Technical University of Crete
School of Electrical and Computer Engineering

Distributed k-Means Streaming Algorithms in Spark

Author:
Ioanna Kiriakidou

Thesis Committee:
Prof. Antonios Deligiannakis (Supervisor)
Prof. Minos Garofalakis
Assoc. Prof. Michail G. Lagoudakis

A thesis submitted in fulfillment of the requirements for the degree of Diploma in Electrical and Computer Engineering

Chania, June 2021

Abstract

K-means is one of the most commonly used clustering algorithms that clusters the multi-dimensional data points into a predefined number of clusters. When data arrives in a stream, there is a need to estimate clusters dynamically, updating them on arrival. In this thesis, we will apply a sampling technique using a data structure called coreset trees, before any approximation algorithm is applied. Coresets are used to obtain a small weighted sample from the data stream. Using coresets in a tree-like form we successfully speed up the process of computing a summary of the original data. The advantage of such a coreset is that we can apply any clustering algorithm on a much smaller sample to compute a solution for the original dataset faster. In the second step, we are using a StreamKM++ to estimate the cluster centers of the summary. We evaluate the algorithm on how the parallelism level impacts the time needed to extract the clusters, finally we compare the consistency within clusters of data conclusions about the usage of coreset trees as a distributed sampling method. In our experiments we set up a real life scenario of data coming from a ride-hailing app. We received geospatial data from drivers around Colombia and applied StreamKM++ to get clusters of driver's location real-time.

Acknowledgements

I would like to thank Prof. Antonios Deligiannakis, my academic supervisor, who provided me with feedback and support throughout this thesis. Furthermore, I am also grateful to the rest of my thesis committee members: Prof. Minos Garofalakis and Assoc. Prof. Michail G. Lagoudakis, for their time to evaluate this work and the knowledge I received from them during my undergraduate years. Last, but not least, I also want to thank my family and all of my friends, for encouraging and supporting me all these years.

Context

Chapter 1

1.1 Introduction

Clustering aims to partition input objects into groups or “clusters” such that objects within a cluster are similar to each other, and objects in different clusters are not. Often the size of the datasets to be clustered, are very big and so clustering algorithms to handle these datasets are basic tools in many different areas like machine learning, data mining, data compression, and database systems. A popular formulation of clustering is Lloyd's algorithm [1] (usually referred to simply as k-means).

Given a set of points S in an Euclidean space and a parameter k, the goal is to partition S into k “clusters” in a way that minimizes a cost metric based on the l_2 distance between points. The k-means formulation is widely used in practice. Starting k “centers” are typically chosen at random as per Lloyd’s algorithm from the original data points. Then, each point is assigned to the nearest center, and the centers are again recomputed as the center of the mass of all points assigned to it. When the process is stabilised we stop repeating the last two steps. By checking that φ is monotonically decreasing, which ensures that no configuration is repeated during the course of the algorithm. There are only k*n possible clusterings, thus the process will always terminate. The algorithm is very fast and simple to perform, and this is what is making it appealing not the accuracy of the result.

However, in real-world problems the data that needs clustering might have a great amount of volume. The size of the data can lead to serious problems for applications that need fast (near) real-time answers. Thus, we use a data structure that helps us create representative synopses of the data we’ve seen so far. This synopsis is extracted using coresets. A coreset is a subset of input, such that we can get a good approximation to the original input. The goal of this thesis is to design a distributed streaming clustering algorithm that significantly improves the clustering runtime query, compared to the current state of the art, while keeping other desirable properties that are enjoyed by other current algorithms, such as provable accuracy and limitations in the memory requirements.

1.2 Thesis Contribution

As per this thesis, we chose to study Coreset Trees, a data structure that Ackermann, Lammersen, Märtens, Raupach, Sohler, Swierkot proposed for the implementation of a distributed streaming k-means algorithm. Since distributed processing is one the most active research areas nowadays, we try to expand their model to work in a distributed and parallel manner.

1.3 Thesis Outline

In Chapter 2 we describe the general idea of the coresets, the coresets tree and how they can be implemented to work in a distributed manner. We also define the outline of the algorithm.
In Chapter 3 we go through the stack, technologies and useful tools that we use as part of our experiments.
In Chapter 4 we present the evaluation methods we used and the experiments we did, both batch and streaming ones. We visualise different metrics to compare how our proposed algorithm performed.
In Chapter 5 and 6 we discuss related work already done in the same topic and how we can improve our solution in the future.

Chapter 2

2.1 Preliminaries

We work with points from the d-dimensional Euclidean space Rd for integer d > 0. A point can have a positive integral weight associated with it. If unspecified, the weight of a point is assumed to be 1. Let $\|\cdot\|$ the l_2-norm for points $x, y \in R^d$, by D(x,y) = llx – yll denote the Euclidean distance between x and y. For point x and a point set$C \subseteq R^d$, the distance of x to C is defined to be D(x,C) = $min_{c \in C}$llx – cll and cost(P, C) = $\sum_{x \epsilon P} w(x)*D^2(x, C)$ for C, P ⊂ R^d. Analogously, for a weighted subset S ⊂ Rd with weight function $w: S \rightarrow Z+$, we use $cost_w(S, C) = \sum_{y \epsilon S} w(y)*D^2(y, C)$.

The Euclidean k-means problem is defined as follows.

Problem 1. (2) Given an input set P ⊆R^d with n points and weight function defined by $w: P \rightarrow Z+$, find a set C ⊆ R^d with |C| = k that minimizes cost(P, C):
Furthermore, by

$$opt_k(P) = min_{C' \subset R^d: |C'|=k} \sum_{x \epsilon P} w(x)*D^2(x, C')$$

we denote the cost of an optimal Euclidean k-means clustering of P .

The most important concept we'll use is the coresets. In general, a coreset for a set P is a relatively small weighted-set, in such a way that for any set of k cluster centers the clustering cost of the coreset is an approximation for the cost of the original set P with a small relative error. The advantages of such data structure is that we can apply any fast approximation algorithm (for the weighted problem) on a much smaller coreset to compute the approximate solution for the original data set P more efficiently. We use the following formal definition.

Theorem 1. (2) Let k ∈ N and ε ≤ 1. A weighted set $S \subset R^d$ with positive weight function $w: P \rightarrow Z+$ and $\sum_{y \epsilon S} w(y)$ = |P| defines a (k, ε) - coreset of P iff for every $C \subset R^d$ of size |C| = k we have

$$(1 - \varepsilon)\, cost(P, C) \leq cost_w(S, C) \leq (1 + \varepsilon)\, cost(P, C)$$

The most important advantage of the coresets is their ability to be constructed in a parallel manner. This can be done easily, since the union of two coresets creates again a coreset. More specifically, if a dataset is split into l chunks that are subsets of it, and we compute a $(1 + \varepsilon')$-coreset of size s for each subset, then the union of the l coresets is a $(1 + \varepsilon')$-coreset of the original set and has size $l \cdot s$. This is especially helpful if the data are distributed into l different machines. Then, each machine will simply compute a coreset. Afterwards, the coresets that are usually smaller can then be sent to a master device which approximately solves the optimization problem. Our coreset results for subspace approximation for one j-dimensional subspace, we can use it for k-means algorithm and for general projective clustering and we can provide a distributed algorithm for solving these problems approximately. The algorithm we implemented follows the same approach and creates a small weighted coreset for every micro batch of the streaming data to extract clusters using k-means algorithm.

Hereafter, we will refer to k-means coresets as simply coresets. For integer $k > 0$, a parameter $0 < \varepsilon < 1$, and a weighted dataset $P \subseteq R^d$, the notation coreset(k, ε, P) means a (k, ε)-coreset of P. From [4] we use the following properties:

Property 1. ([4]) If C1 and C2 are each (k, ε)-coresets for disjoint multi-sets P1 and P2 respectively, then $C1 \cup C2$ is a (k, ε)-coreset for $P1 \cup P2$.

Property 2. ([4]) Let k be fixed. If C1 is ε_1-coreset for C2, and C2 is a ε_2-coreset for P, then C1 is a $((1 + \varepsilon_1)(1+\varepsilon_2) - 1)$ - coreset for P.

def buildCoreset(m: Int, input: List[Point]): List[Point]
For points that are more than m, the process is the following: ● Choose the first centroid with KMeans++ seeding ● Calculate the distance of the other points with the first centroid ● Recursively calculate the final centroids ● Assign points to these centroids while updating the weights and the counts If points are less than m then each one is a center
Algorithm 1 - buildCoreset: Constructs a coreset with size m for a list of points with n dimensions

2.2 Coreset Tree

A new data structure called coreset tree is developed in order to significantly speed up the time necessary for sampling non- uniformly during the coreset construction. After the coreset is extracted from the data stream, a weighted k-means algorithm is applied on the coreset to get the final clusters for the original stream data. To build a coreset tree we use the Merge-and-Reduce technique [5] as described below.

