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 comput- ing 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 CluS- tream, 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 pro- vide more information than previously developed algorithms for data stream clus- tering by that time. It provides a solution for handling streams of data indepen- dently 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 [1]. 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.

[1] 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 loosing 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 $\bar{X}_1 ... \bar{X}_k ...$ where $\bar{X}_i = (x_i^1 ... x_i^d)$.

Definition 1: [**?**]

A micro-cluster for a set of d-dimensional points $X_{i_1} ... X_{i_n}$ with time stamps $T_{i_1} ... T_{i_n}$ is defined as the $2 \cdot d + 3)$ tuple $(\overline{CF2^x}, \overline{CF1^x}, CF2^t, CF1^t, n)$, wherein $\overline{CF2^x}$ and $\overline{CF1^x}$ 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 $\overline{CF2^x}$. Thus, $\overline{CF2^x}$ contains d values. The p-th entry of $\overline{CF2^x}$ is equal to $\sum_{j=1}^{n} (x_{ij}^p)^2$.
- For each dimension, the sum of the data values is maintained in $\overline{CF1^x}$. Thus, $\overline{CF1^x}$ contains d values. The p-th entry of $\overline{CF1^x}$ is equal to $\sum_{j=1}^{n} x_{ij}^p$.
- The sum of the squares of the time stamps $T_{i_1} ... T_{i_n}$ is maintained in $CF2^t$.
- The sum of the time stamps $T_{i_1} ... T_{i_n}$ 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 α^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[2].
- 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 α^{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 $(2^2 + 1) \cdot log_\alpha(100 * 365 * 24 * 60 * 60) \approx 158$ 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 $(2^{10} + 1) \cdot log_\alpha(100 * 365 * 24 * 60 * 60) \approx 32343$ 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 radious or *RMSD*, only to

[2]The proof can be found in the original article [**?**].

then compare the distance to the point to a factor of it: when the distance between a point and its nearest micro-cluster is smaller or equal to the average radiuos (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[3] 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 sustraction 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 sate 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 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.

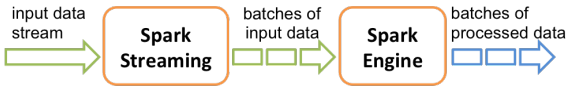

Figure 2. Flow of data in Spark streaming

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.

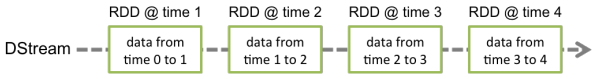

Figure 3. DStreams are Spark streaming's abstraction of a data stream

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 worrking distributed computing and, thus, the algorithm has to be able to work in parallel. Both parts (online and offline) were adapted.

[3]See [**?**] for more details.

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

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, and that is $rdd.map\{a => function(a)\}$. This operation, *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' cetroids 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 assignations 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 assignation process, the points that are going to be processed are located in $dataIn$, which looks as follows:

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

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:

$$dataIn.mapValues(v => (v, v * v, 1)) = \\ \{(id_1, (p_1, p_1^T Ip_1, 1)), (id_2, (p_2, p_2^T Ip_2, 1)), ...\}$$

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

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

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: $(id, (\overline{CF1^x}_{new}, \overline{CF2^x}_{new}, N_{new}))$. 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 mirco-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.

$j \leftarrow 0$
for all $p \in dataOut$ **do**
  $distance, mcID \leftarrow getMinDistanceFromIDs(newMicroClusters, p_2)$
  if $distance < t * mcInfo[mcID]_1.rmsd$ **then**
    $addPointToMicroCluster(mcID, p_2)$
  else
    if $safeDelete[j].isDefined$ **then**
      $replaceMicroCluster(safeDelete[j], p_2)$
      $newMicroClusters.append(j)$
      $j \leftarrow j + 1$
    else
      $index1, index2 \leftarrow getTwoClosestMicroClusters(keepOrMerge)$
      $mergeMicroClusters(index1, index2)$
      $replaceMicroClusters(index2, p_2)$
      $newMicroClusters.append(j)$
      $j \leftarrow j + 1$
    end if
  end if
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 achive 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 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.

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.

