that durable performance is guaranteed for small values. However, determining the suitable parameters for a dataset is a topic of further research.

Our methods can be applied to various applications that use data stream clustering. We have used news clustering as an example of motivation in the introduction. Many studies have applied data stream clustering to various applications such as video [19], spam filtering [33], trajectory [21, 22], emergency management [53], Internet of Things(IoT) [54], community analysis in social media [5, 29, 51], topic detection in social media [28, 48, 62]. Our algorithm can be easily applied to these applications. For example, our dimension reduction technique in CSCS will be effective for large-scale high-dimensional data streams from various sensors in IoT. Applications that uses text data, such as spam filtering and topic detection, uses the same method used in document clustering in Chapter 6. Text data can be converted to distributed representation, and our algorithm can be used directly. In particular, it is a very important issue to process a large and fast data stream in the case of an application that extracts trends of search keywords, popular hashtags for short texts such as search queries and tweets. We consider that our approach is a good solution for developing data stream applications.

Bibliography

[1] Ackerman, M., and Dasgupta, S. Incremental clustering: The case for extra clusters. In *Advances in Neural Information Processing Systems 27* (2014), Curran Associates, Inc., pp. 307–315.

[2] Ackermann, M. R., Märtens, M., Raupach, C., Swierkot, K., Lammersen, C., and Sohler, C. Streamkm++: A clustering algorithm for data streams. *Journal of Experimental Algorithmics 17* (2012), 2.4:2.1–2.4:2.30.

[3] Aggarwal, C. C., Han, J., Wang, J., and Yu, P. S. A framework for clustering evolving data streams. In *Proceedings of the 29th International Conference on Very Large Data Bases* (2003), vol. 29, VLDB Endowment, pp. 81–92.

[4] Aloise, D., Deshpande, A., Hansen, P., and Popat, P. Np-hardness of euclidean sum-of-squares clustering. *Machine Learning 75*, 2 (2009), 245–248.

[5] Anagnostopoulos, A., Ł ącki, J., Lattanzi, S., Leonardi, S., and Mahdian, M. Community detection on evolving graphs. In *Advances in Neural Information Processing Systems 29*. Curran Associates, Inc., 2016, pp. 3522–3530.

[6] Arasu, A., Babu, S., and Widom, J. The cql continuous query language: Semantic foundations and query execution. *The VLDB Journal 15*, 2 (2006), 121–142.

[7] Arthur, D., and Vassilvitskii, S. K-means++: The advantages of careful seeding. In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms* (2007), SODA '07, Society for Industrial and Applied Mathematics, pp. 1027–1035.

[8] Awasthi, P., Charikar, M., Krishnaswamy, R., and Sinop, A. K. The hardness of approximation of euclidean k-means. In *31st International Symposium on Computational Geometry (SoCG 2015)* (2015), vol. 34 of *Leibniz International Proceedings in Informatics (LIPIcs)*, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, pp. 754–767.

[9] Babcock, B., Datar, M., Motwani, R., and O'Callaghan, L. Maintaining variance and k-medians over data stream windows. In *Proceedings of the Twenty-second ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems* (2003), ACM, pp. 234–243.

[10] Badiozamany, S., Orsborn, K., and Risch, T. Framework for real-time clustering over sliding windows. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management* (2016), SSDBM '16, ACM, pp. 19:1–19:13.

[11] Begum, N., Ulanova, L., Wang, J., and Keogh, E. Accelerating dynamic time warping clustering with a novel admissible pruning strategy. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2015), KDD '15, ACM, pp. 49–58.

[12] Bhatnagar, V., Kaur, S., Saxena, R., and Khanna, D. Dasc: data aware algorithm for scalable clustering. *Knowledge and Information Systems 50*, 3 (2017), 851–881.

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

[14] Blei, D. M., Ng, A. Y., and Jordan, M. I. Latent dirichlet allocation. *Journal of Machine Learning Research 3*, Jan (2003), 993–1022.

[15] Bojanowski, P., Grave, E., Joulin, A., and Mikolov, T. Enriching word vectors with subword information. *ArXiv e-prints* (2016).

[16] Bradley, P. S., Fayyad, U., and Reina, C. Scaling clustering algorithms to large databases. In *Proceedings of the Fourth International Conference on Knowledge Discovery and Data Mining* (1998), KDD'98, AAAI Press, pp. 9–15.