2.3 Merge-and-Reduce Technique

We will discuss a popular method to build an efficient streaming algorithm based on coresets as a data structure. As discussed in the previous section, coresets have a very useful property; the union of coresets are also a coreset. Specifically, if S1 and S2 are (1 + ε)-coresets for input P1 and P2 point sets, respectively, then S1 ∪ S2 is a coreset for P1 ∪ P2 (Property 1). In addition, coresets can have nested computations: If S1 is a (1 + ε1)-coreset for S2, and S2 is a (1 + ε2)-coreset for P, then S1 is a (1 + ε1)(1 + ε2)-coreset for P.

So, we can move the coreset construction idea into an Insert Only data stream by splitting the input in chunks, compute a coreset for each partition so as to union the result into the final coreset. When the union gets large, we can reduce it by applying the same algorithm again (i.e. coreset construction). Nevertheless, each union introduces an additional error, so the number of unions that we perform has to be kept small. More specifically, the number of the unions and computations that a point in the set participates must be kept small.

This Merge-and-Reduce technique does exactly what we described. For each partition of the data and the coreset(s) computed from it, this technique keeps the unions of each chunk low. We can achieve that by maintaining a tree-like structure while merging and reducing chunks. We can see how it works in Figure 1. Basically, the first chunk/block B1 is read and reduced to coreset S1, the second chunk B2 to coreset S2. Then, S1 and S2 are merged and reduced to coreset S3. Blocks B3 and B4 induce the coreset computation of S4 and S5 first and S6 then. When S6 is computed, it is merged with S3 and reduced to S7. In this way the computation continues: Each block is then summarized into a coreset which forms a leaf node in the tree, and whenever both children of a node have computed their coresets, then the node computes its coreset, too.

The advantages of this method is that each point in the tree takes part at most in log|P| reduction steps, where P is the number of points that we have seen until the reduction. This means that for each reduction step we compute a (1 + ε)-coreset of the input, so the result coreset is a $(1+\varepsilon)\cdot log\left|P\right|$-coreset. In order to compensate for this accuracy loss, the reduction steps have to be a bit more accurate. They have to compute (1+ε′)-coresets for $\varepsilon' := \frac{\varepsilon}{log|P|}$.

Lastly, we have to notice that we may need to store multiple coresets at the same time. But, we will not need to store more than a coreset at each level and one whenever a new chunk of data is added in the tree.

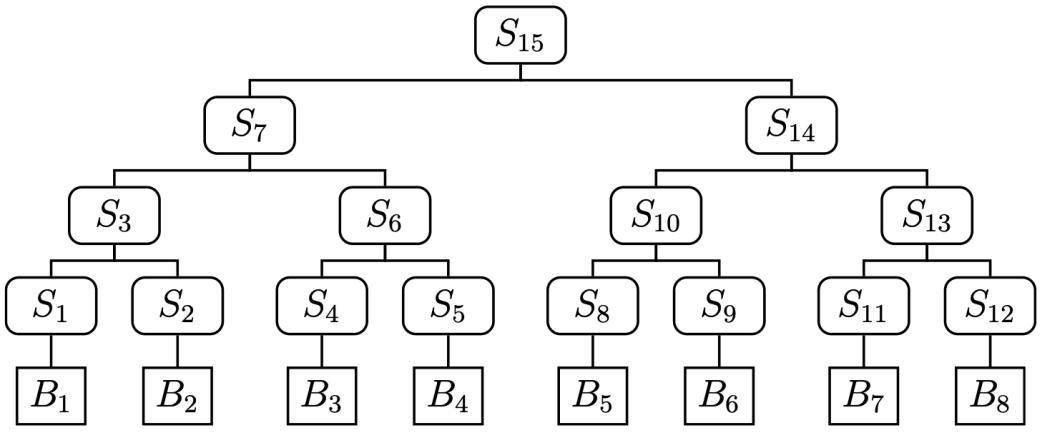


Figure 1 - The coreset tree data structure using merge-reduce technique

2.4 A Streaming Coreset

In order to maintain a small coreset for all points in the data stream we use the merge-and-reduce method that we described above. For a data stream containing n points, the algorithm maintains L buckets, where L is $ceil(log_2(\frac{n}{m}) + 2)$. Bucket B0 can store any number between 0 and m points in contrast, for i >= 1, bucket B1 is either empty or contains exactly m points. The idea of this approach is that at any point of time, if bucket Bi is full it contains a coreset of size m representing 2^(i-1) m points from the data stream.

New data from the data stream are always inserted into the first bucket B0. If bucket B0 is full (i.e., contains m points), all points from B0 need to be moved to bucket B1. If bucket B1 is empty, we are finished. However, if bucket B1 already contains m points, we compute a new coreset Q of size m from the union of the 2m points stored in B0 and B1 by using the coreset construction described above. Now, both buckets B0 and B1 are emptied and the m points from coreset Q are moved into bucket B2 (unless, of course, bucket B2 is also full in which case the process is repeated). Algorithm InsertPoint for inserting a point from the data stream into the buckets is given in Figure 2.

At any point of time, it is possible to compute a coreset of size m for all the points in the data stream that we have seen so far. For this purpose we compute a coreset from the union of the at most $L = m * ceil(log_2(\frac{n}{m}) + 2)$, points that are stored in all buckets B0 to BL-1 by using the coreset tree construction and obtaining the desired coreset of size m.

def buildCoresetTree(m: Int, input: List[Point]): CoresetTree
To add a bucket to the tree: ● Finds the next empty bucket ● Merges all the previous buckets (if any) ● The merged bucket (aka coreset) is added to the next empty bucket ● All previous buckets are cleaned up ● Add the new bucket b to the first bucket ● If all buckets are full merge them and store them to last position
Algorithm 2 - buildCoresetTree: Constructs a coreset tree with size m for a list of points with n dimensions

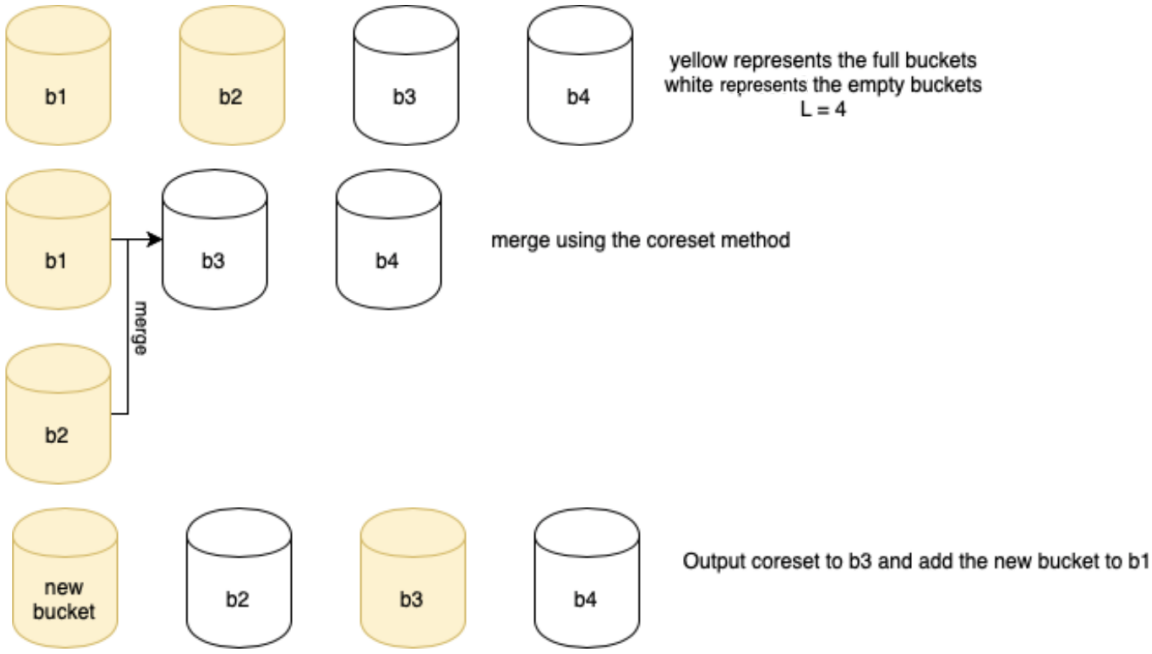


Figure 2 - Visualization of the buildCoresetTree algorithm.

To build the streaming coreset algorithm we used Apache Spark as our data processing framework of choice. More details in the following section.

2.5 Clustering with K-Means++

Clustering is one of the classic problems in machine learning and computational geometry. In the popular k-means formulation, one is given an integer k and a set d of n data points in Rd. The goal is to choose k centers so as to minimize φ, the sum of the squared distances between each point and its closest center.

