

# Scalable Online-Offline Stream Clustering in Apache Spark

Omar Backhoff  
Technische Universität München (TUM)  
Munich, Germany  
omarbackhoff@gmail.com

Eirini Ntoutsi  
Leibniz Universität Hannover  
Hanover, Germany  
ntoutsi@kbs.uni-hannover.de

**Abstract**—Two of the most popular strategies to mine big data are distributed computing and stream mining. The purpose of this thesis is to incorporate both together bringing a competitive stream clustering method into a modern framework for distributed computing, namely, Apache Spark. The method in question is CluStream, a stream clustering method which separates the clustering process into two different phases: an online phase which handles the incoming stream, generating statistical summaries of the data and an offline phase which takes those summaries to generate the final clusters. These summaries also contain valuable information which can be used for further analysis. The main goal is to adapt this method in such a framework in order to obtain a scalable stream clustering method which is open source and can be used by the Apache Spark community.

**Keywords**—big data streams; stream mining; stream clustering; CluStream

## I. INTRODUCTION

The analysis of data streams comes along with important questions: what kind of data is it? What important information is contained in it? How does the stream evolve? The key question for this project among those is the latter, i.e. dealing with the evolution of the stream, because prior to the development of the CluStream [?] method there was not an easy to answer that question as it was one of the first to tackle this issue.

Clustering is one of the main tasks in data mining, also often referred as an exploratory subtask of it. As the name implies, the objective is to find clusters, i.e., collections of objects that share common properties. One can also relate this task to unsupervised machine learning, which intends to classify data when it lacks of labels, i.e., when the data instance does not indicate to which category it belongs. The CluStream method was developed in 2003 [?] and its main purpose is to provide more information than previously developed algorithms for data stream clustering by that time. It provides a solution for handling streams of data independently from the one that finds the final clusters. It consists of two phases (passes) instead of one; the first one deals with the incoming data and stores relevant information over time and the second one is in charge of the clustering using the previously generated information.

## II. RELATED WORK

### +++ Related work on stream clustering

*CluStream* has been implemented in different types of software and libraries, being *SAMOA* - *Scalable Advanced Massive Online Analysis* one of the options. It is also a distributed computing implementation of the algorithm. The difference is that it is not implemented in *Spark*, but rather in a *Distributed Stream Processing Engine* which adapts the *MapReduce* approach to parallel stream processing [?].

Main differences with this adaptation:

- It does not include an offline macro-clustering phase.
- It is developed in *Java* and not designed to work with *Spark*.

*StreamDM* is a collection of algorithms for mining big data streams<sup>1</sup>. One of the included methods for stream clustering is *CluStream*. This collection of algorithms is developed for *Spark*.

Main differences with this adaptation:

- It does not include an offline macro-clustering phase.

## III. BASIC CONCEPTS

### A. *CluStream*

*CluStream* is a method developed in the Watson Research Center at IBM and the University of Illinois, UIUC. This method presented a different approach on the matter of clustering streams of data with respect to a modified version of *K-Means* which was adapted to work also with data streams. The main difference relies on the separation of the clustering process into two parts: one which would handle the data stream itself gathering only statistically relevant information (online part) and another which actually process the results of the former to produce the actual clusters wanted (offline part).

Separating the clustering process provides the user several advantages, among others:

- by saving only statistical data, rather than the original content, it is possible to save physical storage space (e.g. hard drive space) and therefore reducing costs and allowing a wider range in time to be clustered.
- The method also allows the analysis of the evolution of the data, as the necessary information for that is contained in the stored statistical information.

<sup>1</sup>As it is stated by them here: <http://huawei-noah.github.io/streamDM/>

- Because the two parts operate independently it allows the user to select a time horizon, or even a time window, to perform the offline clustering part using the stored statistical information.

1) *The CluStream framework*: This method is built over a few ideas that need to be conceptualized, which will answer fundamental questions and set up a basis of terminology useful along this work.

- **Micro-Clusters**: that is the given name for the statistical information summaries that is computed during the online component. They are a temporal extension of *cluster feature vectors* [?], which benefit from an additive feature that makes them a natural choice for the data stream problem [?].
- **Pyramidal time frame**: micro-clusters are stored periodically following a pyramidal pattern. This allows a nice tradeoff between the ability to store large amounts of information while giving the user the possible to work with different time horizons without losing too much precision. The statistical summaries stored are used by the offline component to compute finally the macro-clusters which are the actual clusters the user intended to get.

It is assumed that a data stream comes in the form of multi-dimensional records  $\bar{X}_1 \dots \bar{X}_k \dots$  where  $\bar{X}_i = (x_i^1 \dots x_i^d)$ .

*Definition 1*: [?]

A micro-cluster for a set of d-dimensional points  $X_{i_1} \dots X_{i_n}$  with time stamps  $T_{i_1} \dots T_{i_n}$  is defined as the  $2 \cdot d + 3$  tuple  $(CF2^x, CF1^x, CF2^t, CF1^t, n)$ , wherein  $CF2^x$  and  $CF1^x$  each correspond to a vector of d entries. The definition of each of these entries is as follows:

- For each dimension, the sum of the squares of the data values is maintained in  $CF2^x$ . Thus,  $CF2^x$  contains d values. The p-th entry of  $CF2^x$  is equal to  $\sum_{j=1}^n (x_{i_j}^p)^2$ .
- For each dimension, the sum of the data values is maintained in  $CF1^x$ . Thus,  $CF1^x$  contains d values. The p-th entry of  $CF1^x$  is equal to  $\sum_{j=1}^n x_{i_j}^p$ .
- The sum of the squares of the time stamps  $T_{i_1} \dots T_{i_n}$  is maintained in  $CF2^t$ .
- The sum of the time stamps  $T_{i_1} \dots T_{i_n}$  is maintained in  $CF1^t$ .
- The number of data points is maintained in n.

The idea behind the pyramidal time frame is that *snapshots* of the micro-clusters can be stored in an efficient way, such that if  $t_c$  is the current clock time and  $h$  is the history length, it is still possible to find an accurate approximation of the higher level clusters (macro-clusters) for a user specified time horizon  $(t_c - h, t_c)$ .

