Preface

This textbook is intended for use by students of physics, physical chemistry, and theoretical chemistry. The reader is presumed to have a basic knowledge of atomic and quantum physics at the level provided, for example, by the first few chapters in our book *The Physics of Atoms and Quanta*. The student of physics will find here material which should be included in the basic education of every physicist. This book should furthermore allow students to acquire an appreciation of the breadth and variety within the field of molecular physics and its future as a fascinating area of research.

For the student of chemistry, the concepts introduced in this book will provide a theoretical framework for that entire field of study. With the help of these concepts, it is at least in principle possible to reduce the enormous body of empirical chemical knowledge to a few basic principles: those of quantum mechanics. In addition, modern physical methods whose fundamentals are introduced here are becoming increasingly important in chemistry and now represent indispensable tools for the chemist. As examples, we might mention the structural analysis of complex organic compounds, spectroscopic investigation of very rapid reaction processes or, as a practical application, the remote detection of pollutants in the air.

April 1995

Walter Olthoff
Program Chair
ECOOP'95

Organization

ECOOP'95 is organized by the department of Computer Science, Univeristy of Århus and AITO (association Internationa pour les Technologie Object) in cooperation with ACM/SIGPLAN.

Executive Commitee

Conference Chair:	Ole Lehrmann Madsen (Århus University, DK)
Program Chair:	Walter Olthoff (DFKI GmbH, Germany)
Organizing Chair:	Jørgen Lindskov Knudsen (Århus University, DK)
Tutorials:	Birger Møller-Pedersen (Norwegian Computing Center, Norway)
Workshops:	Eric Jul (University of Kopenhagen, Denmark)
Panels:	Boris Magnusson (Lund University, Sweden)
Exhibition:	Elmer Sandvad (Århus University, DK)
Demonstrations:	Kurt Nørdmark (Århus University, DK)

Program Commitee

Conference Chair:	Ole Lehrmann Madsen (Århus University, DK)
Program Chair:	Walter Olthoff (DFKI GmbH, Germany)
Organizing Chair:	Jørgen Lindskov Knudsen (Århus University, DK)
Tutorials:	Birger Møller-Pedersen (Norwegian Computing Center, Norway)
Workshops:	Eric Jul (University of Kopenhagen, Denmark)
Panels:	Boris Magnusson (Lund University, Sweden)
Exhibition:	Elmer Sandvad (Århus University, DK)
Demonstrations:	Kurt Nørdmark (Århus University, DK)

Referees

V. Andreev	Braunschweig	P. Dingus
Bärwolff	F.W. Büsser	H. Duhm
E. Barrelet	T. Carli	J. Ebert
H.P. Beck	A.B. Clegg	S. Eichenberger
G. Bernardi	G. Cozzika	R.J. Ellison
E. Binder	S. Dagoret	Feltesse
P.C. Bosetti	Del Buono	W. Flauger

A. Fomenko
G. Franke
J. Garvey
M. Gennis
L. Goerlich
P. Goritchev
H. Greif
E.M. Hanlon
R. Haydar
R.C.W. Henderso
P. Hill
H. Hufnagel
A. Jacholkowska
Johannsen
S. Kasarian
I.R. Kenyon
C. Kleinwort
T. Köhler
S.D. Kolya
P. Kostka
U. Krüger
J. Kurzhöfer
M.P.J. Landon
A. Lebedev
Ch. Ley
F. Linsel
H. Lohmand
Martin
S. Masson
K. Meier
C.A. Meyer
S. Mikocki
J.V. Morris
B. Naroska
Nguyen
U. Obrock
G.D. Patel
Ch. Pichler
S. Prell
F. Raupach
V. Riech
P. Robmann
N. Sahlmann
P. Schleper
Schöning
B. Schwab
A. Semenov
G. Siegmon
J.R. Smith
M. Steenbock
U. Straumann
C. Thiebaux
P. Van Esch
from Yerevan Ph
L.R. West
G.-G. Winter
T.P. Yiou
M. Zimmer

Sponsoring Institutions

Bernauer-Budiman Inc., Reading, Mass.
The Hofmann-International Company, San Louis Obispo, Cal.
Kramer Industries, Heidelberg, Germany

Table of Contents

Scalable Stream Clustering in Apache Spark

Ivar Ekeland[1], Roger Temam[2] Jeffrey Dean, David Grove, Craig Chambers, Kim B. Bruce, and Elsa Bertino

[1] Princeton University, Princeton NJ 08544, USA,
I.Ekeland@princeton.edu,
WWW home page: http://users/~iekeland/web/welcome.html
[2] Université de Paris-Sud, Laboratoire d'Analyse Numérique, Bâtiment 425, F-91405 Orsay Cedex, France

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: Stream mining, Clustering, CluStream

1 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 in- formation over time and the second one is in charge of the clustering using the previously generated information. In other words,

- For each batch of data, statistically relevant summaries of the data are created and stored at a defined pace. This storing pace follows a specific storage scheme such that the disk space requirement reduces drastically; this is necessary as in most cases for data streams one does not want to store everything that arrives, one reason being the big data requires large and expensive computational resources (processing power and storage).
- On user demand, the stored summaries can be used for the end clustering task as they include all necessary information to achieve accurate results. Additionally, as these summaries are stored over time, a user defined time horizon/window can be chosen in order to analyze the data in different time periods, giving the possibility of a better understanding of the evolution of the data.