Usually referred to simply as k-means, Lloyd's algorithm begins with k arbitrary centers, typically chosen uniformly at random from the data points. Each point is then assigned to the nearest center, and each center is recomputed as the center of mass of all points assigned to it. These two steps (assignment and center calculation) are repeated until the process stabilizes. For the purposes of this thesis we are going to use the k-means++ clustering proposed in [6]

2.6 The k-means++ Algorithm

The k-means algorithm begins with an arbitrary set of cluster centers. We used the proposed way of [6] to choose the centers thus, we chose a first random center and then we use the next centers with the D^2 weighting step as shown in Algorithm 3. More specifically, at any given time, let D(x) denote the shortest distance from a data point x to the closest center we have already chosen. We call the following algorithm we call k-means++.

def kmeans++(input: List[Point], k: Int)
Input: a sequential of N population Output: a random sample of size n (n N) 1. a. Choose an initial center c1 uniformly at random from X. b. Choose the next center ci, selecting ci = x′ ∈ X with probability $\frac{D(x')^2}{\sum_{x \epsilon X} D(x)^2}$ c. Repeat Step 1b until we have chosen a total of k centers. 2. For each i ∈ {1,...,k}, set the cluster Ci to be the set of points in X that are closer to ci than they are t o cj for all j $\neq i$ 3. For each i ∈ {1,...,k}, set ci to be the center of mass of all points in Ci: $ci = \frac{1}{\lvert Ci \rvert} \sum_{x \in Ci} x$ 4. Repeat Steps 3 and 4 until C no longer changes.
Algorithm 3. k-means++

Chapter 3

3.1 Stack Overview

3.1.1 Apache Spark

Apache Spark is an open-source distributed general-purpose cluster-computing framework. Spark provides an interface for programming entire clusters with implicit data parallelism and fault tolerance.
Apache Spark has a well-defined layer architecture which is designed on two main abstractions:

- Resilient Distributed Dataset (RDD): RDDs are immutable (read-only) collections of objects that can be processed on many devices at the same time. It's Spark the fundamental collection of Spark. We can divide each dataset of an RDD into logical portions that can exist in different nodes in a cluster.
- Directed Acyclic Graph (DAG): DAG is the scheduling layer of the Apache Spark architecture that implements stage-oriented scheduling. Compared to MapReduce that creates a graph in two stages, Map and Reduce, Apache Spark can create DAGs that contain many stages.

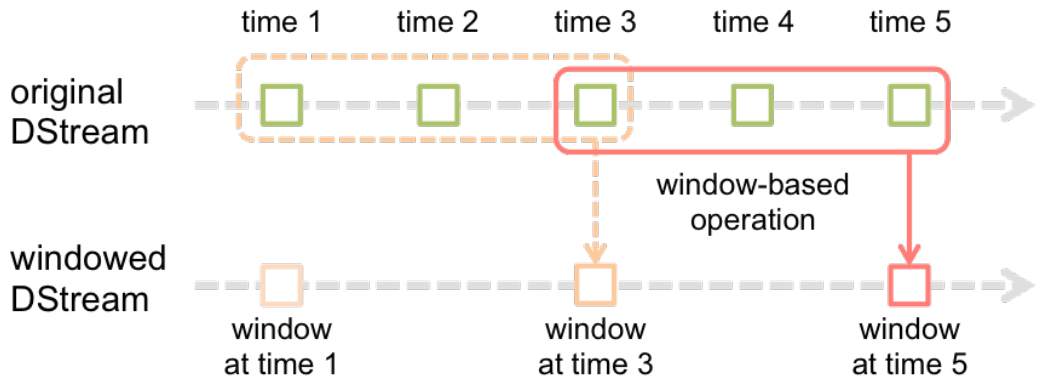
RDDs are the main logical data units in Spark. They are a distributed collection of objects, which are stored in memory or on disks of different machines of a cluster. A single RDD can be divided into multiple logical partitions so that these partitions can be stored and processed on different machines of a cluster.

RDDs are immutable, this means that they are read only in nature. You cannot change an RDD, instead a new RDD can be created as a result of coarse-grain operations that are performed in the original RDD. Such operations can be transformations on an already existing RDD, actions etc. Another useful property of RDDs in Spark is that they can be cashed in memory and used again for future transformations, having the huge benefit of reducing the time for reading and transforming again the RDD. They are lazily evaluated, this means that we perform all the logic that can affect an RDD when it's needed. This saves time and can significantly improve efficiency.

Spark Streaming is an extension of the core Spark API that enables stream processing of live data streams. Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data.
Internally, it works as follows. Spark Streaming receives live input data streams and divides the data into batches, which are then processed by the Spark engine to generate the final stream of results in batches.
Spark Streaming provides a high-level abstraction called discretized stream or DStream, which represents a continuous stream of data. DStream, the basic abstraction in Spark Streaming, consists of sequential RDDs that represents a continuous stream of data and they can be created either from input data streams from sources such as Kafka, and Kinesis, or by applying high-level operations on other DStreams. Internally, a DStream is represented as a sequence of RDDs.

Window Functions

Every time the window slides over a source DStream, the source RDDs that fall within the window are combined and operated upon to produce the RDDs of the windowed DStream.



In the example shown above a function is applied over the last three RDDs, and slides by two RDDs.

The implementation and the experiments are built with Spark 2.2 since this thesis started when the above version was introduced. We also used Kafka as our messaging system of choice.

3.1.2 Apache Kafka

Kafka is a distributed streaming platform that is used to publish and subscribe to streams of records. Kafka is used for fault tolerant storage. Kafka replicates topic log partitions to multiple servers. Kafka is designed to allow your apps to process records as they occur.

3.1.3 Hadoop Distributed File System - HDFS

A distributed file system that handles large data sets running on commodity hardware. Holds a very large amount of data and provides easier access. To store such huge data, the files are stored across multiple machines. HDFS is fault-tolerant and provides high throughput access to application data and is suitable for applications that have large data sets; like spark applications etc.

3.1.4 Prometheus + Grafana

Prometheus is an open-source systems monitoring and alerting toolkit. The metrics are scraped from the job that we instrument directly or by using a push gateway (intermediary) this means that the job sends the metrics in the gateway and then prometheus scraps it. This can be applied to short lived jobs. After Prometheus scraps the metrics it stores them locally and runs queries to them. Aggregating data, or applying rules and it can generate alerts. Then, Grafana, a tool that can consume Prometheus data and visualise them. Grafana allows us to query, visualise and alert on metrics and logs even from other sources like prometheus. Both tools can help instrument jobs and check performance along with other useful metrics.

3.1.5 Tableau

Tableau is another visualisation tool used for Data Analytics. It can be used to visualise - among other data storages - distributed file systems such as HDFS, this can help us understand huge amounts of data. It also provides features that can help to project coordinates in a map. Thus, we used it to visualise the results of our clustering method.

3.2 Algorithm Implementation

Architectural Overview

The algorithm is implemented and evaluated using the architecture shown in Figure 2. First, we consumed from a Kafka topic and using Spark Streaming we processed the incoming data. Each received micro-batch was reduced using the coreset tree algorithm that we proposed and then we performed k-means clustering. For the evaluation part we stored the k-clusters in HDFS and visualized them using Tableau. For the streaming experiments we also kept some metrics in Prometheus that we later used to create graphs in grafana.

Figure 3 - Architecture overview

Using Spark's Scala Api we treat each incoming RDD as follows:

As shown in Figure 4 we converted each incoming rdd to chunks and extracted a coreset using the property 2 and the coreset construction algorithm (algorithm 1).

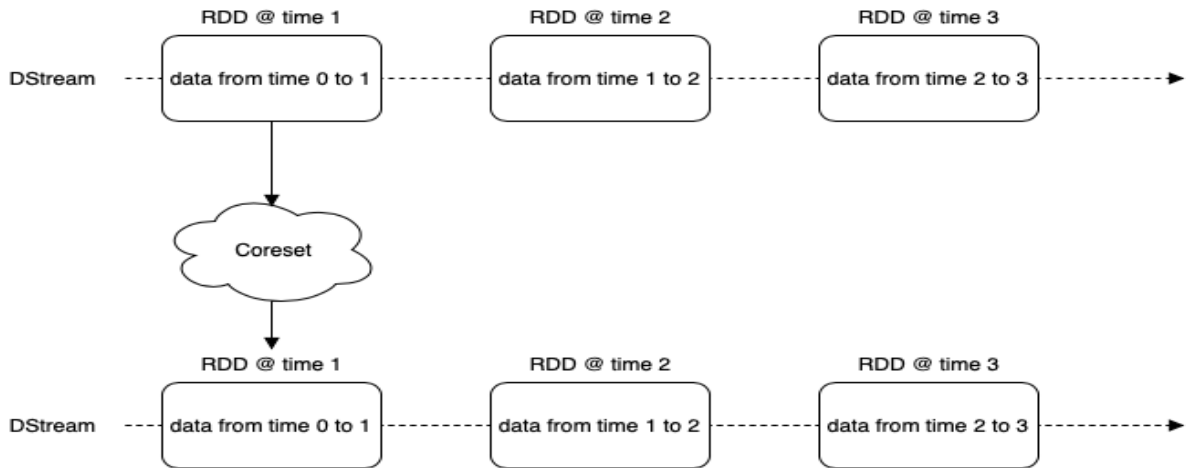


Figure 4 - RDD transformation with buildCoreset algorithm

More specifically, to minimise the time needed to build a coreset for each partition of the RDD we calculated a sub coreset for each one with size m. Parameter m was set dynamically depending on the partition size. As described in figure 5 the union of each coreset provides the final coreset of size.