In this time frame, the snapshots are stored at different levels of granularity that depends upon the *recency* of them. These are classified in different orders which can vary from 1 to  $\log(T)$ , where  $T$  is the clock time elapsed since the beginning of the stream. The snapshots are maintained as

follows:

- A snapshot of the  $i$ -th order occur at time intervals  $\alpha^i$ , for  $\alpha \geq 1$ . In other words, a snapshot occurs when  $T \bmod \alpha^i = 0$ .
- At any given moment, only the last  $\alpha^l + 1$  snapshot for any given order are stored, where  $l \geq 1$  is a modifier which increases the accuracy of the time horizon with the cost of storing more snapshots. This allows redundancy, but from an implementation point of view only one snapshot must be kept.
- For a data stream, the maximum number of snapshots stored is  $(\alpha^l + 1) \cdot \log_\alpha(T)$ .
- For any specified time window  $h$ , it is possible to find at least one stored snapshot within  $2 \cdot h$  units of the current time<sup>2</sup>.
- The time horizon  $h$  can be approximated to a factor of  $1 + (1/\alpha^{l-1})$ , whose second summand is also referred as the accuracy of the time horizon.

Order of Snapshots	Clock Times (Last 5 Snapshots)
0	55 54 53 52 51
1	54 52 50 48 46
2	52 48 44 40 36
3	48 40 32 24 16
4	48 32 16
5	32

Figure 1: Example of snapshots stored for  $\alpha = 2$  and  $l = 2$

With the help of Figure 1 it is possible to observe how this pyramidal time frame works: snapshots of order 0 occur at odd time units, these need to be retained as are non-redundant; snapshots of order 1 which occur at time units not divisible by 4 are non-redundant and must be retained; in general, all the snapshots of order  $i$  which are not divisible by  $\alpha^{i+1}$  are non-redundant. Another thing to note is that whenever a new snapshot of a particular order is stored, the oldest one from that order needs to be deleted.

To illustrate the effect on the accuracy of storing more snapshots, the following example is given: supposing that a stream is running for 100 years, with a time granularity of 1 second. The total number of snapshots stored would be  $(2^2 + 1) \cdot \log_2(100 * 365 * 24 * 60 * 60) \approx 158$  with an accuracy of  $1/2^{2-1} = 0.5$  or 50% of a given time horizon  $h$ . Increasing the modifier  $l$  to 10 would yield to  $(2^{10} + 1) \cdot \log_2(100 * 365 * 24 * 60 * 60) \approx 32343$  maximum snapshots stored with an accuracy of  $1/2^{10-1} \approx 0.00195$  or  $\approx 0.2\%$  which is a significant improvement.

2) *Maintaining the micro-clusters*: Whenever a new point arrives, it is necessary to find its nearest micro-cluster. It is possible to calculate an average radius or *RMSD*, only to then compare the distance to the point to a factor of it: when

<sup>2</sup>The proof can be found in the original article [?].

the distance between a point and its nearest micro-cluster is smaller or equal to the average radius (of the micro-cluster in question) times a user defined factor, then this point is added to the micro-cluster. Adding a point to a micro-cluster means that the properties of the micro-cluster change, such as RMSD and size (number of points).

Whenever a point (outlier) does not fulfill the mentioned condition, then a new micro-cluster has to be created in order to give this point a chance as a potential new cluster. In order to do so, an older micro-cluster has to be deleted or two micro-clusters have to be merged. To determine which solution is appropriate a recency value for each micro-cluster has to be determined<sup>3</sup> and until all the micro-clusters which have an older recency value than a user specified parameter are deleted, it is possible to start merging the micro-clusters which are closest to one another.

3) *Offline macro-clusterig*: The macro-clustering part is done by selecting a time window and then performing a modified version of *K-Means* to cluster the center of the current micro-clusters using the size as weights.

Selecting a time window implies using two snapshots to determine what happened in such time window: the most recent snapshot is used as the base and the older one is used to determine overlapping micro-clusters. Once this is determined, a simple subtraction of their properties is performed to obtain the correct set of micro-clusters to use in the macro-clustering step.

## B. SPARK

*Apache Spark* is an open source framework developed in the AMPLab at the University of California, campus Berkeley [?]. It is a fast and general engine for large-scale data processing, as they describe it themselves. The original goal was to design a new programming model that supports a wider class of applications than MapReduce and at the same time keeping the fault tolerance property of it. They claim MapReduce is inefficient for applications that require a multi-pass implementation and a low latency data sharing across parallel operations, which are common in data analytics nowadays, such as:

- Iterative algorithms: many machine learning and graph algorithms.
- Interactive data mining: multiple queries on data loaded into RAM.
- Streaming applications: some require an aggregate state over time.

Traditionally, MapReduce and DAG engines are based on an acyclic data flow, which makes them non optimal for these applications listed above. In this flow, data has to be read from a stable storage system, like a distributed file system, and then processed on a series of jobs only to be written back to the stable storage. This process of reading

<sup>3</sup>See [?] for more details.

and writing data on each step of the workflow causes a significant rise in computational cost.

The solution proposed offers *resilient distributed datasets (RDDs)* to overcome this issue efficiently. RDDs are stored in memory between queries (no need of replication) and they can rebuild themselves in case of failure as they remember how they were originally built from other datasets by transformations such as *map*, *group*, *join*.

## C. SPARK streaming

For this project, Spark streaming plays an important role as it takes a raw data stream and transforms it so that it is possible to process it within the framework. A raw stream of data can come in different forms and through different channels: from a very simple file stream, where whenever a new file is added to a specific location it is recognized as the input, a socket stream where the data comes through the network using a TCP protocol and also integrates with more elaborated sources such as *Kafka*, *Flume*, *Twitter*, *HDFS/S3*, etc.



Figure 2: Flow of data in Spark streaming

Figure 2 shows the general idea of Spark streaming [?], a raw stream is linked to this module and it converts it to batches of data at user-defined intervals. These batches of data are then treated as RDDs, thus it gets distributed over the cluster where Spark runs. The abstraction of a data stream in Spark is called *DStream*, which stands for Discretized Stream, and is continuous series of RDDs. In a *DStream*, each RDD contains data from a specific interval of time, as it can be seen in Figure 3.



Figure 3: DStreams are Spark streaming's abstraction of a data stream

## IV. CLUSTREAM EXTENSION IN APACHE SPARK

There are some modifications which had to be done in order to adapt *CluStream* in Spark. Working with Spark means working distributed computing and, thus, the algorithm has to be able to work in parallel. Both parts (online and offline) were adapted.

