DistStream: An Order-Aware Distributed Framework for Online-Offline Stream Clustering Algorithms

Lijie Xu†, Xingtong Ye†, Kai Kang†, Tian Guo‡, Wensheng Dou†, Wei Wang†, Jun Wei†
†Institute of Software, Chinese Academy of Sciences
‡Worcester Polytechnic Institute

***Abstract*—Stream clustering is an important data mining technique to capture the evolving patterns in real-time data streams. Today's data streams, e.g., IoT events and Web clicks, are usually high-speed and contain dynamically-changing patterns. Existing stream clustering algorithms usually follow an online-offline paradigm with a *one-record-at-a-time* update model, which was designed for running in a single machine. These stream clustering algorithms, with this sequential update model, can not be efficiently parallelized and fail to deliver the required high throughput for stream clustering.**

In this paper, we present *DistStream*, a distributed framework that can effectively scale out online-offline stream clustering algorithms. To parallelize these algorithms for high throughput, we develop a mini-batch update model with several efficient parallelization approaches. To maintain high clustering quality, *DistStream*'s mini-batch update model preserves the update order in all the steps during parallel execution, which can reflect the recent changes for dynamically-changing streaming data. We implement *DistStream* atop Spark Streaming, as well as four representative stream clustering algorithms based on *DistStream*. Our evaluation on three real-world datasets shows that *DistStream*-based stream clustering algorithms can achieve sublinear throughput gain and comparable (99%) clustering quality with their single-machine counterparts.

I. Introduction

Today, we are increasingly leveraging data mining techniques to extract insights from real-time data streams created by IoT sensors, Web clicks, etc. *Stream clustering* is a key technique to capture the evolving patterns, e.g., network intrusion patterns and online shopping behavior patterns, through grouping similar records into clusters continuously.

Different from batch-mode clustering, stream clustering needs to process *unbounded* data streams in one pass with limited memory capacity. Under these constraints, most stream clustering algorithms adopt a two-phase *online-offline* paradigm [11, 19], which keeps an up-to-date sketch of the streaming data in memory during the online phase and computes the final clusters offline when necessary. By leveraging this online-offline paradigm, researchers have proposed more than 20 stream clustering algorithms including *CluStream* [11], *DenStream* [16], *D-Stream* [17], and *ClusTree* [24].

These stream clustering algorithms leverage a *one-record-at-a-time* update model that has strict sequential update constraint, i.e., a new record cannot be clustered until its previous record has been processed. Thus, these algorithms are suitable to work in a single machine, leading to low clustering throughput. For example, *CluStream*'s throughput only achieves ∼5K records per second on TCP connection stream in a single machine [11]. However, today's data streams are orders of magnitude faster. For instance, the online shopping site Alibaba needs to process 256K transactions per second [1], and a leading Web company was hit by a DDoS attack at the peak of 292K requests per second [5]. Further, these data streams often have dynamically-changing patterns, e.g., changing customer shopping behaviors or evolving attack patterns.

Currently, there is not an effective approach for online-offline stream clustering algorithms to achieve high throughput and high clustering quality on dynamically-changing data streams. Existing batch-mode distributed machine learning frameworks, such as Spark MLlib [27] and Petuum [30], are inadequate to support the unique characteristics of stream clustering algorithms. Recently, researchers proposed several approaches to implement stream clustering algorithms on distributed stream processing systems [13, 23]. However, these approaches are designed for a specific algorithm (i.e., *CluStream*) and cannot be extended to other algorithms like *DenStream* [16] and *D-Stream* [17]. In addition, these approaches do not distinguish the data arrival orders and cannot precisely reflect the latest changes for dynamically-changing patterns, leading to degraded clustering quality.

In this paper, we propose a distributed framework *DistStream* to parallelize the online-offline stream clustering algorithms, with the goal of achieving both high throughput and good clustering quality. To achieve this, we mainly address two key challenges.

The *first* challenge is how to design an efficient update model for stream clustering algorithms to achieve high throughput. We design a *mini-batch update model* to overcome the throughput inefficiency of one-record-at-a-time model, by introducing a new batch-by-batch feedback loop and multiple parallelization dimensions for the update steps. Through analyzing the latency and network communication, we judiciously choose the most efficient dimension for parallelizing each step of our mini-batch update model.

The *second* challenge is how to achieve good clustering quality, comparable to that of one-record-at-a-time model. Directly parallelizing the mini-batch model without distinguishing the data arrival orders will lead to low clustering quality, because it fails to reflect the pattern changes in data streams. To mitigate this problem, we design an *order-aware update mechanism* in all update steps during parallel execution. Further, we theoretically demonstrate that our order-aware update mechanism can reflect the recent changes (over

unordered update mechanism) and mitigates the quality loss associated with the mini-batch update model.

DistStream is implemented on top of Spark Streaming [31] and exposes four high-level APIs for algorithm developers to migrate and parallelize single-machine stream clustering algorithms. We also implement four representative stream clustering algorithms, i.e., *CluStream*, *DenStream*, *D-Stream*, and *ClusTree*, atop *DistStream* for evaluation. The experiments on three real-world datasets with different data distributions show that our *DistStream*-based implementations achieve comparable clustering quality of 99% and sublinear throughput gain of up to 13X (with 32 cores) to their single-machine counterparts. Our experiments also demonstrate the importance of our order-aware update mechanism—unordered mini-batch implementations suffer from about 60% lower clustering quality. In summary, we make the following key contributions.

- We design a mini-batch update model with efficient parallelization approaches for stream clustering algorithms to overcome the scalability inefficiency of the traditional *one-record-at-a-time* model.
- In our mini-batch update model, we design an order-aware update mechanism and theoretically demonstrate its importance in maintaining the clustering quality for the dynamically-changing data streams.
- We implement our *DistStream* framework atop widely-used Spark Streaming, as well as four representative algorithms based on *DistStream*. The experiments on real-world datasets show that *DistStream*-based implementations can achieve sublinear throughput gain and comparable clustering quality with their single-machine counterparts.

II. Background

This section describes the online-offline paradigm and one-record-at-a-time update model of stream clustering algorithms.

A. Online-offline paradigm

The objective of a clustering algorithm is to group a set of d-dimensional data records $\{x_1, x_2, \ldots, x_m\}$ into a number of clusters $C = \{c_1, c_2, \ldots, c_n\}$, in which intra-cluster records have high similarity and inter-cluster records have low similarity. The similarity among records is typically measured by distance functions such as Euclidean distance.

Different from batch-mode clustering algorithms [20, 27], stream clustering algorithms process an *unbounded* sequence of ordered data records that arrive in real time. Each data record x_i is associated with a timestamp t_i, and stream clustering algorithms incrementally update the clusters with the arrival of new records. The clusters C at time t represent the clustering results of the records arriving before t, where the recent records are assumed to be more important to the clustering results than the older records. The *clustering quality* is measured by the compactness of each cluster and how well recent records have been grouped to the correct clusters.

To satisfy the latency and throughput requirements, stream clustering algorithms are designed to process the streaming records in one pass with limited memory capacity, and update

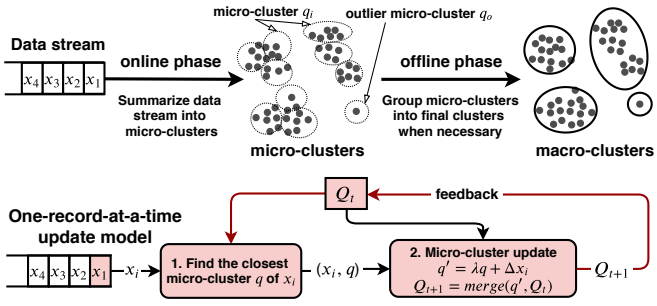


Fig. 1. The online-offline paradigm with one-record-at-a-time update model.

the clustering results as fast as possible. To fulfill these requirements, most of stream clustering algorithms [19], such as *CluStream* [11] and *DenStream* [16], adopt a two-phase *online-offline* clustering paradigm, as shown in Figure 1. The key idea is to keep an up-to-date sketch (i.e., micro-clusters) of the streaming data in memory at *online* phase and compute the final clusters (i.e., macro-clusters) *offline* when necessary.

Take the network intrusion pattern detection as an example. The incoming TCP connection streams contain normal connections and multiple types of attack connections with varying distributions. To identify the evolving patterns, e.g., which attack types are emerging, dominating and disappearing, CluStream [11] summarizes the connections into a number of in-memory micro-clusters and constantly updates these micro-clusters. This online update procedure may create new micro-clusters for new types of attacks, merge micro-clusters together for emerging attacks, and decay/delete micro-clusters for vanishing attacks. To identify recent intrusion patterns and adjust defense strategies accordingly, security analyst can invoke offline clustering to obtain recent clustering results (i.e., macro-clusters). In this case, the clustering quality can be measured by the number of correctly clustered TCP connections.