2 Related work

2.1 SAMOA

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.

2.2 StreamDM

StreamDM is a collection of algorithms for mining big data streams [3]. 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.

[3] As it is stated by them here: http://huawei-noah.github.io/streamDM/

3 Basic notions

3.1 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.*

3.2 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 1 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 2.

Fig. 1. Flow of data in Spark streaming

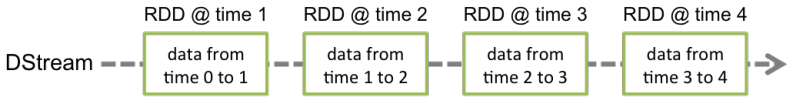


Fig. 2. DStreams are Spark streaming's abstraction of a data stream

3.3 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.
- 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.

3.4 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 $\sum_{j=1}^{n} (x_{i_j}^p)^2$.*
- *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 $\sum_{j=1}^{n} x_{i_j}^p$.*
- *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[4].
– 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

Fig. 3. Example of snapshots stored for $\alpha = 2$ and $l = 2$

With the help of Figure 3 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.

3.5 Online Micro-clustering

During this process, the incoming information is analyzed and transformed into statistical summaries that are easier to handle. This process is divided in three parts: initialization, classification and assignation, these are described as follows:

Initialization Before the main maintenance of the micro-clusters, the initialization of q micro-clusters must be performed and each of them are identified

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

with a unique *id*. The number q is taken as a user input and highly depends on the number of final macro-clusters to be calculated and the capabilities of the system where the algorithm runs.

- Random initialization:
  1. generate q random d-dimensional vectors to use as centroids of the micro-clusters so that data can be randomly assigned to different micro-clusters.
  2. Assign the first incoming batch of data to their closest random centroids[5].
- KMeans initialization:
  1. take the first n_i points, number which is specified by the user, and perform the KMeans clustering method for q clusters. By doing this it is possible to ensure that the first n_i points will be better clustered than after a random initialization.
  2. Assign the first incoming batch of data to their closest found centroids.

Classification When a new batch of data arrives, it is necessary to verify point by point whether they belong to a micro-cluster or not. This means that a point has to be located within a factor t of the RMS deviation[6] (RMSD) of the nearest micro-cluster, otherwise it is considered an outlier.

Outliers are taken into account as possible new micro-clusters, this is because the stream of data might change over time. To do so, one new micro-cluster should be created with an entirely new *id*, containing the outlier. The number of micro-clusters q is fixed, so one micro-cluster must be freed either by deleting an old micro-cluster of by merging two of them. To decide whether a micro-cluster is safe to delete, it is necessary to compute its value of *recency* (RV) and compare it to a user defined threshold δ, if no micro-cluster is safe to delete, then the two closest micro-clusters are merged.

1. Check if a point X_{i_k} is not an outlier: $distance(\bar{X}_{i_k}, \hat{M}_j) \leq t \cdot RMSD$, where $\hat{M}_j$ is the nearest micro-cluster.
   - If X_{i_k} is an outlier: compute RV_{M_j}, if $N_j < 2 * m$ then the mean the time stamps is used, otherwise compute an approximation for the last m points of each micro-cluster assuming its points are normally distributed[7] for the $1 - m/2N_j$ percentile.
     (a) If $RV_{M_j} < T - \delta$, where RV_{M_j} is the *recency* value for a given micro-cluster and T the current clock time units elapsed since the beginning of the stream, then M_j is safe to delete and a new micro-cluster M_{newID} must replace it containing X_{i_k}.

[5] See the assignation process for the details.
[6] The RMSD can be obtained using $\overline{CF2^x}$ and $\overline{CF1^x}$ as shown in **??**
[7] It is possible to derive the mean and the standard deviation from $CF2^t$ and $CF1^t$ as shown in **??**

(b) When there is no M_j safe to delete, then merge M_j and M_{jj} such that $min\{M_j, M_{jj} : M_j, M_{jj} \in M_Q, distance(M_j, M_{jj})\}$, where M_Q is the set of all micro-clusters. The new merged micro-cluster will now have a list of IDs $\{j, jj\}$ and a new micro-cluster M_{newID} must replace M_{jj} containing X_{i_k}.

