Data Stream Clustering: An In-depth Empirical Study

XIN WANG[†‡*], Ohio State University, United States
ZHENGRU WANG[†‡¶], Nvidia, China
ZHENYU WU[‡], University of Manchester, United Kingdom
SHUHAO ZHANG[§], Singapore University of Technology and Design, Singapore
XUANHUA SHI[¶], Huazhong University of Science and Technology, China
LI LU, Sichuan University, China

Data Stream Clustering (*DSC*) plays an important role in mining continuous and unlabeled data streams in real-world applications. Over the last decades, numerous *DSC* algorithms have been proposed with promising clustering accuracy and efficiency. Despite the significant differences among existing *DSC* algorithms, they are commonly built around four key design aspects: summarizing data structure, window model, outlier detection mechanism, and offline refinement strategy. However, there is a lack of empirical studies on these key design aspects in the same codebase using real-world workloads with distinct characteristics. As a result, it is difficult for researchers to improve upon the state-of-the-art. In this paper, we conduct such a study of *DSC* on its four key design aspects. We implemented state-of-the-art variants of all of these design choices in an open-sourced platform from scratch and evaluated them using both real-world and synthetic workloads. Our analysis identifies the fundamental issues and trade-offs of each design choice in terms of both accuracy and efficiency. We even find that combining flexible design choices led to the development of a new algorithm called *Benne*, which can be tuned to achieve either better accuracy or better efficiency compared to the state-of-the-art.

CCS Concepts: • **Information systems** → **Clustering**; **Data stream mining**.

Additional Key Words and Phrases: data stream, clustering algorithm, empirical study

ACM Reference Format:
Xin Wang, Zhengru Wang, Zhenyu Wu, Shuhao Zhang, Xuanhua Shi, and Li Lu. 2023. Data Stream Clustering: An In-depth Empirical Study. *Proc. ACM Manag. Data.* 1, 2, Article 162 (June 2023), 26 pages. https://doi.org/10.1145/3589307

1 INTRODUCTION

Data Stream Clustering (*DSC*) is one of the most important data stream mining operations. In the last decades, *DSC* has been widely applied in various real-world scenarios, including network intrusion detection [35], social network analysis [18], and weather forecast [32]. *DSC* aims at grouping input tuples according to their attribute similarities on the fly. In contrast to batch clustering

*Author is also affiliated with Sichuan University, China.
†Co-first author.
‡Work is done while visiting Shuhao Zhang's IntelliStream group at SUTD.
§Corresponding author, contact at shuhao_zhang@sutd.edu.sg.
¶Authors are further affiliated with Team Heptagon at HUST, China.

Authors' addresses: Xin Wang, Ohio State University, United States; Zhengru Wang, Nvidia, China; Zhenyu Wu, University of Manchester, United Kingdom; Shuhao Zhang, Singapore University of Technology and Design, Singapore; Xuanhua Shi, Huazhong University of Science and Technology, China; Li Lu, Sichuan University, China.

1111-1111/2023/6-ART162 $15.00
https://doi.org/10.1145/3589307

algorithms such as KMeans [24, 26] or DBSCAN [16], *DSC* algorithms are required to handle some data stream characteristics, such as *cluster evolution* and *outlier evolution* [6, 18, 21, 25, 28, 29, 33]. Furthermore, in addition to clustering accuracy, processing efficiency is also a critical concern of *DSC* algorithms [12, 27].

The complex use case scenarios and varying performance metrics motivate the rapid development of numerous *DSC* algorithms [4, 5, 7, 9, 13, 18, 19, 36, 36, 37]. Despite its prosperity, it is still difficult for researchers and practitioners to determine a suitable *DSC* algorithm on real-world workloads with varying characteristics. In particular, there are several fundamental design choices of *DSC* algorithms that have different trade-offs and performance behaviors. These design choices are also highly dependent on each other. Thus, it is non-trivial to discern which ones are better than others and why.

Some prior works [12, 27] have conducted evaluations of *DSC* algorithms, but they suffer from several drawbacks, making them not able to fairly reflect and reason the behavior of *DSC* algorithms. (1) *Coarse-grained comparison among DSC algorithms:* Existing studies are limited to giving a coarse-grained analysis of absolute performance among *DSC* algorithms. They fail to investigate the design similarities and differences in algorithm design aspects and pinpoint those attributed to the performance differences. (2) *Problematic benchmark settings:* Existing studies lead to misguided experimental results that may be caused by various programming languages, compilers, and implementation tricks. Furthermore, previous benchmark settings only contain accuracy metrics while ignoring processing efficiency metrics, which are at least equally crucial. Moreover, the used workloads are not able to cover some key characteristics of real-world scenarios.

In this paper, we perform an in-depth study of key design aspects of *DSC* algorithms: (1) *summarizing data structure*, (2) *window model*, (3) *outlier detection mechanism*, and (4) *offline refinement strategy*. For each design aspect, there are multiple design choices that lead to different *DSC* algorithms that behave differently under varying workloads. As part of this investigation, we made a good faith effort to implement all of these approaches in the same framework ***Sesame*** from scratch using C++, eliminating the differences among existing implementations caused by programming languages and compilers. *Sesame* follows a modular design, which clearly separates key design aspects. We open-source *Sesame* at https://github.com/intellistream/Sesame.

Based on *Sesame*, we conduct our study using four real-world and two synthetic workloads with varying characteristics. Through extensive evaluation, we are able to make a number of novel findings, including (a) for each design aspect, none of the design choices can always guarantee good performance under varying workload characteristics and/or optimization targets; (b) each combined selection of design choices from four design aspects has its own strength and limitation and none can achieve the highest accuracy and efficiency at the same time; (c) algorithm configuration and correlations among design aspects bring further complex influence on the clustering behaviour. Our findings even promote a novel *DSC* algorithm called *Benne*, that combines flexible design choices from four design aspects. It can achieve either a better accuracy or a better efficiency compared to the state-of-the-art on all of our tested workloads.

The remaining of this paper is organized as follows: Section 2 provides some preliminary knowledge of *DSC* algorithms and the four design aspects. In Sections 3~6,we provide a brief description of every design aspect of *DSC* algorithms and the performance trade-offs among design choices. In Section 7, we discuss the methodology of our experimental study including the algorithm selection, dataset selection, and the design of our testbed – *Sesame*. We show the comprehensive evaluation results in Section 8. Section 9 discusses related work, and Section 10 concludes this study. Note that, this work focuses on single-thread execution, and exploring parallelization for DSC algorithms is a subject for future study.

2 PERLIMINARY KNOWLEDGE

This section begins with an overview of the high-level concepts of *DSC* and its algorithmic design aspects.

2.1 Data Stream Clustering

The analysis of large-scale datasets that evolve over time has gained considerable attention, with a particular focus on stream processing methods. Among the vital tasks in data stream analysis is the clustering of data streams [6]. This task involves partitioning data in streams into clusters such that similar data are grouped together while dissimilar data are separated into distinct clusters.

Unlike traditional clustering algorithms [16, 24, 26] that work on the entire dataset, *DSC* algorithms have to analyze each data point as it arrives in sequential order and perform necessary processing or learning steps in an online fashion. In particular, *DSC* algorithms commonly maintain *temporal clusters* that temporarily hold the current computed clustering results. Furthermore, there are many unique characteristics that *DSC* algorithms need to handle for the evolving data stream, such as 1) cluster evolution [18, 21, 25, 28, 29], which refers to the five types of cluster activities that occur among temporal clusters including emerge, merge, split, adjust, and delete, and 2) outlier evolution, which refers to the dynamic role exchange between outlier clusters and temporal clusters during stream clustering procedure [6, 33].

2.2 Design Aspects of DSC Algorithms

Numerous *DSC* algorithms have been proposed [4, 5, 9, 13, 18, 19, 22, 36, 37], and are all built with four key design aspects: summarizing data structure, window model, outlier detection mechanism, and offline refinement strategy. Table 1 summarize the design choices of every design aspect.

(1) Summarizing Data Structure stores the intermediate clustering information. Since data streams are typically infinite, it is impractical to store the entire input data stream for clustering. Hence, developing suitable data structures for effectively summarizing the data stream is a crucial step for any *DSC* algorithm.

(2) Window Model is used to determine the most recent input data for processing. In most cases, more recent information from the stream better reflects the evolving activities in clusters. Thus, setting up a window to store this information for clustering can effectively improve the algorithm's clustering capability.

(3) Outlier Detection Mechanism identifies the incoming data points that seem to be different from the historical stream. Existing *DSC* algorithms all identify the new data points which are far from the temporal clusters as outliers. However, when outlier evolution occurs in data stream, some previous outliers may become part of clusters and some clustered points may become outliers. Therefore, the detection of outliers over data streams is always a challenging task for all of the *DSC* algorithms.

(4) Offline Refinement Strategy refers to the process of applying offline clustering algorithms to refine the clustering results from online clustering. Different from the other three design aspects that aim to keep execution continuously in real-time, the offline refinement strategy only applies once before getting the final clustering result. Thus, it usually does not bring a significant influence on efficiency but hopefully improves accuracy.

3 SUMMARIZING DATA STRUCTURE

Generally there are two main catalogues for summarizing data structure: the hierarchical catalogue and the partitional catalogue. In the following, we discuss six representative summarizing data

Table 1. Summary of design aspects

Design Aspect	Design Choices	Efficiency	Accuracy	Notes
Summarizing Data Structure	CFT	high	high	efficient data insertion and some additional operations for handling cluster evolution
	CoreT	low	High	need to rebuild the whole tree for updating lazily
	DPT	high	high	contain additional density information for clustering
	MCs	low	high	keep basic structure of CFT and additional time information; needs to search for every clusters during data insertion
	Grids	high	low	no need for frequent distance calculation but not so much accurate during data insertion
	AMS	low	High	need to frequently rebuild the structure
Window Model	LandmarkWM	depends	depends	difficult to determine a suitable landmark configuration
	SlidingWM	high	low	fixed window size results in fewer clusters for computation
	DampedWM	low	depends	process all the data points with a decay function
Outlier Detection	NoOutlierD	high	low	do not use any outlier detection mechanism
	OutlierD	high	high	detect outlier clusters through density and reduce the number of temporal clusters
	OutlierD-B	low	high	well handle cluster evolution among outliers and avoid cluster pollution but consumes much time to maintain the buffer
	OutlierD-T	high	high	prevent removing active clusters and be more focused on recent cluster information
	OutlierD-BT	low	high	improve the algorithms' ability for clustering evolving stream but still be low efficient in maintaining buffer
Offline Refinement	Refine	minor overhead	minor impact	any existing batch-based clustering algorithms may apply to refine results
	NoRefine	no impact	no impact	do not apply any further refinement

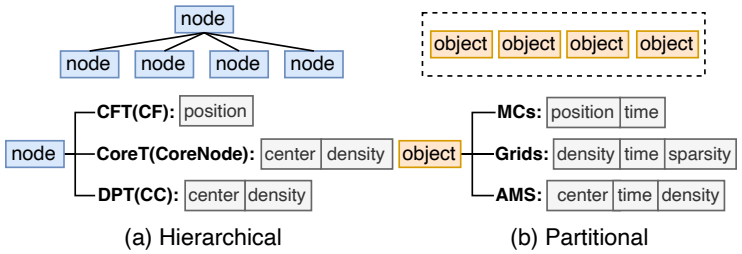


Fig. 1. Six Types of Summarizing Data Structure.

structures commonly used in *DSC* algorithms. Three are hierarchical-based, and three are partitional-based. Their main features is summarized in Figure 1.