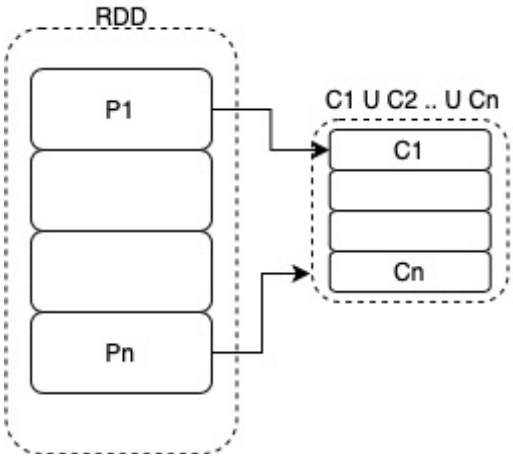


Figure 5 - Coreset construction over an RDD.

Then, (Figure 6) we end up having a DStream in which each RDD is a coreset. Using a sliding window stream, we added each coreset of the rdd to the Tree using the Coreset Tree algorithm (algorithm 2).

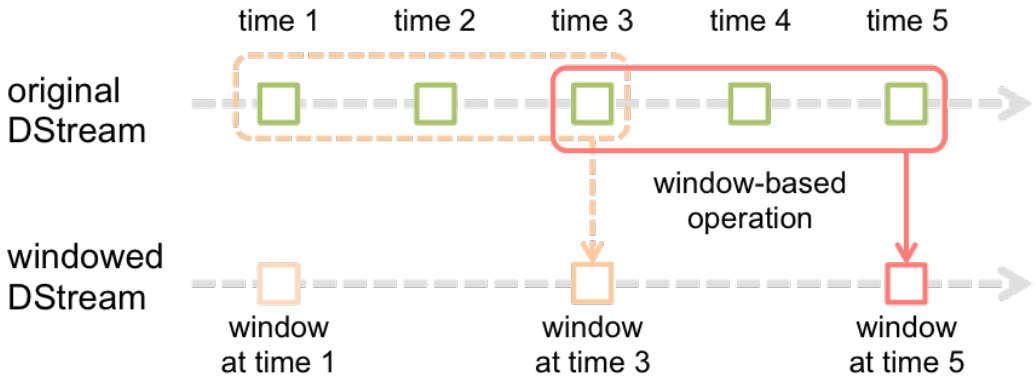


Figure 6 - Window Stream using

Finally, we have a stream that contains the coreset tree of the input so far. When we query to get the k - clusters, a weighted k-means clustering algorithm is applied on the coreset, thus, on a smaller set of points that represents the input.

After that, we can run k-means clustering to obtain k clusters at any given point of time on the m-size coreset. Note that since the size of it is much smaller and independent of the size of the original data stream we can now perform algorithms that can require random access of their input data. We will use k-means++ algorithm in our implementation from [6] on our coreset. Our algorithm follows the same principles as [9] but with some notable differences as described in Chapter 5.

Chapter 4

4.1 Evaluation and Experiments

4.1.1 Silhouette Coefficient

In general silhouette coefficient or silhouette score is a metric used to calculate the goodness of a clustering technique. Its value lies between -1 to 1 inclusive. Where 1 means the clusters are strongly distinguished and apart from one another. 0 represents indifferent clusters without significant distance between them. While -1 means clusters are assigned in a wrong way and there isn't any similarity between points of the same cluster.

Usually, the right number of clusters is not known in advance. Since the k-means objective function drops monotonically as k increases, one needs a different measure for the quality of a clustering that is independent of k. Such a measure is provided by the average silhouette coefficient [8] of the clustering. The silhouette coefficient of a point pi is computed as follows.

- First compute the average distance of pi to the points in the same cluster as pi.
- Then for each cluster C that does not contain pi compute the average distance from pi to all points in C.
- Let bi denote the minimum average distance to these clusters.
- Then the silhouette coefficient of pi is defined as:

$$\text{SC} = \frac{b_i - a_i}{max(a_i,\ b_i)}$$

As stated above the value of the silhouette coefficient of a point varies between –1 and 1.

More specifically, there is a rather subjective interpretation of the silhouette coefficient which is described in the above table.

SC	Proposed Interpretation
0.71 - 1.00	Data have a strong similarity
0.51 - 0.70	A reasonable similarity has been found
0.26 - 0.50	Weak structure that can be artificial
<= 0.25	Data in the same cluster do not have a strong relationship

Table 1 - Subjective Interpretation of Silhouette Coefficient (SC), Defined as the Maximal Average Width for the Entire Data Set

In all of our experiments we evaluate the clusters using the Silhouette Coefficient.

4.2 Experiments

In our experiments we focus on the time needed to extract the coreset from the original dataset while at the same time we observe the quality of the clusters. We leave out the part of the final k-means clustering for these experiments in order to observe the runtime of the coreset construction algorithm and the accuracy of its results.

Our set up consists of a cluster running Spark 2.2

- 10 slaves x 30GB RAM
- 1 master x 15GB RAM
- OS: Ubuntu 16 64 bit
- 1TB Disk / slave
- 4 CPU Cores / slave

4.2.1 Parallelization Experiments

First, we evaluate the pipeline over the same data so we can experiment with the parallelization level of the algorithm. The data are stored in HDFS as orc files, using these files we create a stream and we run our application over this input. The algorithm basically splits the data in buckets using hashing and computes the coresets as we described in the

previous section. All the transformations are done in the executors of each worker, the only time we use the driver is to store the output in HDFS and use the data to evaluate the results. For the evaluation process we computed the silhouette coefficient of the centers and we visualised the centers over the original dataset.

Our input is a dataset with 100.000.000 rows of raw data approximately 2.4 GB that contain latitude and longitude stored in HDFS.