2. Assign the point X_{i_k} to $\hat{M}_j$.

Assignation When a point X_{i_k} is assigned to a micro-cluster, the tuple $(\overline{CF2^x}, \overline{CF1^x}, CF2^t, CF1^t, n)$ for M_j needs to be maintained, due to its addition properties it is possible to do so directly as follows:

- $\overline{CF2^x}_{new} = \overline{CF2^x}_{old} + \overline{CF2^x}_{X_{i_k}} = \overline{CF2^x}^p_{old} + (x^p_{i_k})^2 \ \forall p \in \{1, 2...d\}$
- $\overline{CF1^x}_{new} = \overline{CF1^x}_{old} + \overline{CF1^x}_{X_{i_k}} = \overline{CF2^x}^p_{old} + x^p_{i_k} \ \forall p \in \{1, 2...d\}$
- $\overline{CF2^t}_{new} = \overline{CF2^t}_{old} + \overline{CF2^t}_{X_{i_k}} = \overline{CF2^x}_{old} + (T_{i_k})^2$
- $\overline{CF1^t}_{new} = \overline{CF1^t}_{old} + \overline{CF1^t}_{X_{i_k}} = \overline{CF2^x}_{old} + T_{i_k}$
- $n_{new} = n_{old} + 1$

3.6 Offline Macro-Clustering

While the online process transforms the stream into statistical summaries in the form of micro clusters, the offline process uses this result to deliver the final micro-clusters[8] for a given time horizon. This means that the micro clusters in the snapshots stored are now the input data for the macro-clustering process.

Assuming that t_c is the current time and h is the user defined horizon, the time window of information would be $(t_c, t_c - h)$. Having snapshots means that probably $t_c - h$ will not exist in an exact form but the snapshot that occurred just before that time is chosen. The pyramidal time frame ensures that it is always possible to find a snapshot $t_c - h'$ within the user specified tolerance for any h.

If $S(t_c - h')$ is the set of micro-clusters at time $t_c - h$ and $S(t_c)$ is the set of micro-clusters at time t_c, it is possible to find the final set of micro-clusters $N(t_c, h')$ by subtracting from $S(t_c)$ each corresponding micro-cluster in $S(t_c - h')$. This is possible due to the fact that each micro-cluster is associated with a list of *ids*. Doing this ensures that micro-clusters created before the user specified time horizon do not dominate the the results of the clustering process.

Property 1. Let C_1 and C_2 be two sets of points. Then the cluster feature vector $\overline{CFT(C_1 \cup C_2)}$ is given by of $\overline{CF2(C_1)} + \overline{CF2(C_2)}$

Property 2. Let C_1 and C_2 be two sets of points such that $C_1 \supseteq C_2$. Then the cluster feature vector $\overline{CFT(C_1 - C_2)}$ is given by $\overline{CF2(C_1)} - \overline{CF2(C_2)}$

[8] In K-Means this know as the k clusters.

Properties 1 and 2 show, respectively, the additive and subtractive nature of the cluster feature vectors. This is particularly helpful as only two snapshots are required to approximate any user specified time horizon or window.

Once $N(t_c, h')$ is constructed, the micro-clusters in it are treated as *pseudo-points* K-Means can be used to determine the higher level and final clusters after slightly adjusting it:

- At its initialization step, the seeds are sampled with probability proportional to the number of points each micro-cluster has instead of picking them randomly, which correspond to the centroids of the micro-clusters.
- The distance from a seed to a *pseudo-point* is equal to the distance between the seed and the centroid of the corresponding micro-cluster.
- When the seeds are adjusted, the new seed is defined as the weighted centroid of the micro-clusters in that partition.

As a matter of fact, following the previous modifications, many traditional clustering algorithms could be used if needed.

4 Spark-CluStream

5 Experiments

5.1 Validation

Two cases are analyzed in this section. These cases are two of the ones used by the developers of *CluStream* to test its clustering capabilities. The tests differ in two ways: the speed of the stream and the time window used. The desired result would be to obtain comparable results for *Spark-CluStream*.

Case 1 The first case uses a stream speed of 2000 points per time unit and a horizon $H = 1$. The dataset contains exactly 494021 points, meaning that the online phase would require $\frac{494021}{2000} \approx 247$ time units to complete.

The measurement is the sum of squares (SSQ) of the euclidean distances from the points to their nearest macro-cluster. In fact, the case is run a total of 4 times for *Spark-CluStream* to compute an average. The SSQ is defined as:

$$SSQ = \sum_{i=1}^{N} distance(p_i, c)^2, \tag{1}$$

where N is the number of points used in the horizon $H = 1$, which should average $N \approx 2000 \cdot H \approx 2000$ points.

The parameters used in [**?**] are: $\alpha = 2, l = 10, InitNumber = 2000, \delta = 512, t = 2$.