### A. CluStreamOnline class (online phase)

Two processes were modified: processing the stream and updating the micro-clusters. As this adaptation uses Spark Streaming, the points coming from the stream are processed in batches at user specified time intervals. This contrasts with the original methodology which indicates to process point by point. The main difference with the batch processing method is that now points laying in current micro-clusters are processed before than the ones that are not, this also includes updating the micro-clusters before processing the points laying out. The reason for this is that as part of the strategy chosen to parallelize the algorithm, the micro-clusters are maintained locally and processing the outliers is also performed locally because deleting and merging requires to modify the micro-clusters for every point.

1) *Finding nearest micro-cluster*: The maintenance of the micro-clusters starts with this operation. After initialization (described in [?]) is performed, finding the nearest micro-clusters for all the points is the very first thing to be done for every new batch of data.

---

#### Algorithm 1 Find nearest micro-cluster.

---

**Input:** *rdd*: RDD[breeze.linalg.Vector[Double]], *mcInfo*: Array[(McI,Int)]— *rdd* is an RDD containing data points and *mcInfo* is the collection of the micro-clusters information.

**Output:** *rdd*: RDD[(Int, breeze.linalg.Vector[Double])] — returns a tuple of the point itself and the unique ID of the nearest micro-cluster.

```

1: for all  $p \in rdd$  do
2:    $minDistance \leftarrow Double.PositiveInfinity$ 
3:    $minIndex \leftarrow Int.MaxValue$ 
4:   for all  $mc \in mcInfo$  do
5:      $distance \leftarrow squaredDistance(p, mc_1.centroid)$ 
6:     if  $distance \leq minDistance$  then
7:        $minDistance \leftarrow distance$ 
8:        $minIndex \leftarrow mc_2$ 
9:     end if
10:  end for
11:   $p = (minIndex, p)$ 
12: end for return rdd

```

---

Finding the nearest micro-clusters is an operation of complexity  $O(n * q * d)$ , where  $n$  is the number of points,  $q$  the number of micro-clusters and  $d$  the dimension of the points;  $q$  and  $d$  remain constant during runtime but  $n$  might vary. Algorithm 1 describes a simple search for the minimum distance for every point in the RDD to the micro-clusters. This is also a good opportunity to show how this works using Spark and Scala:

```

1 def assignToMicroCluster(rdd: RDD[Vector[Double]],
2   mcInfo: Array[(MicroClusterInfo, Int)]): RDD
3   [(Int, Vector[Double])] = {
4   rdd.map { a =>
5     var minDist = Double.PositiveInfinity
6     var minIndex = Int.MaxValue
7     for (mc <- mcInfo) {
8       val dist = squaredDistance(a, mc._1.
9         centroid)
10      if (dist < minDist) {
11        minDist = dist
12        minIndex = mc._2
13      }
14    }
15    (minIndex, a)
16  }
17 }

```

---

Scala allows operating the elements of a collection, e.g. an array, through a map operation. The code above represents the actual function in the source code of this project that does what Algorithm 1 says for the initialization process. The variable *mcInfo* contains a summarized information taken from the micro-clusters; this variable is broadcasted to always have updated information for each batch. In this programming language, the last line defines what the function returns; it can be seen in line 2 of the mentioned function that there is actually only one instruction called, *map*, requires a function to be passed, which at the same time receives as argument each element *a* of a collection, in this case the collection is *rdd*, and returns anything resulting from that function which will replace *a*. In other words, it is possible to operate and change every element of a collection through *map* with a specified function and get a new "updated" collection in return.

To find the nearest micro-cluster of a point, in this case *a*, an iterative process is used, which computes the squared distance from *a* to all micro-clusters' centroids and stores only smaller distances and the unique ID of such so-far-nearest micro-cluster. When the iteration finishes, the function returns the tuple (*minIndex*, *a*) which replaces the element *a*.

Spark uses this *map* operation to serialize the function passed so that all nodes in the cluster get the same instruction, this is exactly how computations are parallelized within this framework. At this point, every node performs this operation to find the nearest micro-cluster for all the points they locally have.

2) *Processing points*: The points are separated in two: points within micro-clusters and outliers, it is possible to compute the necessary information from them to update the micro-clusters. It is important to perform this step before handling the outliers because this adaptation process the points for batches of data and not points individually as they arrive, and the reasons are:

- Every point for a batch is treated equally in terms of temporal properties. A batch of points gets distributed among the cluster and they all get the same time stamp. As the points are processed in parallel, and there is

no constant communication among nodes, there is no reason to assume that a point is older or newer within that batch because then race conditions would occur resulting in unpredictable results.

- Taking into account the previous statement, the process of handling outliers involve deleting and merging micro-clusters and not processing the points first would lead to invalid assignments as some micro-clusters might not exist anymore or might have changed while being merged.

These two points are some of the key differences between the original *CluStream* method and this adaptation. For the original, it is possible to handle point by point as each have different clear time stamps.

Processing the points means two things: to compute the values required to update the micro-clusters, i.e. the *cluster feature vector*, and actually updating the micro-clusters. This is another good opportunity to show a piece of code because it will be important for the performance analysis later on, and also to show how it is possible to reduce the amount of code needed for such operation. First, computing the *cluster feature vector*:

---

```

1  val aggregateFunction =
2    (aa: (Vector[Double], Vector[Double], Long),
3     bb: (Vector[Double], Vector[Double], Long)) =>
4    (aa._1 :+ bb._1, aa._2 :+ bb._2, aa._3 + bb._3)
5
6  val sumsAndSumsSquares =
7    dataIn.mapValues(v => (v, v :* v, 1L))
8    .reduceByKey(aggregateFunction).collect()

```

---

First a function called *aggregateFunction* is defined, and this is a generic function that can get passed as an argument to another function. It has the purpose of taking any two given values, in this case two given tuples, and return a tuple that contains the sum of every element of one tuple with the respective element of the other tuple:

$$aggregateFunction = ((v_1, u_1, k_1), (v_2, u_2, k_2)) => (v_1 + v_2, u_1 + u_2, k_1 + k_2)$$