3.1 Hierarchical Summarizing Data Structure

Hierarchical summarizing data structure groups the incoming data points into a tree structure. As shown in Figure 1(a), each temporal cluster is depicted as a *node*, and the parent cluster comprises multiple sub-clusters within the tree. To insert a new data point, it starts by searching for a suitable temporal cluster from the root node to one of the leaf nodes. Such a tree structure makes the insertion fast as there is no need for checking every existing temporal cluster. In the following, we discuss three popular and representative hierarchical summarizing data structures different from each other for the implementation of each tree node.

Clustering Feature Tree (CFT). CFT [37] is one of the oldest but representative hierarchical summarizing data structures. Every tree node of CFT is named *clustering feature (CF)*, which consists of the position information of a temporal cluster. Although CF is quite simple, it is adequate to support many basic operations of stream clustering, such as distance calculation and cluster

updating. Due to the additivity of the CF [37], CFT can be updated incrementally rather than starting from scratch again. Moreover, it also offers some advanced operations such as merging or splitting for handling cluster evolution during stream clustering.

Coreset Tree (CoreT). CoreT [4] is a binary tree structure used for extracting a core subset from a large volume of stream data. Compared with CF in CFT, the tree node of CoreT named *CoreNode* provides additional information about the density of the corresponding cluster. However, since CoreT is unable to update incrementally like CFT, it needs to rebuild the whole tree structure during the clustering procedure, which might influence the clustering efficiency of the whole algorithm. To overcome this shortage, CoreT reduces the time of rebuilding via lazily performing the update. In exchange, it may not be able to provide real-time clustering information for accurate clustering, especially when cluster evolution happens frequently.

Dependency Tree (DPT). DPT [18] is one of the most recent hierarchical summarizing data structures. It is specifically designed for handling cluster evolution. The tree node of DPT is called *cluster cell* (CC). DPT is built based on the cluster density and it requires every cluster with low density to be connected to its nearest denser cluster. Although CC contains similar clustering information as CoreNode in CoreT, it can partially adjust itself for adapting to the evolving activities in the stream, helping to achieve a higher clustering efficiency.

3.2 Partitional Summarizing Data Structure

As shown in Figure 1(b), partitional summarizing data structures do not organize temporal clusters into hierarchical structures. They hence denote temporal clusters as *objects*, rather than nodes. When inserting a new data point, they need to check every object. This brings significant overhead compared to hierarchical structures but may help to achieve a higher clustering accuracy. In the following, we discuss three typical partitional summarizing data structures, different from each other by the implementation of objects.

Micro Clusters (MCs). MCs [5] is one of the oldest partitional summarizing data structures. It contains the same positional information as CF in CFT. To keep track of the clusters' activities in real-time, it has two additional elements used for summarizing the timestamps for the clusters' updates. Therefore, it is more comprehensive and robust for stream clustering with high accuracy, especially under cluster evolution.

Grids (Grids). Grids [13] performs its clustering simply based on checking whether the data lies in a specific grid of d-dimensional space and grouping the dense grids into temporal clusters. Compared to other data structures, Grids is more computationally efficient because it does not require frequent distance calculations between new data and the grid. Additionally, it periodically removes sparse grids, which further boosts its clustering efficiency. However, since every grid has a fixed position in the space and the information about the relationship among grids is only updated lazily, Grids may not work well when some evolving activities occur frequently among the grid and thus leads to lower clustering accuracy.

Augmented Meyerson Sketch (AMS). AMS [9] is the extension of traditional Meyerson Sketch [31] which can better summarize the clustering information using the limited number of data points. AMS mainly consists of a set of temporally weighted centres and the weight is mainly influenced by both the time and density information of the corresponding cluster. Similar to CoreT, AMS may need to delete and reconstruct the created sketch to adapt to the evolving data stream. This brings a huge efficiency cost and temporal information loss. Another drawback of AMS is that it needs a predefined number of clusters (K) that is hard to determine for clustering of real-world evolving data streams.

Discussion. In general, hierarchical summarizing data structure improves the clustering efficiency while the partitional structure is expected to bring in better accuracy. The unique

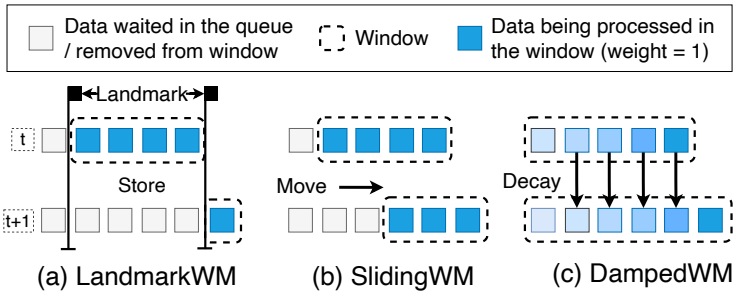


Fig. 2. Three Types of Window Model.

design of each specific structure, however, leads to further differences in their clustering behaviour. For example, although CFT, CoreT, DPT and MCs promise high clustering accuracy due to the comprehensive information stored in their structure, their efficiency behaviour is quite different. Only CFT and DPT can guarantee high efficiency due to their flexible updating. CoreT and AMS are slow due to frequently rebuilding the whole structure while MCs is slow since it needs to scan every temporal cluster for data insertion. For Grids, although it may not be as accurate as the other structures, it largely saves its computation time due to eliminating the distance calculation operation.

4 WINDOW MODEL

The window model can be categorized into three main types as illustrated in Figure 2. For each kind of window model, there are further two types of window implementations: count-based and time-based. We base on the count-based window when implementing *DSC* algorithms in our benchmark as it is more frequently used [6].

4.1 Landmark Window Model (LandmarkWM)

As shown in Figure 2(a), LandmarkWM collects the data between two landmarks into a window for clustering. When reaching the current landmark, the temporal clustering result will be stored before clearing the current summarizing data structure. After that, the algorithm will continue performing clustering based on the new 'blank' structure until reaching the next landmark. It is hard to define a proper landmark configuration for LandmarkWM. If the two landmarks are spaced very close together, many evolving activities in the stream will not be captured inside the window, leading to poor clustering accuracy. On the contrary, if the two landmarks are set far from each other, a great number of data points will be involved inside the window for computation and the efficiency will decrease significantly.

4.2 Sliding Window Model (SlidingWM)

SlidingWM is usually implemented as a queue following a First-In-First-Out processing strategy as depicted in Figure 2(b). Similar to the LandmarkWM, the SlidingWM only considers data objects whose timestamp falls within the current window range with older ones discarded. In contrast to the LandmarkWM whose window size can be dynamically determined by the landmark, the size of the SlidingWM is predefined with a fixed value. The non-discriminate nature of data insertion and deletion results in a smaller number of clusters by comparison and thus reduces computation time and resources. However, this also means the clustering accuracy will greatly decrease especially under a small window size configuration.

4.3 Damped Window Model (DampedWM)

For both the LandmarkWM and the SlidingWM, all data objects in the window are associated with equal importance. In contrast, to ensure the more recent data points in the window are always being treated with higher priority, the DampedWM associate objects in the data stream with weights that decay with a function $f(\alpha, \lambda)$ over time. Meanwhile, unlike the other two types of window models, DampedWM does not discard objects as time goes on. While taking all data into account may bring in high accuracy, this causes efficiency issues. Furthermore, since the α and λ parameters are predefined, the decay function is always kept the same to determine the priority, and the DampedWM may be more susceptible to some stream characteristics such as outlier evolution.

Discussion. Although these window models all try to prioritize the most recent data from streams, their specific strategies are quite different leading to different clustering behaviours. For a better clustering efficiency, SlidingWM may be the best choice since its fixed window size largely reduces the number of clusters for maintenance. On the contrary, although both LandmarkWM and DampedWM may not be as efficient as the SlidingWM, they can bring in more accurate clustering results. Finally, for all three types of window models, their clustering behaviour is sensitive to their specific window configurations, which have not been studied in depth in the literature.

5 OUTLIER DETECTION MECHANISM

Unlike the former two design aspects, the outlier detection mechanism is an optional design aspect. It further contains two optional independent optimizations: the use of a buffer and a timer, respectively. Put them together, including not using any outer detection mechanism (NoOutlierD), we have five different choices for this design aspect.

5.1 Outlier Detection (OutlierD)

As depicted in Figure 3(a), the basic outlier detection mechanism (OutlierD) is to periodically select temporal clusters that are sparse in density (denoted in light colour in the figure) to transform into the outlier clusters and remove them from the memory [5, 13]. Compared with NoOutlierD, its clustering accuracy will greatly improve. Moreover, the whole clustering efficiency may not even get influenced by adding this additional detection operation since the number of temporal clusters will be reduced, saving time for future data insertion and cluster updating.

5.2 Outlier Detection with Buffer (OutlierD-B)

Due to the influence of outlier evolution, some outliers, which can be grouped into sparse clusters with other outliers, need to be tracked during the clustering procedure. This is because these outlier clusters may become dense enough after adding more and more outliers and will be transformed into temporal clusters. In this case, setting a buffer to store the summary of these outlier clusters for later detection (OutlierD-B), as shown in Figure 12(b), is one of the most commonly used optimization strategies in outlier detection [18, 36]. Another advantage of applying a buffer for detection is that incoming data once detected as outliers are immediately inserted into the outlier buffer rather than temporal clusters avoiding pollution. However, maintaining the outlier buffer consumes additional computational time and resources.