Figure 4 shows the results used by the original *CluStream* to show its capabilities against an older method *STREAM*, which is a modified version of K-Means

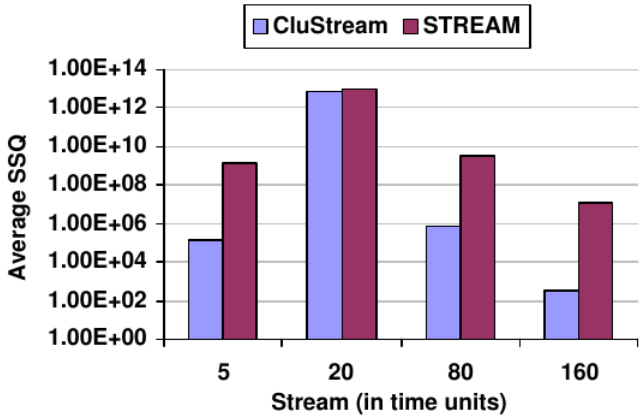


Fig. 4. Results for the original *CluStream*[**?**]. Stream speed = 2000, H=1

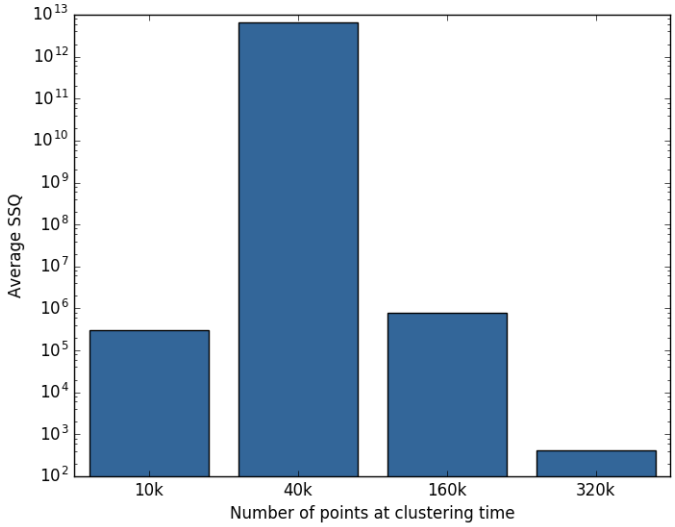


Fig. 5. Validation results for $Spark - CluStream$. Stream speed = 2000, H=1

for data streams. The average SSQ for *CluStream* is the most relevant to this test.

The parameters used for *Spark-CluStream* were matched.

The parameter m, for m last points, was the only one not provided. Here, $m = 20$ was chosen. For this case, 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 q is 50, a 10 times the number of final clusters (5) is enough for

the vast majority of cases[**?**]. The rest of the parameters were matched, with the only remaining thing to point out is that *fakeKMeans()* used 5000 sampled points.

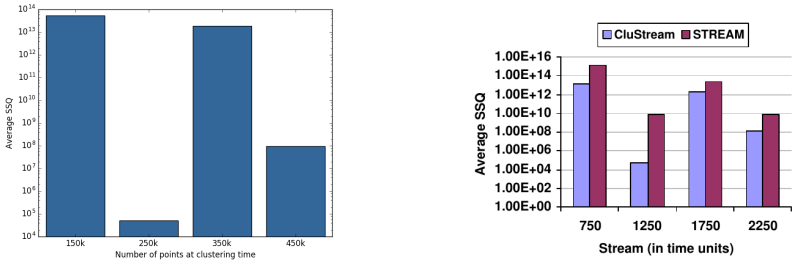
Figure 5 shows the results obtained by *Spark-CluStream*. There is a difference in the labels of the horizontal axis, while Figure 4 shows the time units of the stream, Figure 4 shows the number of points that had been streamed and processed. This is done because Spark streaming libraries in combination with the streaming simulation do not always deliver the same amount of points every single time unit, leading to inaccurate results comparing only by clustering on certain time units. 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.

Comparing the results, it is possible to deduce that they are very similar. The exact values for Figure 4 are not available but it suffices to compare the magnitudes of the average SSQ.

Case 1: 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

Case 2 The second case uses a stream speed of 200 points per time unit and a horizon $H = 256$. The dataset contains exactly 494021 points, meaning that the online phase would require $\frac{494021}{200} \approx 2470$ time units to complete.

The measurement is the SSQ again and the same circumstances apply for this case as in the first one with the difference that here δ and m are relevant. The parameter m is again chosen to be 20: if 200 points are processed every time unit and there are 50 micro-clusters, assuming all 200 points should be distributed uniformly at least every 5 time units leads to $5\frac{200}{50} = 20$. An in-depth analysis of the behavior of *CluStream* for different δ's and m's is out of the scope of this work.



(a) *Spark-CluStream* (b) Original *CluStream*[**?**]

Fig. 6. Validation results: case 2. Stream speed = 200, H = 256

Again, the comparison is for the average SSQ. The test ran 4 times for *Spark-CluStream* to average the results, which are very similar to the original *CluStream* in this case as well:

Case 2: 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

5.2 Performance

Scalablity 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.