From the assignment process, the points that are going to be processed are located in *dataIn*, which looks as follows:

$$dataIn = \{(id_1, p_1), (id_2, p_2), \dots\},$$

where,  $id_i$  is the unique identifier of the micro-cluster the point  $p_i$  belongs to. Then a *map* is performed only on the "values" of the tuple, considering that Spark can interpret tuples as (*key, value*) pairs, to replace each point  $p_i$  with a tuple containing  $p_i$ , squared elements of  $p_i$  and the *Long* value of 1:

$$dataIn.mapValues(v => (v, v * v, 1)) = \{(id_1, (p_1, p_1^T I p_1, 1)), (id_2, (p_2, p_2^T I p_2, 1)), \dots\}$$

To square the elements means to element-wise square the values of a vector  $v$ , therefore this is represented by  $v^T I v$ . This is done in order to perform a *reduceByKey* operation, which is how Spark combines and operates the distributed elements of an RDD:

$$dataIn.mapValues(...).reduceByKey(aggregateFunction) = \{(id_1, (p_{1,1} + p_{1,2} + \dots, p_{1,1}^T I p_{1,1} + p_{1,2}^T I p_{1,2} + \dots, 1 + 1 + \dots)), \dots\}$$

The (*key, value*) pairs are important here because then all the points belonging to the same micro-cluster are "reduced" together, resulting in tuples containing the element-wise sum, square sum and count of points:  $(id, (\overline{CF1}_{new}^x, \overline{CF2}_{new}^x, N_{new}))$ . After having computed the values to update the *cluster feature vector* of the micro-clusters which get new points.

3) *Handling outliers*: First the micro-clusters which are safe to delete are determined, then the outliers can be handled. The first thing that happens in this operation, is to check whether an outlier can be absorbed by a newly created micro-cluster as a result from other outlier, this compensates an issue which batch processing brings: if this does not happen, then equal (or extremely near) points would create a new micro-cluster of their own, not resembling the behavior of the original method. The first outlier skips this step simply because the array *newMicroClusters* is empty, and only grows every time a new micro-cluster is created. In general, there are three possible scenarios:

- If the point lies within the restriction regarding the RMSD for its nearest micro-cluster in *newMicroClusters*, the point is added to it.
- If the point does not lie within any of the new micro-clusters, then it replaces a micro-cluster from the *safeDelete* array, assuming there are safe-to-delete micro-clusters. This is done until every safe-to-delete micro-cluster is deleted. There is no further method to prioritize deletions.
- If none of the previous scenarios are viable, then the two micro-clusters that are closest to each other get merged, freeing one spot to create the new micro-cluster. This is the the most computationally expensive scenario. The function *getTwoClosestMicroClusters()* has a complexity of  $O(p_m d \cdot \frac{q!}{2!(q-2)!})$ , where  $p_m$  is the number of outliers that require a merge,  $d$  the dimension of the points, and  $q$  the number of micro-clusters.

It is important in the procedure described in Algorithm 2 to locally update the *mcInfo* every time a point is added to

---

**Algorithm 2** handle outliers.

---

```
1:  $j \leftarrow 0$ 
2: for all  $p \in dataOut$  do
3:    $distance, mcID \leftarrow$ 
      $getMinDistanceFromIDs(newMicroClusters, p_2)$ 
4:   if  $distance < t * mcInfo[mcID]_1.rmsd$  then
5:      $addPointToMicroCluster(mcID, p_2)$ 
6:   else
7:     if  $safeDelete[j].isDefined$  then
8:        $replaceMicroCluster(safeDelete[j], p_2)$ 
9:        $newMicroClusters.append(j)$ 
10:     $j \leftarrow j + 1$ 
11:   else
12:      $index1, index2 \leftarrow$ 
        $getTwoClosestMicroClusters(keepOrMerge)$ 
13:      $mergeMicroClusters(index1, index2)$ 
14:      $replaceMicroClusters(index2, p_2)$ 
15:      $newMicroClusters.append(j)$ 
16:      $j \leftarrow j + 1$ 
17:   end if
18: end if
19: end for
```

---

a micro-cluster, two micro clusters are merged and when a new micro-cluster is created. There could be a lot of change, depending on the outliers, and this loop requires up-to-date information for each iteration, otherwise merges and the RMSD check would be inaccurate.

### B. CluStream class (offline phase)

Using a *weighted K-Means* approach, as described in [?] was not directly possible, and for that reason, a new adaptation had to be done in order to achieve similar results.

1) *The fakeKMeans solution:* The original *CluStream* method suggests to use a slightly modified version of K-Means, a version for which one can initialize the seeds (initial clusters) by sampling from the micro-clusters' centroids taking into account the number of points each micro-cluster has and for which one can use these centroids as weighted input points. These weights, again, are related to the number of points they absorbed. Spark's (current) implementation of K-Means does allow to initialize the seeds but unfortunately it is not possible to predefine the weights for the input points.

In order to solve this issue, a new version of K-Means needs to be implemented. This version uses, in fact, Spark's own version, but to overcome the problem of not being able to define the weights at the beginning, this new version uses as input many points sampled from the micro-clusters' centroids. Algorithm 3 shows this procedure. Remarks on the *fakeKMeans* algorithm:

- The *getCentersFromMC()* function returns an array with the centroids of the micro-clusters as follows: for

---

**Algorithm 3** The fakeKMeans algorithm.

---

**Input:**  $sc$ : *SparkContext*,  $k$ : *Int*,  $n$ : *Int*,  $mcs$ : *Array[MicroCluster]* —  $sc$  is the Spark Context in which the clustering is performed,  $k$  is the number of macro-clusters,  $n$  is the number of points to be sampled and  $mcs$  is the array of micro-clusters.

**Output:**  $model$  : *KMeansModel* — returns the K-Means model resulting from the clustering process. This model is default to Spark.

```
1:  $centers \leftarrow getCentersFromMC(mcs)$ 
2:  $weights \leftarrow getWeightsFromMC(mcs)$ 
3:  $points \leftarrow$ 
    $\{sample(centers, weights)_1, \dots, sample(centers, weights)_n\}$ 
4:  $initialClusters \leftarrow$ 
    $\{sample(centers, weights)_1, \dots, sample(centers, weights)_k\}$ 
5:  $KMeans.setK(k)$ 
6:  $KMeans.setInitialModel(initialClusters)$ 
7:  $trainingSet \leftarrow sc.parallelize(points)$ 
8:  $model \leftarrow KMeans.run(trainingSet)$ 
   return  $model$ 
```

---

each micro-cluser the operation  $\frac{1}{N} \overline{CF1^x}$  is performed, where  $N$  is the number of points of the micro-cluster in question.