[17] Braverman, V., Lang, H., Levin, K., and Monemizadeh, M. Clustering problems on sliding windows. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms* (2016), Society for Industrial and Applied Mathematics, pp. 1374–1390.

[18] Cao, F., Ester, M., Qian, W., and Zhou, A. Density-based clustering over an evolving data stream with noise. In *2006 SIAM Conference on Data Mining* (2006), SIAM, pp. 328–339.

[19] Chandrasekhar, V., Tan, C., Min, W., Liyuan, L., Xiaoli, L., and Hwee, L. J. Incremental graph clustering for efficient retrieval from streaming egocentric video data. In *22nd International Conference on Pattern Recognition* (2014), pp. 2631–2636.

[20] Chen, Y., and Tu, L. Density-based clustering for real-time stream data. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (2007), ACM, pp. 133–142.

[21] d. Silva, T. L. C., Zeitouni, K., and d. Macêdo, J. A. F. Online clustering of trajectory data stream. In *17th IEEE International Conference on Mobile Data Management (MDM)* (2016), vol. 1, pp. 112–121.

[22] da Silva, T. L. C., Zeitouni, K., de Macêdo, J. A. F., and Casanova, M. A. Cutis: Optimized online clustering of trajectory data stream. In *Proceedings of the 20th International Database Engineering & Applications Symposium* (2016), IDEAS '16, pp. 296–301.

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

[24] Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. S. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the Twentieth Annual Symposium on Computational Geometry* (2004), ACM, pp. 253–262.

[25] Ding, S., Zhang, J., Jia, H., and Qian, J. An adaptive density data stream clustering algorithm. *Cognitive Computation 8*, 1 (2016), 30–38.

[26] Ester, M., Kriegel, H.-P., Sander, J., and Xu, X. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining* (1996), AAAI Press, pp. 226–231.

[27] Farnstrom, F., Lewis, J., and Elkan, C. Scalability for clustering algorithms revisited. *SIGKDD Explorations Newsletter 2*, 1 (2000), 51–57.

[28] Feng, W., Zhang, C., Zhang, W., Han, J., Wang, J., Aggarwal, C., and Huang, J. Streamcube: Hierarchical spatio-temporal hashtag clustering for event exploration over the twitter stream. In *IEEE 31st International Conference on Data Engineering* (2015), pp. 1561–1572.

[29] Folino, F., and Pizzuti, C. An evolutionary multiobjective approach for community discovery in dynamic networks. *IEEE Transactions on Knowledge and Data Engineering 26*, 8 (2014), 1838–1852.

[30] Frahling, G., and Sohler, C. Coresets in dynamic geometric data streams. In *Proceedings of the Thirty-seventh Annual ACM Symposium on Theory of Computing* (2005), STOC '05, ACM, pp. 209–217.

[31] Frey, B. J., and Dueck, D. Clustering by passing messages between data points. 972–976.

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

[33] Georgala, K., Kosmopoulos, A., and Paliouras, G. Spam filtering: An active learning approach using incremental clustering. In *Proceedings of the 4th International Conference on Web Intelligence, Mining and Semantics (WIMS14)* (2014), WIMS '14, pp. 23:1–23:12.

[34] Guha, S., Meyerson, A., Mishra, N., Motwani, R., and O'Callaghan, L. Clustering data streams: Theory and practice. *IEEE Transactions on Knowledge and Data Engineering 15*, 3 (2003), 515–528.

[35] Guha, S., Mishra, N., Motwani, R., and O'Callaghan, L. Clustering data streams. In *Proceedings of 41st Annual Symposium on Foundations of Computer Science* (2000), pp. 359–366.

[36] Hahsler, M., Bolaños, M., and Forrest, J. Introduction to stream: An extensible framework for data stream clustering research with R. *Journal of Statistical Software 76*, 14 (2017), 1–50.

[37] Hahsler, M., and Bolaños, M. Clustering data streams based on shared density between micro-clusters. *IEEE Transactions on Knowledge and Data Engineering 28*, 6 (2016), 1449–1461.

[38] Har-Peled, S., and Kushal, A. Smaller coresets for k-median and k-means clustering. In *Proceedings of the Twenty-first Annual Symposium on Computational Geometry* (2005), SCG '05, ACM, pp. 126–134.