$centers \leftarrow getCentersFromMC(mcs)$
$weights \leftarrow getWeightsFromMC(mcs))$
$points \leftarrow \{sample(centers, weights)_1, ..., sample(centers, weights)_n\}$
$initialClusters \leftarrow \{sample(centers, weights)_1, ..., sample(centers, weights)_k\}$
$KMeans.setK(k)$
$KMeans.setInitialModel(initialClusters)$
$trainingSet \leftarrow sc.parallelize(points)$
$model \leftarrow KMeans.run(trainingSet)$
return $model$

V. Experiments

A. Experiments setting

For the experiments we used the Network Intrusion dataset [4], 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 $v = 2,0000$ points per timestamp and a horizon $H = 1$. This implies that the stream lasts for $\frac{494{,}021}{2{,}000} \approx 247$ time units. The second one, denoted as $DS2$, has a speed of $v = 200$ points per timestamp and a horizon $H = 256$. Therefore the stream lasts for a period of $\frac{494021}{200} \approx 2470$ time points.

[4] @Omar: Add a link to the dataset

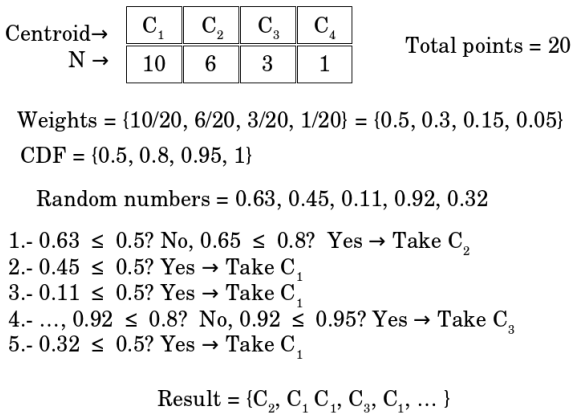

Figure 4. Demonstration: sampling the centroids from weights

To evaluate the clustering quality, we report in Section V-B on the sum of squares distance (SSQ) from the points to their nearest micro-cluster, using Euclidean distance as the distance function, within a horizon H.

With respect to the efficiency aspect, we report on +++, in Section V-C.

B. Clustering quality

1) Quality of $Spark - CluStream$ vs original $CluStream$: The SSQ for $DS1$ is shown in Figure 7 for the original $CluStream$ and our $Spark - CluStream$. We used the same parameters as in [**?**], i.e., $\alpha = 2, l = 10, InitNumber = 2000, \delta = 512, t = 2$. The parameter m, for m last points, was the only one not provided, we set it to $m = 20$. For $DS1$, both m and δ are irrelevant and the reason is that the threshold is never reached (247 time units vs. 512). The number of micro-clusters was set to $q = 50$, 10 times the number of final clusters (5). @Omar: they suggest 50 and 5 in the original paper?. Also, *fakeKMeans()* used 5000 sampled points.

For the original $CluStream$ we show the results from the original paper [**?**] - in Figure 5 we also see the performance of the STREAM algorithm, a modified version of k-Means for data streams [**?**] - the only relevant for the current work is $CluStream$.

Figure 6 shows the results obtained by $Spark - CluStream$. There is a difference in the labels of the horizontal axis; while Figure 5 shows the time units of the stream, Figure 5 shows the number of points that had been streamed and processed. The reason is that Spark and the simulated stream do not deliver the same amount of points for each time unit. A basic multiplication was used to determine the exact moment in terms of points: $2000 \cdot 5 = 10000$, $2000 \cdot 20 = 40000$ and so on.I dont understand this sentence Comparing the results though, it is possible to deduce that they are very similar. We don't know the exact values for Figure 5 but it suffices to compare the magnitudes of the average SSQ. The exact SSQ scores for $Spark - CluStream$ and the approximated ones from $CluStream$ based on [**?**] are shown in Table I.

@Omar: Can you change the y-axis labels in the second image to match the format of the first? Also, can you change the x-axis label of the second image to "Stream (#points at clustering time))

The SSQ for $DS2$ is shown in Figure 10. Parameters were set as for $DS1$.

The test ran 4 times for *Spark-CluStream* to average the results. Same for DS1? You run the streams x4 and reported the avg values over the 4 runs?

The results of $CluStream$ and $Spark - CluStream$ are shown in Figure 10; again, they perform similarly. The exact SSQ scores for $Spark - CluStream$ and the approximated ones from $CluStream$ based on [**?**] are shown in Table II.