Formally, each micro-cluster is a statistical structure denoted as $q_i = \{S_i, T_i, N_i\}$, where S_i represents the spatial locality, i.e., the compactness of the records in q_i. For example in CluStream, S_i is measured by two d-dimensional feature vectors as $(\overline{CF^2}, \overline{CF^1})$, which represent the *squared sum* and *linear sum* of the records in the micro-cluster q_i, respectively. The temporal locality T_i measures the temporal weight of the records in the micro-cluster by favoring newer records. Micro-clusters with low temporal locality will be deleted during update. N_i is the number of records in q_i. DenStream regards the micro-clusters with high temporal localities as density-connected micro-clusters, and groups them together to find arbitrary shapes of clusters. D-Stream partitions the feature space into grids (i.e., micro-clusters) and groups the adjacent grids with high T_i and large N_i as macro-clusters. ClusTree organizes micro-clusters as a tree structure for better data summarization and fast record insertion.

B. One-record-at-a-time update model

For the online phase as shown in Figure 1, current streaming algorithms use a *one-record-at-a-time* update model to process

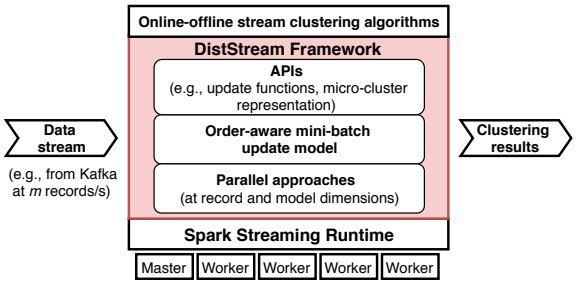


Fig. 2. The architecture of *DistStream*.

each incoming data record sequentially. For initialization, stream clustering algorithms leverage batch-mode clustering algorithms like K-means to generate n micro-clusters on user-defined first m data records. Then, for each incoming record x_i, the algorithms decide whether to assign it to an existing micro-cluster q based on spatial locality, i.e., the distance between record x_i and the centroid of each micro-cluster. If x_i falls within the maximum boundary (e.g., the radius) of q, the algorithms update both the spatial and temporal locality of q as $q' = \lambda_i q + \Delta x_i$, where Δx_i is the spatial and temporal increment of record x_i such as $\Delta x_i = (x_i^2, t_i^2)$, and λ_i denotes a time decaying factor that controls the importance of newer record over old record as $\beta^{-\Delta t_i}$ (e.g., $\beta = 1.2$). Otherwise, the algorithms create a new micro-cluster for record x_i. Before processing the next record, the algorithms merge the newly created (or updated) micro-cluster with other micro-clusters to form a new micro-cluster set Q_{t+1}. This one-record-at-a-time model can adapt to the dynamically-changing data stream but cannot be efficiently parallelized due to its one-by-one feedback loop, which is detailed in Section IV-A. For the offline phase, the final clustering results can be generated directly from the micro-clusters [19] using batch-mode algorithms such as K-means and DBSCAN [20].

III. DistStream overview

In this paper, we investigate the problem of how to design a distributed framework to efficiently parallelize online-offline stream clustering algorithms. Note that we focus on parallelizing the *online phase* to achieve high throughput while maintaining good clustering quality. We omit the discussion of *offline phase* because it does not impose real-time requirement and can be efficiently parallelized using existing batch-mode implementations such as distributed K-means [27].

The architecture of *DistStream* is shown in Figure 2. The core of *DistStream* is an order-aware mini-batch update model (Section IV), which generates computation steps for the implemented stream clustering algorithms. *DistStream* parallelizes these computation steps using efficient parallelization approaches (Section V), and launches Spark Streaming tasks to perform the computation (Section VI). *DistStream* provides APIs for algorithm developers to easily parallelize online-offline stream clustering algorithms.

IV. Order-aware mini-batch update model

In this section, we present the key design principles of our *order-aware mini-batch update model*. To improve throughput, we leverage a relaxed *batch-by-batch* feedback loop, instead of the original *one-by-one* feedback loop. We also decouple the update step into local and global update steps, and parallelize the local update step to improve update throughput. To achieve good clustering quality, we design an order-aware update mechanism, i.e., preserving the update order of records and micro-clusters in both update steps. We theoretically demonstrate that order-aware update mechanism can reflect the recent changes (over unordered update mechanism) and mitigate the quality loss associated with the mini-batch update model.

A. Batch-by-batch feedback loop

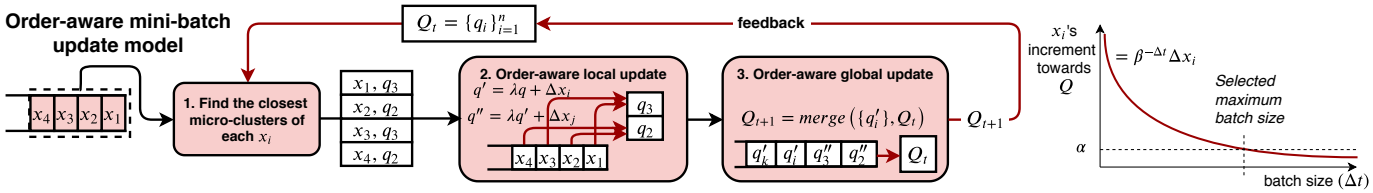
As shown in Figure 1, existing stream clustering algorithms use a one-record-at-a-time model with two sequential update steps. When a new record x_i arrives, the first step is to identify one micro-cluster q that is closest to record x_i. In the second step, the closest micro-cluster q is updated and then merged with all existing micro-clusters Q_t through operations such as merging or deletion. The updated micro-clusters Q_{t+1} are then made available to handle the next record. This *one-by-one feedback loop* limits the clustering throughput as new record can not be clustered until its previous record is finished.

To address the throughput limitation of one-record-at-a-time model, our key insight is to relax its strict sequential update-feedback constraint, by introducing a new *batch-by-batch* feedback loop. Specifically, we allow a batch of records to be processed on the *same*, albeit stale, micro-clusters Q_t and update Q_t at the end of each batch, as shown in Figure 3(a). This batch-by-batch feedback loop is also denoted as *mini-batch update model*, which introduces more parallelization dimensions for us to improve the throughput of each step. For example, in the first step of Figure 3(a), we can compute the closest micro-cluster of $\{x_1, x_2, x_3, x_4\}$ in parallel.

However, this mini-batch update model incurs three challenges. *First*, the update step needs to update a batch of new records to the stale micro-clusters at a time, while leveraging traditional one-by-one update suffers from low throughput. We describe how our decoupled update design improves the update throughput in Section IV-B. *Second*, it is challenging to achieve comparable clustering quality to the one-record-at-a-time model, given that all records in a mini-batch use stale micro-clusters for computation. We describe how our order-aware update mechanism addresses this staleness problem in Section IV-C. The *third* challenge is how to select batch size since it is related to clustering quality and throughput. We discuss this problem in Section IV-D.

B. Decoupling local and global update steps

When using mini-batch update model, the first step is to compute the closest micro-clusters for all records within the same mini-batch, e.g., the closest micro-clusters $\{q_3, q_2, q_3, q_2\}$ for records $\{x_1, x_2, x_3, x_4\}$ in Figure 3(a). The second step is to update these chosen micro-clusters together



(a) We decouple the computation execution into three sequential steps and enforce the update order within each batch. (b) Batch size selection.

Fig. 3. *DistStream*'s order-aware mini-batch update model and batch size selection.

with existing micro-clusters. If we were to follow the update step of one-record-at-a-time model, i.e., updating each chosen micro-cluster and immediately merging it with the existing micro-clusters, the overall throughput can be very low.

To improve the update throughput, our key idea is to decouple the update step to two independent sub-steps, i.e., a *parallel* local update step and a global update step. The local update step only updates the chosen micro-clusters, by adding the increments of records to them. As different micro-clusters can be independently updated, we can parallelize local update step using model-based parallelism as described in Section V-B. The next global update step performs *global* operations, such as merging these updated/newly created micro-clusters with existing micro-clusters, to reflect the data stream changes. We run global update step in a single node, because this step needs to collect all the updated micro-clusters together to perform merging/deletion operations. To reduce the computation latency, we optimize this step by pre-merging the newly created micro-clusters as described in Section V-C.

C. Order-aware update mechanism

In this section, we describe how we enforce the order-aware update in both local and global update steps to achieve good clustering quality as that of the one-record-at-a-time model. Our key insight is to reflect the impacts of records on clustering results based on their arrival orders.