- The *getWeightsFromMC()* function returns an array with the weights of the micro-clusters as follows: for each micro-cluser the operation  $\frac{N}{totalPoints}$  is performed, where  $N$  is the number of points of the micro-cluster in question and *totalPoints* is the sum of all  $N$ 's. By doing this, a frequency distribution is generated.
- The *sample()* function takes the centroids and their weights to randomly sample centroids for the given frequency distribution: the more points a micro-cluster has, the more likely its centroid will be sampled, as shown in Figure 4.

## V. EXPERIMENTS

### A. Experiments setting

For the experiments we used the Network Intrusion dataset<sup>4</sup>, which consists of 494,021 instances. For the analysis, we used only the numerical attributes (#34 out of #43 attributes). We vary the speed of the stream and the horizon and we derive two different stream configurations. The first one, denoted as *DS1*, has a speed  $v = 2,000$  points per timestamp, which implies that it lasts for  $\frac{494,021}{2,000} \approx 247$  time units, and a horizon  $H = 1$ . The second one, denoted as *DS2*, has a speed of  $v = 200$  points per timestamp, thus lasting for a period of  $\frac{494,021}{200} \approx 2470$  time points, and a horizon  $H = 256$ . To evaluate the clustering quality,

<sup>4</sup>Source: <http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html>



Centroid→	C <sub>1</sub>	C <sub>2</sub>	C <sub>3</sub>	C <sub>4</sub>	
N →	10	6	3	1	Total points = 20

Weights = {10/20, 6/20, 3/20, 1/20} = {0.5, 0.3, 0.15, 0.05}

CDF = {0.5, 0.8, 0.95, 1}

Random numbers = 0.63, 0.45, 0.11, 0.92, 0.32

- 1.-  $0.63 \leq 0.5$ ? No,  $0.65 \leq 0.8$ ? Yes → Take C<sub>2</sub>
- 2.-  $0.45 \leq 0.5$ ? Yes → Take C<sub>1</sub>
- 3.-  $0.11 \leq 0.5$ ? Yes → Take C<sub>1</sub>
- 4.- ...,  $0.92 \leq 0.8$ ? No,  $0.92 \leq 0.95$ ? Yes → Take C<sub>3</sub>
- 5.-  $0.32 \leq 0.5$ ? Yes → Take C<sub>1</sub>

Result = {C<sub>2</sub>, C<sub>1</sub>, C<sub>1</sub>, C<sub>3</sub>, C<sub>1</sub>, ... }

Figure 4: Demonstration: sampling the centroids from weights

we report in Section V-B on the sum of square distances (SSQ) from the points to their nearest micro-cluster, using Euclidean distance as the distance function, within a horizon  $H$ . Efficiency and scalability is discussed in Section V-C.

### B. Clustering quality

We first compare *CluStream-Spark* to the original *CluStream* (Section V-B1) and then against other stream clustering approaches in Spark (Section V-B2).

#### 1) Quality of Spark – CluStream vs original CluStream:

*Results for DS1:* We used the same parameters as in [?], i.e.,  $\alpha = 2, l = 10, InitNumber = 2,000, \delta = 512, t = 2$ . The parameter  $m$ , for  $m$  last points which is used to estimate the recency of a microcluster, was the only one not provided, we set it to  $m = 20$ . For *DS1*, both  $m$  and  $\delta$  are irrelevant as the threshold is never reached (247 time units vs. 512). The number of micro-clusters was set to  $q = 50$ , which is 10 times the number of final clusters (5).

In Table I we show the SSQ scores for our *CluStream-Spark* and the results from *CluStream* as reported in [?] - these are approximate results “extracted” from the original paper. Although we don’t know the exact values for *CluStream*, we can see that the magnitudes of the average SSQ are comparable.

*Results for DS2:* Parameters were set as for *DS1*. The exact scores of *CluStream-Spark* and the approximate values from *CluStream* are reported in Table II. Again the quality scores are comparable, i.e., our *CluStream-Spark* achieves similar quality clusterings to the original *CluStream*.

2) *Spark-CluStream vs other clustering approaches in SPARK:* We compare our *CluStream-Spark* against available solutions for stream clustering in Spark and in particular

against *Streaming K-Means*<sup>5</sup> and *StreamDM-CluStream*<sup>6</sup>. We report here on their clustering quality, the efficiency issue is discussed in Section V-C.

*Results on DS1:* For the *DS1* stream the results are shown in Figure 5. For *Streaming K-Means*, the horizon  $H = 1$  is transformed to *halfLife* = 1,000 points, as the stream speed is 2,000 points. The *decayFactor* is set to 0, i.e., only the last 2,000 points will influence on the clusters. *StreamDM-CluStream* is set up with its default parameters, only changing the horizon to 1 and the number of micro-clusters to 50 to match those of *CluStream-Spark*. As we can

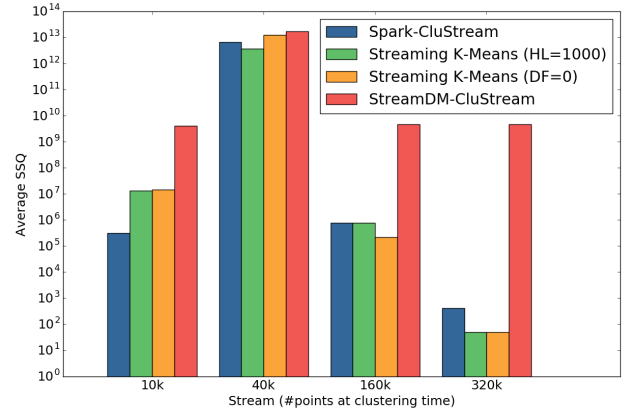


Figure 5: Average SSQ for different stream clustering methods in SPARK. *DS1* (Stream speed  $v = 2,000, H=1$ )

see, our *CluStream-Spark* delivers results which are very close to those of *Streaming K-Means*, whereas *Streaming K-Means* with the *decayFactor* (DF) is the best. The surprising results come from *StreamDM-CluStream*, which is the worse among the tested methods, especially for the last two points, i.e., at 160k and 320k. To further investigate this, we conduct another experiment where we compare *CluStream-Spark* without snapshots against *StreamDM-CluStream*. The results are shown in Figure 6. As we can see, *CluStream-Spark* outperforms *StreamDM-CluStream* even if we remove the snapshot part, but the quality is lower comparing to the snapshot version.