2) $Spark - CluStream$ vs other clustering approaches in SPARK: We compare our $Spark - CluStream$ against available solutions for stream clustering in Spark and in particular against i)*Streaming K-Means* from Spark and *StreamDM-CluStream*, which is another adaptation of *CluStream* for Spark. We report here on their clustering quality, the efficiency issue is discussed in Section **??**.

We roughly overview these methods hereafter.

- *Streaming K-Means* [5]:
  - In order to have comparable results, the time horizon H must be interpreted differently. There are two strategies: the first option is to use the parameter *halfLife*, which can be configured to let the algorithm to completely adjust the clusters after HL points or batches.
  - The alternative would be to set the $decayFactor$, which sets the weight for the clusters of the "old" data (only the current batch is considered "new" data). This is a number between 0 and 1, such that if it is 0 then only the clusters for "new" data determine the final clusters, if it is set to 1, then the clusters of past data will have the same influence on the final clusters. It is important to notice that this *decayFactor* also considers the number of points of the "new" and "old" data, so in the last case, after a long time, "new" data will have little influence as the number of points of the current batch will be considerable smaller than the points clustered so far.
- *StreamDM-CluStream* [6]:
  - This adaptation of *CluStream* does not include the offline part as a separate module, meaning that it does not save snapshots and therefore it has to perform the macro-clustering process for every

[5]Add reference or link
[6]Add reference or link

$DS1$ - **avg SSQ**	**10k**	**40k**	**160k**	**320k**
$CluStream$	10^5-10^6	10^{12}-10^{13}	$\approx 10^6$	10^2-10^3
$Spark - CluStream$	3.099×10^5	6.676×10^{12}	7.833×10^5	4.191×10^2

Table I
$DS1$ - AVERAGE SSQ VALUES

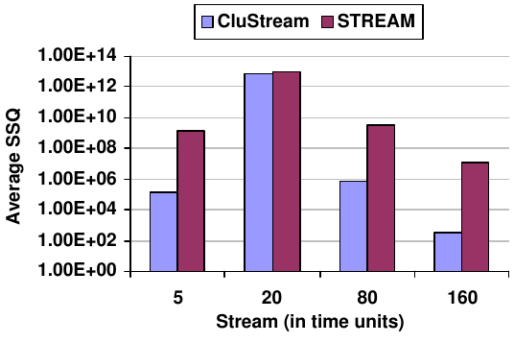

Figure 5. SSQ for the original $CluStream$ [**?**] vs STREAM []

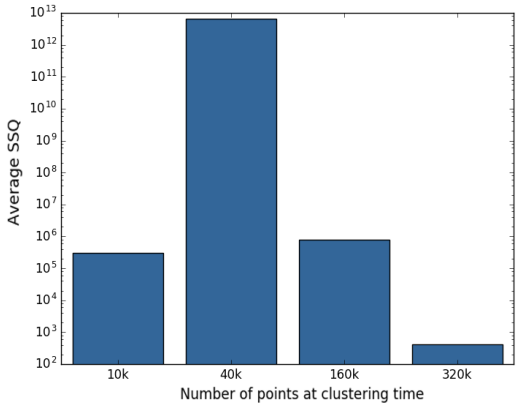

Figure 6. SSQ for our $Spark - CluStream$

Figure 7. Original $CluStream$ vs SPARK-Clustream. $DS1$ (Stream speed v = 2,000, H=1).

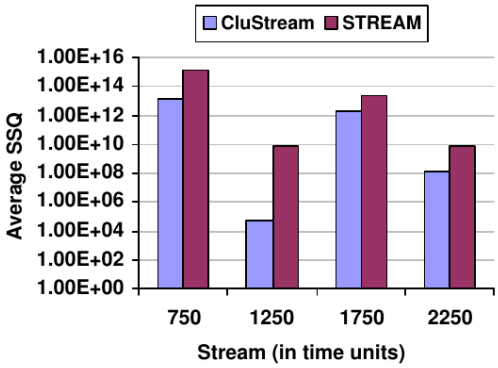

Figure 8. SSQ for the original $CluStream$ [**?**] vs STREAM []