1) Order-aware local update: To adhere to the same update principle of one-record-at-a-time model, we design an *order-aware* local update step. In this step, records that map to the same micro-cluster are updated in the same order as their arrivals. As shown in Figure 3(a), for each record x_i and its closest micro-cluster q, we compute x_i's increment Δx_i towards q and then update q. If record x_i arrives before record x_j and both identify micro-cluster q as the closest, we will insert x_i to q before x_j. This is to guarantee that each record's increment is correctly decayed according to its arrival order and appropriately updated to its closest micro-cluster. Thus, the impact of each record on the updated micro-clusters is consistent with that of the one-record-at-a-time model, leading to similar update effects.

Next, we provide a theoretical analysis that demonstrates the importance of update order by comparing to an unordered mini-batch update model [13]. Empirical comparisons between these two update models are presented in Section VII.

Importance of update ordering. Let us denote a stream of records as $\{x_1, x_2, \ldots\}$. Because the order of records only matters when records map to the same micro-cluster, we assume that two records that map to the closest micro-cluster q arrive as $\{\ldots, x_i, \ldots, x_k, \ldots\}$. Without loss of generality, we compare the impact of Δx_i and Δx_k (where $i < k$) on the micro-cluster q when updating in arrival order of $\{\ldots, x_i, \ldots, x_k, \ldots\}$ and in the reverse order of $\{\ldots, x_k, \ldots, x_i, \ldots\}$.

First, when updating in the order $\{x_i, x_k\}$ based on an update function as $q_{\text{new}} = \lambda_i q + \Delta x_i$, we have new micro-cluster $q'_{\rightarrow}$ as:

$$q'_{\rightarrow} = \lambda_k(\lambda_i q + \Delta x_i) + \Delta x_k = \lambda_k \lambda_i q + \lambda_k \Delta x_i + \Delta x_k,$$

where Δx_i and Δx_k represent the spatial and temporal increments of x_i and x_k towards micro-cluster q; λ_i and λ_k represent the decaying factor for record x_i and x_k, respectively. The decaying factor $\lambda_i = \beta^{-\Delta t_i}$ is a function of Δt_i that denotes the time interval between x_i and the previous record that was updated to q. Since β is a constant and $\beta \geq 1$, we have $\lambda \leq 1$.

We can then quantify the impact of data record x_k (the most recently updated data) on micro-cluster $q'_{\rightarrow}$ as:

$$Impact_j^{\rightarrow} = \left[\frac{\Delta x_k}{q'_{\rightarrow}}\right]_j = \left[\frac{\Delta x_k}{\lambda_k \lambda_i q + \lambda_k \Delta x_i + \Delta x_k}\right]_j.$$

Since Δx_k and $q'_{\rightarrow}$ are vectors, we use $\left[\frac{\Delta x_k}{q'_{\rightarrow}}\right]_j$ to quantify the impact of x_k on $q'_{\rightarrow}$ at the j-th dimension. When updating in the reverse order, the new micro-cluster $q'_{\leftarrow}$ can be represented as:

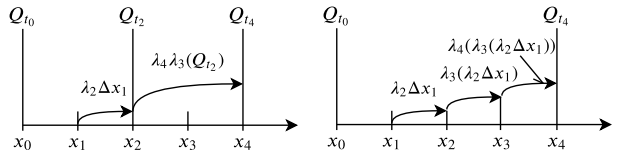
$$q'_{\leftarrow} = \lambda_i(\lambda_k q + \Delta x_k) + \Delta x_i = \lambda_i \lambda_k q + \lambda_i \Delta x_k + \Delta x_i.$$

Similarly, we can then quantify the impact of data record x_k on micro-cluster $q'_{\leftarrow}$ as:

$$Impact_j^{\leftarrow} = \left[\frac{\lambda_i \Delta x_k}{q'_{\leftarrow}}\right]_j = \left[\frac{\Delta x_k}{\lambda_k q + \Delta x_k + \frac{\Delta x_i}{\lambda_i}}\right]_j.$$

The reason we use $\lambda_i \Delta x_k$ instead of Δx_k is that record x_k's increment Δx_k has been decayed by λ_i after updating the next record x_i to $q'_{\leftarrow}$. Further, because we have $\lambda_i \leq 1$ and $\lambda_k \leq 1$, we have $\lambda_k \lambda_i q \leq \lambda_k q$ and $\lambda_k \Delta x_i \leq \frac{\Delta x_i}{\lambda_i}$. By substituting these inequalities, we then have:

$$Impact_j^{\rightarrow} \geq Impact_j^{\leftarrow}.$$



(a) Update with batch size = 2s. (b) Update with batch size = 4s.

Fig. 4. The record x_i's increment towards the micro-clusters Q_{t_4} does not vary with batch sizes, suppose each record arrives at 1 second interval.

That is, for a given record, updating it in different orders leads to different impacts on the updated micro-cluster. This necessitates the need for ensuring update order in mini-batch model, in order to match with the update principle of one-record-at-a-time model and catch up with the dynamically changing data distribution.

2) Order-aware global update: We design an order-aware global update step, in which each micro-cluster is updated based on its created/updated time. In detail, the global update step needs to merge a set of newly created/updated micro-clusters $\{q'_i\}$ with the old ones Q_t. Since micro-cluster operations such as deletion and merging are irreversible (e.g., deleted micro-clusters cannot be recovered), it is important to perform these operations on micro-clusters by the order of their updated/created time. If we were to process these micro-clusters out-of-order, we may delete important micro-clusters that reflect the current data pattern. For example, suppose $\{q'_1, q'_2\}$ are two newly created micro-clusters and q'_1 was created before q'_2. The anticipated update outcome towards the overall micro-clusters is to incorporate both q'_1 and q'_2, or at least q'_2. However, if updating with the reverse order $\{q'_2, q'_1\}$, the overall micro-clusters may only include q'_1 but not q'_2—when q'_2 is merged into an older micro-cluster that is later deleted due to its lower temporal locality.

By preserving the update order in local and global update steps, our approach is more resilient to the clustering quality loss introduced by the *stale* micro-clusters. Concretely, as the first step operates on stale micro-clusters, some of the chosen micro-clusters are not *necessarily* the closest ones, referred to as non-optimal micro-clusters. As such, the local update step might operate on some non-optimal micro-clusters. However, since the global update step has a global view of all the existing micro-clusters, it can mitigate this problem by merging non-optimal micro-clusters with existing ones. For example, record x_4 in Figure 3(a) is mapped to a non-optimal micro-cluster q_2 even though the true closest micro-cluster is q_3. In cases when the optimal micro-cluster q_3 and non-optimal micro-cluster q_2 come from the same natural cluster, they tend to have small distance. Therefore, q_2 and q_3 could be merged in global update step.

D. Determining batch size

To determine the appropriate batch size, we first need to understand its impact on clustering quality. When using the mini-batch update model, the clustering quality of micro-clusters Q_t is mainly determined by the increment of each record in the batch and the previous micro-clusters Q_{t-1}. Based on our order-aware update mechanism, we can show that each record's increment does not vary with batch sizes. When using small batch size as shown in Figure 4(a), record x_1's increment towards Q_{t_2} (i.e., Δx_1) will be decayed by λ_2 when updating x_2 to Q_{t_2} with $Q_{t_2} = \lambda_2(\lambda_1 Q_{t_0} + \Delta x_1) + \Delta x_2$ in the first batch. Likewise, in the second batch, Q_{t_2} will be decayed twice to be $\lambda_4\lambda_3(Q_{t_2})$ when updating x_3 and x_4 to Q_{t_4}. Since x_1 is absorbed in Q_{t_2}, x_1's increment towards Q_{t_4} is decayed accordingly to be $\lambda_4\lambda_3(\lambda_2\Delta x_1)$, which equals to record x_1's increment towards Q_{t_4} when using large batch size as shown in Figure 4(b). Therefore, batch size has limited impact on clustering quality which is also empirically evaluated in Section VII-B2.

With large batch size, the record's increment tends to be decayed to a small value. As shown in Figure 4(b), x_1's increment towards Q_{t_4} is decayed as $\lambda_4\lambda_3\lambda_2\Delta x_1 = \beta^{-(t_4-t_1)}\Delta x_1$ (β is a constant), which will become a small value if $\Delta t = t_4 - t_1$ is very large as shown in Figure 3(b). Small increments can be problematic as they make the updated micro-clusters such as Q_{t_4} only reflect the latest records, but omit the intermediate clustering results like Q_{t_2}. To mitigate this problem, one potential way is to set the batch size by bounding the decaying factor in order to control the impact of records on clustering. Concretely, given a user-defined threshold α, we have $\beta^{-\Delta t} > \alpha$ which leads to a maximum batch size of $log_\beta \frac{1}{\alpha}$. For example, the maximum batch size is about 25 seconds when $\alpha = 0.01$ and $\beta = 1.2$.