[39] Har-Peled, S., and Mazumdar, S. On coresets for k-means and k-median clustering. In *Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing* (2004), STOC '04, ACM, pp. 291–300.

[40] Hyde, R., Angelov, P., and MacKenzie, A. Fully online clustering of evolving data streams into arbitrarily shaped clusters. *Information Sciences 382–383* (2017), 96–114.

[41] Isaksson, C., Dunham, M. H., and Hahsler, M. Sostream: Self organizing density-based clustering over data stream. 264–278.

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

[43] Kullback, S., and Leibler, R. A. On information and sufficiency. *The Annals of Mathematical Statistics 22*, 1 (1951), 79–86.

[44] Langone, R., Barel, M. V., and Suykens, J. Efficient evolutionary spectral clustering. *Pattern Recognition Letters 84* (2016), 78–84.

[45] Le, Q., and Mikolov, T. Distributed representations of sentences and documents. In *Proceedings of the 31st International Conference on Machine Learning* (2014), vol. 32 of *Proceedings of Machine Learning Research*, PMLR, pp. 1188–1196.

[46] Li, J., Maier, D., Tufte, K., Papadimos, V., and Tucker, P. A. No pane, no gain: Efficient evaluation of sliding-window aggregates over data streams. *SIGMOD Record 34*, 1 (2005), 39–44.

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

[48] Lu, Z., Lin, Y.-R., Huang, X., Xiong, N., and Fang, Z. Visual topic discovering, tracking and summarization from social media streams. *Multimedia Tools and Applications 76*, 8 (2017), 10855–10879.

[49] Mai, S. T., Assent, I., and Le, A. *Anytime OPTICS: An Efficient Approach for Hierarchical Density-Based Clustering.* 2016, pp. 164–179.

[50] Mikolov, T., Chen, K., Corrado, G., and Dean, J. Efficient estimation of word representations in vector space. *ArXiv e-prints* (2013).

[51] Nguyen, N. P., Dinh, T. N., Shen, Y., and Thai, M. T. Dynamic social community detection and its applications. *PLOS ONE 9*, 4 (2014), 1–18.

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

[53] Pohl, D., Bouchachia, A., and Hellwagner, H. Online indexing and clustering of social media data for emergency management. *Neurocomputing 172* (2016), 168–179.

[54] Puschmann, D., Barnaghi, P., and Tafazolli, R. Adaptive clustering for dynamic iot data streams. *IEEE Internet of Things Journal 4*, 1 (2017), 64–74.

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

[56] Rodrigues, P. P., Gama, J., and Pedroso, J. P. Odac: Hierarchical clustering of time series data streams. In *Proceedings of the 2006 SIAM International Conference on Data Mining* (2006), SIAM, pp. 499–503.

[57] Rosman, G., Volkov, M., Feldman, D., Fisher III, J. W., and Rus, D. Coresets for k-segmentation of streaming data. In *Advances in Neural Information Processing Systems 27*, Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, Eds. Curran Associates, Inc., 2014, pp. 559–567.

[58] Silva, J. A., Faria, E. R., Barros, R. C., Hruschka, E. R., Carvalho, A. C. P. L. F. d., and Gama, J. a. Data stream clustering: A survey. *ACM Computing Surveys 46*, 1 (2013), 13:1–13:31.

[59] Sun, L., and Guo, C. Incremental affinity propagation clustering based on message passing. *IEEE Transactions on Knowledge and Data Engineering 26*, 11 (2014), 2731–2744.

[60] Tu, L., and Chen, Y. Stream data clustering based on grid density and attraction. *ACM Transactions on Knowledge Discovery from Data 3*, 3 (2009), 12:1–12:27.

[61] Wan, L., Ng, W. K., Dang, X. H., Yu, P. S., and Zhang, K. Density-based clustering of data streams at multiple resolutions. *ACM Transactions on Knowledge Discovery from Data 3*, 3 (2009), 14:1–14:28.

[62] Xie, W., Zhu, F., Jiang, J., Lim, E. P., and Wang, K. Topicsketch: Real-time bursty topic detection from twitter. In *2013 IEEE 13th International Conference on Data Mining* (2013), pp. 837–846.