*Results on DS2:* For the *DS2* stream the results are shown in Figure 7. All parameters remained the same for all methods, except for the *halfLife* parameter for *Streaming K-Means*, which is set to *halfLife* = 25,600 ( $200 \cdot 256 = 51,200$ ). We calculate the *decayFactor* as follows: At 150,000 points, the ratio of the points to cluster to the total number of points at that particular time is  $\frac{51,200}{150,000} \approx 0.3413$ . At 250,000 points, this equals to  $\frac{51,200}{250,000} \approx 0.2048$ . At 350,000 points, this equals to  $\frac{51,200}{350,000} \approx 0.1462$ . At

<sup>5</sup>More information: <https://databricks.com/blog/2015/01/28/introducing-streaming-k-means-in-spark-1-2.html>

<sup>6</sup>More information: <http://huawei-noah.github.io/streamDM/>

<i>DS1 - avg SSQ</i>	<b>10k</b>	<b>40k</b>	<b>160k</b>	<b>320k</b>
<i>CluStream</i>	$10^5$ - $10^6$	$10^{12}$ - $10^{13}$	$\approx 10^6$	$10^2$ - $10^3$
<i>Spark - CluStream</i>	$3.099 \times 10^5$	$6.676 \times 10^{12}$	$7.833 \times 10^5$	$4.191 \times 10^2$

Table I: *DS1* - Average SSQ values

<i>DS2 - avg SSQ</i>	<b>150k</b>	<b>250k</b>	<b>350k</b>	<b>450k</b>
<i>CluStream</i>	$10^{13}$ - $10^{14}$	$\approx 10^5$	$10^{12}$ - $10^{13}$	$\approx 10^8$
<i>Spark-CluStream</i>	$5.402 \times 10^{13}$	$5.143 \times 10^4$	$1.892 \times 10^{13}$	$9.646 \times 10^7$

Table II: *DS2* - Average SSQ values

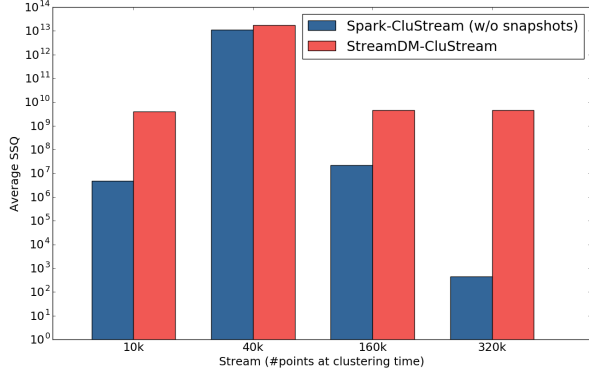


Figure 6: *CluStream-Spark* without snapshots vs *StreamDM-CluStream*. *DS1* (Stream speed  $v = 2,000$ ,  $H = 1$ ,  $m = 100$ ).

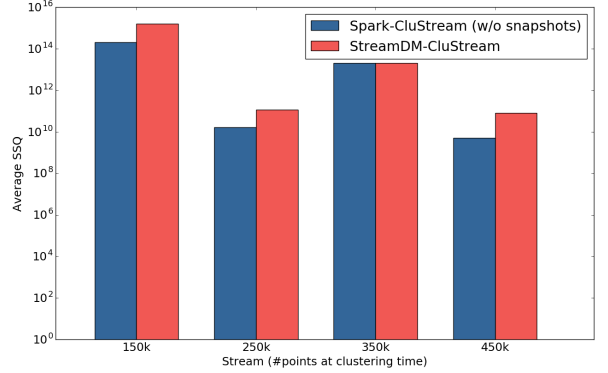


Figure 8: *CluStream-Spark* without snapshots vs *StreamDM-CluStream*. *DS2* (Stream speed  $v = 200$ ,  $H = 256$ ,  $m = 100$ ).

450,000 points, this equals to  $\frac{51,200}{450,000} \approx 0.1137$ . Averaging those ratios leads to a  $decayFactor = 0.2015$ . As we can

indicate that *StreamDM-CluStream* benefits from larger horizons.

### C. Scalability

@Omar: Describe how you generated the data We test the scalability with respect to data dimensionality and number of microclusters.

The scalability tests are performed in two different scenarios: one being an analysis of how it scales for different number of attributes (dimensions of the data points) using only 20 micro-clusters and the other one using 200 micro-clusters. The reason behind this is that the number of attributes and the number of final clusters for a specific purpose are two key factors which determine the complexity of *Spark-CluStream*. The speed of the stream is controlled for 10000 points for every batch of data because it is easier to test the scalability when many computations have to be done.

Any application using Spark streaming assigns one core exclusively to handle the stream, therefore the minimum number of processors required is two, this also means that using 2 processors is equivalent to using a single processor to execute the application. The number of processors mentioned in these tests is the total, but the real number of processors used for the computations is that number minus

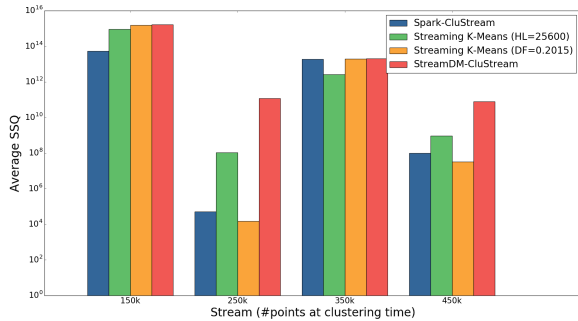


Figure 7: Average SSQ for different stream clustering methods in SPARK. *DS2* (Stream speed  $v = 200$ ,  $H=256$ )

see, *CluStream-Spark* performs consistently well and better than *StreamDM-CluStream*. As expected *Streaming K-Means* with the  $decayFactor$  achieves the best performance.

We repeat the without-snapshot experiment, the results are shown in Figure 8. As we can see, *CluStream-Spark* delivers better results than *StreamDM-CluStream* but the difference is reduced significantly comparing to *DS1*. These results might



one.