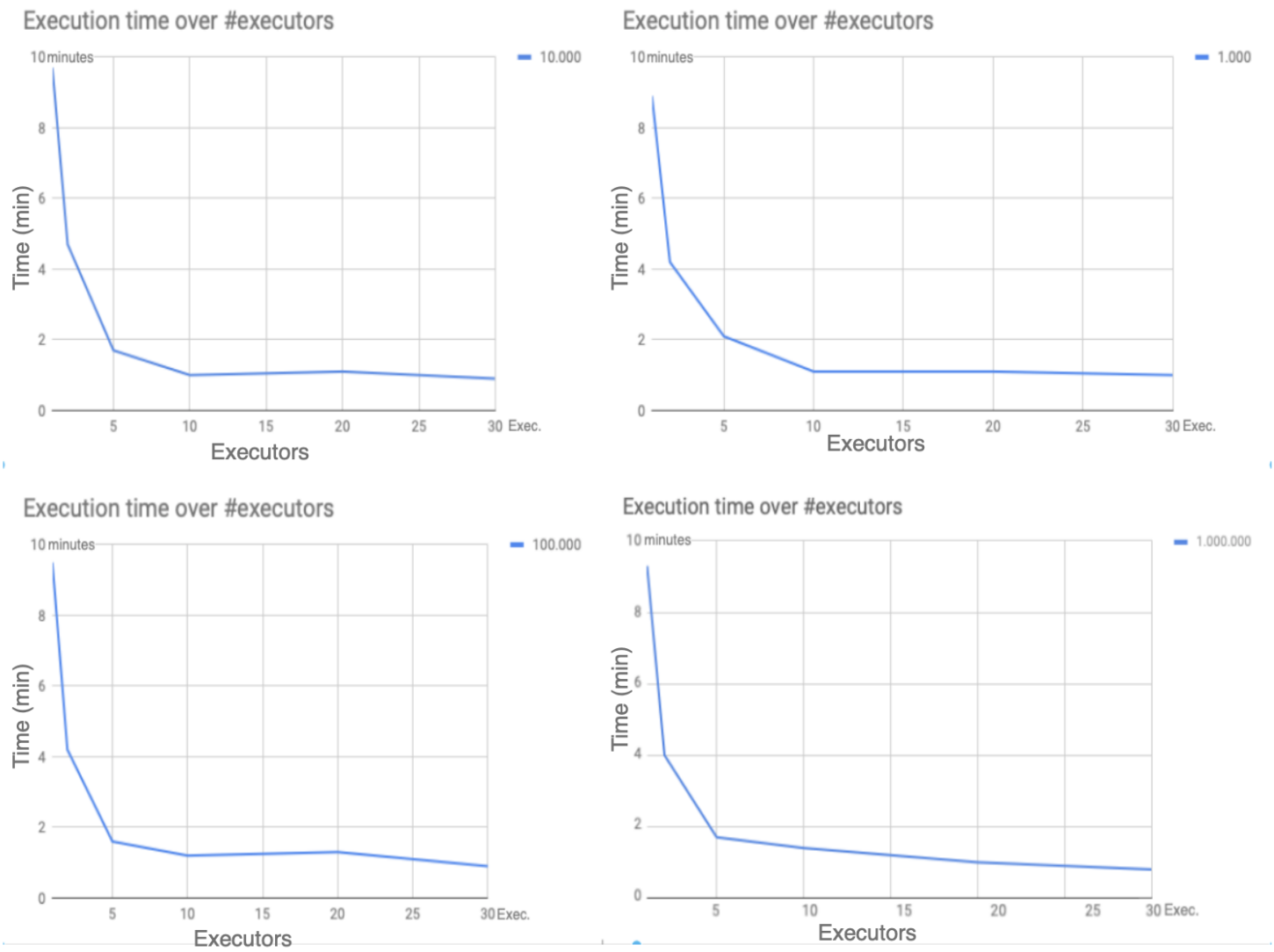


Figure 7 - Comparison between the execution time and the level of parallelization for different coreset sizes. As shown in the graphs with only 1 executor the running time to compute a coreset in different size for an input of 2GB was about 10 minutes. When we increased the executors the time needed was reduced by almost 50%. By adding even more executors we managed to reduce the execution time by more than 90%.

We compare the reduction percentage across different coreset sizes and number of executors. In the table above and using the decrease in percentage formula that is:

decrease% = (starting value - final value) / starting value * 100

Coreset size	1.000.000		100.000		10.000		1.000	
Executors	Time (min)	Decrease in percentage from a single executor (%)	Time (min)	Decrease in percentage from a single executor (%)	Time (min)	Decrease in percentage from a single executor (%)	Time (min)	Decrease in percentage from a single executor (%)
1	9,3	-	9,5	-	9,7	-	8,9	-
2	4	56,98	4,2	55,78	4,7	51,54	4,2	52,80
5	1,7	81,72	1,6	83,15	1,7	82,47	2,1	76,40
10	1,4	84,94	1,2	87,36	1	89,69	1,1	87,64
20	1	89,24	1,3	86,31	1,1	88,65	1,1	88,76
30	0,8	91,39	0,9	90,52	0,9	90,72	1	88,75

Using Spark's User Interface and the available metrics there, we were able to see how different stages are spread in the different executors.

Executor ID ▲		Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Shuffle R
1	stdout stderr	co-dev-hadoop-slave-3.beat:36228	1.0 min	5	0	0	5	125.2 ME
10	stdout stderr	co-dev-hadoop-slave-1.beat:44093	1.1 min	7	0	0	7	175.2 ME
11	stdout stderr	co-dev-hadoop-slave-2.beat:45953	1.0 min	6	0	0	6	150.2 ME
12	stdout stderr	co-dev-hadoop-slave-0.beat:38965	1.1 min	9	0	0	9	225.4 ME
13	stdout stderr	co-dev-hadoop-slave-3.beat:42802	1.1 min	6	0	0	6	150.2 ME
14	stdout stderr	co-dev-hadoop-slave-1.beat:45395	1.0 min	6	0	0	6	150.2 ME
15	stdout stderr	co-dev-hadoop-slave-2.beat:33273	1.1 min	7	0	0	7	175.4 ME
2	stdout stderr	co-dev-hadoop-slave-1.beat:34395	1.1 min	6	0	0	6	150.2 ME
3	stdout stderr	co-dev-hadoop-slave-2.beat:44343	1.1 min	7	0	0	7	175.2 ME
4	stdout stderr	co-dev-hadoop-slave-0.beat:43293	1.1 min	9	0	0	9	225.5 ME
5	stdout stderr	co-dev-hadoop-slave-3.beat:45394	1.0 min	5	0	0	5	125.2 ME
6	stdout stderr	co-dev-hadoop-slave-1.beat:40916	1.0 min	6	0	0	6	150.3 ME
7	stdout stderr	co-dev-hadoop-slave-2.beat:44800	1.1 min	7	0	0	7	175.2 ME
8	stdout stderr	co-dev-hadoop-slave-0.beat:35940	1.1 min	9	0	0	9	225.4 ME
9	stdout stderr	co-dev-hadoop-slave-3.beat:37828	1.0 min	5	0	0	5	125.2 ME

Figure 8 - In all the 15 executors the task time was approximately 1 minute. The tasks along with the input size were equally splitted the different executors. This shows that all of the transformations we performed to extract the coreset on the input were performed wi distributed manner.

We visualised the raw data in the map to check the quality of the clusters after the coreset-tree reduction.

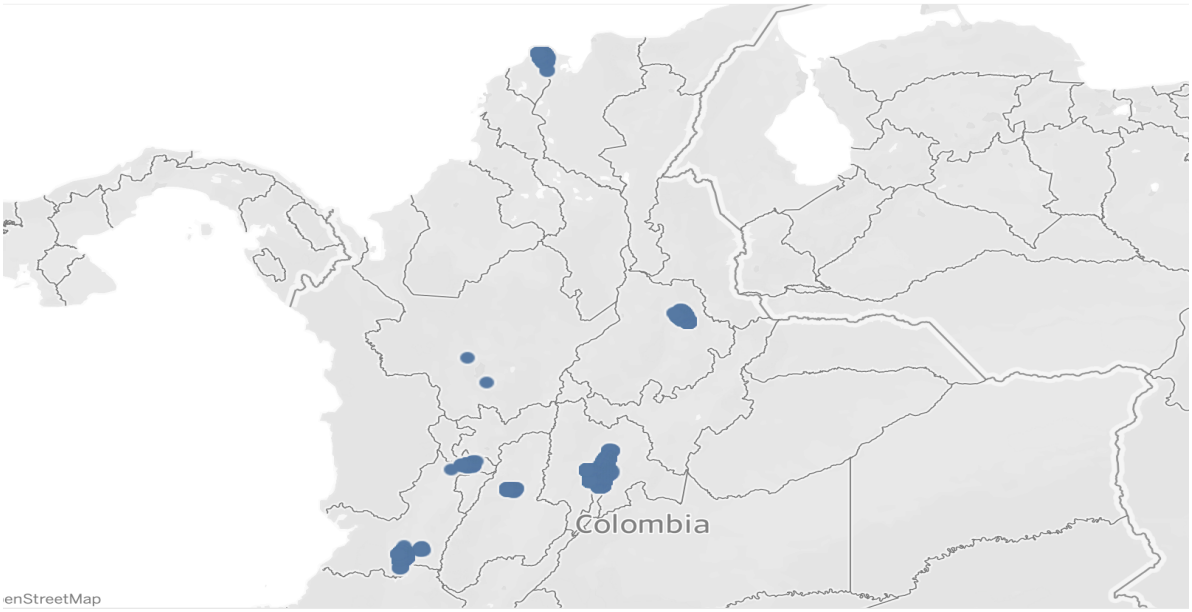


Figure 9 - The raw dataset with coordinates spread across Colombian cities. Most of the occurrences are located in the capital, Bogota.