[63] Yang, C., Bruzzone, L., Guan, R., Lu, L., and Liang, Y. Incremental and decremental affinity propagation for semisupervised clustering in multispectral images. *IEEE Transactions on Geoscience and Remote Sensing 51*, 3 (2013), 1666–1679.

[64] Youn, J., Choi, J., Shim, J., and Lee, S.-g. Partition-based clustering with sliding windows for data streams. In *Proceedings of 22nd International Conference on Database Systems for Advanced Applications* (2017), Springer, pp. 289–303.

[65] Zhang, T., Ramakrishnan, R., and Livny, M. Birch: An efficient data clustering method for very large databases. *SIGMOD Record 25*, 2 (1996), 103–114.

[66] Zhang, X., Furtlehner, C., Germain-Renaud, C., and Sebag, M. Data stream clustering with affinity propagation. *IEEE Transactions on Knowledge and Data Engineering 26*, 7 (2014), 1644–1656.

[67] Zheng, L., Huo, H., Guo, Y., and Fang, T. Supervised adaptive incremental clustering for data stream of chunks. *Neurocomputing 219* (2017), 502–517.

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

[69] Zhu, Y., and Shasha, D. Statstream: Statistical monitoring of thousands of data streams in real time. In *Proceedings of the 28th International Conference on Very Large Data Bases* (2002), VLDB '02, VLDB Endowment, pp. 358–369.

Chapter A

Appendix

A.1 Experimental Results of GFCS and CSCS

We use shortened keywords for space reasons. SWC is SWClustering, and SKM is StreamKM++.

Table A.1: Average SSQ of census1990

k	CSCS	GFCS	SWC
20	8.8505E+06	9.3138E+06	8.9378E+06
30	7.6405E+06	7.8354E+06	7.9978E+06
40	6.7664E+06	7.6281E+06	7.4453E+06

k	ClusTree	SKM	G2CS	k-means
20	1.0148E+07	8.3228E+06	9.7137E+06	8.0083E+06
30	8.7936E+06	7.1130E+06	9.2349E+06	6.6271E+06
40	8.0679E+06	6.1326E+06	8.9809E+06	5.8743E+06

Table A.2: Average SSQ of covtype

k	CSCS	GFCS	SWC	ClusTree	SKM	G2CS	k-means
20	2.8286E+09	2.8878E+09	2.8358E+09	3.6256E+09	4.4738E+09	3.0436E+09	2.6960E+09
30	2.1889E+09	2.2976E+09	2.2366E+09	2.9445E+09	3.1881E+09	2.2910E+09	2.0747E+09
40	1.8408E+09	1.8900E+09	1.8597E+09	2.5698E+09	2.8284E+09	1.8705E+09	1.7186E+09

Table A.3: Average SSQ of kddcup99

k	CSCS	GFCS	SWC	ClusTree	SKM	G2CS	k-means
20	1.0430E+11	2.5194E+11	2.6776E+11	3.5853E+11	2.8699E+11	1.8070E+11	6.9180E+10
30	5.6704E+10	1.5937E+11	1.4428E+11	2.3303E+11	1.4971E+11	8.4830E+10	3.0549E+10
40	3.4376E+10	1.0375E+11	1.0257E+11	1.9127E+11	1.2833E+11	6.0752E+10	1.7818E+10

Table A.4: Average SSQ of tower

k	CSCS	GFCS	SWC
20	1.1271E+07	1.4861E+07	1.4795E+07
30	8.1690E+06	1.1033E+07	1.1218E+07
40	6.9199E+06	8.8254E+06	8.6413E+06

k	ClusTree	SKM	G2CS	k-means
20	1.2106E+07	1.1984E+07	1.4537E+07	9.5419E+06
30	9.3675E+06	8.4648E+06	1.0947E+07	6.9855E+06
40	7.8346E+06	6.8399E+06	9.2053E+06	5.6697E+06

Table A.5: Processing time(ms) by window size for census1990

RANGE	CSCS	GFCS	SWC
20000	122.08	847.54	1694.55
40000	226.32	922.09	2457.67
60000	335.00	936.89	2851.47
80000	478.71	947.04	3208.32
100000	581.78	992.54	3526.11

RANGE	ClusTree	SKM	G2CS	k-means
20000	768.18	2995.06	3308.96	1494.69
40000	1492.68	6431.91	3432.00	2991.52
60000	2205.52	10048.61	3556.27	5738.04
80000	2964.00	13766.73	3679.38	7262.44
100000	3709.95	17632.28	3838.79	9645.54