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 7 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[9] for up to 16-17 processors, as it can be seen in Figure 8. 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

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

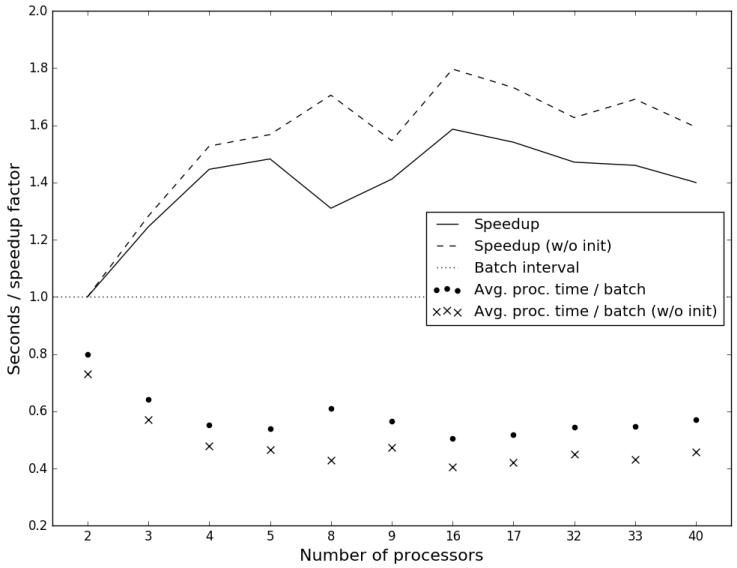


Fig. 7. Scalability: Stream speed = 10000, q = 20, d = 2

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 9.

Finally, when the number of clusters and the number of attributes are both increased significantly, Figure 10 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.

Comparison against alternatives It is important for this project to know how *Spark-CluStream* stands against some of the other alternatives for stream clustering available for Spark, in particular: *Streaming K-Means* from Spark and *StreamDM-CluStream*, which is another adaptation of the *CluStream* method for Spark. There are two aspects of interest in this tests, one being their clustering capabilities and the other their performance.

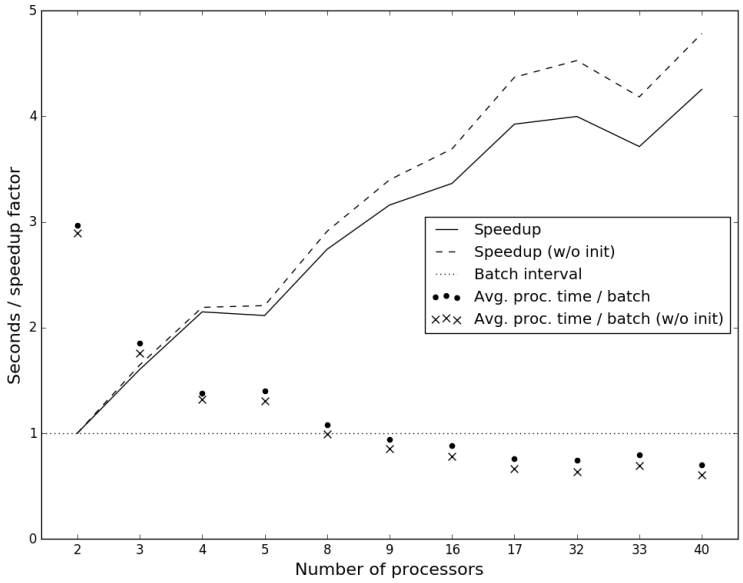


Fig. 8. Scalability: Stream speed = 10000, q = 20, d = 100

5.3 Clustering

The setup and the dataset are the same as in **??**, as having already verified results provides the possibility of using those tests to directly compare the results against the other methods. Again, the used measurement is the sum of squares (SSQ).

Before looking at the results, here are some key considerations for the other methods:

– *Streaming K-Means*:
  - 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

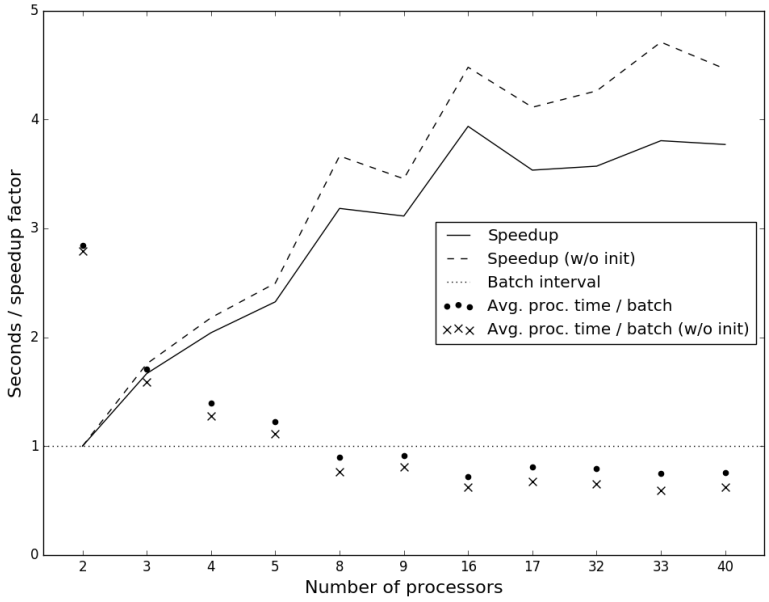


Fig. 9. Scalability: Stream speed = 10000, q = 200, d = 2

points of the current batch will be considerable smaller than the points clustered so far.