Lastly, we empirically study the batch size impact on clustering throughput in Section VII-D3 and discuss the potential ways to select best batch size for improving throughput.

V. Parallelizing order-aware mini-batch model

In this section, we describe our approaches to parallelize the order-aware mini-batch update model for high throughput and good clustering quality. We first formalize the computation problem in each step and analyze the computation dependency. We then select the most efficient parallel approach for the first two steps, based on the theoretical analysis of computation latency and network communication of three potential parallelism dimensions. Specifically, we consider record-based, model-based, and feature-based parallelisms that leverage the observations of independent computation for each data record, each micro-cluster, and each feature, respectively. We also explain our optimizations for performing the global update step in a single machine. Below we use *task* to denote the computation unit that can run in parallel in different machines.

A. Finding the closest micro-cluster

The first step is to assign each incoming record within the *same* mini-batch to an appropriate micro-cluster, based on the spatial locality. Formally, for each record x_i, this step computes its distance with the centroid of each micro-cluster, selects the closest micro-cluster q, and performs outlier check by verifying whether x_i falls within the maximum boundary of micro-cluster q.

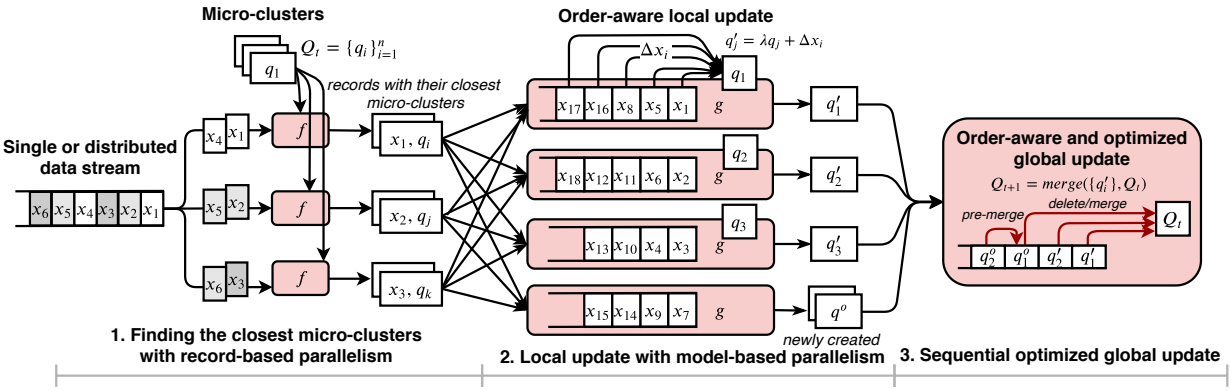


Fig. 5. The parallelized order-aware mini-batch update model.

Because the goal is to pick the closest micro-clusters of all the data records, the distance computation and verification procedure depends on all the records and all existing micro-clusters. Since the distance computation is independent for each record and for each micro-cluster, we can parallelize this computation by each data record, referred to *record-based parallelism*, and each micro-cluster, referred to *model-based parallelism*. Moreover, the distance function depends on the computation between each feature of x_i and the corresponding dimension of q. This makes it possible to parallelize the computation by each feature, referred to *feature-based parallelism*. As shown in Table I, we compare the performance of these three possible dimensions and select the most efficient record-based parallelism to parallelize this step. Another advantage of record-based parallelism is that it maintains the temporal order of the outputed results, which further improves the parallelization efficiency of next local update step.

Using the record-based parallelism: As shown in Figure 5, when using record-based parallelism, incoming records are assigned to different tasks that perform the distance computation and verification of each record locally. To do so, we first broadcast (copy) the entire micro-cluster set $Q^t = \{q_i\}_{i=1}^n$ to each task at the beginning of each batch-by-batch feedback loop. We then assign incoming records with different timestamps into different tasks in a *round-robin* way. This is to facilitate the goal of maintaining the relative orders between the input data records and the output micro-cluster results. Other assignment approaches, such as *range partition* that assigns a range of incoming records to each task, fail to produce the records' closest micro-clusters in the desired temporal order. For each assigned record x_i, the task computes distances between x_i and all the micro-clusters $\{q_i\}_{i=1}^n$, selects and verifies the closest micro-cluster q_j, and finally outputs $\langle x_i, q_j \rangle$.

Compare with other parallelism dimensions: Apart from record-based parallelism, we can parallelize this step by dividing micro-clusters (or *features*) into different tasks, and perform distance computation on each subset of micro-clusters (or *features*) in parallel, namely *model-based* (or *feature-based*) parallelism. Record-based parallelism outperforms the others because it avoids unnecessary inter-task communication and achieves higher parallelism degree.

First, record-based parallelism naturally fits the distributed data stream processing with the *lowest* network communication. Record-based parallelism divides the data stream into parallel streams according to data record arrival time and keeps the order of the records in each stream. The only network communication is to copy the most update-to-date micro-cluster set Q^t to all N tasks at the beginning of each feedback loop, leading to $O(N|Q|)$ network communication where $|Q|$ denotes the size of Q^t. Model-based parallelism divides the micro-clusters into different tasks and each task is only responsible for a subset of micro-clusters. Thus, we need to copy each incoming record to N tasks, leading to $O(mN|x|)$ network communication where m is the record number in a batch (and often larger than N) and $|x|$ is the size of a record. Moreover, since each task only computes partial distance results (on a subset of micro-clusters), additional inter-task data shuffling and synchronization are needed to aggregate the partial results to select the closest micro-cluster of each record. This leads to an additional $O(m|Q|)$ network communication. The feature-based parallelism first distributes a subset of data features to parallel tasks with $O(m|x|)$ network communication, and then aggregates the partial results from all tasks, further leading to an additional $O(m|Q|)$ network communication as model-based parallelism. In summary, both model-based and feature-based parallelisms lead to more network communication since mini-batch size m is usually larger than the parallelism degree (task number N).

Second, the record-based parallelism has the lowest per-batch latency of $O(\frac{mn}{N})$ for a batch of *m* incoming records and *n* micro-clusters. This is because all the required computation is done in the same task. On the contrary, both the model-based and feature-based parallelisms require an *additional* stage to aggregate partial results from all *N* parallel tasks with time complexity of $O(merge)$. This merge time depends on the amount of data that need to be shuffled and network condition.

Third, record-based parallelism provides the highest flexibility in varying the parallelism degree. Intuitively, record-

TABLE I
THEORETICAL ANALYSIS ON DIFFERENT PARALLEL APPROACHES FOR STEP ONE: FINDING THE CLOSEST MICRO-CLUSTER.

Characteristics	Record-based	Model-based	Feature-based
Network communication	**Low** $O(m\|x\|)$ $+O(N\|Q\|)$	**High** $O(mN\|x\|)$ $+O(m\|Q\|)$	**Medium** $O(m\|x\|)$ $+O(m\|Q\|)$
Computation latency	**Low** $O(mn/N)$	**High** $O(mn/N)$ $+O(merge)$	**High** $O(mn/N)$ $+O(merge)$
Parallelism degree	**Flexible** m	**Less Flexible** n	**Fixed** d

TABLE II
THEORETICAL ANALYSIS ON DIFFERENT PARALLEL APPROACHES FOR STEP TWO: LOCALLY UPDATING THE CLOSEST MICRO-CLUSTER.

Characteristics	Model-based	Record-based	Feature-based
Clustering quality	**High**	**Low**	**High**
Computation latency	**Low** $O_{sort}\left(\frac{m}{N}\right)+$ $O_{update}\left(\frac{m}{N}\right)$	**High** $O_{update}\left(\frac{m}{N}\right)$ $+O_{sort}\left(\frac{m}{N}\right)$ $+O_{merge}\left(\frac{m}{N}\right)$	**High** $O_{sort\text{-}merge}\left(\frac{m}{N}\right)$ $+O_{update}\left(\frac{m}{d}\right)$ $+O_{merge}\left(\frac{m}{N}\right)$
Network communication	**Low** $O(m\|q\|)$	**Low** $O(m\|q\|)$	**High** $O(2m\|q\| + \|q\|)$
Parallelism degree	**Less Flexible** n	**Flexible** m	**Fixed** d

based parallelism is only bounded by the batch size *m*, while model-based and feature-based parallelisms are limited by the number of micro-clusters *n* or the number of features *d*. In general, we can adjust the batch size to increase the parallelism while it is less common to adjust the number of micro-clusters as it can lead to degraded clustering quality, and the number of features is fixed given a dataset. That said, the parallelism degree often satisfies the following inequality: $m > max(n, d)$.