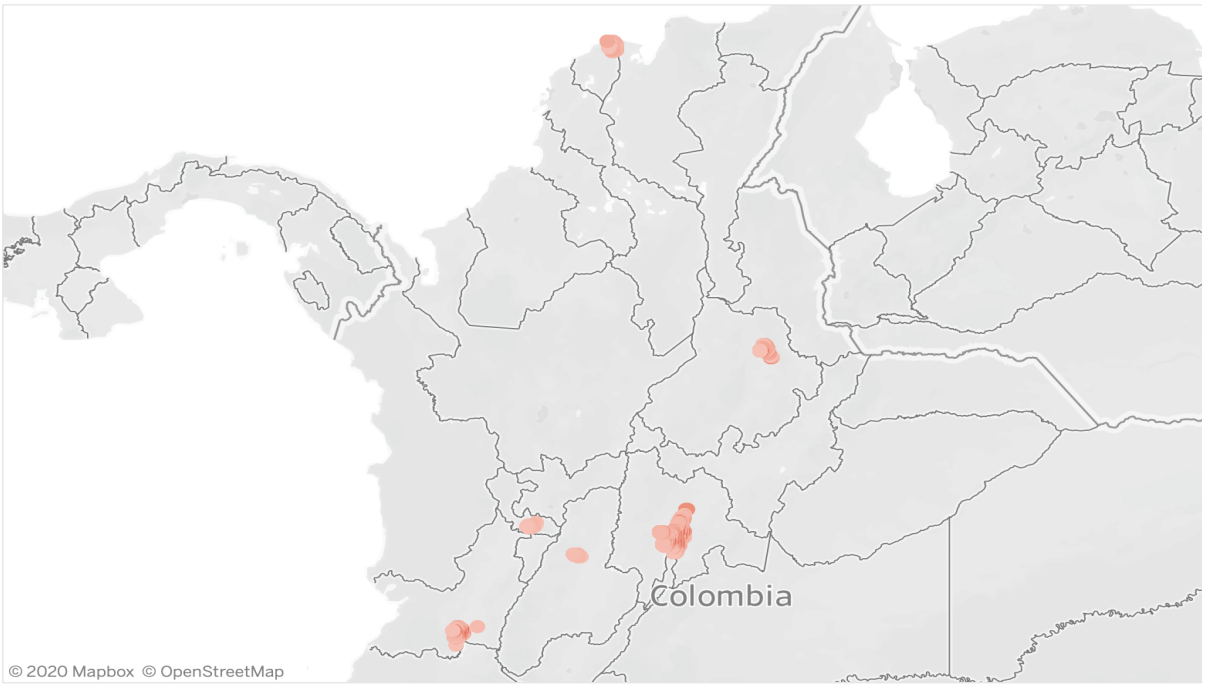


Figure 10 - The representative points of the raw dataset. The size of the resulting dataset after the coreset extraction is 10.000 points. With darker red we can spot the centroids with bigger weight.

Using the aforementioned silhouette coefficient we measured the quality of the centroids.

For the above example we obtained 0.98. This means that all points of the original dataset were correctly assigned to the closest centroid.

We also visualised a small dataset of raw data along with the cluster representatives after the coreset construction. The Birch 1 dataset is a synthetic 2-d data with N=100,000 vectors and we extracted k=1000 clusters. Score measured with Silhouette Coefficient: 0.978

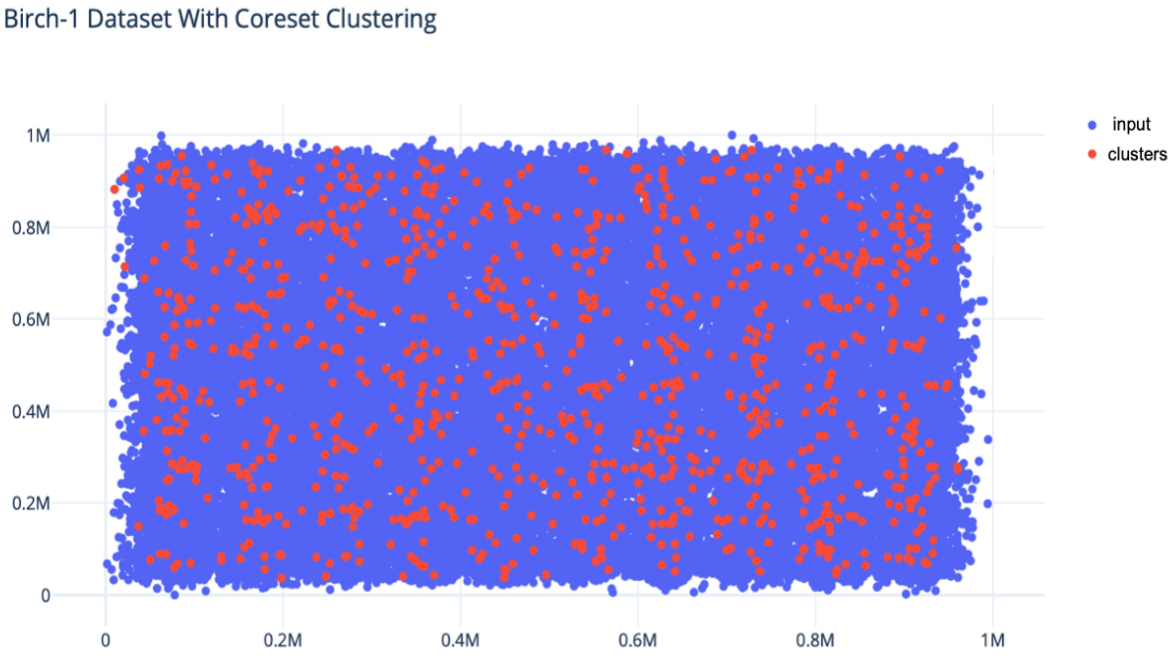


Figure 11 - With the blue dots we represent the 100.000 input points of the birth 1 dataset [7]. Using this input we applied the coreset tree algorithm to extract the 1000 clusters

4.2.2 Streaming Experiments

We also set up a real life scenario where data arrives on a Kafka topic, as our data source we used anonymised data that contained coordinates of different vehicles spread around a city. Each vehicle creates an event every 3 seconds and sends it over a kafka topic. Using Spark we consumed this topic and applied the coreset tree algorithm to reduce the data. The amount of data we got on a rush hour for that city was 200.000 events per 5 minutes. We decided to use a large interval of 5 minutes to increase the amount of data that needed to be processed and observe the time needed for the algorithm to compute the final coreset. The weights of the centroids get updated for every interval. We are able to remember the previous centroids by using the window function we described earlier. This way in every window we use the previous RDD to compute the new centroids and keep track of the changes over time. Finally, we measure the silhouette coefficient of each coreset which is again above 0.9. We were able to calculate the clusters in under 1 second.

The settings we used for this example in our Spark Streaming Application was:

- batch interval - 5 minutes
- coreset size - 500 points
- number of executors - 2

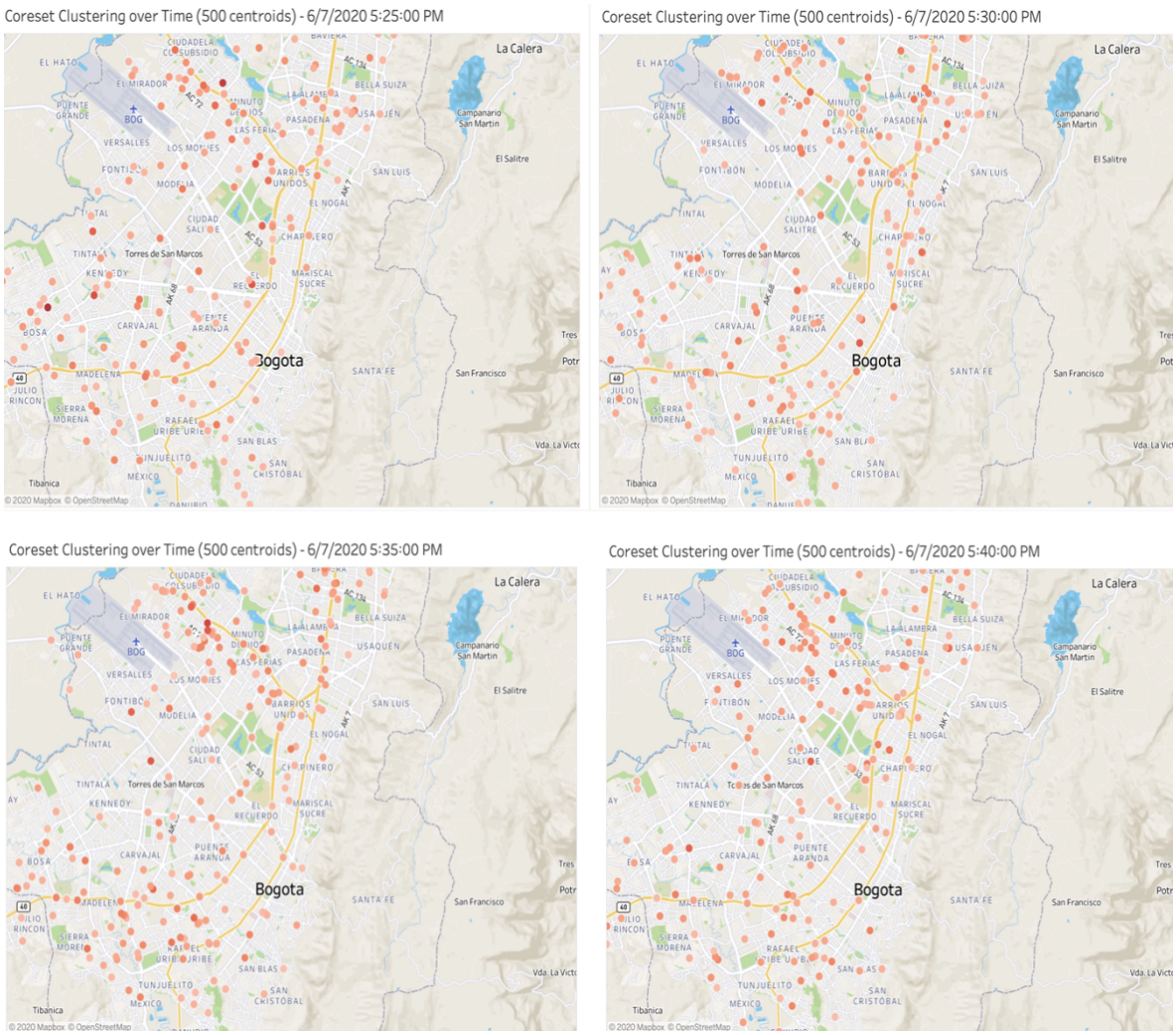


Figure 12 - Extracted coresets over 5 minute intervals in Tableau. Darker red refers to centroids with bigger weight Score measured with Silhouette Coefficient was

Through Prometheus and Grafana we also kept the time needed to create the coreset between the windows. For the aforementioned configuration and a load on Kafka with average 200k messages which included latitude and longitude fields we got the following graph.