– *StreamDM-CluStream*:
  - 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 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.

Case 1 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.

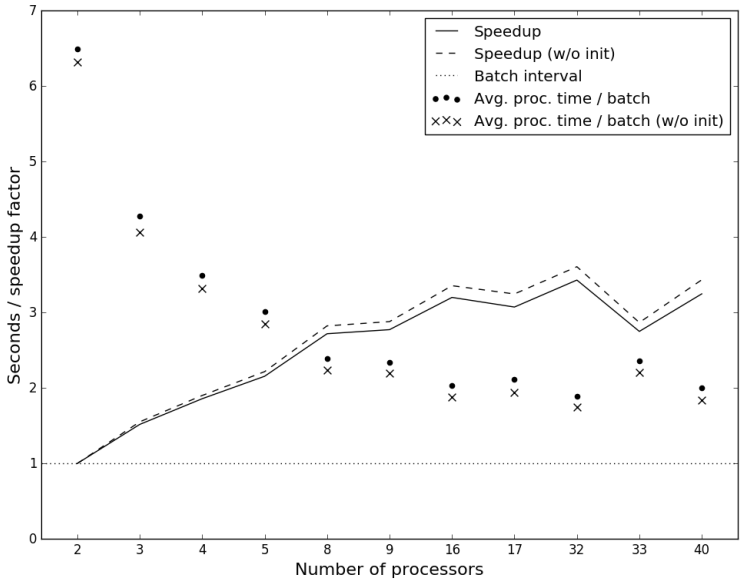


Fig. 10. Scalability: Stream speed = 10000, q = 200, d = 100

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 *Streaming 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 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.

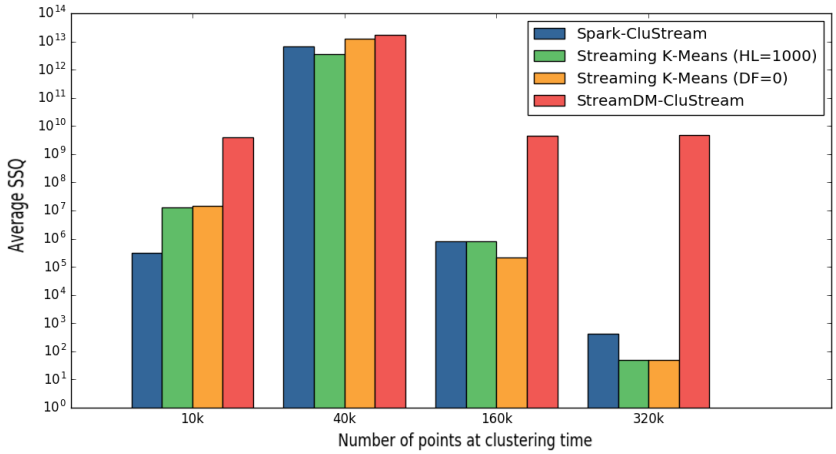


Fig. 11. Comparison results: all methods. Stream speed = 2000, H=1

Case 2 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 $halfLife = 25600$.

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

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.