B. Locally updating the closest micro-cluster

After finding the closest micro-cluster of each incoming record x_i, the next step is to update the corresponding micro-cluster of x_i. Formally, this local update step computes the increment of record x_i towards its closest micro-cluster q as Δx_i, and update q to q' with $q' = \lambda q + \Delta x_i$. This function leverages the additivity property of the micro-cluster, and λ denotes a time fading factor that controls the importance of newer records over this old record. As discussed in Section IV-C, it is important to enforce the local update order based on the arriving order of the records, for maintaining good clustering quality. If the incoming record is identified as an outlier record, a new micro-cluster q_o is created for it.

Because the goal is to update the closest micro-clusters of all the records, we can also parallelize this local update step by each record, each micro-cluster, and each feature, similar to that in the first step. As shown in Table II, model-based parallelism outperforms the others in performance and can strictly keep the update sequence. Thus, we select the model-based parallelism to parallelize this step.

Using the model-based parallelism: Since the local update procedure is independent for each micro-cluster, we divide the closest micro-clusters of incoming records into different tasks and update each micro-cluster in parallel. As shown in Figure 5, the output results of the first step (i.e., records with their closest micro-clusters) are grouped by the micro-cluster *id* as $\{q_j, [x_a, x_b, \dots x_k]\}_{j=1}^{n}$ and then distributed into different tasks. Each task first *sorts* the absorbed records $[x_a, x_b, \dots x_k]$ of each micro-cluster q_j based on the timestamps to enforce that update orders comply with arrival orders. Then, it computes the record's increment Δx_i and updates q_j to q'_j by $q'_j = \lambda q_j + \Delta x_i$ iteratively, one record at a time. Finally, the task outputs the updated micro-cluster q'_j for the next computation step of global update.

Compare with other parallelism dimensions: We choose to parallelize the computation in this step using model-based parallelism for two key reasons.

First, model-based parallelism allows us to strictly keep the update order with minimal efforts. When using model-based parallelism, each task groups all the absorbed records of a micro-cluster together, sorts them, and updates the micro-cluster in sequence. Thus, the update order of each micro-cluster is guaranteed. On the contrary, record-based parallelism not only breaks the sequential update constraint by updating the closest micro-cluster of each record in parallel, but also requires access to carefully designed function for merging these partial results. Feature-based parallelism updates each dimension of micro-cluster in parallel and needs to merge all the records into a single stream to keep the update order.

Second, model-based parallelism has lower computation latency and network communication. In model-based parallelism, the task computes record increment and updates each micro-cluster locally. The average latency is the sum of sorting and updating as $O_{sort}(\frac{m}{N})+O_{update}(\frac{m}{N})$, where *m* is the number of records in a batch and *N* is the number of tasks. The only network communication occurs when grouping the closest micro-cluster of each record together, which incurs $O(m|q|)$ amount of data shuffling. Here, $|q|$ denotes the size of a micro-cluster. Compared to the model-based parallelism, record-based parallelism incurs an additional latency of $O_{merge}(\frac{m}{N})$. The reason is that record-based parallelism first updates the closest micro-cluster of each record in parallel with update latency $O_{update}(\frac{m}{N})$, and then requires additional tasks to merge the partially updated micro-clusters with $O_{sort}(\frac{m}{N})+O_{merge}(\frac{m}{N})$ latency. Furthermore, record-based parallelism incurs a network communication of $O(m|q|)$ when merging the updated micro-cluster of each record. Moreover, it is hard to design an accurate merge function since different tasks may add different number of record increments to the same micro-cluster. Feature-based parallelism requires three steps, including merging all the records into a single stream to keep the update order, updating each micro-cluster dimension, and merging the partial update results together. Therefore, the total latency is $O_{sort\text{-}merge}(\frac{m}{N})+O_{update}(\frac{m}{d})+O_{merge}(\frac{m}{N})$ and the total network communication is $O(m|q| + m|q| + |q|)$, which are higher than the other two approaches.

C. Globally updating all the micro-clusters

During the global update step, we need to merge a set of newly created/updated micro-clusters with the stale micro-clusters set. As described in Section IV-C, we need to keep the update sequence of these new micro-clusters, i.e., merging with stale micro-clusters according to their created/updated time. To do so, we perform the following three sequential computations. First, the old micro-clusters that were not updated during the local update step are decayed to decrease their importance. Second, we replace the rest old micro-clusters with the corresponding newly updated ones. Third, in order to accommodate each newly created micro-cluster, we first try to delete as many outdated micro-clusters as possible and then merge any of the two closest micro-clusters.

Optimized global update in a single node: The global update step has the least computation demand among the three steps and requires sequential update to guarantee the clustering quality. As such, we choose to perform the global update step in a single powerful node. The detailed reasons as well as optimization approach are as follows. *First*, we need to collect newly created/updated micro-clusters together to find the old micro-clusters to decay and then to perform the *deletion* and *merging* operations. *Second*, the input micro-clusters of each operation depend on the results of the last operation. For example, the input of merging operation depends on the results of deletion operation. *Third*, the number of micro-clusters *n* is often much smaller than the incoming records *m*. Thus, the global update step has low latency, except when there are many newly created micro-clusters. More newly created (i.e., outlier) micro-clusters means that we need to perform more operations to find two closest micro-clusters for merging, which increase the latency. To mitigate this problem and reduce latency, we perform a *pre-merge* operation on outlier micro-clusters by letting current outlier micro-cluster (e.g., q_2^o in Figure 5) merge with the previously created outlier micro-clusters (e.g., q_1^o). The key intuition is that many outlier micro-clusters are from the same natural cluster when data distribution is evolving from one cluster to another, and therefore this pre-merge operation can reduce the number of outlier micro-clusters. Consequently, our optimization can reduce computation latency with decreased number of outlier micro-clusters.

VI. DistStream implementation

We have implemented *DistStream* as a distributed framework for parallelizing online-offline stream clustering algorithms, in a total of 5.3K lines of code. We choose to build *DistStream* atop Spark Streaming [31] instead of Flink [2] and Storm [3], because Spark Streaming supports dividing data streams into mini-batches and aggregating results of parallel tasks at the end of each batch. These features facilitate our implementation of the batch-by-batch feedback and global update step. Concretely, we leverage existing Spark Streaming operators such as *window*(), *map*(), *groupByKey*(), and *sort*() to implement our order-aware mini-batch update model.

DistStream exposes four APIs, including micro-cluster representation, distance computation, local update, and global update, which abstract the computational flow of distributed stream clustering algorithms. These APIs allow algorithm developers to implement any stream clustering algorithms that comply with the online-offline update paradigm, because such algorithms only differ in their micro-cluster representations and micro-cluster update functions [19].

We have implemented four representative stream clustering algorithms atop *DistStream*, namely CluStream, DenStream, D-Stream, ClusTree. For instance, to implement the partition-based algorithm CluStream and the density-based algorithm DenStream, we define micro-cluster representations as $(\sum_{i=1}^{n} x_i^2, \sum_{i=1}^{n} x_i, \sum_{i=1}^{n} t_i^2, \sum_{i=1}^{n} t_i)$ for CluStream and as $(\sum_{i=1}^{n} w_i x_i^2, \sum_{i=1}^{n} w_i x_i)$ for DenStream, respectively. Here, w_i is the temporal weight of record x_i. For both algorithms, we use Euclidean distance function for distance computation. We define the local update functions for CluStream and DenStream as $\Delta x_i = (x_i^2, x_i, t_i^2, t_i)$; $\lambda_i = 1$ and $\Delta x_i = (x_i^2, x_i)$; $\lambda_i = \beta^{-\Delta t_i} < 1$, respectively. Finally, for the global update, we use two different temporal thresholds for deleting older micro-clusters.

DistStream uses Spark Streaming runtime to distribute and schedule *DistStream* tasks to a cluster of machines. *DistStream* leverages Spark Streaming's *parallel recovery* mechanism for fault tolerance [31]. *DistStream* executes the global update step using Spark Streaming driver in a single machine.

VII. Evaluation

In this section, we evaluate the clustering quality and performance of stream clustering algorithms implemented on *DistStream*, and compare with that implemented on one-record-at-a-time model and unordered mini-batch update model.

We select four representative stream clustering algorithms, namely *CluStream*, *DenStream*, *D-Stream* and *ClusTree*, and implement them atop *DistStream*. For all four algorithms, we implement two sets of baselines: the first baselines (with prefix *MOA-*) are implemented using the single-machine MOA library [15], and the second baselines (with prefix *unordered-*) are implemented based on an existing work [13] for *CluStream* and based on *DistStream* for the other three algorithms. We configure each algorithm with default values from original papers [11, 16]. For example in *CluStream*, the number of micro-clusters is set to ten times of the real cluster numbers. In *DenStream*, we set $\beta = 2^{0.25} \approx 1.2$ and μ = 10, where μ denotes radius threshold. Given that we observe similar results for all four algorithms and consider space limitation, we only detail the results of *CluStream* and *DenStream* from Section VII-B to VII-D, and summarize the results of *D-Stream* and *ClusTree* in Section VII-E.