5.3 Outlier Detection with Timer (OutlierD-T)

Apart from periodically identifying outlier clusters via density information, *DSC* algorithms may further check whether the temporal clusters are not active anymore before transforming them to outlier clusters with a timer (OutlierD-T) as illustrated in Figure 12(c). Using a timer makes the detection more robust than OutlierD since it helps to prevent accidentally removing temporal

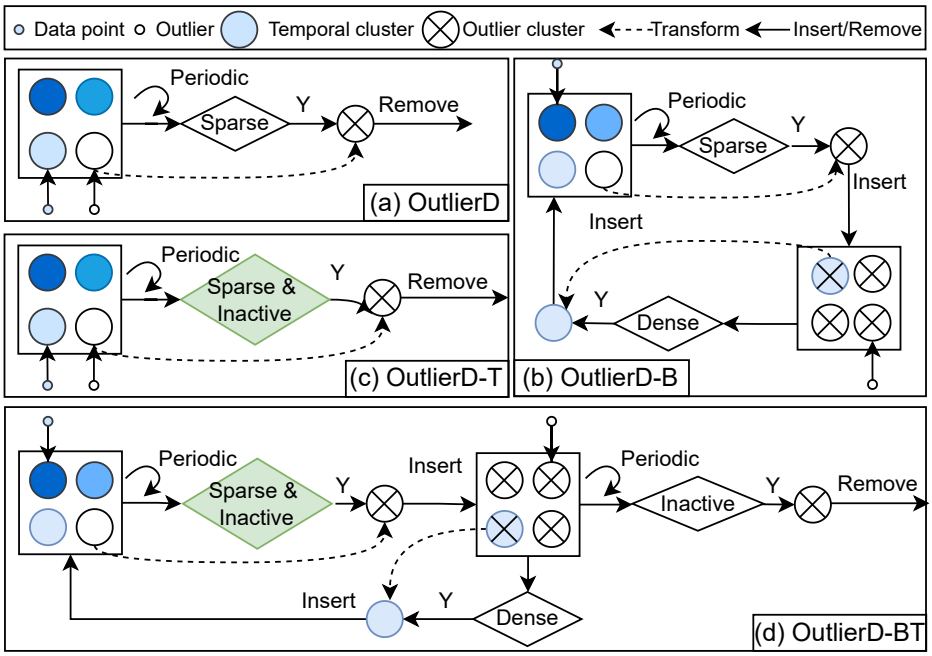


Fig. 3. Four Types of Outlier Detection Mechanisms.

clusters that may soon become dense. Additionally, using a timer helps the *DSC* algorithm to be more focused on the recent cluster changes, thus bringing higher efficiency.

5.4 Outlier Detection with Buffer and Timer (`OutlierD-BT`)

Outlier buffer and outlier timer can be also used in combination (`OutlierD-BT`) as depicted in Figure 12(d). Similar to `OutlierD-B`, `OutlierD-BT` maintains a buffer to temporarily store outlier clusters, which may transform back to temporal clusters. Different from `OutlierD-B` though, `OutlierD-BT` additionally checks if a sparse temporal cluster is inactive before transforming it to an outlier cluster with the timer. Furthermore, the algorithm will also periodically check and remove some inactive outlier clusters in the buffer. This helps the algorithm improve its clustering accuracy for handling outlier evolution but also consumes a lot of time for buffer maintenance.

Discussion. The buffer and timer optimizations can be applied either separately or simultaneously, resulting in different decisions regarding the outlier detection mechanism. The use of a buffer can greatly improve the clustering quality due to its better ability for handling outlier evolution. However, maintaining the outlier buffer in real-time will also bring down the efficiency. On the contrary, the use of the timer may be beneficial for both better clustering accuracy and efficiency, especially under outlier evolution. Using a buffer and timer together in outlier detection further complicates the clustering behaviour and motivates an in-depth study.

6 OFFLINE REFINEMENT STRATEGY

The basic idea of the offline refinement strategy is to utilize offline clustering algorithms to improve the online clustering results. Being another optional design aspect in *DSC* algorithm, we study two design choices, `NoRefine` and `Refine`. As shown in Figure 4, if apply `NoRefine`, the algorithm will directly output the temporal clusters C as the final clustering results. Instead, if `Refine` is applied,

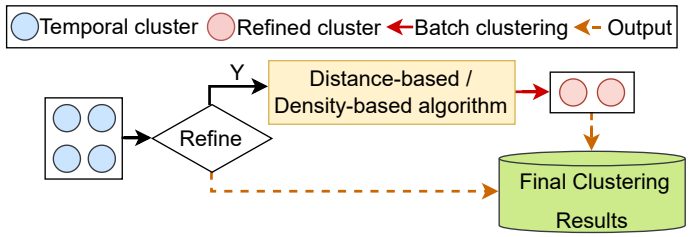


Fig. 4. Offline Refinement Strategy.

Table 2. Specification of our evaluation platform

Component	Description
Processor	Intel(R) Xeon(R) Gold 6338 CPU @ 2.00GHz
L3 Cache Size	48MiB
Memory	256GB DDR4 RAM
OS	Ubuntu 20.04.4 LTS
Kernel	Linux 5.4.0-122-generic
Compiler	GCC 11.3.0 with -O3

some batch clustering algorithms, such as *density-based* (DBSCAN) or *distance-based* (KMeans), will be applied on C for further refining the results before output.

Discussion. Offline refinement is optional and can be easily applied to refine online clustering results before outputting the result. The common wisdom is that it may further improve clustering quality. Unfortunately, there is no prior study about the real impact of this design aspect on both accuracy and efficiency. Even in recent years, some choose [4] to involve this design aspect in their proposed *DSC* algorithms, and some choose not [18]. We will empirically show its impact via extensive evaluations.

7 METHODOLOGY

In this section, we provide a detailed description of our experimental setting. In Section 7.1, we first discuss our selection of eight existing *DSC* algorithms as references to our study. Next, we describe our workload selection and the benchmark testbed implementation in Section 7.2 and Section 7.3, respectively. All experiments are carried out on an Intel Xeon processor. Table 2 summarizes the detailed specification of the hardware and software used in our experiments.

7.1 Algorithm Selection

We base our study on eight existing *DSC* algorithms. The summary of these algorithms is shown in Table 3. BIRCH [37] can be considered a primitive algorithm in this area. In particular, the use of a *clustering feature (CF)* for summarizing large amounts of data was first introduced in the BIRCH algorithm in 1996. The CF was later extended and named *micro cluster (MC)* in the CluStream [5] algorithm, which is considered the first *DSC* algorithm for efficient clustering of evolving data streams with an online-offline strategy. DenStream [36] is a density-based algorithm that also utilizes CF. It further introduces *outlier detection* in order to reduce the impact of noise on stream clustering. DStream [13] partitions the n-dimensional feature space into density cells, and maps each data stream object into density grid cells. Different from prior algorithms, StreamKM++ [4] applies a two-step mechanism called *"merge and split"*, building around a data structure called *Coreset Tree*. DBStream [19] addresses the issue that dense areas formed by online clusters may

Table 3. A summary of representative *DSC* algorithms and their design decisions. The year attribute for each algorithm is when it was first published.

Algorithm	Year	Summarizing Data Structure		Window Model	Outlier Detection	Offline Refinement
		Name	*Catalog*			
BIRCH [37]	1996	CFT	Hierarchical	LandmarkWM	OutlierD	NoRefine
CluStream [5]	2003	MCs	Partitional	LandmarkWM	OutlierD-T	Refine
DenStream [11]	2006	MCs	Partitional	DampedWM	OutlierD-BT	Refine
DStream [13]	2007	Grids	Partitional	DampedWM	OutlierD-T	NoRefine
StreamKM++ [4]	2012	CoreT	Hierarchical	LandmarkWM	NoOutlierD	Refine
DBStream [19]	2016	MCs	Partitional	DampedWM	OutlierD-T	Refine
EDMStream [18]	2017	DPT	Hierarchical	DampedWM	OutlierD-BT	NoRefine
SL-KMeans [9]	2020	AMS	Partitional	SlidingWM	NoOutlierD	NoRefine

Table 4. Characteristics differences of selected workloads. Note that the outliers column refers to whether there are outliers in the final clustering results.

Workload	Length	Dimension	Cluster Number	Outliers	Evolving Frequency
FCT [1]	581012	54	7	False	Low
KDD99 [35]	4898431	41	23	True	Low
Insects [34]	905145	33	24	False	Low
Sensor [2]	2219803	5	55	False	High
EDS	245270	2	363	False	Varying
ODS	100000	2	90	Varying	High

be separated by small, low-density areas in micro-cluster-based clustering algorithms during refinement. EDMStream [18] uses a damped window model thus the stored cluster density decays over time. SL-KMeans [9] presents the first algorithms for the k-clustering problem on sliding windows with space linear in k and empirically shows that it performs better than analytic bounds. We select these algorithms based on two key criteria: 1) as illustrated in Table 3, these algorithms cover a wide range of design decisions of all four design aspects; 2) the eight selected *DSC* algorithms are either representative (BIRCH [37] proposed in 1996) or proposed recently (e.g., SL-KMeans [9] proposed in 2020) covering a long history in this field.

7.2 Dataset Selection

Table 4 summarizes the characteristics of our selected workloads. We carefully select six datasets for evaluation with two key considerations. First, datasets originally used for evaluating any one of the eight algorithms in Table 3 are inconsistent. For a fair and thorough evaluation, we selected the three most commonly used datasets in our experiment. Specifically, *FCT* (Forest CoverType) is used by SL-KMeans, StreamKM++, EDMStream, and DBStream. *KDD99* is used by StreamKM++, DStream, EDMStream, DenStream, CluStream, and DBStream. *Sensor* is used by DBStream. Besides these three classical datasets, we added another recent dataset named *Insects* proposed in 2020 [34]. Second, although some prior works proposed synthetic datasets for evaluation, they are not publicly available. To better evaluate the algorithm under changing workload characteristics as shown in Table 4, we further designed two synthetic datasets, *EDS* and *ODS*. Specifically, *EDS* contains varying frequencies of the occurrence of cluster evolution, while *ODS* contains a time-varying number of outliers at different stages.

A detailed description of our selected workloads is provided as follows. ***FCT*** (Forest CoverType) [1] contains tree observations from four areas of the Roosevelt National Forest in Colorado. It is a high-dimensional dataset with 54 attributes. Every data point has its cluster label representing its belonging tree type, and there are no outliers in the dataset. ***KDD99*** [35] contains a large volume of network intrusion detection stream data collected by the MIT Lincoln Laboratory.

The dimension of this workload is also high, however, different from *FCT*, it contains a lot of outliers and has been frequently used to test algorithms' ability to identify outliers. ***Insects*** [34] is the most up-to-date stream workload which is proposed in 2020. It is generated by an optical sensor that measures insect flight characteristics and is specifically designed to test the clustering of evolving data streams. ***Sensor*** [2] contains temperature, humidity, light, and voltage information collected from sensors deployed in Intel Berkeley Research Lab. It is a low dimensional workload with only 5 attributes. Different from *FCT*, *KDD99* and *Insects*, the frequency of cluster evolution is very high in this workload. Thus, many prior works have used it to measure their algorithms' ability to handle cluster evolution [7, 12, 19]. ***EDS*** is a synthetic workload applied by some previous works [36] to further study cluster evolution. It is made by combining three sub-synthetic datasets with different cluster evolution frequencies. In our experiment, we divide *EDS* into five stages according to the evolving frequency. Comparing algorithms' behaviour from phase 1 to 5, we can further know the changes in their clustering ability with the increase of cluster evolution frequency. ***ODS*** is also made through the combinations of several sub-synthetic datasets with the same low dimension. Different from *EDS*, the second half part of *ODS* is purely made up of outliers. We can hence further analyze algorithms' clustering ability under various numbers of outliers using *ODS*.