The charts here presented show the speedup obtained by increasing the number of processors from 2 to 40, which in reality means that 1 to 39 processors were used for the computations. It also shows the average processing time for each batch of data. Because the initialization takes the most amount of time, it is also convenient to show these values without considering that process: by doing so it is possible to see what would be the expected results for a longer run, where the initialization is no longer dominant. Finally it shows the interval time for which Spark process a new batch of data, in particular all these tests processed batch every second.

Figure 9 shows that using only 20 micro-clusters and 2 dimensions has poor scalability, not even being able to perform twice as fast as for a single processor (2 in total). Even for this high speed streaming, one processor is enough to process the batches of data before a new batch is processed, meaning that the setup is stable.

Increasing the dimensionality of the points increases the computational effort needed to process the points in every batch of data and here is where *Spark-CluStream* shows its scalability, which is almost linear<sup>7</sup> for up to 16-17 processors, as it can be seen in Figure 10. From the average processing time per batch, it can be seen that from 32 to 40 processors it does not improve much anymore and the speedup does not increase quasi-linearly anymore. Here a total of 9 processors were required to stabilize *Spark-CluStream*.

Interestingly, increasing the number of micro-clusters by a factor of 10 for 2 attributes resulted in good scalability, similarly to the scenario with 20 micro-clusters and 100 attributes. Here a total of 8 processors were enough for a stable run, as shown in Figure 12.

Finally, when the number of clusters and the number of attributes are both increased significantly, Figure 13 shows for *Spark-CluStream* quasi-linear scalability but this time only up to about 8-9 processors. After that point, the speedup slows down showing almost no improvement after 16 processors. This test never reached a stable configuration.

#### D. Performance

In this section, the scalability of *Spark-CluStream* is compared to that of *StreamDM-CluStream* and Spark's *Streaming K-Means* using the Spark cluster setup for  $q = 20$  and  $d = 2, 100$ , for the *CluStream* method. Also, a test on a single machine is performed, using the setup and dataset as in ??.

In Figure 15 it can be seen that *Spark-CluStream* took the most time on average to process a batch of data and being *Streaming K-Means* the fastest among the three.

<sup>7</sup>By linear scalability does not mean it scales with a 1 to 1 ratio, but rather linearly proportional.

When it comes to higher dimensions, *Spark-CluStream* shows a significant improvement over *StreamDM-CluStream*, which never got to the point where it was stable (below the 1 second mark), as shown in 16, it seems to scale as fast as *Spark-CluStream* but it was not enough even with 40 processors.

Surprisingly, in Figure 17, *StreamDM-CluStream* shows to be able to scale even for this tests, while both *Spark-CluStream* and *Streaming K-Means* seem to struggle taking advantage of using more processors.

Figure 18 shows that all three algorithms are able to scale similarly for this test, being *Spark-CluStream* the one having a very slight advantage as it does not slow down as quickly as the other two.

Another interesting comparison, is the processing time per batch of data for a single machine, using a real dataset such as the *Network Intrusion*. Here, communication is less of an issue as all the partitions lie in the same shared memory space, and still there are 4 virtual cores in disposition for the algorithms to run.

The test was performed using a stream speed of 2000 points per batch and with a horizon  $H = 1$ , to match one of the validation tests.

The results shown in Figure 19 are quite remarkable. As *StreamDM-CluStream* shows a very significant disadvantage when using greater numbers of micro-clusters and higher dimensions.

For this single machine test, *Spark-CluStream* was about 18 times faster on average than *StreamDM-CluStream* and about two times slower than *Streaming K-Means* on average.

Another consideration to be made, is that *Spark-CluStream* saves a snapshot for every batch of data, having to write to disk, while the other two algorithms never access the disk on this matter.

## VI. CONCLUSIONS

We proposed a variation of *CluStream* tailored to Apache Spark, *CluStream-Spark*. Our experiments show that *CluStream-Spark* achieves similar quality to the original approach [?], while it is far more efficient. Comparing to other stream clustering approaches in Spark, our results show that *Streaming K-Means* is the fastest algorithm among the three tested (highly optimized for Spark), but it does not offer the flexibility of the online-offline clustering approaches like *CluStream* that better fit evolving data streams. Comparing *CluStream-Spark* to *StreamDM-CluStream*, *CluStream-Spark* delivers more consistent and accurate results, and outperforms *StreamDM-CluStream* in most cases, including one up to around 18 times faster.

As part of our future work, we will focus on minimizing the communication cost between the different nodes to further improve the efficiency and also on dealing with outliers to improve quality.