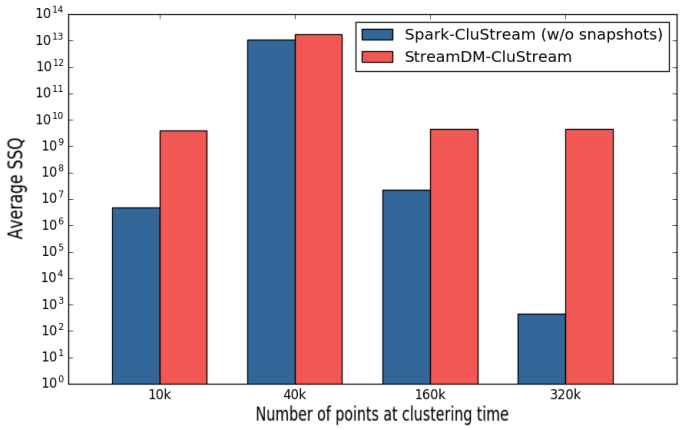


Fig. 12. *Spark-CluStream* without snapshots. Stream speed=2000, H=1, m=100

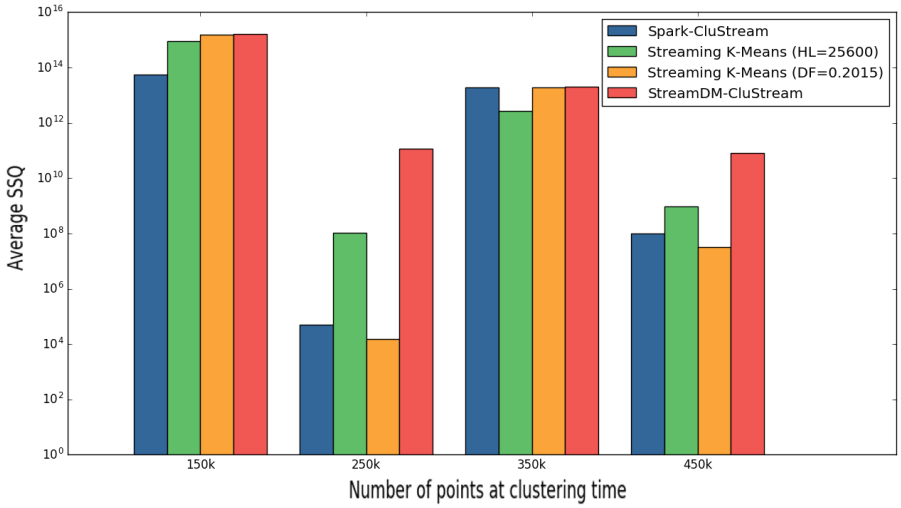


Fig. 13. Comparison results: all methods. Stream speed = 200, H=256

5.4 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 **??**.

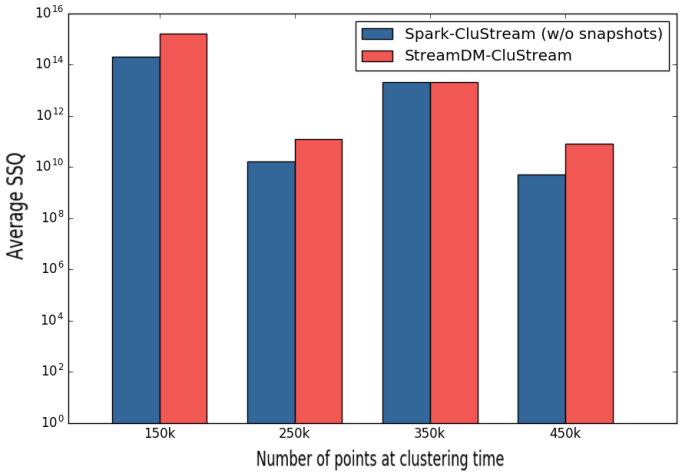


Fig. 14. *Spark-CluStream* without snapshots. Stream speed = 200, H=256, m=100

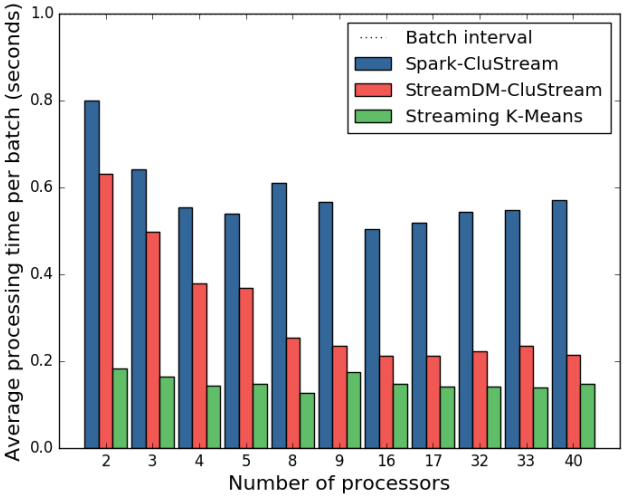


Fig. 15. Processing time comparison: $q = 20$, $d = 2$

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 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 16, it seems to scale as fast as *Spark-CluStream* but it was not enough even with 40 processors.

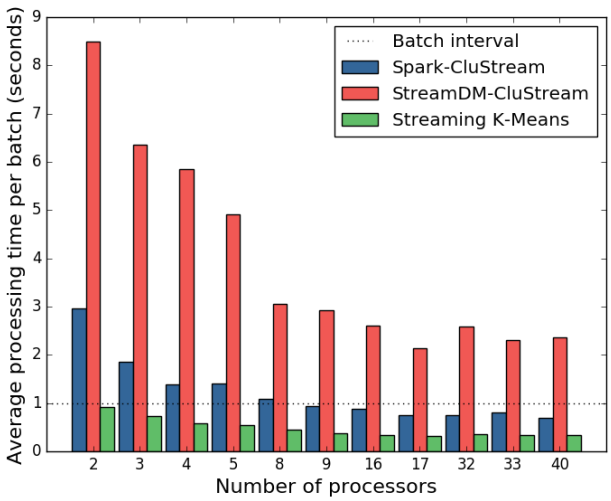


Fig. 16. Processing time comparison: $q = 20$, $d = 100$

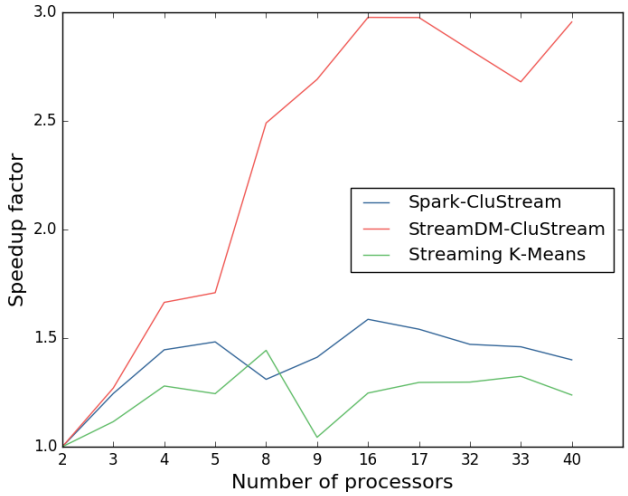


Fig. 17. Scalability comparison: $q = 20$, $d = 2$

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.