7.3 Benchmark Testbed

Many open-source projects have created frameworks or libraries for data stream mining, including implementations for *DSC* algorithms, but they only support a biased subset of *DSC* algorithms and are not easily extensible. For example, massive online analysis (MOA) [8] and scalable advanced massive online analysis (SAMOA) [14] are popular data stream mining frameworks. However, they do not provide a clear modular architecture to separate design aspects of *DSC* algorithms making the extension to include novel algorithm design options difficult. Another repeating issue in previous empirical studies of *DSC* algorithms [18] is that they do not clearly separate the data producer (i.e., producing the data stream) and consumer (i.e., running the *DSC* algorithm) in their experiments.

We made a good faith effort to implement various design choices of *DSC* algorithms in the same testbed, called ***Sesame***, written in C++. For a more realistic setting, *Sesame* is made up of three main threads forming a pipeline for processing the data stream. The communication between threads is realized via a shared-memory queue to eliminate the impact of network transmission.

(1) *Data producer thread* loads benchmark workloads in memory, and then continuously inserts each input data point into a queue. We configure a high input arrival rate, i.e., every data point arrives immediately, so that the algorithm does not spend time waiting for the input data.

(2) *Data consumer thread* runs a *DSC* algorithm to process the data stream. The input tuple is continuously fetched from the queue to be processed. Temporal clustering results are produced and sent to the next thread. All efficiency measurements are conducted in the second thread to ensure a fair comparison among *DSC* algorithms. We use throughput for efficiency comparison. We reference DStream, DBStream, CluStream and DenStream to corresponding papers and the SAMOA benchmark [14, 20]. We reference BIRCH, StreamKM++, EDMStream, and SL-KMeans to corresponding papers and their open-sourced code.

(3) *Result collector thread* is used to store the temporal clustering results produced from the second thread. We run accuracy measurements in this thread to minimize the interference of efficiency measurement. We use purity [15] to measure the general clustering quality and also use CMM [23] to test the design aspects' ability to handle cluster evolution.

8 EXPERIMENTAL ANALYSIS

We now present our evaluation and analysis of the stream clustering design decisions discussed in this paper. As pointed out by Hahsler et al. [19], average purity depends on the number of clusters. We hence tune each of the algorithms such that the generated number of clusters is close to the ground truth while maximizing its purity. Specifically, we apply a command line tool called ticat [3] which automatically launches *Sesame* with different algorithms, datasets, and parameter settings. We then select the parameter setting that leads to the highest clustering quality for every study following the previous works [7, 12].

In the following, we first summarize our key findings in Section 8.1. We present our comprehensive evaluation of every design aspect in Section 8.2 ~ Section 8.5. Based on the modularized experiments, we obtain a number of observations about the behaviour of every design choice. To further verify these observations, we finally compare the behaviour of the eight existing *DSC* algorithms and a novel *DSC* algorithm called *Benne* derived from our study in Section 8.6.

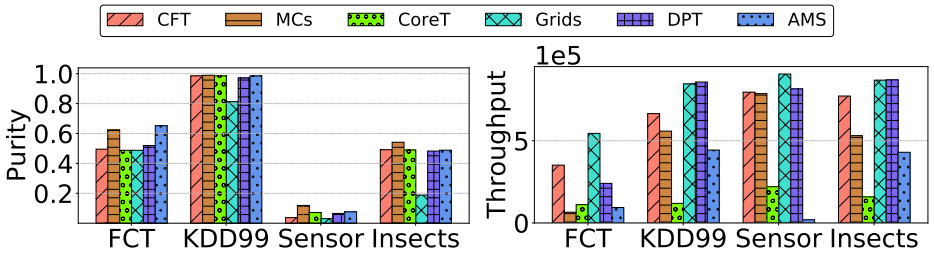
8.1 Key Findings

We summarize our key findings based on a number of observations in the experiment as follows:

K1 **There is Still No Silver Bullet (O3-4, O6-8, O13-14, O16):** For each design aspect, none of the design choices can always guarantee good performance under varying workload characteristics and/or optimization targets. In particular, workload characteristics such as *cluster evolution* (**O3-4**, **O6-7**), *outlier evolution* (**O6, O8**, **O13-O14**), and *workload dimensions* (**O16**) bring non-trivial impacts to stream clustering process. Notably, our results indicate that (1) `MCs` and `CFT` guarantee high accuracy while `CFT` and `DPT` guarantee high efficiency for handling cluster evolution than other types of data structure (**O3-4**). (2) The efficiency `DampedWM` becomes worse with the increase of cluster evolution frequency while the accuracy of `LandmarkWM` becomes better under outlier evolution (**O6-8**). (3) Utilizing a timer in the outlier detection mechanism can guarantee good clustering accuracy and efficiency under outlier evolution (**O13-14**).

K2 **It is Non-trivial to Determine the Best Overall Design (O1-2, O5, O15, O18):** Each overall design (i.e., a combined selection of design decisions from all design aspects) has its own strength and limitation and none can achieve the highest accuracy and efficiency at the same time. Notably, our results indicate that (1) `Grids` summarizing data structure and `SlidingWM` window model can both guarantee a high clustering efficiency but low accuracy while `AMS` and `CoreT` focus on high accuracy but achieve low efficiency (**O1**, **O5**). (2) Hierarchical summarizing data structures can bring in better efficiency while partitional structures bring higher accuracy (**O2**). (3) Applying an offline refinement strategy has little impact on both clustering accuracy and efficiency (**O15**). (4) None of the existing algorithms is able to achieve high accuracy for workloads with high cluster evolution frequency (**O18**). (5) Composing suitable design choices from each design aspect, we obtain a novel *DSC* algorithm (i.e., *Benne*) that can be reconfigured to achieve either the highest accuracy or highest efficiency, but not at the same time (**O18**).

K3 **Algorithm Configuration and Correlations among Design Aspects Bring Further Complex Influence on the Clustering Behaviour (O9-10, O17):** In our experiment, we find that different algorithm configurations bring non-trivial trade-offs in terms of accuracy and efficiency (**O9-10**). Additionally, we observe that an unsuitable summarizing data structure dominantly leads to poor performance of *DSC* algorithms regardless of the selection of other design choices (**O17**), which indicates a main correlation among selections of different design aspects influencing their clustering behaviour.



(a) Accuracy (b) Efficiency

Fig. 5. The Behaviour Comparison of Different Data Structures on Four Real-world Workloads.

8.2 Summarizing Data Structure

We first compare *DSC* algorithms' behaviour with varying summarizing data structures. We fix algorithms to use (1) `LandmarkWM`, (2) `OutlierD`, and (3) `NoRefine`. After that, we conduct further experiments to specifically study the impact of key workload characteristics including cluster evolution on the behaviour of different summarizing data structures.

General Comparison: The clustering behaviour of different summarizing data structures is shown in Figure 5. There are three major observations:

O1: Both `AMS` and `CoreT` data structures focus on improving accuracy at the cost of efficiency, whereas `Grids` prioritizes efficiency over accuracy. As shown in Figure 5a and Figure 5b, `AMS` and `CoreT` both achieve comparable purity but at least 30% lower throughput than average. As discussed in section 3.2, their similar behaviour is attributed to the frequent reconstruction of the main structure, which improves the whole clustering quality but consumes too much time. On the contrary, the throughput of `Grids` is at least 25% higher than the average, which confirms its higher efficiency. However, it achieves the lowest accuracy when compared with others on all four workloads. As discussed in Section 3.2, this is due to its unique data insertion method, which saves computation time, but leads to lower clustering accuracy. Therefore, when the user cares more about the clustering efficiency, `Grids` may be a great design decision.

O2: Hierarchical summarizing data structures can bring in better efficiency while partitional structures bring higher accuracy. `CFT`, `CoreT`, and `DPT` are hierarchical structures, while `MCs`, `Grids`, and `AMS` are partitional structures. It is clear to see that neither hierarchical nor partitional structure can guarantee better accuracy and efficiency at the same time. Although the hierarchical summarizing data structures (i.e., `CFT`, `CoreT`, and `DPT`) are about ~70% faster than the partitional structures (i.e., `MCs` and `AMS`) except `Grids`, their accuracy is worse than the partitional structures. The selection of a suitable structure is determined by the optimization target. As discussed in Section 3, since no information about the relationship between clusters is stored in a partitional structure, the algorithm needs to check every existing cluster during data insertion. On the contrary, the algorithm with a hierarchical structure only needs to search for a subset of child nodes (clusters) along the path from the root node to the leaf nodes (clusters), which helps to achieve higher efficiency. However, the hierarchical data structure may fail to find the optimally suitable cluster for insertion when some of the hierarchical connection is misleading or outdated.

O3: The clustering accuracy and efficiency of summarizing data structure are highly influenced by cluster evolution frequency. We observe that the clustering behaviour of summarizing data structure varies significantly with different workload characteristics. In Figure 5a and Figure 5b, we see that `MCs` achieves ~80% higher purity while both `CFT` and `DPT` even achieve ~40% higher throughput than average on *Sensor* workload with high cluster evolution frequency. To

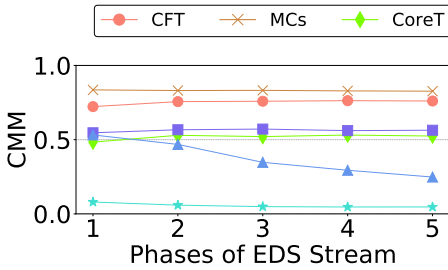


Fig. 6. CMM (*EDS*).

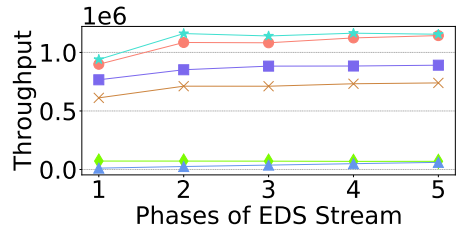
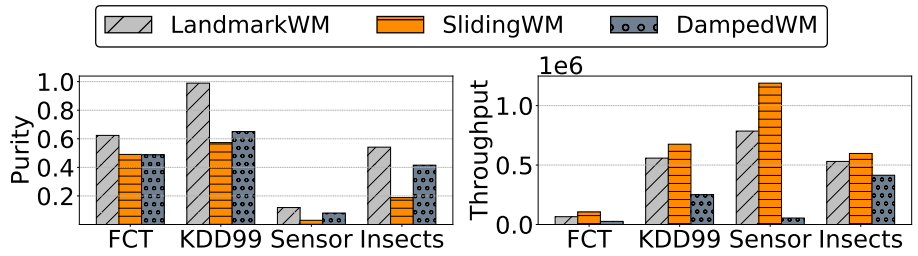


Fig. 7. Throughput (*EDS*).