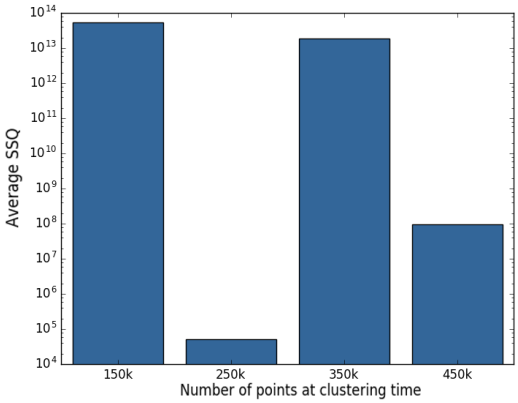

Figure 9. SSQ for our $Spark - CluStream$

Figure 10. Original $CluStream$ vs SPARK-Clustream. $DS2$ (Stream speed v = 200, H=256).

batch. This brings some limitations, the horizon H no longer has the same meaning: the δ parameter is used instead as an equivalent, relying on the micro-clustering part only and its ability to delete and create new micro-clusters.

3) Results on $DS1$: For the $DS1$ stream the results are shown in Figure 11.

The parameters used for *Spark-CluStream* are the same as in **??**. The number of clusters k is always 5 for this dataset and these tests for all methods.

For *Streaming K-Means*, the horizon $H = 1$ was transformed to $halfLife = 1000$ points. This is because the speed of the stream is 2000 points per time unit, if the horizon is 1, then only 2000 points are desired to be clustered, and half of that results in 1000 points. For the *decayFactor*, it is safe to choose 0, as that would mean that only the last 2000 points have influence on the clusters, which is exactly what it's desired.

StreamDM-CluStream is set up with its default parameters, only changing the horizon to 1 and the number of micro-clusters to 50 in order to match those of *Spark-CluStream*.

From Figure 11 it can be seen that *Spark-CluStream* delivers results which are very close to those of *Stream-*

$DS2$ - **avg SSQ**	**150k**	**250k**	**350k**	**450k**
CluStream	10^{13}-10^{14}	$\approx 10^5$	10^{12}-10^{13}	$\approx 10^8$
Spark-CluStream	5.402×10^{13}	5.143×10^4	1.892×10^{13}	9.646×10^7

Table II
$DS2$ - AVERAGE SSQ VALUES

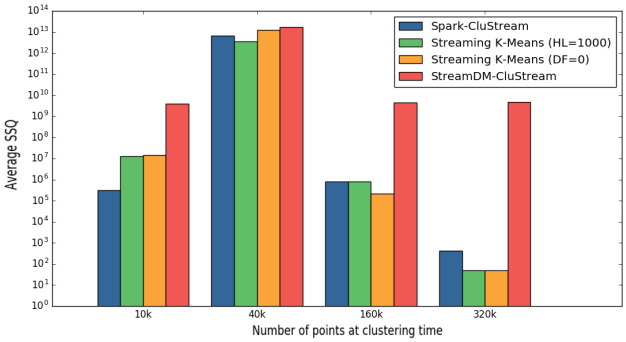

Figure 11. Comparison results: all methods. Stream speed = 2000, H=1

ing K-Means, which performs significantly better than the older method *STREAM*. Also, *Streaming K-Means* with the *decayFactor* (DF) is expected to do well on this test as it could be configured to cluster exactly as it was intended for this dataset.

The surprising results came from *StreamDM-CluStream*, as it performed noticeably, and significantly, worse than the rest of the methods. Specially for the last two marks at $160k$ and $320k$ it shows poor performance, which are where the other methods performed the better on average.

To find out whether this behavior is due to not using the snapshots plus offline macro-clustering, another test was performed using *Spark-CluStream* with the same conditions as for *StreamDM-CluStream*: using $\delta = 1$ as the horizon and $m = 100$ to match both methods

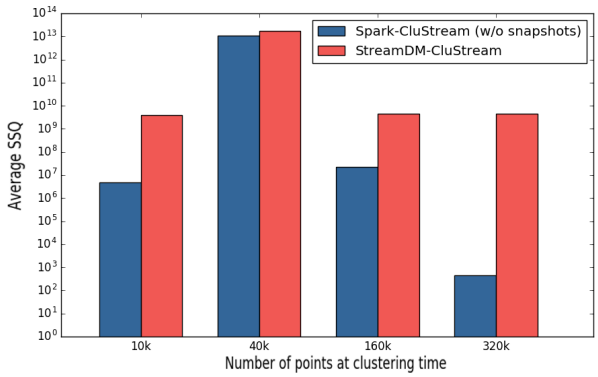

Figure 12. *Spark-CluStream* without snapshots. Stream speed=2000, H=1, m=100

Figure 12 shows poorer results for *Spark-CluStream* in comparison to its original behavior with snapshots, but still delivers noticeably better results than *StreamDM-CluStream*, even though all these tests were executed 4 times and the SSQ erros were averaged to get a better representation of how these methods perform.