A. Experimental setup

We perform the evaluation on a local cluster of ten nodes, as shown in Figure 6. *DistStream* runs on a master-slave architecture with one master and eight workers. To generate the data stream, the last node runs a Apache Kafka producer

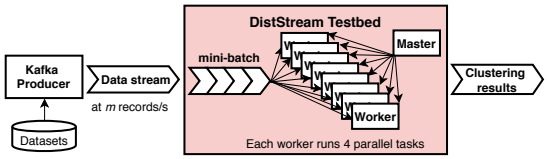


Fig. 6. Testbed of *DistStream*.

TABLE III
THE CHARACTERISTICS OF THE THREE DATASETS.

Dataset	#Records	#Used features	#Clusters (a% b% c%)
KDD-99 [7]	494,021	34	23 (57%, 22%, 20%)
CoverType [4]	581,012	54	7 (49%, 36%, 6%)
KDD-98 [6]	95,412	315	5 (95%, 1.5%, 1.4%)

* We normalize each feature of the three datasets to have zero mean and unit variance, to avoid biasing any features [21]. (a%, b%, c%) denotes the record percentages of the three largest real clusters.

that reads data records from local disk sequentially and outputs the records at a user-defined rate. *DistStream* pulls the data stream in mini-batches and distributes the data stream into eight workers. Each worker has 4 physical cores and 64GB memory, and we configure each *DistStream* task with a physical core and 8GB memory. Therefore, the maximum parallelism degree (i.e., the number of parallel tasks) is thirty-two. We use Spark 2.4.0 standalone version with Hadoop HDFS 2.7.4 and Apache Kafka 1.1.0, running on Ubuntu 16.04 to perform all the experiments.

We choose three real-world datasets that are widely used for evaluating stream clustering algorithms [11, 16, 22], as shown in Table III. These datasets have different data distributions and feature dimensions, making them ideal for studying clustering quality. We convert each dataset to a data stream through first setting the timestamp for each record and then streaming them in chronological order. We use the cluster labels as the ground truth for clustering quality measurement.

(1) KDD-99 is a network intrusion detection dataset from KDD Cup 1999 [7], describing the dynamically changing TCP connections in a network attack environment in MIT. This dataset forms 23 clusters, including one normal connection and 22 different attack types such as buffer overflow and rootkit.

(2) CoverType is a cartographic dataset, describing forest areas with different elevations in Colorado [4]. These records have seven clusters that each represents one forest cover type.

(3) KDD-98 is a charitable donation dataset from KDD Cup 1998, describing people's response to the donation promotion mailing from a not-for-profit organization [6]. We group the records into five clusters based on donation amount, ranging from [\$0], (\$0, \$10], (\$10, \$15], (\$15, \$20], and (\$20, $+\infty$).

For performance and scalability evaluation, we prepare a larger dataset by instructing Kafka to read from the same dataset ten times. Finally, we construct three larger datasets, namely *large-KDD99*, *large-CoverType*, and *large-KDD98*.

B. Clustering quality comparison

1) Methodology: We use standard Clustering Mapping Measure (CMM) criterion [25] to measure the stream clustering quality, because it is more accurate than batch-oriented metrics such as SSQ, Purity, and F-measure [25, 19]. CMM does so by decaying the weights of aging records and penalizing three common clustering errors caused by evolving clusters, namely *missed records*, *misplaced records*, and *noise records*. CMM normalizes these errors as well as the difference between clustering results and ground truth to a value between 0 and 1, where larger value denotes higher clustering quality.

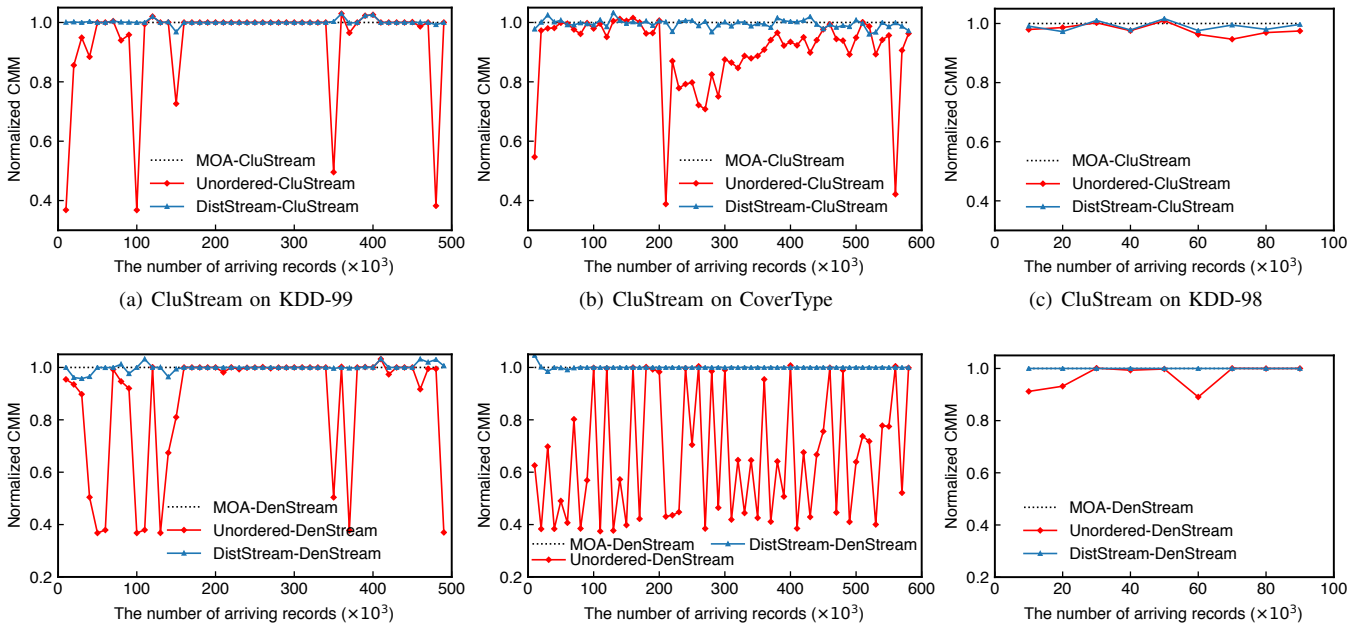
For each dataset, we stream the data records at a rate of 1K records/s. To fairly compare with MOA-based single machine implementations, we run all distributed implementations in a single machine with parallelism degree of one. We set the batch size to be 10 seconds and calculate the CMM values at the end of each batch using the clustering results generated offline. We compute the CMM values of MOA-based algorithms at the same interval.

2) Clustering quality results: Figure 7 plots the normalized CMM values along the entire data stream, for all the algorithm implementations on the three datasets. The normalized CMM values are raw CMM values divided by that achieved by corresponding MOA-based implementations. Thus, the normalized CMM values of MOA-based implementations are always 1.0.

For both CluStream and DenStream algorithms, our *DistStream*-based implementations achieve comparable (average 99%) clustering quality to the MOA-based ones on all three datasets. Specifically, the achieved clustering quality difference between *DistStream*-CluStream (*DistStream*-DenStream) and *MOA*-CluStream (*MOA*-DenStream) is 1.1% (0.3%) on average. In contrast, both *unordered*-CluStream and *unordered*-DenStream suffer from up to 60% lower clustering quality than MOA-based and DistStream-based implementations.

The clustering quality gaps between unordered and *DistStream*-based implementations suggest the importance of maintaining update order in mini-batch update model. For KDD-99 and CoverType datasets, unordered implementations suffer from significantly lower clustering quality. We analyze these clustering quality differences and find that the number of *missed records* directly impact CMM values. In particular, for KDD-99 and CoverType datasets, unordered implementations cause on average 2.6X and 1.8X more *missed records* than that of our order-aware implementations. The reason is that unordered implementations mislabel 1.5-3.2X more incoming records to be outliers than that of our order-aware implementations. The root cause is that unordered mini-batch model fails to favor recent records, making the updated micro-clusters cannot capture the recent data patterns. Further, unordered implementations suffer from unpredictable (highly fluctuant) clustering quality, because the data records within a mini-batch have random impacts on the updated micro-clusters.