(a) Accuracy (b) Efficiency

Fig. 8. The Behaviour Comparison of Different Window Models on Four Real-world Workloads.

better comprehend the impact of these workload characteristics, we use *EDS* workload to conduct a further investigation as follows.

Cluster Evolution: The *EDS* workload has five phases with varying cluster evolution frequency. In this study, we use CMM [23] as the accuracy metric instead of purity since it is specifically designed to measure algorithms' ability for handling cluster evolution. The results are shown in Figure 6 and Figure 7. `Grids` achieves the lowest CMM but the highest throughput while `AMS` and `CoreT` achieve the average CMM but quite low throughput, which reaffirms our analysis in **O1**. Besides, there is one additional observation as follows.

O4: `MCs` and `CFT` guarantee high accuracy while `CFT` and `DPT` guarantee high efficiency for handling cluster evolution than other types of data structure. As shown in Figure 6, the CMM of `MCs` and `CFT` are much higher than other types of data structures, this is due to their advanced operations additionally provided to handle the evolving data stream. For efficiency comparison on *EDS* according to Figure 7, it shows that `MCs` is not as fast as `CFT` since it is a partitional-based structure, which has been discussed in **O2**. Moreover, as expected, `DPT`, which is specifically designed for well detecting and adapting to cluster evolution [18], also outperforms others since it is also able to partially adjust its structure for clustering the data stream.

8.3 Window Model

We next evaluate the window model. For all of these experiments, we fix the algorithms to use `MCs` summarizing data structure with `OutlierD` and `NoRefine`. We begin with an analysis of different window models on real-world workloads. After that, we specifically study two key workload characteristics, cluster evolution and outlier evolution on their individual impacts.

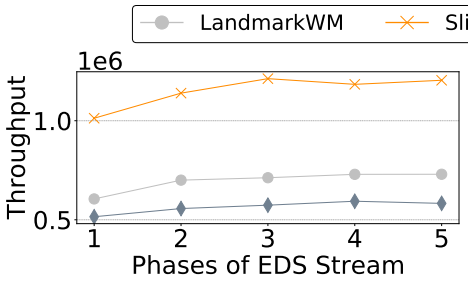


Fig. 9. Throughput (*EDS*).

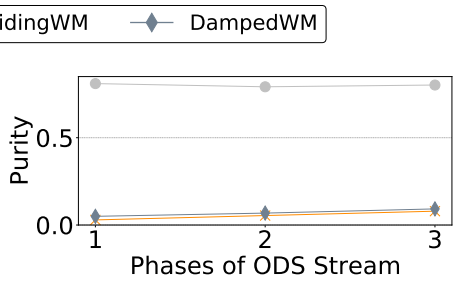


Fig. 10. Purity (*ODS*)

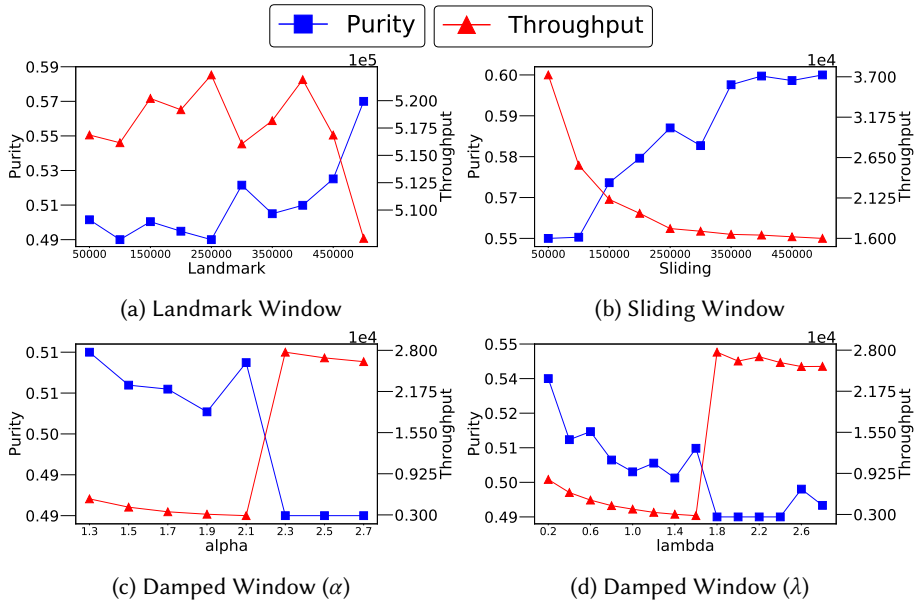
General Comparison: This experiment compares the behaviour of different types of window models on four real-world workloads. According to the experimental results shown in Figure 8a and Figure 8b, there are two general observations shown as follows.

O5: The `SlidingWM` achieves the lowest clustering accuracy but the highest efficiency among the three types of window models. As shown in the figure, the `SlidingWM` achieves at least ~15% higher throughput but much lower purity than `LandmarkWM` and `DampedWM`. As discussed in Section 4.2, the higher efficiency and lower accuracy are attributed to the fixed size of the window, which reduces the time for inserting the new data point but limited the clustering information retained in the window.

O6: The behaviour of a window model is highly dependent on cluster evolution and outlier evolution frequency in the workload. We can see that the purity of the `LandmarkWM` is much higher than the `DampedWM` and `SlidingWM` on *KDD99* compared with values on the rest three workloads . This difference in clustering behaviour on different workloads is attributed to the outlier evolution according to table 4. In addition, the `SlidingWM` achieves ~75% higher throughput than avearage on *Sensor*, the dataset with the highest cluster evolution frequency among the four workloads. This difference on *Sensor* is quite larger than on the other three workloads. The different behaviour of these window models on *Sensor* might be attributed to the impact of cluster evolution frequency.

Cluster Evolution: We next evaluate the impact of cluster evolution on the clustering efficiency of different window models. For comparison, we record the throughput of handling every phase of *EDS* with varying cluster evolution frequency. The comparison result is shown in Figure 9, we can see that the throughput of `SlidingWM` is much higher than `LandmarkWM` and `DampedWM`, which reaffirms our previous analysis in **O5**. Apart from that, there is one major observation:

O7: The efficiency of `DampedWM` is much worse than other two types of window model with the increase of cluster evolution frequency. The result shows that `DampedWM` achieves the lowest throughput among the three types of window models. As discussed in Section 2.1, one of the major cluster changes resulting from cluster evolution is the emergence of new clusters. For *EDS*, the number of clusters becomes larger from phases 1 to 5. Due to the fact that `DampedWM` only eliminates a few outdated data using its decayed function, more clusters, including the outdated ones will be captured and stored by `DampedWM`. Therefore, the total efficiency will become quite low since too much time is required for frequently updating the weight of clusters and inserting the new stream data during the clustering procedure. Additionally, when cluster evolution becomes more frequent from phases 1 to 5, `LandmarkWM` and `SlidingWM` generally become more and more efficient while the `DampedWM` always keeps slow, which also shows that `DampedWM` can not guarantee a great clustering efficiency especially under the frequent cluster evolution.



(a) Landmark Window (b) Sliding Window

(c) Damped Window (α) (d) Damped Window (λ)

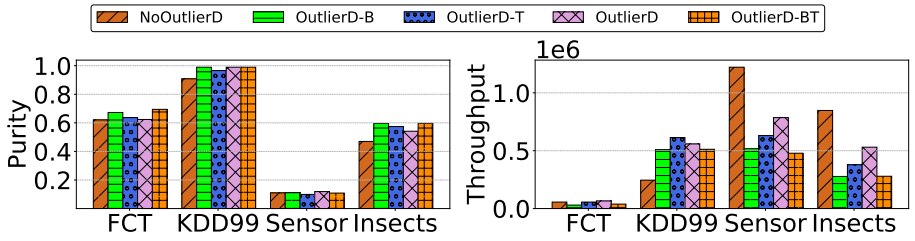
Fig. 11. The Change of Behaviour When Tuning Different Window Settings.

Outlier Evolution: We then evaluate the impact of outlier evolution on the clustering accuracy of different window models. Similar to the evaluation on *EDS*, we record the purity of handling every phase of *ODS* with varying outlier evolution frequency. The comparison result is shown in Figure 9. As discussed in **O5**, `SlidingWM` achieves the lowest purity. Additionally, we obtain one major observation:

O8: `LandmarkWM` has a much better ability to accurately handle outlier evolution than other types of window models. Surprisingly, as shown in Figure 9, the purity of `LandmarkWM` is ~500% higher than `DampedWM` on *ODS*. This is because of the fixed decay function of `DampedWM`. Under the occurrence of outlier evolution in the stream, the fix function will often lead to the false identification of outliers which should be either absorbed by those expired clusters or become the start of a new cluster. On the contrary, the `LandmarkWM` treats every historical data equally, which can better help the algorithm identify outliers in a wider time range, especially when data needs to be inserted into some old clusters. Therefore, with the increase of outlier evolution frequency, the `LandmarkWM` guarantees a higher accuracy.

Window Configuration Study: We now conduct a further study on the changes in clustering behaviour with varying window configurations. Specifically, we evaluate the landmark and size for the `LandmarkWM` and `SlidingWM`, respectively. For the `DampedWM`, we evaluate α and λ, which are the key parameters of its decaying weight. Evaluation results based on the *FCT* workload are shown in Figure 11 and there are two main observations.

O9: With the increase in window length, the clustering accuracy increases but the efficiency decreases. We can see from the results that there is a non-trivial trade-off between clustering accuracy and efficiency when tuning the landmark of `LandmarkWM` or the size of `SlidingWM`, which reaffirms our analysis in Section 4.1 and 4.2. Since these two parameters control the length of the corresponding window, they both show the common phenomenon that processing more data inside the window will bring in better accuracy but lower efficiency. As discussed in **O5**, a larger window can store more temporal information bringing higher accuracy. However, more data points are



(a) Accuracy (b) Efficiency

Fig. 12. The Behaviour Comparison of Different Outlier Detection Mechanisms on Four Real-world Workloads.

required for processing with increasing window length. Therefore, how to select the best window length depends on the optimization target.

O10: The clustering accuracy decreases the when the data point decays too fast. Figure 11d and Figure 11c show the changes of two key parameters of the DampedWM's decaying weight on clustering accuracy and efficiency. Obviously, we can see that the increase of λ and α (i.e., faster decay as discussed in Section 4.3), make clustering accuracy worse. As discussed in **O8**, if the data point decays too fast in the DampedWM, many of them will be regarded as outliers and get removed from the window. Although this may reduce the total workloads and finally improve the efficiency, which is also shown in the figure that there is a sharp increase of the throughput, it is still highly possible that many useful points are frequently discarded, which makes the clustering accuracy worse. As discussed in Section 4.3, this non-trivial trade-off is due to the fixed decaying configuration of DampedWM, which makes the window less capable to handle evolving data stream. Therefore, for a better clustering quality, the decaying weight needs to be dynamically tuned during the clustering procedure.