4) Results on $DS2$: For the $DS2$ stream the results are shown in Figure 11.

Repeating the experiment for the stream with a speed of 200 and a horizon $H = 256$ revealed unexpected results. While most parameters for all methods remained the same, for *Streaming K-Means* a new $halfLife$ has to be calculated: multiplying the speed of the stream to the horizon, $200 \cdot 256 = 51200$ shows how many points of the stream are supposed to be clustered at each time, indicating that the parameter should be set to $halfLife = 25600$.

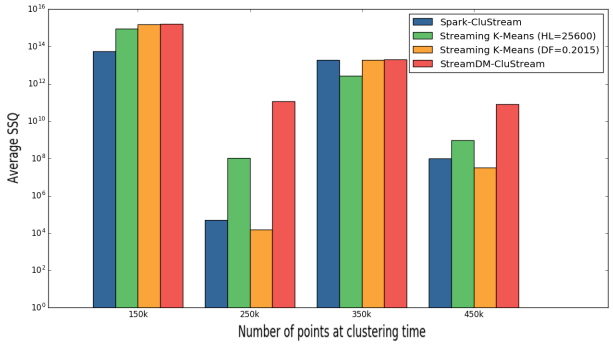

Figure 13. Comparison results: all methods. Stream speed = 200, H=256

The *decayFactor* strategy at first seems that does not work for such experiment, but considering that the total number of entries is known and exactly the marks at which the clustering process happens, it is possible to calculate an average value to use as a $decayFactor$:

- At 150000 points: $\frac{51200}{150000} \approx 0.3413$, which is the ratio of the points to cluster to the total number of points at that particular time.
- At 150000 points: $\frac{51200}{250000} \approx 0.2048$.
- At 150000 points: $\frac{51200}{350000} \approx 0.1462$.
- At 150000 points: $\frac{51200}{450000} \approx 0.1137$.

Averaging those ratios leads to a *decayFactor = 0.2015*, which is a way to determine how important the old data is in comparison to the new one.

Figure 13 shows that while *Spark-CluStream* still performs consistently good, *Streaming K-Means* with the *decay-*

Factor outperformed its relative with the $halfLife$ strategy. Another thing to notice is that *StreamDM-CluStream* still delivered the worse results.

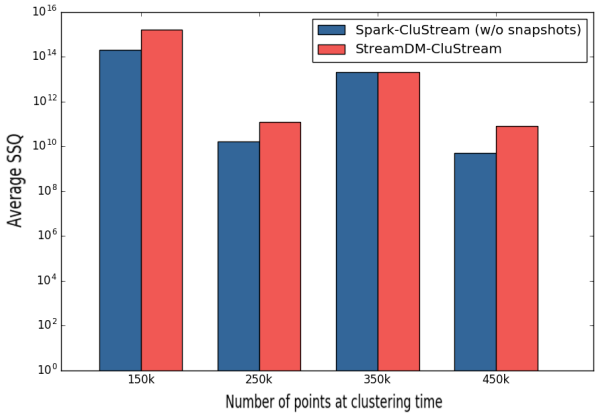

Figure 14. *Spark-CluStream* without snapshots. Stream speed = 200, H=256, m=100

Testing *Spark-CluStream* again without the use of snapshots, showed once more that it delivers better results than *StreamDM-CluStream*, as it can be seen in Figure 14, but the difference was reduced significantly. These results might indicate that *StreamDM-CluStream* does not benefit from shorter horizons.

C. Scalability

@Omar: Describe how you generated the dataWe 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 one.