Table A.6: Processing time(ms) by window size for kddcup99

RANGE	CSCS	GFCS	SWC
20000	104.26	275.40	2830.96
40000	203.04	280.23	4588.76
60000	231.79	346.19	6139.57
80000	270.96	417.25	7156.14
100000	352.14	539.43	7895.67

RANGE	ClusTree	SKM	G2CS	k-means
20000	407.51	1214.25	1535.96	828.52
40000	821.89	2468.37	1617.33	1965.90
60000	1228.34	3780.82	1763.08	2709.21
80000	1657.62	5268.05	1886.74	4066.53
100000	2106.44	6783.82	2015.95	5666.28

Table A.7: Processing time(ms) at each timestamp of census1990

TS	CSCS	GFCS	SWC	ClusTree	SKM	G2CS	k-means
10000	170.56	827.90	1030.40	983.51	457.21	3317.06	774.65
20000	115.84	983.14	1479.27	1196.41	1451.08	3356.67	1486.74
30000	134.37	957.11	1919.91	1399.46	4795.37	3290.41	2362.01
40000	236.67	1040.17	2286.91	2006.97	5905.38	3329.06	3617.45
50000	381.76	1035.62	2443.99	1915.31	8662.06	3602.80	5138.80
60000	221.89	980.49	2725.91	2245.35	9873.57	3183.88	8973.48
70000	459.88	848.51	2962.10	2635.99	11271.47	3694.99	5575.80
80000	593.95	1028.05	3187.00	2844.07	13249.01	3417.61	4149.74
90000	504.81	900.39	3339.04	3294.06	15351.53	3430.01	7477.47
100000	656.24	1020.62	3494.62	3627.17	17231.54	3609.80	7389.64
110000	581.23	878.59	3278.54	3794.84	17535.96	3951.13	6161.46
120000	545.09	987.88	3361.54	3672.15	17659.49	3781.13	5745.93
130000	904.65	932.23	3454.38	3656.20	17690.32	3904.35	8161.40
140000	598.64	1064.54	3518.96	3808.69	17510.49	3707.01	8208.25
150000	716.82	847.39	3593.87	3770.89	17659.70	3915.25	8090.25
160000	444.69	942.83	3642.27	3632.66	17645.99	4001.46	22279.48
170000	475.02	1143.23	3613.82	3692.31	17689.93	3578.66	9323.88
180000	559.37	1078.88	3494.34	3697.78	17696.24	3959.97	10410.96
190000	464.54	1045.07	3637.23	3712.65	17597.71	3694.36	6992.69
200000	527.74	1004.81	3666.09	3661.35	17637.00	3894.62	11081.06

Table A.8: Processing time(ms) at each timestamp of kddcup99

TS	CSCS	GFCS	SWC	ClusTree	SKM	G2CS	k-means
10000	175.89	421.73	1617.08	730.66	263.25	1944.67	417.22
20000	137.13	321.30	3607.87	710.99	758.51	1656.02	872.39
30000	191.32	272.44	4721.34	939.82	2364.14	1639.20	854.61
40000	242.84	512.21	5531.82	959.54	2814.99	1981.05	1129.41
50000	211.28	515.38	5732.05	1069.47	4186.95	1861.24	1584.13
60000	609.12	172.63	6115.74	1229.66	4669.59	1624.99	1756.87
70000	547.57	509.87	6497.59	1362.94	5374.90	1843.95	3680.53
80000	285.49	720.01	7162.54	1548.30	6419.69	1927.19	2230.41
90000	421.01	13.52	7601.70	1696.57	7185.66	377.47	2940.18
100000	316.32	14.17	7466.85	1947.12	7697.06	356.90	3822.83
110000	238.33	180.69	6713.72	2233.45	7407.35	558.73	7263.08
120000	435.21	571.65	6894.55	2082.54	6950.95	1548.38	5465.83
130000	330.26	422.42	7409.49	2133.57	6746.17	2143.69	5260.75
140000	517.39	473.36	7587.45	2059.41	6643.48	2109.58	3243.64
150000	363.69	444.98	7381.54	2153.92	6602.62	2087.60	5481.41
160000	424.72	869.68	7731.93	2164.12	6529.87	2253.85	9862.38
170000	317.48	416.84	8432.81	2081.97	6510.84	2328.72	5565.38
180000	188.73	524.79	8272.63	2032.48	6149.98	2137.74	5342.04
190000	374.79	690.74	9200.26	2067.98	6601.21	2428.86	4187.74
200000	330.79	799.15	9332.34	2054.96	7695.77	2562.38	4990.54