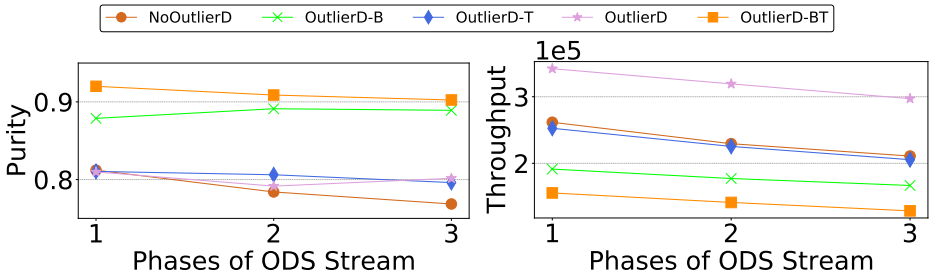
8.4 Outlier Detection Mechanism

We now evaluate different outlier detection mechanisms in detail. We first give a general comparison of different mechanism combinations, and then we discuss the impact of outlier evolution on their clustering behavior. For these experiments, we used the MCs as the summarizing data structure, the LandmarkWM window model, and do not apply Refine.

General Comparison: The results of the general comparisons of different types of outlier detection mechanisms are shown in Figure 12. There are three main observations as follows.

O11: Applying outlier detection mechanisms, especially with buffer improves clustering accuracy. Compared to NoOutlierD, applying outlier detection with or without a buffer and/or a timer can bring at least ~8% higher accuracy, even for workloads without outliers at the end of processing (i.e., *FCT*, *Insects*, and *Sensor*). This is because all workloads contain "temporal outliers" during the processing, while those outliers may transform into clusters upon finishing processing. We also note that both timer and buffer can bring higher efficiency than OutlierD and NoOutlierD. When they are utilized together, *DSC* algorithms achieve even higher accuracy compared to using either of them. Unfortunately, not all *DSC* algorithms utilize outlier detection mechanisms, including the recent proposal such as SL-KMeans.

O12: Counter-intuitively, applying an outlier detection mechanism may bring even higher efficiency. Surprisingly, we can see from Figure 12b that the throughput of OutlierD (w/o buffer, w/o timer) is even greater than NoOutlierD on *FCT* and *KDD99*, which indicates that applying outlier detection even improves the clustering efficiency of *DSC* algorithms. This is because many



(a) Accuracy (b) Efficiency

Fig. 13. The Behaviour Comparison of Different Outlier Detection Mechanism on *ODS* Workload.

sparse clusters are removed as a result of being detected as outliers, and thus the total number of online clusters is reduced, reducing total workloads. However, when utilizing buffers in outlier detection (i.e., `OutlierD-B` and `OutlierD-BT`), the throughput decreases a lot. Therefore, for ensuring a high clustering efficiency of the *DSC* algorithm, it is not recommended to apply a buffer for detecting outliers.

O13: The behaviour of outlier detection mechanism is highly influenced by the outlier evolution in the data stream. We find that the changing workload characteristic brings a significant impact on outlier detection mechanisms. For example, applying any types of mechanisms can all achieve at least ~100% better efficiency while keeping up to 0.97 clustering purity than `NoOutlierD` only on *KDD99* workload, which is the workload with high outlier evolution frequency.

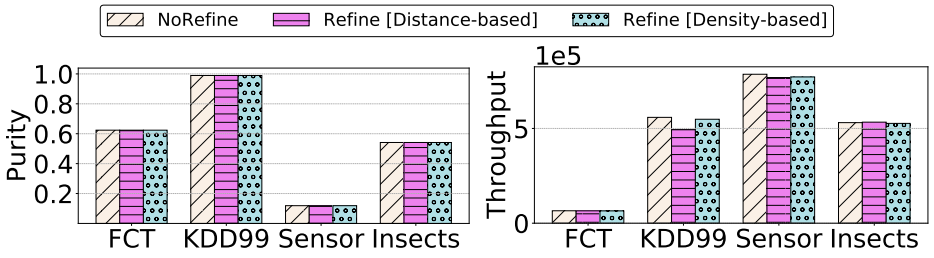
Outlier Evolution: We now evaluate the impact of the outlier evolution frequency on outlier detection mechanisms. Similar to the previous experiment, we use *ODS* workload for evaluation. According to the result shown in Figure 13, applying buffer for detection (i.e., `OutlierD-B` and `OutlierD-BT`) can greatly improve the clustering accuracy but also decrease the efficiency, which reaffirms the previous discussion in **O11** and **O12**. Besides, there is one additional observation:

O14: Utilizing a timer in the outlier detection mechanism can guarantee good clustering accuracy and efficiency under outlier evolution. From phases 1 to 5, `OutlierD-T` achieves higher accuracy than `OutlierD` while maintaining a comparable efficiency. As discussed in Section 5.3, utilizing timers in outlier detection helps to regularly delete the clusters which have not been used or updated for a certain amount of time. When outlier evolution becomes more and more frequent in the stream, many temporal clusters become outliers since they are not being updated. In this case, deleting these outdated clusters with the help of a timer can not only reduce workloads, but also reduces the possibility that some new data points are mis-grouped with those deleted outliers, increasing accuracy. Therefore, utilizing a timer in outlier detection mechanisms is recommended, especially with the increasing outlier evolution frequency in workloads.

8.5 Offline Refinement Strategy

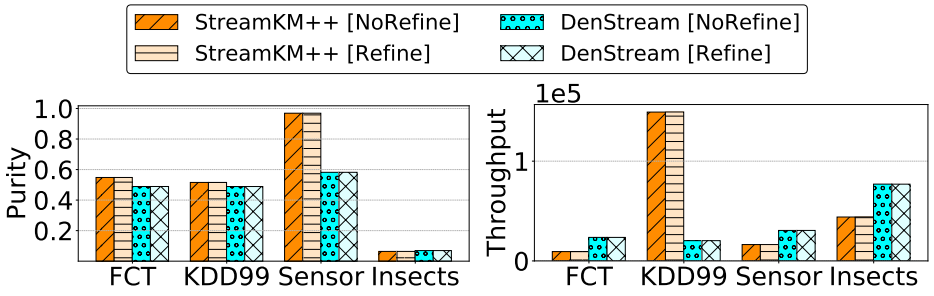
Lastly, we study the impact of the offline refinement strategy. We configured the algorithms to use `MCs` summarizing data structure, `LandmarkWM` window model and `OutlierD` outlier detection mechanism.

General Comparison: In this experiment, we use KMeans++ [26] and DBSCAN [16] to implement distance-based and density-based offline refinement strategies (i.e., `Refine`), respectively. Results are shown in Figure 14. There is one main observation as follows:



(a) Accuracy (b) Efficiency

Fig. 14. The Behaviour Comparison of Different Offline Refinement Strategies on Four Real-world Workloads.



(a) Accuracy (b) Efficiency

Fig. 15. The Behaviour Comparison when Turning On or Off the Offline Refinement Strategy of Two Existing Algorithms.

O15: In most cases, applying offline refinement is unnecessary as it does not bring any improvement to the clustering results. As shown in Figure 14, both distance-based and density-based refinement strategies do not bring accuracy or performance improvements. In fact, when being applied on *KDD99* and *Insects* workloads, they even bring down the clustering accuracy. This is surprising since many existing *DSC* algorithms [4, 5, 19, 36] have applied either distance-based or density-based refinement strategies into their clustering procedure as discussed in Section 6.

To get a better understanding of the phenomenon, we select two existing algorithms, StreamKM++ [4] and DenStream [36], covering the two types of `Refine` strategies. For comparison, we disabled the `Refine` in these two algorithms and test the changes in their clustering behaviour. In Figure 15, we found that both algorithms' accuracy and efficiency are not influenced by their offline refinement strategy for all workloads. This result contradicts the common wisdom [5] and encourages the researchers to design *DSC* algorithms without `Refine`.

8.6 Overall Algorithm Comparison

To verify the previous observations on four design aspects, we performed the last group of experiments for overall comparison among *DSC* algorithms. In addition to the eight *DSC* algorithms listed in Table 3, we propose a novel *DSC* algorithm called *Benne*, derived by composing suitable design choices from four design aspects.

8.6.1 How Benne Works? The high-level execution flow of *Benne* is presented in Algorithm 1. The overall execution follows a two-phase execution strategy. In the online phase, a *Window Fun.* is

Algorithm 1: Execution flow of *Benne*.

```
   Data: p // Input point
   Data: s // Summarizing data structure
   Input: struc. // Selected type of summarizing data structure
   Input: win. // Selected type of window model
   Input: out. // Selected type of outlier detection mechanism
   Input: ref. // Selected type of refinement strategy
   // Online Phase
1  while !stop processing of input streams do
2      Window Fun. (...);
3      if out. != NoOutlierD then
4          b ← Outlier Fun. (...);
5          if b = false then
6              Insert Fun. (...)// Insert p to s and update s
7      else
8          Insert Fun. (...)// Insert p to s and update s
   // Offline Phase
9  if ref. != NoRefine then
10     Refine Fun. (ref.);
```