@Omar: Please put the d=2 and d=100 dimensions side by side in a two-columns, it is easier to compare

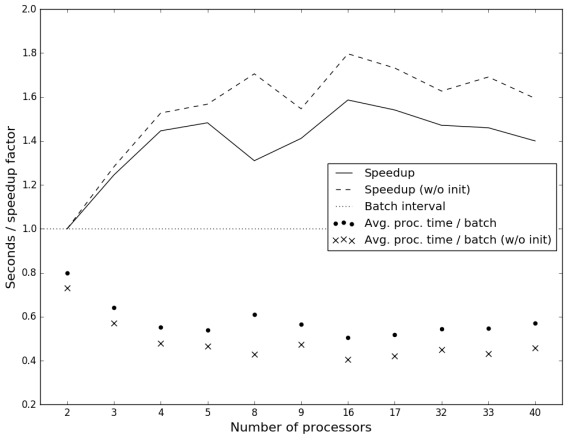

Figure 15. Scalability: Stream speed v= 10,000 points per timestamp, q = 20, d = 2.

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 where 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.

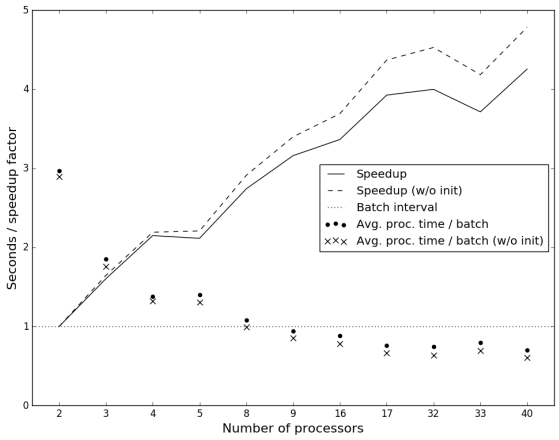

Figure 16. Scalability: Stream speed = 10000, q = 20, d = 100

Figure 15 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.

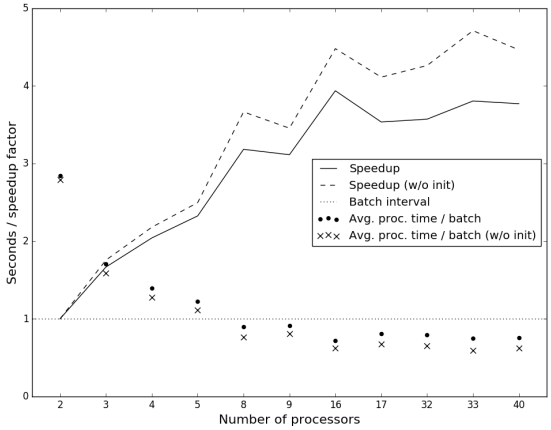

Figure 17. Scalability: Stream speed = 10000, q = 200, d = 2

@Omar: Please put the d=2 and d=100 dimensions side by side in a two-columns, it is easier to compare

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[7] for up to 16-17 processors, as it can be seen in Figure 16. 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 17.

Finally, when the number of clusters and the number of attributes are both increased significantly, Figure 18 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* unsing the Spark cluster setup for $q = 20$ and $d = 2, 100$, for the $CluStream$ method. Also, a test on a signle machine is performed, using the setup and dataset as in **??**.

[7]By linear scalability does not mean it scales with a 1 to 1 ratio, but rather linearly proportional.