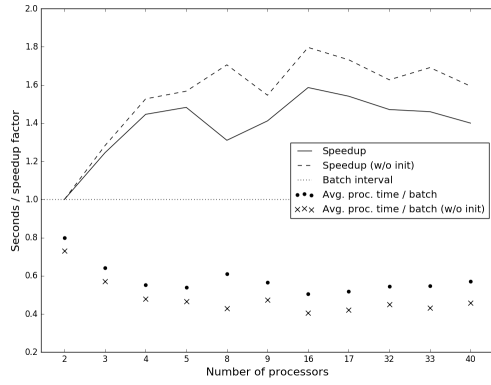


Figure 9: Dimension:  $d = 2$

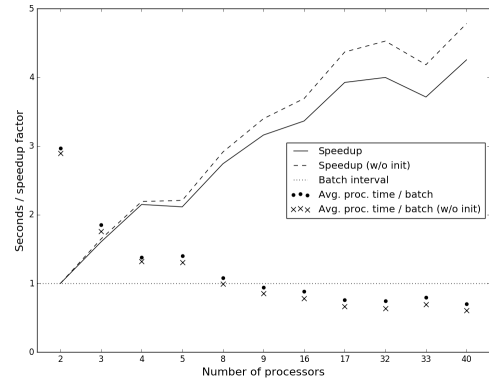


Figure 10: Dimension:  $d = 100$

Scalability-dimensionality comparison for Stream speed = 10000 and  $q = 20$

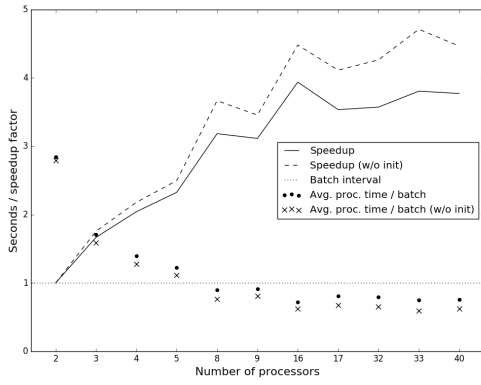


Figure 12: Dimension:  $d = 2$

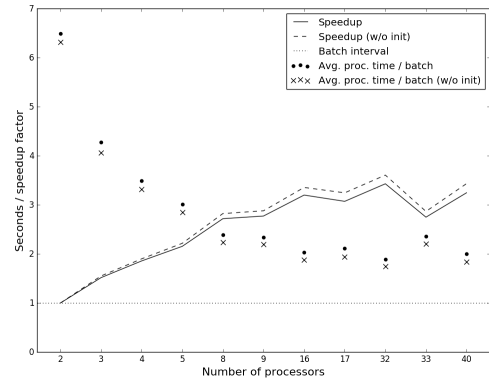


Figure 13: Dimension:  $d = 100$

Scalability-dimensionality comparison for Stream speed = 10000 and  $q = 200$

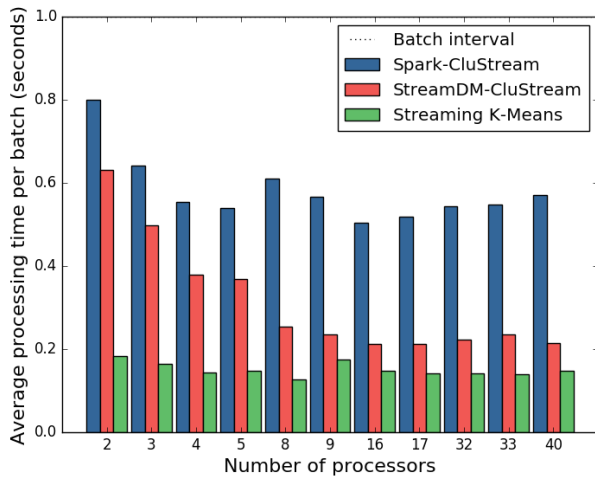


Figure 15: Processing time comparison:  $q = 20$ ,  $d = 2$

## REFERENCES

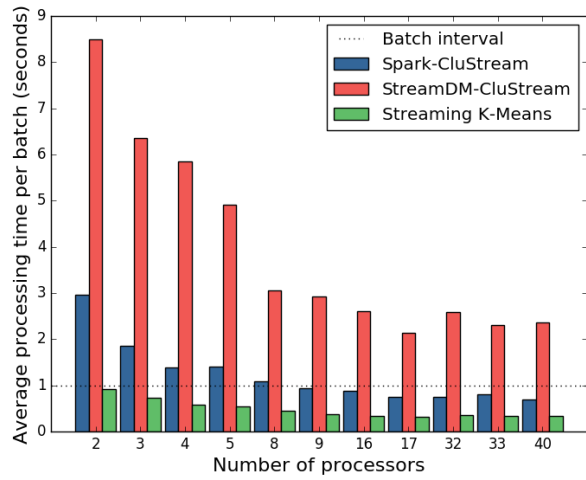


Figure 16: Processing time comparison:  $q = 20$ ,  $d = 100$

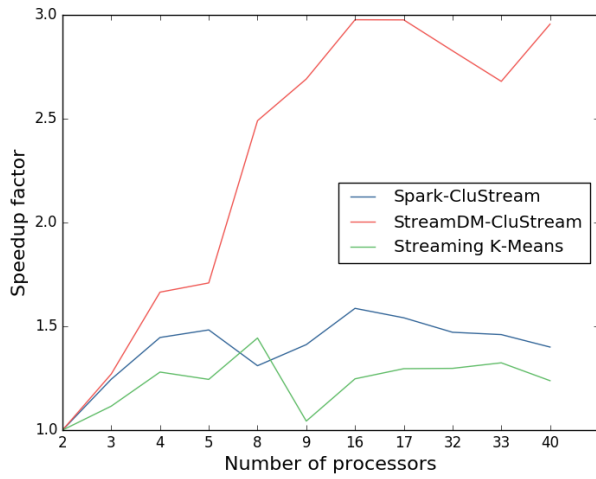


Figure 17: Scalability comparison:  $q = 20$ ,  $d = 2$

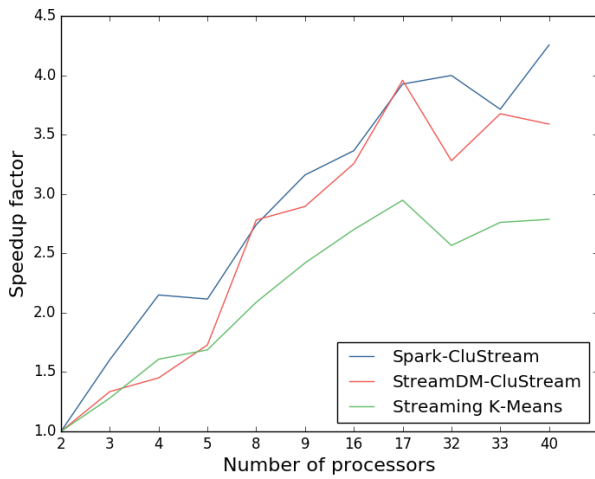


Figure 18: Processing time comparison:  $q = 20$ ,  $d = 100$

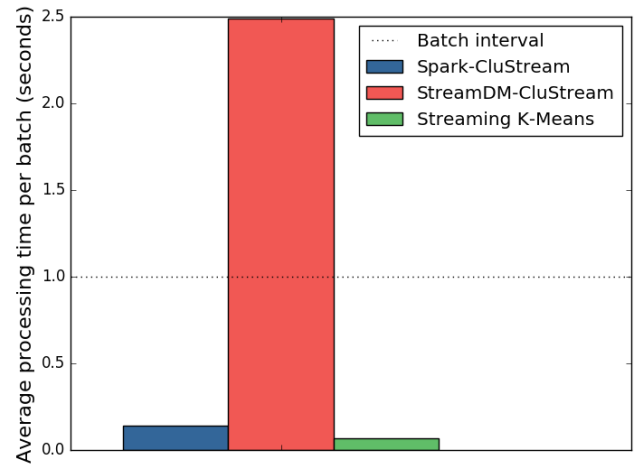


Figure 19: Processing time comparison for a single machine:  $q = 50$ ,  $d = 34$