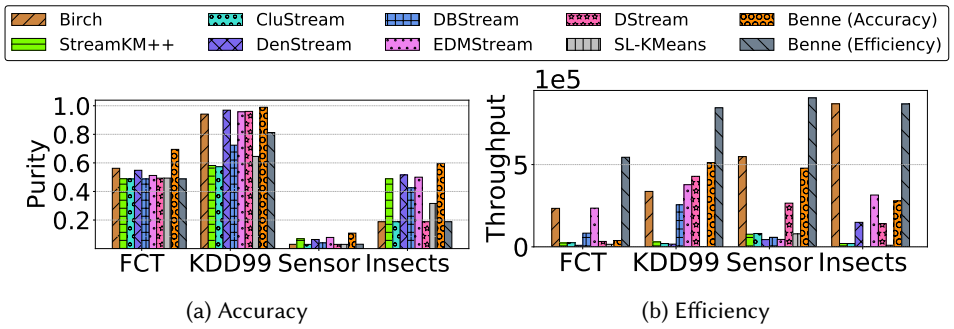
Algorithm 2: Window Fun. of *Benne*.

```
   /* c: counter (initialized with 0), m: landmark, ws: sliding window size, dc = (α, λ): decay parameters */
1  Function Window Fun.(s, m, ws, dc):
2      c++;
3      if win. = LandmarkWM then
4          if c > m then
5              Sink the clustering results from s;
6              Clear all of the clustering information;
7              m ← m_next; /* update current landmark                                    */
8      else
9          if win. = DampedWM then
10             Update weight with dc;
11         else // win. = SlidingWM
12             if c > ws then
13                 Remove the earliest point from window;
```

first applied to collect the most recent data for clustering (Line 2). The *Window Fun.* (Algorithm 2) invokes different computing logic depending on the selected type of window model as described in Section 4. After that, the algorithm runs the outlier detection mechanism, to detect if the current input point (p) is an outlier. Specifically, if outlier detection is enabled (Line 3), then the *Outlier Fun.* (Algorithm 3) is invoked (Line 4). The *Outlier Fun.* executes different logics depending on the selected outlier detection mechanisms (i.e., w/ or w/o buffer and timer) as discussed in Section 5. The input point will then be added to the summarizing data structure if it is not identified as an outlier (Line 6) or the algorithm does not use any outlier detection mechanisms (Line 8). We omit the corresponding pseudocode of the *Insert Fun.* as it similarly follows our discussion in Section 3. When there is no more input data to process, the algorithm enters the offline phase. An offline refinement of the clustering results can be applied (Line 8) as discussed in Section 6. For simplicity, we also omit the corresponding pseudocode of the *Refine Fun.*. Being a generic algorithm, *Benne* can be easily reconfigured into different variants by setting different types of $struc.$, $win.$, $out.$, and $ref.$, respectively. To achieve the highest accuracy, *Benne (Accuracy)* is made up of `MCs` summarizing data structure, `LandmarkWM`, `OutlierD-BT`, and `NoRefine`. In contrast, to achieve the highest efficiency,

Algorithm 3: Outlier Fun. of *Benne*.

```
   /* Buffer: outlier buffers, t: timer threshold, d: density threshold */
1  Function Outlier Fun.(p, s, Buffer, t, d):
2      if out. = (OutlierD-B Or OutlierD-BT) then
           /* use buffer optimization */
3          if (p Is Outlier) = true then
4              Select cluster cl closest to p from Buffer;
5              Insert p to cl and update cl;
6              bool dense ← Check (cl);
               /* check if cl is dense enough */
7              if dense = true then
8                  Move cl from Buffer to s;
9      if REGULAR CHECK TRIGGERED then
10         cls ← Extract(s);/* extract all clusters from s */
11         for cl ∈ cls do
12             bool active = false;
13             bool dense ← Check (cl, d);
               /* check if cl is dense enough */
14             if out. = (OutlierD-T Or OutlierD-BT) then
                   /* use timer optimization */
15                 bool active ← Check(cl.timer, t);
                   /* check if cl is active enough */
16             if dense = false and active = false then
17                 if BUFFER ENABLED then
18                     Insert cl to Buffer;
19                 Remove cl from s;
20         if OutlierD-BT then
21             for cl ∈ Buffer do
22                 bool active ← Check(cl.timer, t);
23                 if active = false then
24                     Remove cl from Buffer;
25     Return (p Is Outlier);
```



(a) Accuracy (b) Efficiency

Fig. 16. The Behaviour Comparison of Eight Existing *DSC* Algorithms on Four Real-world Workloads.

Benne (Efficiency) is made up of Grids summarizing data structure, LandmarkWM, OutlierD, and NoRefine.

8.6.2 Comparison Results. We now compare the selected eight algorithms and *Benne* for handling four real-world workloads. Results are shown in Figure 16. We obtain a list of observations, mostly reaffirming our previous analysis. From the summarizing data structure aspect, we can see that both

SL-KMeans and StreamKM++ achieve low efficiency and relative great accuracy while DStream has the highest one but low accuracy on all workloads efficiency. This reaffirms our previous observations **O1** since SL-KMeans applies the AMS, StreamKM++ applies CoreT and DStream applies Grids as their data structure respectively. Besides, BIRCH and EDMStream achieve relatively higher efficiency than others on *FCT*, *KDD99* and *Insects*, which matches **O2** for their summarizing data structures are hierarchical-based. Finally, BIRCH achieves much higher efficiency than other algorithms on *Sensor* workload with high cluster evolution frequency. This also partially reaffirms our previous analysis on **O3-4**.

From the window model aspect, we can see that algorithms with a DampedWM, including DenStream, DBStream, DStream and EDMStream all achieve poor clustering efficiency on *Sensor* workload, which reaffirms our previous discussion on **O6-7**. Moreover, the purity of BIRCH, which uses LandmarkWM is high on the *KDD99* workload, which matches our analysis discussed in **O8**.

From the outlier detection mechanism aspect, we can see that the accuracy of DenStream, which utilizes OutlierD-B, is higher than many other algorithms such as DBStream and DStream without using a buffer. This partially reaffirms the previous discussion of **O11**. However, its throughput is lower than the average, which is also caused by the overhead of maintaining the buffer as discussed in **O12**. Additionally, DBStream, EDMStream and DStream achieves high clustering efficiency and relative great accuracy on *KDD99* with high outlier evolution frequency, which partially reaffirms **O13-14** since they apply timers for detecting outliers.

Finally, from the offline refinement strategy aspect, we can see that there is no clear difference between those without Refine strategies such as BIRCH, EDMStream and SL-KMeans and those with Refine strategies, such as StreamKM++ and DenStream. This matches **O15** that it is unnecessary to use offline refinement for the clustering of data streams.

Apart from reaffirming our previous analysis, we make three additional observations as follows.

O16: The clustering behavior of *DSC* algorithms is affected by workload dimensions. Despite the significant differences among *DSC* algorithms, they all achieve lower throughput for higher workload dimensionalities. We observe that the complexity of many operations is highly related to workload dimension, such as updating the summarizing data structure and searching for the suitable cluster for data insertion based on distance. Therefore, with the increase of workload dimension, more time is spent on these operations bringing down the clustering efficiency. Higher dimensional workloads are expected to be harder to cluster correctly due to the well-known *Curse of Dimensionality* issue. However, we observe that algorithms achieve higher purity for handling high dimensional workloads (*FCT* and *KDD99*). We suspect that dimensionality is not a dominant factor for the clustering accuracy here. Nevertheless, we plan to evaluate the impact of dimensionality with more extensive datasets in future.

O17: The selection of summarizing data structure has a large impact on the clustering behaviour of window models. First, as discussed in **O5**, the SlidingWM guarantees a better clustering efficiency. However, although SL-KMeans utilizes a SlidingWM, its efficiency is still poor due to the usage of the AMS data structure. Second, as discussed in **O7**, the DampedWM achieves poorer efficiency with the increase of cluster evolution frequency. However, the efficiency of EDMStream, which also uses the DampedWM, is higher than others on handling *Sensor* workload. We observe that it is because of the usage of DPT, which guarantees a high clustering efficiency as discussed in **O4**. When selecting the proper window model, we should also take the selection of summarizing data structure into consideration.

O18: By selecting suitable design choices, we are able to compose a novel algorithm called *Benne*, which can either achieves the highest accuracy or highest efficiency compared to the state-of-the-art on all real-world workloads. As shown in Figure 16, *Benne (Accuracy)* and *Benne (Efficiency)* achieve at most ~50% and ~130% higher accuracy and throughput, respectively,

than any of the existing *DSC* algorithms on all real-world workloads. Nevertheless, we are not able to compose an algorithm that achieves the highest accuracy and efficiency at the same time, reaffirming our analysis that existing design options are trading off between accuracy and efficiency. Moreover, there is still a large room for further improvement as none of the *DSC* algorithms including *Benne* is able to achieve good accuracy results for handling the *Sensor* workload. We envision novel *DSC* algorithms to be designed in future, to better handle workloads with high cluster evolution frequency such as *Sensor*.

9 RELATED WORK

In the following, we first discuss the previous experimental studies on data stream clustering, then we outline related research efforts divided into the four key design aspects of *DSC* algorithms.

Previous Experimental Study. To the best of our knowledge, there are only two existing empirical studies on *DSC* algorithms [12, 27]. The first is proposed in 2017 [12]. In this work, the researchers implement ten popular *DSC* algorithms in `R` with interfaces to `C++`, and evaluate their clustering accuracy on workloads with varying cluster shapes. It lacks the efficiency comparison, which is at least equally important as the clustering accuracy. The second study [27] discussed the impact of some *DSC* design aspects in their evaluation. However, their experiment is based on a coarse-grained algorithm comparison rather than studying the impact of each design aspect in depth. Both prior studies are not sufficient to comprehend the behaviours of *DSC* algorithms under varying characteristics of data stream workload and provide fine-grained insights into the four algorithm design aspects.

Summarizing Data Structure. Proposing better data structures for summarizing data streams has received much attention in the literature. Zhang et al. [38] proposed Clustering Feature Tree (`CFT`) with incrementality and additivity properties well suited for streaming workloads. Aggarwal et al. [5] propose to extend CF structure and call it microclusters (`MCs`) with additional summary information about timestamps and respective weight, which are used to cooperate with the window models. Differs significantly from `CFT`, Ackermann et al. [4] proposed the Coreset Tree (`CoreT`), which utilizes a *merge-and-reduce* algorithm over the binary tree, in order to reduce leaf objects in the tree. Chen et al. [13] proposed to use of a grid data structure, rather than a tree-like structure for efficiency concerns. In contrast, Gong et al. [18] proposed Dependency Tree (`DPT`) that can deliver satisfying efficiency and high accuracy at the same time. Nevertheless, we observe that `DPT` leads to sub-optimal results in accurately handling cluster evolution. Most recently, Borassi et al. [9] proposed Augmented Meyerson Sketch (`AMS`) for stream clustering claiming it performs empirically better than analytic bounds. However, we found that `AMS` is not a good choice for the clustering of data streams, due to its unsatisfying accuracy and low efficiency compared with other data structures.

Window Model. Various window models are proposed in *DSC* algorithms to determine the subset of (potentially infinite) data streams to be involved in processing. Metwally et al. [30] proposed the landmark window model (`LandmarkWM`) that defines data arriving after the landmark is kept. Considering the most recent records to be more critical, Zhou et al. [39] and Borassi et al. [9] propose to store only the most recent data in the stream, which leads to the sliding window model (`SlidingWM`). Cao et al. [11] and Chen et al. [13] proposed the damped window model (`DampedWM`), which keeps all the data while still maintaining interest in the most recent information, by associating each object with different weights. Different from the aforementioned two window models, `DampedWM` better captures the dynamic changing of clusters, ensuring a higher quality in clustering. To the best of our knowledge, there is still a lack of an in-depth study that compares the efficiency and/or accuracy impacts of different window models using a comprehensive benchmark until our work.