For KDD-98 dataset, the clustering quality differences between unordered and our order-aware implementations are less obvious (up to 11%). The number of *missed records* of unordered implementations is also much lower (up to 6%). The key reason is that KDD-98 dataset is more stable than KDD-99 and CoverType datasets. Here, a dataset is stable if its data distribution has small change over time. For KDD-



(a) CluStream on KDD-99 (b) CluStream on CoverType (c) CluStream on KDD-98

(d) DenStream on KDD-99 (e) DenStream on CoverType (f) DenStream on KDD-98

Fig. 7. The clustering quality of CluStream and DenStream algorithms on different datasets using different update models.

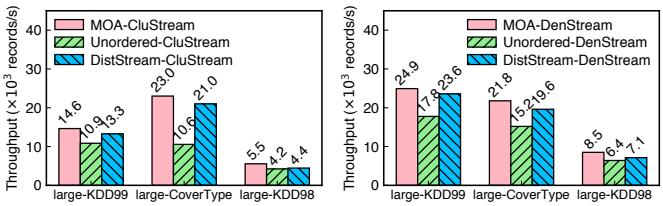
98, this translates to that 95% of its data records belong to a long-standing dominating cluster. In this case, most of the updated micro-clusters map to the dominating cluster, reducing the probability of mislabeling outliers. This reveals that the update order exerts *higher* impacts on clustering quality for datasets with more dynamically changing distributions.

Next, we repeat the experiments for different batch sizes, including smaller batch size (5s) and larger batch sizes (15s to 30s with 5s interval). We observe on average 2.79% clustering quality differences between *DistStream*-based and MOA-based implementations, for all three datasets. This suggests that the batch size has limited impact on clustering quality when using order-aware mini-batch update model. The reason is that the records' increments, which directly impact the clustering quality, will stay the same with different batch sizes, as long as the update order is maintained. More theoretical analysis can be found in Section IV-D.

Summary: *DistStream*-based implementations achieve comparable (average 99%) clustering quality with MOA-based counterparts, while unordered ones suffer from up to 60% lower clustering quality. We also identify that more stable datasets are less sensitive to update order. Further, batch sizes have limited impact (average 2.79% quality difference) on our order-aware mini-batch update model.

C. Performance comparison in a single machine

Since MOA library only provides single-machine implementations, we compare the throughput of *MOA*-, *unordered*-, and *DistStream*-based implementations in a single machine, i.e., one task with one physical core and 8GB RAM.

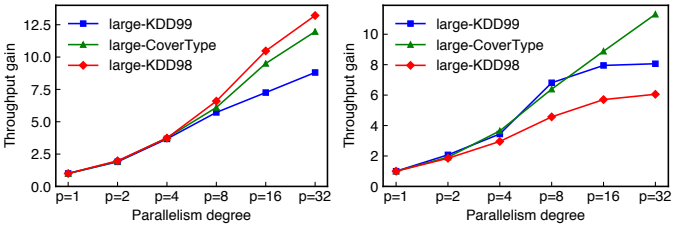


(a) CluStream throughput (b) DenStream throughput

Fig. 8. Throughput of CluStream and DenStream in a single machine. The per record latency is the inverse of the throughput.

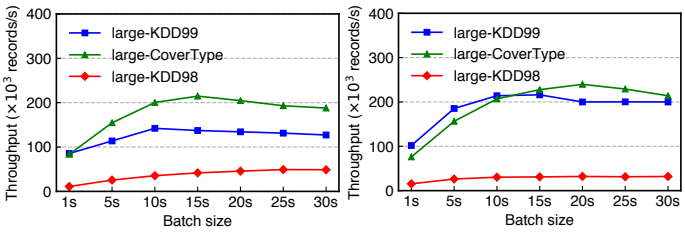
1) Methodology: For stress throughput test, we measure and use the Kafka's maximum stable streaming rates on all three large datasets, i.e., 100K/s on the low dimensional large-KDD99 and large-CoverType, and 10K/s on the high dimensional large-KDD98. Further, to factor out the network overhead, we co-locate all the data records with the task. We also configure the batch size to be 10 seconds for both unordered and *DistStream*-based implementations. We calculate the average throughput by dividing total data records by the total processing time, and calculate per record latency as the inverse of the throughput. We repeat each experiment five times and report the average throughput.

2) Performance results: Figure 8 illustrates the throughput of three implementations of *CluStream* and *DenStream* on three large datasets. Compared to MOA-based implementations, mini-batch based implementations have an average of 10.6% lower throughput. This performance difference can be attributed to additional works that are associated with mini-batch based implementations, such as starting, serializing/de-



(a) *DistStream*-CluStream (b) *DistStream*-DenStream

Fig. 9. Throughput gains of *DistStream*-CluStream and *DistStream*-DenStream algorithms with different parallel degrees on three datasets.



(a) *DistStream*-CluStream (b) *DistStream*-DenStream

Fig. 10. The throughput with different batch sizes at p=32. The throughput on large-KDD98 is lower than others due to its higher feature dimension.

serializing and scheduling tasks. For example, we observe this latency overhead is 4μs per record on large-CoverType.

Next, compared to *unordered* implementations, our order-aware implementations have on average 1.3X higher throughput, across all algorithm-dataset combinations. Intuitively, *DistStream*-based implementations should incur *higher* processing latency than unordered implementations due to additional sorting overhead. However, we observe that this sorting overhead is offset by the unordered implementations' tendency to generate and handle more outliers (as described in Section VII-B2), leading to about 12μs per record latency.

Summary: Compared to MOA-based implementations, mini-batch based implementations have an average of 10.6% lower throughput. Such overhead is attributed to system overhead such as starting, serializing and scheduling tasks. In addition, *DistStream*-based implementations outperform *unordered* ones because of less outlier micro-clusters to process in global update step, justifying our order-aware design choice.

D. Scalability

This section investigates the scalability of *DistStream*-based implementations. We pinpoint the key factors that impact *DistStream*'s scalability through an in-depth bottleneck analysis, and also study the impacts of batch sizes.

1) Methodology: Similar to Section VII-C, we use Kafka to produce the maximum stable stream rate and measure the clustering throughput of *DistStream*-based implementations. We vary the parallelism degree, i.e., the number of parallel tasks, from 1 to 32. For large-KDD99 and large-CoverType datasets, we set the batch size to be 10 seconds. For high-dimensional large-KDD98 dataset, we set the batch size to be 20 seconds due to its lower rate, i.e., 10K/s compared to 100K/s of others described previously in Section VII-C1. We discuss how to vary batch size in Section VII-D3.

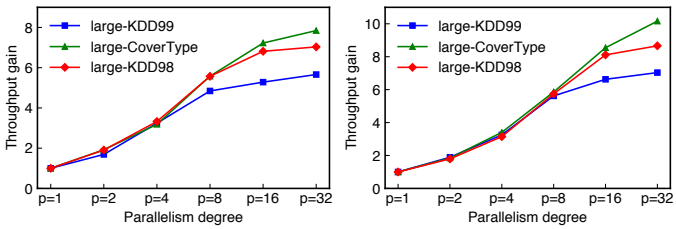
2) Scalability results: Figure 9 depicts the *throughput gains* for both *DistStream*-Clustream and *DistStream*-DenStream implementations. The throughput gain is calculated as the ratio of the achieved throughput at parallelism degree p=k to the throughput at p=1. Our key observation is that *DistStream*-based implementations achieve sublinear throughput gain of 13.2X when the parallelism degree is 32.

We identify three potential bottlenecks that lead to *DistStream*'s sublinear throughput gain. The *first* bottleneck comes from performing the global update step in a single machine without parallelization for ensuring the clustering quality. For example, we find that the latency of global update stays relatively constant, instead of decreasing, when parallelism degree increases for both large-KDD99 (∼6μs per record) and large-KDD98 (∼2μs per record). To improve the scalability, we can run this global update step in a more powerful server while still ensuring the clustering quality.

The *second* bottleneck is the overhead of system and network, which does not decrease with the increasing parallelism degree. For example, we observe similar overhead of ∼5μs per record when increasing parallelism degree from p=16 to p=32. The *third* bottleneck comes down to the increasing number of straggler tasks, e.g., tasks with execution time that exceed 1.2X of the average. Because *DistStream* currently uses a synchronous update protocol, straggler tasks can prolong the execution time of the first two parallelized steps. For example, when running *DistStream*-CluStream on large-KDD99, the percentage of straggler tasks increases from 12% to 25% as parallelism degree increases from p=16 to p=32. The potential optimization is to design new asynchronous update protocol.

Summary: *DistStream*-based implementations can achieve sublinear throughput gain of 13.2X when the parallelism degree is 32. The bottlenecks of *DistStream* include the single-machine global update step, system and network overhead, and the straggler tasks under a synchronous update protocol.