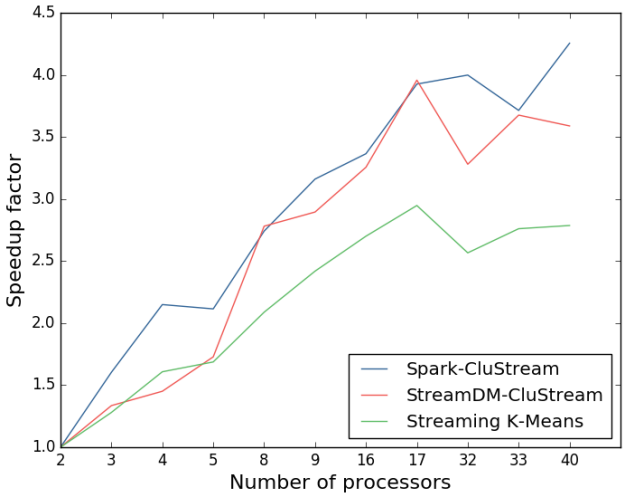


Fig. 18. Processing time comparison: $q = 20$, $d = 100$

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 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 or this matter.

6 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

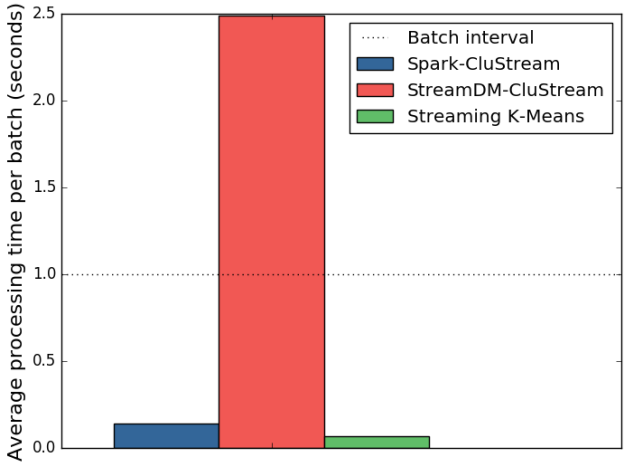


Fig. 19. Processing time comparison for a single machine: $q = 50$, $d = 34$

problem and this knowledge itself is applicable to an uncountable number of problems.

The typical workflow for doing so should follow a similar structure: from understanding the problem as described in chapter **??**, to going through the process of achieving the desired goals as shown in chapters **??** and **??**.

7 Goals review

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

7.1 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 and the proof of that lies in the validation section **??**: 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*.

7.2 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, as shown in section **??**, 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, as shown in the chapter section **??**, 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.

Notes and Comments. The first results on subharmonics were obtained by Rabinowitz in [5], who showed the existence of infinitely many subharmonics both in the subquadratic and superquadratic case, with suitable growth conditions on H'. Again the duality approach enabled Clarke and Ekeland in [2] to treat the same problem in the convex-subquadratic case, with growth conditions on H only.

Recently, Michalek and Tarantello (see [3] and [4]) have obtained lower bound on the number of subharmonics of period kT, based on symmetry considerations and on pinching estimates, as in Sect. 5.2 of this article.

Table 1. This is the example table taken out of *The TEXbook,* p. 246

Year	World population
8000 B.C.	5,000,000
50 A.D.	200,000,000
1650 A.D.	500,000,000
1945 A.D.	2,300,000,000
1980 A.D.	4,400,000,000

References

1. Clarke, F., Ekeland, I.: Nonlinear oscillations and boundary-value problems for Hamiltonian systems. Arch. Rat. Mech. Anal. 78, 315–333 (1982)
2. Clarke, F., Ekeland, I.: Solutions périodiques, du période donnée, des équations hamiltoniennes. Note CRAS Paris 287, 1013–1015 (1978)
3. Michalek, R., Tarantello, G.: Subharmonic solutions with prescribed minimal period for nonautonomous Hamiltonian systems. J. Diff. Eq. 72, 28–55 (1988)
4. Tarantello, G.: Subharmonic solutions for Hamiltonian systems via a $\mathbb{Z}_p$ pseudoindex theory. Annali di Matematica Pura (to appear)
5. Rabinowitz, P.: On subharmonic solutions of a Hamiltonian system. Comm. Pure Appl. Math. 33, 609–633 (1980)

Subject Index