Table A.9: Average SSQ by coreset size in CSCS

Dataset	m=1000	m=5000	m=10000
kddcup99	9.2728E+10	6.3772E+10	5.4588E+10
census1990	9.2694E+06	8.1583E+06	7.7391E+06
syn1k30d40	4.2370E+07	4.6713E+07	2.2013E+07
syn1k30d80	9.0768E+07	6.6412E+07	3.7812E+07
syn1k30d160	1.2302E+08	1.0571E+08	8.1632E+07
syn1k30d200	1.2856E+08	1.3369E+08	1.0286E+08

Dataset	m=20000	m=40000	k-means
kddcup99	5.6728E+10	5.5603E+10	3.0549E+10
census1990	7.4702E+06	7.2098E+06	7.3706E+06
syn1k30d40	2.7494E+07	2.9273E+07	1.2952E+07
syn1k30d80	5.1012E+07	3.8848E+07	2.5895E+07
syn1k30d160	8.2504E+07	8.2609E+07	5.1794E+07
syn1k30d200	1.0373E+08	1.6649E+08	6.4770E+07

Table A.10: Processing time(ms) by coreset size in CSCS

Dataset	m=1000	m=5000	m=10000	m=20000	m=40000	k-means
kddcup99	2315.39	4317.65	7338.51	13880.80	23316.16	125775.72
census1990	3456.08	6839.92	11018.32	23505.86	29514.62	147256.09
syn1k30d40	3251.30	3140.57	4671.65	4809.68	5372.74	12552.11
syn1k30d80	5217.83	5647.37	5404.89	6026.60	5819.08	20561.32
syn1k30d160	9779.78	10237.67	10226.23	9644.21	10325.18	33782.32
syn1k30d200	11673.59	11750.32	12247.54	12063.57	11565.22	40321.91

Table A.11: Average SSQ by the number of hash functions in CSCS

Dataset	l=5	l=10	l=15
kddcup99	7.5048E+10	5.9187E+10	5.3440E+10
census1990	9.1618E+06	8.0500E+06	7.5898E+06
syn1k30d40	3.2970E+07	2.3710E+07	3.2448E+07
syn1k30d80	1.0051E+08	5.2281E+07	6.2865E+07
syn1k30d160	6.3351E+08	8.3686E+07	1.0928E+08
syn1k30d200	1.2313E+09	1.1673E+08	1.0042E+08

Dataset	l=20	l=25	k-means
kddcup99	5.3564E+10	5.4036E+10	3.0549E+10
census1990	7.6407E+06	7.7107E+06	7.3706E+06
syn1k30d40	3.2959E+07	3.8425E+07	1.2952E+07
syn1k30d80	6.1303E+07	7.6795E+07	2.5895E+07
syn1k30d160	1.3842E+08	1.6024E+08	5.1794E+07
syn1k30d200	1.0184E+08	1.6651E+08	6.4770E+07

Table A.12: Processing time(ms) by the number of hash functions in CSCS

Dataset	l=5	l=10	l=15	l=20	l=25	k-means
kddcup99	5989.30	7044.11	7925.03	7320.34	8887.46	125775.72
census1990	3225.94	7757.71	11667.63	16444.13	12951.85	147256.09
syn1k30d40	2366.11	2786.21	3957.87	3877.89	4371.88	12552.11
syn1k30d80	4224.97	4621.25	6113.88	6594.45	7802.85	20561.32
syn1k30d160	7879.54	8495.50	9954.07	11022.37	13616.37	33782.32
syn1k30d200	9163.49	10480.87	11881.20	13233.34	14957.23	40321.91

A.2 Experimental Results of Document Clustering

Table A.13: Average purity by RANGE size for knews-pv

RANGE	Den-CS	DenStream	D-STREAM	DBSTREAM	DBSCAN	AP-CS	AP
2000	0.3419	0	0.3225	0.3205	0.3476	0.4109	0.3222
5000	0.3074	0	0.3056	0.31472	0.3094	0.3678	-
10000	0.3015	0	0.2938	0.31612	-	0.3400	-