3) Impacts of batch sizes: Finally, we investigate whether varying batch size can improve *DistStream*'s currently achieved throughput, i.e., the throughput with parallelism degree of 32. Figure 10 shows the achieved throughput, when varying the batch size from 1s to 30s at fixed p=32. We observe that the clustering throughput first increases with the batch size and then drops at very large batch size (e.g., 30s on large-CoverType dataset). This indicates the potential to tune the batch size for higher throughput. The key reason for lower throughput when the batch size is small comes down to the lower task computation time. For example, when the batch size was set to 1s, each task only received about 3K records on large-CoverType. In this case, the tasks spend less time doing useful work, leading to higher percentage of system and network overhead compared to that of larger batch sizes. However, if the batch size is too large, the straggler tasks can lead to degraded performance. For example, when the



(a) *DistStream*-DStream (b) *DistStream*-ClusTree

Fig. 11. Throughput gains of *DistStream*-DStream and *DistStream*-ClusTree.

batch size was set to 30s, each task has long computation time and the straggler task can lead to 2s wait time in *DistStream*-CluStream on large-CoverType dataset.

In summary, to determine the optimal batch size for throughput, we need to balance factors such as task computation and wait time. In addition, to account for the potential quality degradation, we also need to limit the batch size as discussed previously in Section IV-D. Currently, we configure batch size statically based on a user-defined threshold (Section IV-D) but will explore adaptive batch sizing approaches in future work.

E. Results on the other two algorithms

We also evaluate *D-Stream* and *ClusTree* on *DistStream*. The experiments show that our *DistStream*-based implementations still achieve comparable (on average 99.1%) clustering quality with MOA-based counterparts. Figure 11 shows that our *DistStream*-based implementations achieve sublinear throughput gain with the increasing parallelism degree. The only differences are that D-Stream and ClusTree algorithms are more effective in finding the closest micro-clusters using grid mapping [17] and tree-based search [24], and thus achieve 1.1-1.3X higher throughput than CluStream and DenStream, when implementing using *DistStream*.

VIII. Related Work

Stream clustering algorithms. Stream clustering plays an important role in real-time data analysis. With the one-pass and memory capacity limitation, most existing stream clustering algorithms [10, 19, 26] follow the online-offline paradigm [11]. Even so, these algorithms are designed to capture different clustering shapes with different micro-cluster representations and update functions. For example, partition-based algorithms [9, 11] group the data stream into a number of partitions and are suitable for finding spherical clusters, while density-based algorithms like *DenStream* [16] are good at finding arbitrary shapes. Grid-based algorithms [17, 29] map the streaming data into discretized grids and group connected dense grids into clusters. Researchers also improve these algorithms for high-dimensional streams [12, 29], for exploring the evolution of density mountain [22], etc. Our work proposes an efficient framework for parallelizing stream clustering algorithms that comply with the online-offline paradigm.

To support high-speed distributed data streams, researchers have extended some batch-mode clustering algorithms to work in a distributed fashion [18, 32, 33]. These algorithms, such as distributed *k-center* [18] and distributed probabilistic algorithms [33], follow a different update paradigm called two-phase merge paradigm. Recently, researchers [13, 14, 23, 28] begin to implement a partition-based *CluStream* algorithm on distributed stream processing systems such as Spark Streaming and Storm. They adopt unordered mini-batch paradigm [13, 14] or similar periodical update paradigm [23, 28]. Our work improves clustering throughput and maintain the clustering quality through order-aware mini-batch update model.

Parallel approaches for machine learning algorithms. Currently, there are two types of parallel approaches for batch-mode machine learning algorithms [30]. One is data-parallel approach that horizontally splits the static big data into many workers, iteratively trains model parameters in each worker, and then aggregates the different model parameters together in server node(s). This approach is widely supported by distributed machine learning frameworks like Spark MLlib [27]. The other approach is model-parallel that splits the large model (e.g., parameter vectors) into different workers, iteratively trains the partitioned model in each worker on all the dataset, and then merges the trained partial models together. This approach can be used in parameter-server based frameworks such as Petuum [30] and Tensorflow [8]. We extend these approaches to stream clustering by considering both the data stream characteristics, namely one-pass, unbounded, ordering, and computation characteristics such as sequential update constraints and micro-cluster merging operation.

IX. Conclusions

Today's data streams require high throughput that cannot be delivered by single-machine stream clustering algorithms. In this paper, we present *DistStream* to scale out the widely-used online-offline stream clustering algorithms. *DistStream* provides a new order-aware mini-batch update model with efficient parallel approaches. *DistStream* is implemented atop Spark Streaming and provides APIs for migrating single-machine stream clustering algorithms to *DistStream*. Our evaluation shows that *DistStream*-based stream clustering algorithms can achieve sublinear throughput gain and comparable clustering quality with the single-machine counterparts.

References

[1] Alibaba: Key highlights from the 2017 11.11 Global Shopping Festival. https://www.alibabagroup.com/en/news/article?news=p171112.
[2] Apache Flink. http://flink.apache.org/.
[3] Apache Storm. http://storm.apache.org/.
[4] Covertype Dataset. http://archive.ics.uci.edu/ml/datasets/covertype.
[5] Imperva blocks DDoS. https://www.imperva.com/blog/imperva-blocks-our-largest-ddos-l7-brute-force-attack-ever-peaking-at-292000-rps.
[6] KDD98 Data. http://kdd.ics.uci.edu/databases/kddcup98/kddcup98.html.
[7] KDD99 Data. http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html.
[8] M. Abadi and et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, 2016.
[9] M. R. Ackermann and et al. Streamkm++: A clustering algorithm for data streams. *ACM Journal of Experimental Algorithmics*, 17(1), 2012.
[10] C. C. Aggarwal. A survey of stream clustering algorithms. In *Data Clustering: Algorithms and Applications*, pages 231–258. 2013.
[11] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for clustering evolving data streams. In *VLDB*, 2003.

[12] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for projected clustering of high dimensional data streams. In *VLDB*, 2004.

[13] O. Backhoff and E. Ntoutsi. Scalable online-offline stream clustering in apache spark. In *ICDM Workshop*, pages 37–44, 2016.

[14] A. Bifet and et al. Streamdm: Advanced data mining in spark streaming. In *ICDM Workshop*, pages 1608–1611, 2015.

[15] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer. MOA: massive online analysis. *Journal of Machine Learning Research*, 2010.

[16] F. Cao, M. Ester, W. Qian, and A. Zhou. Density-based clustering over an evolving data stream with noise. In *SIAM SDM*, 2006.

[17] Y. Chen and L. Tu. Density-based clustering for real-time stream data. In *KDD*, 2007.

[18] G. Cormode, S. Muthukrishnan, and W. Zhuang. Conquering the divide: Continuous clustering of distributed data streams. In *ICDE*, 2007.

[19] J. de Andrade Silva and et al. Data stream clustering: A survey. *ACM Comput. Surv.*, 46(1):13:1–13:31, 2013.

[20] M. Ester and et al. A density-based algorithm for discovering clusters in large spatial databases with noise. In *KDD*, 1996.

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

[22] S. Gong, Y. Zhang, and G. Yu. Clustering stream data by exploring the evolution of density mountain. *PVLDB*, 11(4):393–405, 2017.

[23] P. Karunaratne and et al. Distributed stream clustering using micro-clusters on apache storm. *J. Parallel Distrib. Comput.*, 108:74–84, 2017.

[24] P. Kranen, I. Assent, C. Baldauf, and T. Seidl. The clustree: indexing micro-clusters for anytime stream mining. *Knowl. Inf. Syst.*, 2011.

[25] H. Kremer and et al. An effective evaluation measure for clustering on evolving data streams. In *KDD*, 2011.

[26] S. Mansalis, E. Ntoutsi, N. Pelekis, and Y. Theodoridis. An evaluation of data stream clustering algorithms. *Statistical Analysis and Data Mining*, 11(4):167–187, 2018.

[27] X. Meng and et al. Mllib: Machine learning in apache spark. *Journal of Machine Learning Research*, 17:34:1–34:7, 2016.

[28] G. D. F. Morales and A. Bifet. SAMOA: scalable advanced massive online analysis. *Journal of Machine Learning Research*, 2015.

[29] I. Ntoutsi and et al. Density-based projected clustering over high dimensional data streams. In *SIAM SDM*, 2012.

[30] E. P. Xing and et al. Petuum: A new platform for distributed machine learning on big data. In *KDD*, 2015.

[31] M. Zaharia and et al. Discretized streams: fault-tolerant streaming computation at scale. In *SOSP*, 2013.

[32] Q. Zhang, J. Liu, and W. Wang. Approximate clustering on distributed data streams. In *ICDE*, 2008.

[33] A. Zhou, F. Cao, Y. Yan, C. Sha, and X. He. Distributed data stream clustering: A fast em-based approach. In *ICDE*, 2007.