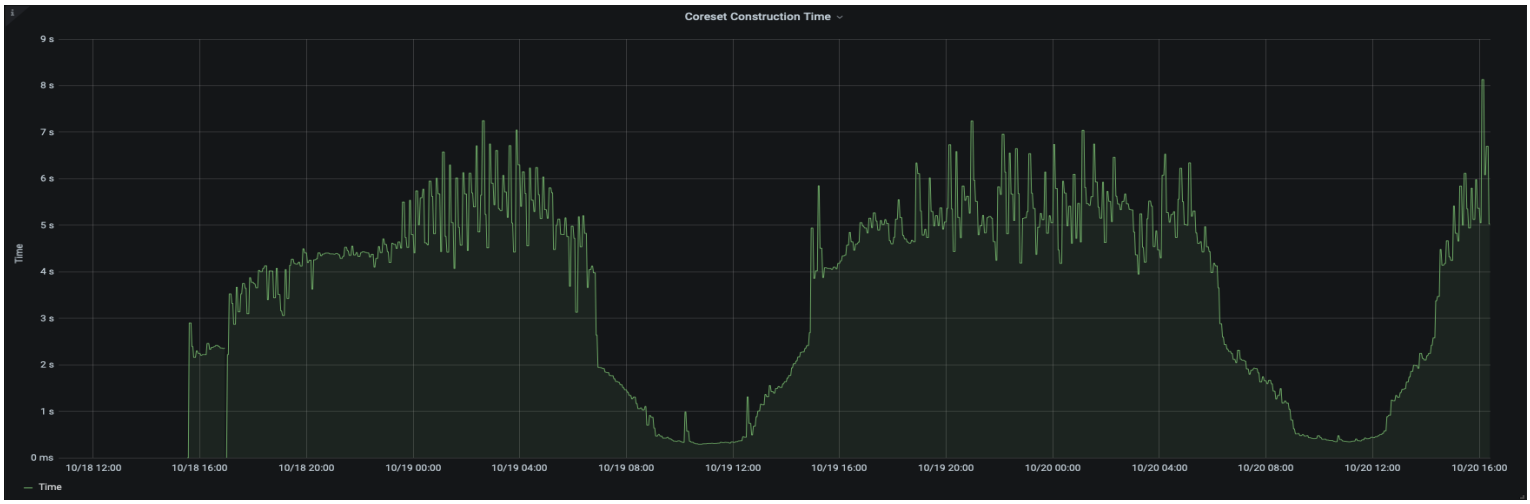


Figure 13 - Coreset construction time over 5 minute batches visualised in Grafana. The required time to extract the

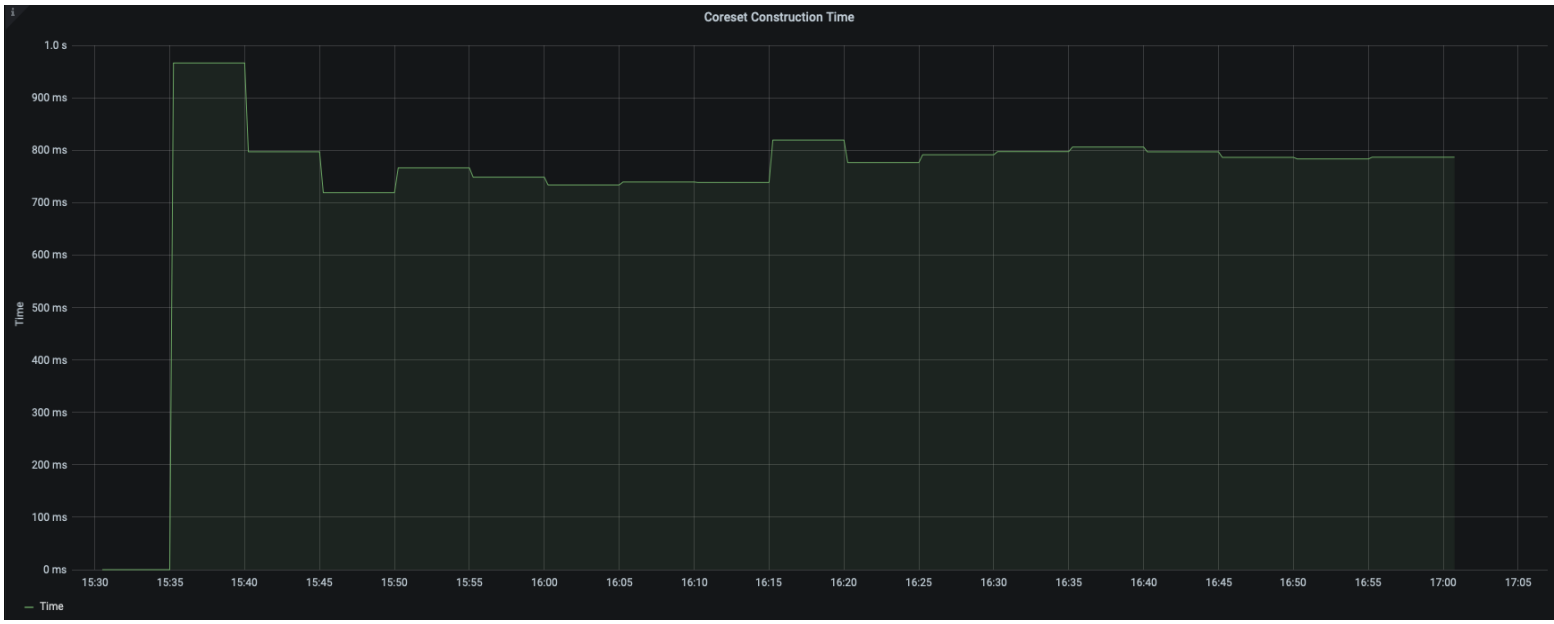


Figure 14 - The time needed to construct a coreset tree of all upcoming batches keeping track of the previous extracted ones. We see that time varies on 3-7 seconds with minimal resources and a parallelisation level equal to 2. In the range of 3-6 seconds we had a volume of 200-300k messages in the Kafka. Note that by the time of the

Completed Batches (last 569 out of 569)

Batch Time	Input Size	Scheduling Delay (?)	Processing Time (?)
2020/10/20 13:20:00	241224 records	0 ms	4 s
2020/10/20 13:15:00	241855 records	0 ms	4 s
2020/10/20 13:10:00	240327 records	0 ms	4 s
2020/10/20 13:05:00	232958 records	1 ms	7 s
2020/10/20 13:00:00	231521 records	0 ms	3 s
2020/10/20 12:55:00	230855 records	0 ms	3 s
2020/10/20 12:50:00	227063 records	0 ms	3 s
2020/10/20 12:45:00	225118 records	0 ms	3 s
2020/10/20 12:40:00	223137 records	1 ms	3 s
2020/10/20 12:35:00	222588 records	0 ms	4 s
2020/10/20 12:30:00	220117 records	1 ms	3 s
2020/10/20 12:25:00	214798 records	0 ms	3 s
2020/10/20 12:20:00	211228 records	0 ms	3 s
2020/10/20 12:15:00	213899 records	0 ms	3 s
2020/10/20 12:10:00	208748 records	1 ms	3 s
2020/10/20 12:05:00	197233 records	0 ms	3 s
2020/10/20 12:00:00	187029 records	1 ms	3 s
2020/10/20 11:55:00	179378 records	0 ms	3 s
2020/10/20 11:50:00	175904 records	0 ms	3 s
2020/10/20 11:45:00	173593 records	1 ms	3 s
2020/10/20 11:40:00	167204 records	0 ms	3 s
2020/10/20 11:35:00	161103 records	0 ms	3 s
2020/10/20 11:30:00	154807 records	1 ms	3 s
2020/10/20 11:25:00	148862 records	1 ms	4 s
2020/10/20 11:20:00	141305 records	0 ms	3 s
2020/10/20 11:15:00	131233 records	0 ms	3 s
2020/10/20 11:10:00	119874 records	1 ms	4 s
2020/10/20 11:05:00	107935 records	0 ms	2 s
2020/10/20 11:00:00	95436 records	1 ms	3 s
2020/10/20 10:55:00	90363 records	0 ms	2 s
2020/10/20 10:50:00	86779 records	0 ms	2 s

Figure 15 - The number of records along with the processing time of a sample of batches visualised in Figure 14.