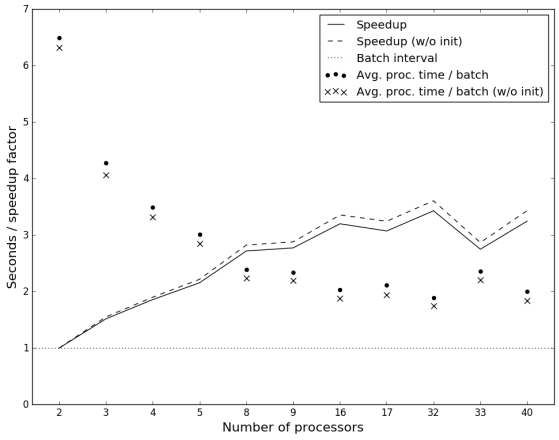

Figure 18. Scalability: Stream speed = 10000, q = 200, d = 100

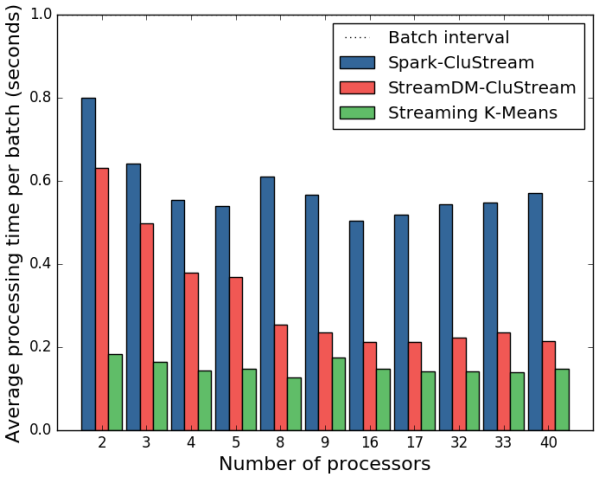

Figure 19. Processing time comparison: $q = 20$, $d = 2$

In Figure 19 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 fastets among the three.

When it comes to higher dimensions, *Spark-CluStream* shows a significant improvement over *StreamDM-CluStream*, which never got to the point were it was stable (below the 1 second mark), as shown in 20, it seems to scale as fast as *Spark-CluStream* but it was not enough even with 40 processors.

Surprisingly, in Figure 21, *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 22 shows that all three algorithms are able to scale