Table A.14: Average purity by RANGE size for knews-ft

RANGE	Den-CS	DenStream	D-STREAM	DBSTREAM	DBSCAN	AP-CS	AP
2000	0.3687	0.3091	0.3200	0.3194	0.3697	0.5716	0.8046
5000	0.3311	0.2990	0.3700	0.3832	0.3401	0.4762	-
10000	0.3132	0.2990	0.3288	0.3827	-	0.4515	-

Table A.15: Average purity by `RANGE` size for covtype

RANGE	Den-CS	DenStream	D-STREAM	DBSTREAM
2000	0.4917	0.4370	0.5045	0.5046

RANGE	DBSCAN	AP-CS	AP
2000	0.4577	0.6087	0.6808

Table A.16: Processing time(ms) of a window of knews-pv

RANGE	Den-CS	DenStream	D-STREAM	DBSTREAM
1000	63.42	354.83	35.0	344.0
2000	78.68	424.80	944.0	1148.0
5000	154.62	614.25	6355.0	6124.0
10000	363.29	850.78	23760.0	23564.0

RANGE	DBSCAN	AP-CS	AP
1000	474.41	67.12	9514.65
2000	1950.62	99.81	75547.88
5000	-	374.02	-
10000	-	1778.93	-

Table A.17: Processing time(ms) by window size for knews-ft

RANGE	Den-CS	DenStream	D-STREAM	DBSTREAM
1000	50.60	254.79	261.0	289.0
2000	52.55	412.33	695.0	805.0
5000	129.38	826.28	4042.0	4568.0
10000	311.54	1366.81	17525.0	19130.0

RANGE	DBSCAN	AP-CS	AP
1000	460.07	55.03	33375.04
2000	1774.16	80.62	220890.96
5000	-	643.19	-
10000	-	1278.96	-

Table A.18: Processing time(ms) by window size for covtype

RANGE	Den-CS	DenStream	D-STREAM	DBSTREAM
1000	14.61	119.72	18.0	47.0
2000	17.93	765.45	39.0	279.0
5000	39.69	6863.96	162.0	688.0
10000	97.18	26021.98	535.0	1938.0

RANGE	DBSCAN	AP-CS	AP
1000	150.66	20.38	13746.40
2000	547.66	42.79	64085.58
5000	-	246.34	-
10000	-	1158.06	-

초 록

슬라이딩 윈도우를 고려한 데이터 스트림 클러스터링은 윈도우가 이동할 때마다 클러스터링 결과를 생성해야 한다. 이러한 반복적인 클러스터링은 메모리 공간 및 계산 시간 측면에서 매우 비효율적이다. 본 논문에서는 슬라이딩 윈도우 집계 기법 및 최근접 이웃 탐색 기법을 활용하여 슬라이딩 윈도우 상에서의 클러스터링 문제를 해결하고자 한다. 우선, 우리의 알고리즘은 윈도우 집계 기법에 기반하여, 윈도우를 겹쳐지지 않은 일정한 크기의 조각으로 나누고, 각각에 대한 GF(group feature)를 구한다. 이렇게 구해진 GF들을 합하여 윈도우가 포함하는 모든 데이터들에 대한 요약을 생성한다. 하지만 일정 크기의 요약을 유지하기 위해서, 알고리즘은 GF의 최근접 이웃들을 병합하는 방식으로 데이터의 크기를 줄인다. 이때, 알고리즘은 LSH(Locality-Sensitive Hashing) 기법을 활용하여 최근접 이웃 탐색을 빠르게 수행한다. 추가적으로, 반복적인 클러스터링 연산을 줄이기 위해서, 클러스터링 수정 전략을 제안한다. 클러스터링 수정 전략은 새로 생성된 GF들을 가장 근접한 클러스터로 할당한다. 이 과정으로 생성된 클러스터들의 품질이 크게 떨어지지 않았다면 클러스터링 연산을 하지 않는다. 실험으로 우리가 제안하는 방법이 슬라이딩 윈도우를 활용한 데이터 스트림 클러스터링에 매우 효과적임을 밝힌다.

주요어: 클러스터링, 데이터 스트림, 슬라이딩 윈도우, 실시간 처리

학번: 2008-20919