Outlier Detection Mechanism. Outlier detection aims to identify objects that deviate from others, it is one crucial design aspect in many *DSC* algorithms. As early as BIRCH [37] has one optional phase of scanning its data storage and looking for outlier candidates whose object density is lower than the threshold, and it periodically checks whether the entries in outlier candidates can be absorbed back into the real clusters. Aggarwal et al. [5] introduce *outlier timer* to better identify outliers (`OutlierD-T`). Wan et al. [36] further introduce the concept of *outlier buffer* (`OutlierD-B`) to ease the transformation between outliers and clustered points. We observed that the timer mechanism can also be used along with the buffer to identify and discard outlier candidates which have not been updated for a long time. However, unlike ours, there is no prior study to comprehensively evaluate the usage of outlier timer, outlier buffer, and their combinations.

Offline Refinement Strategy. Since the first mention by Aggarwal et al. [5] in their proposed online-offline scheme for stream clustering, most subsequent *DSC* algorithms require offline refinement strategies (`Refine`) to get better final results. Frequently used clustering algorithms for offline refinement include KMeans [26] and its many variants such as Scalable k-means [10] and Singlepass k-means [17]. In contrast to common belief, our evaluation results show that adopting offline refinement is often unnecessary and only brings overheads.

10 CONCLUSION

In this paper, we present results from an extensive experimental analysis of different design choices of *DSC* algorithms. With a more realistic benchmark configuration and a detailed in-depth analysis, we are able to identify many important insights that have not been revealed in prior studies. Our findings even promote a novel *DSC* algorithm *Benne*, which can be configured by making a flexible decision at each design aspect to achieve either the highest accuracy or highest efficiency compared to the state-of-the-art *DSC* algorithms on all real-world workloads. Given the encouraging results, we envision that our work will motivate future studies in terms of optimizing or developing new *DSC* algorithms.

ACKNOWLEDGMENTS

We thank Prof. Dongxiang Zhang, Xianzhi Zeng, Haolan He, Zhonghao Yang, Yuhao Wu for joining the discussions, and the anonymous SIGMOD reviewers for their constructive feedback. This work is supported by the National Research Foundation, Singapore and Infocomm Media Development Authority under its Future Communications Research & Development Programme FCP-SUTD-RG-2022-006, a MoE AcRF Tier 2 grant (MOE-T2EP20122-0010), and a SUTD-ZJU IDEA Grant for Visiting Professor (SUTD-ZJU (VP) 202101). Xuanhua Shi's work is supported in part by the National Key R&D Program of China under Grant 2020AAA0108501, and the Key R&D Program of Hubei under Grant 2020BAA020.

REFERENCES

[1] [n.d.]. *Covertype. http://archive.ics.uci.edu/ml/datasets/Covertype.*
[2] [n.d.]. *Sensor. https://www.cse.fau.edu/~xqzhu/stream.html.*
[3] [n.d.]. *Ticat, https://github.com/innerr/ticat.*
[4] Marcel R. Ackermann and et al. 2012. StreamKM++: A Clustering Algorithm for Data Streams. *ACM J. Exp. Algorithmics* 17 (May 2012), 30.
[5] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. 2003. A Framework for Clustering Evolving Data Streams. In *Proceedings of the 29th International Conference on Very Large Data Bases - Volume 29* (Berlin, Germany) *(VLDB '03)*. VLDB Endowment, 81–92.
[6] Daniel Barbará. 2002. Requirements for Clustering Data Streams. *ACM SIGKDD Explorations Newsletter* 3, 2 (Jan. 2002), 23–27. https://doi.org/10.1145/507515.507519
[7] Alessio Bechini and et al. 2020. TSF-DBSCAN: a Novel Fuzzy Density-based Approach for Clustering Unbounded Data Streams. *IEEE Transactions on Fuzzy Systems* (2020).

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

[9] Michele Borassi, Alessandro Epasto, Silvio Lattanzi, Sergei Vassilviskii, and Morteza Zadimoghaddam. 2020. Sliding Window Algorithms for K-Clustering Problems. In *Proceedings of the 34th International Conference on Neural Information Processing Systems* (Vancouver, BC, Canada) *(NIPS'20)*. Curran Associates Inc., Red Hook, NY, USA, Article 731, 12 pages.

[10] Paul Bradley, Johannes Gehrke, Raghu Ramakrishnan, and Ramakrishnan Srikant. 2002. Scaling mining algorithms to large databases. *Commun. ACM* 45, 8 (2002), 38–43.

[11] Feng Cao, Martin Estert, Weining Qian, and Aoying Zhou. 2006. Density-based clustering over an evolving data stream with noise. In *Proceedings of the 2006 SIAM international conference on data mining*. SIAM, 328–339.

[12] Matthias Carnein, Dennis Assenmacher, and Heike Trautmann. 2017. An Empirical Comparison of Stream Clustering Algorithms. *In Proceedings of the Computing Frontiers Conference.(CF' 17)* 17 (2017), 361–366.

[13] Yixin Chen and Li Tu. 2007. Density-Based Clustering for Real-Time Stream Data. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD '07)*. Association for Computing Machinery, New York, NY, USA, 133–142.

[14] Gianmarco De Francisci Morales and Albert Bifet. 2015. SAMOA: Scalable Advanced Massive Online Analysis. *J. Mach. Learn. Res.* 16, 1 (Jan. 2015), 149–153.

[15] M. Deepa, P. Revathy, and P. G. Student. 2012. Validation of Document Clustering based on Purity and Entropy measures.

[16] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. AAAI Press, 226–231.

[17] Fredrik Farnstrom, James Lewis, and Charles Elkan. 2000. Scalability for clustering algorithms revisited. *ACM SIGKDD Explorations Newsletter* 2, 1 (2000), 51–57.

[18] Shufeng Gong, Yanfeng Zhang, and Ge Yu. 2018. Clustering Stream Data by Exploring the Evolution of Density Mountain. *Proc. VLDB Endow.* 11, 4 (oct 2018), 393–405. https://doi.org/10.1145/3164135.3164136

[19] Michael Hahsler and Matthew Bolaños. 2016. Clustering Data Streams Based on Shared Density between Micro-Clusters. *IEEE Transactions on Knowledge and Data Engineering* 28, 6 (2016), 1449–1461. https://doi.org/10.1109/TKDE.2016.2522412

[20] Michael Hahsler, Matthew Bolanos, and John Forrest. 2015. streamMOA: Interface for MOA Stream Clustering Algorithms. https://cran.r-project.org/web/packages/streamMOA/

[21] Ahsanul Haque, Latifur Khan, Michael Baron, Bhavani Thuraisingham, and Charu Aggarwal. 2016. Efficient handling of concept drift and concept evolution over Stream Data. In *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*. 481–492. https://doi.org/10.1109/ICDE.2016.7498264

[22] Philipp Kranen, Ira Assent, Corinna Baldauf, and Thomas Seidl. 2011. The ClusTree: indexing micro-clusters for anytime stream mining. *Knowledge and Information Systems* 29, 2 (2011), 249–272.

[23] Hardy Kremer and et al. 2011. An Effective Evaluation Measure for Clustering on Evolving Data Streams. In *Proceedings of the 11th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/2020408.2020555

[24] S. Lloyd. 1982. Least squares quantization in PCM. *IEEE Transactions on Information Theory* 28, 2 (1982), 129–137. https://doi.org/10.1109/TIT.1982.1056489

[25] Jie Lu and et al. 2019. Learning under Concept Drift: A Review. *IEEE Transactions on Knowledge and Data Engineering* 31, 12 (2019), 2346–2363. https://doi.org/10.1109/TKDE.2018.2876857

[26] J. Macqueen. 1967. Some methods for classification and analysis of multivariate observations. In *In 5-th Berkeley Symposium on Mathematical Statistics and Probability*. 281–297.

[27] Stratos Mansalis, Eirini Ntoutsi, Nikos Pelekis, and Yannis Theodoridis. 2018. An evaluation of data stream clustering algorithms. *Statistical Analysis and Data Mining: The ASA Data Science Journal* 11 (06 2018). https://doi.org/10.1002/sam.11380

[28] Mohammad Masud, Jing Gao, Latifur Khan, Jiawei Han, and Bhavani M. Thuraisingham. 2011. Classification and Novel Class Detection in Concept-Drifting Data Streams under Time Constraints. *IEEE Transactions on Knowledge and Data Engineering* 23, 6 (2011), 859–874. https://doi.org/10.1109/TKDE.2010.61

[29] Mohammad M. Masud, Qing Chen, Latifur Khan, Charu Aggarwal, Jing Gao, Jiawei Han, and Bhavani Thuraisingham. 2010. Addressing Concept-Evolution in Concept-Drifting Data Streams. In *2010 IEEE International Conference on Data Mining*. 929–934. https://doi.org/10.1109/ICDM.2010.160

[30] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2005. Duplicate detection in click streams. In *Proceedings of the 14th international conference on World Wide Web*. 12–21.

[31] A. Meyerson. 2001. Online facility location. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*. 426–431. https://doi.org/10.1109/SFCS.2001.959917

[32] K. Namitha and G. Santhosh Kumar. 2020. Concept Drift Detection in Data Stream Clustering and its Application on Weather Data. *Int. J. Agric. Environ. Inf. Syst.* 11, 1 (2020), 67–85. https://doi.org/10.4018/IJAEIS.2020010104

[33] Jonathan A. Silva, Elaine R. Faria, Rodrigo C. Barros, Eduardo R. Hruschka, André C. P. L. F. de Carvalho, and João Gama. 2013. Data Stream Clustering: A Survey. *ACM Comput. Surv.* 46, 1, Article 13 (jul 2013), 31 pages. https://doi.org/10.1145/2522968.2522981

[34] V. M. A. Souza, D. M. Reis, A. G. Maletzke, and G. E. A. P. A. Batista. 2020. Challenges in Benchmarking Stream Learning Algorithms with Real-world Data. *Data Mining and Knowledge Discovery* 34 (2020), 1805–1858. https://doi.org/10.1007/s10618-020-00698-5

[35] Mahbod Tavallaee, Ebrahim Bagheri, Wei Lu, and Ali A. Ghorbani. 2009. A detailed analysis of the KDD CUP 99 data set. In *IEEE Symposium on Computational Intelligence for Security and Defense Applications*. 1–6.

[36] Li Wan, Wee Keong Ng, Xuan Hong Dang, Philip S. Yu, and Kuan Zhang. 2009. Density-Based Clustering of Data Streams at Multiple Resolutions. *ACM Trans. Knowl. Discov. Data* 3, 3 (July 2009), 28.

[37] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1996. BIRCH: An Efficient Data Clustering Method for Very Large Databases. *SIGMOD Rec.* 25, 2 (jun 1996), 103–114. https://doi.org/10.1145/235968.233324

[38] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. 1997. BIRCH: A new data clustering algorithm and its applications. *Data mining and knowledge discovery* 1, 2 (1997), 141–182.

[39] Aoying Zhou, Feng Cao, Weining Qian, and Cheqing Jin. 2008. Tracking clusters in evolving data streams over sliding windows. *Knowledge and Information Systems* 15, 2 (2008), 181–214.

Received October 2022; revised January 2023; accepted February 2023