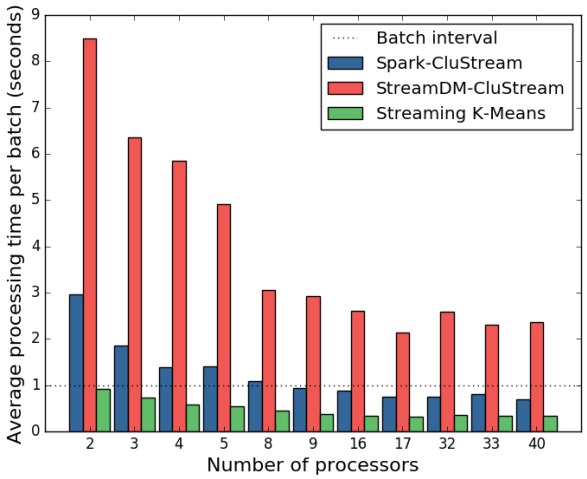

Figure 20. Processing time comparison: $q = 20$, $d = 100$

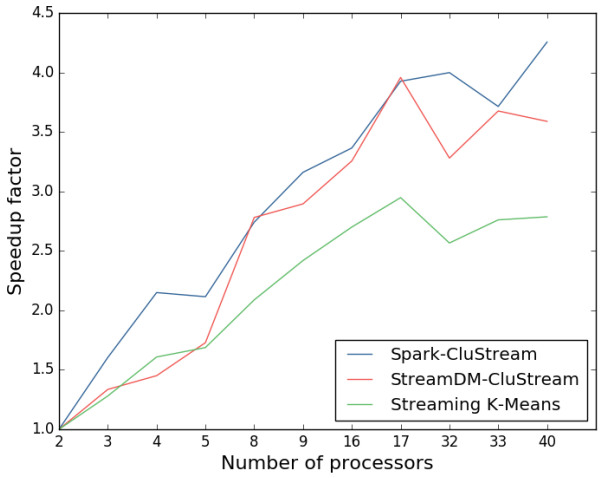

Figure 22. Processing time comparison: $q = 20$, $d = 100$

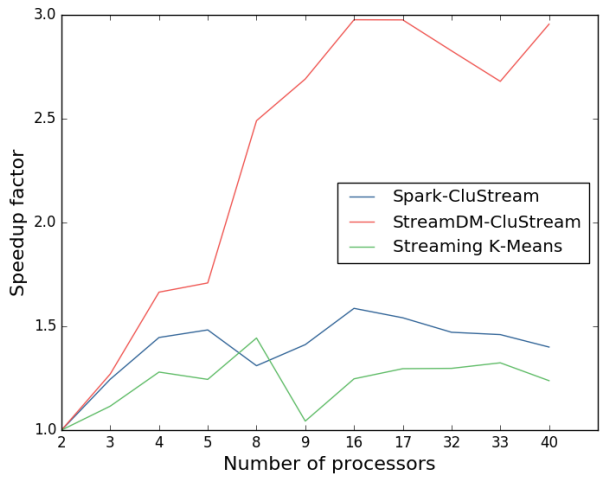

Figure 21. Scalability comparison: $q = 20$, $d = 2$

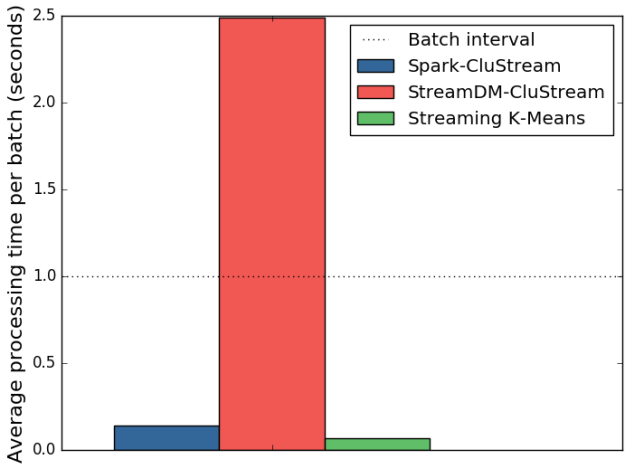

Figure 23. Processing time comparison for a single machine: $q = 50$, $d = 34$

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 share 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 23 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 or this matter.

VI. Conclusions

Bringing successfully the *CluStream* method to Spark has provided valuable information about how similar methods could be adapted by reviewing some of the challenges which might occur. It has also provided deep understanding of this method itself and how stream clustering differs from other clustering methods, which might not need to adapt to changing data, of Apache Spark and how distributed

systems work in general, but most importantly it has provided the experience to parallelize algorithms for the specific requirements of a given problem and this knowledge itself is applicable to an uncountable number of problems.

A. Goals achieved

It is rewarding to conclude that the goals were met satisfactorily. Here, the conclusions for every one of them.

1) Adapt CluStream in Spark (Spark-CluStream): This is the main goal of this thesis, and none of the other goals would have been met if this one failed. Adapting *CluStream* in Spark brought many challenges: one being the fact that the streaming library in Spark handles streams as batches and not individual points with time stamps, forcing this adaptation to change a few things that differentiate it from the original method, and the other one being the parallelization of the algorithm in order to take advantage of distributed computing.

Even though this adaptation changed the way data is processed, i.e. processing the stream in batches of data in parallel as much as possible, the results indicate that this was done correctly: it showed that it is not only capable of correctly clustering streams of data but it was able to match the quality of the original method described in [**?**]. It was shown for two different scenarios that this is true, when comparing the errors obtained after replicating the tests done by the authors of *CluStream*.

2) Understanding its advantages and disadvantages: The second most important goal was to make this adaptation as scalable as possible, and for this reason many tests were made using different scenarios. There are clearly cases where it is not fully scalable, but for the most part it was shown comparable scalability as some of the alternatives for Spark, including a method native of Spark and a similar method developed by a team from *Huawei* for a set of stream mining algorithms called *StreamDM*.

Some of its limitations were also understood , such as bottlenecks that might reduce the scalability and performance in general, such as:

- Outliers: handling with outliers in sequential code is expected to be a bottleneck, depending on the stream, a batch of data might contain points which do not belong to any micro-cluster and therefore, they have to be handled differently. Depending on the number of outliers, in particular the ones which require two micro-clusters to be merged, the total processing time for that batch will be affected negatively. In general there are three situations where this would normally occur: at the beginning of the stream if the initialization is not accurate, when the incoming data is very noisy and when the data dramatically changes.
- Communication costs: running in parallel requires certain communication between processing units. This affects the scalability negatively when few computations are required and too many processing units are used, as most of the time will be spent on communication. Also, even when it is expected to be scalable, increasing the number of micro-clusters used and the dimensionality of the data results in a bigger amount of information to communicate, and therefore not allowing greater speedups after a certain amount of processing units.

The results also showed that the *Streaming K-Means* algorithm is the fastest among the three tested (highly optimized for Spark), delivering good results in certain scenarios as it does not count with the flexibility of *CluStream* to better fit to evolving streams. *Spark-CluStream* on the other hand, showed that it not only delivers quality clustering, but also outperformed the similar *CluStream* implementation in *StreamDM*. Quality-wise it delivered more consistent and accurate results, and performance-wise it outperformed it in most cases, including one up to around 18 times faster.