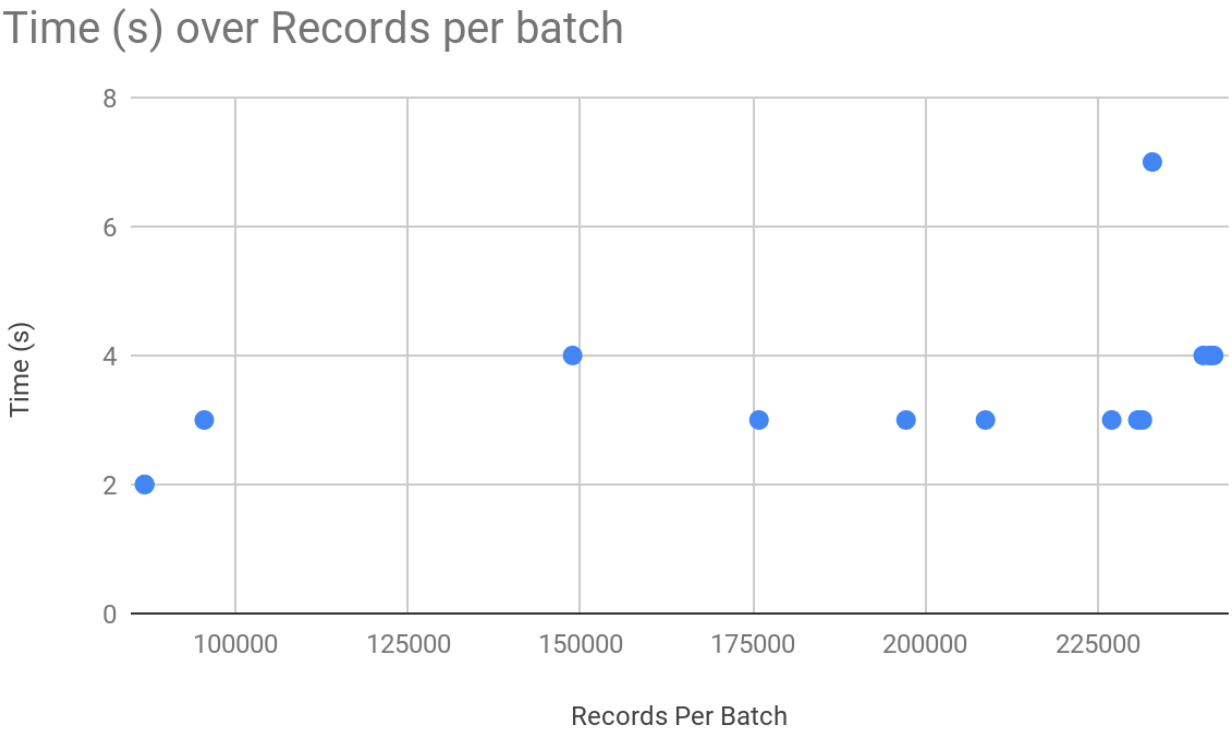


Figure 16 - We compare the number of records with the time needed to process them.

Chapter 5

5.1 Related Work

5.1.1 The StreamKM++ Clustering

StreamKM++ [9] computes a small weighted sample of the data stream, called the coreset of the data stream. Coreset of a set P with respect to some problem- Small subset that approximates the original set P. Solving the problem for the coreset provides an approximate solution for the problem on P.

The open source library called StreamDM [10] is created and maintained by Huawei Noah's Arc. It is written in Scala using Apache Spark 2.3.

In this thesis we follow a similar approach to implement our Streaming Coreset algorithm. Due to some bugs that existed in the original repository the algorithm didn't work as expected. In our implementation we applied some fixes to overcome those issues. Some of them were also applied in the original repository. There were many differences between the two solutions, the most important one was the Tree implementation. To address the issue of keeping the clusters of each RDD and passing them to the incoming input we used a sliding window DStream.

5.1.2 MOA (Massive Online Analysis)

Massive On-line Analysis, here in after called MOA [11, 13] is an open-source framework for mining data streams. It contains a collection of machine learning algorithms and tools to evaluate them. Its source code is written in Java and is related with the WEKA project and it also addresses more demanding problems. MOA aims to create a benchmark framework for streaming data mining algorithms by providing storable settings for real and synthetic data streams for repeatable experiments, implementations for existing algorithms and measures

from the literature for comparison and an easily extendable framework for new streams, algorithms and evaluation methods. The coreset construction algorithm was included in MOA's framework. StreamDM mentioned above was created on top of it.

5.1.3 Streaming k-Means Clustering with Fast Queries

[4] presents the k-means clustering on a stream focusing on providing quick responses to clustering queries. The method provides significant improvements in the query run time for getting cluster centers while in the meantime maintaining all the desirable properties with low space usage and a provably small approximation error in comparison to the current state-of-the-art. The algorithms rely on reusing systematically summaries of the original data (coresets) that are computed for recent queries in answering the current clustering query. Zhang et al. present in depth theoretical analysis and experiments that demonstrate their correctness and efficiency. We used the same approach as in [4] to implement the coreset construction algorithm, using a bucketizer logic as described in the aforementioned paper that is also presented in their repository [12].

5.1.4 Apache Spark MLlib Clustering Algorithms - k-Means++

[26] Spark's mllib is a subproject providing machine learning primitives. Their implementation includes a parallelized variant of the k- means++ method called k-means parallel. A parallel version of the k-means++ initialization algorithm is obtained and empirically demonstrates its practical effectiveness. The main idea is that instead of sampling a single point in each pass of the k-means++ algorithm, O(k) points in each round are sampled and repeat the process for approximately O(log n) rounds.. At the end of the algorithm, O(k log n) points are left from a solution that is within a constant factor away from the optimum. These O(k log n) points into k initial centers for the Lloyd's iteration are clustered again. This initialization algorithm, which is called k-means||, is quite simple and lends itself to easy parallel implementations. However, the analysis of the algorithm turns out to be highly non-trivial, requiring new insights, and is quite different from the analysis of k-means++.

Chapter 6

6.1 Future Work

Our proposed distributed implementation of Streaming Coresets algorithm for clustering data streams was developed in an outdated Apache Spark version. Using Spark Streaming and DStreams we manage to create a streaming pipeline that transforms raw data to coresets. Spark Streaming is a separate library in Spark to process continuously flowing streaming data. The DStream API, which works on top of Spark RDDs. In DStreams data are divided in chunks as RDDs, they are received from a streaming source in order to be processed and afterwards sent to the destination.

Spark Streaming works with the notion of micro batches. We register the streaming pipeline and some lazily evaluated operations, Spark polls that source every predefined interval called batch interval/duration, then a batch with the data is created that contains the incoming records up until the interval is done. Each one of the batches represent an RDD.

There is room for improvement by using Spark's component called Structure Streaming. The same procedure of polling data every interval is applied to Structured Streaming. A distinction makes Structured Streaming a bit more inclined towards real time streaming. There, we do not have the batch concept. The data received, every trigger interval, are appended in a continuously flowing data stream. The result looks like an unbounded table and each row can be processed with the same group of transformations that DStream has. It also provides the functionality to complete, update and append data in the result.

There is also improved performance in the Structured Streaming API, since they are not based on the old RDDs. To sum it up, Structure Streaming would provide lower latency and better performance to our algorithm. Spark Structured Streaming has still microbatches used in the background.

However, it supports event-time processing, quite low latency (but not as low as Apache Flink), there is related work that was proposed by Theodoros Bitsakis on Clustering Big Data Streams in Apache Flink.

References

[1] S. Har-Peled and S. Mazumdar, “On coresets for k-means and k-median clustering,” in STOC, 2004, pp. 291–300.

[2] Ackermann, Lammersen, Märtens, Raupach, Sohler, Swierkot. StreamKM++: A Clustering Algorithm for Data Streams. In Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX 2010)

[3] Turning Big data into tiny data: Constant-size coresets for k-means, PCA and projective clustering Dan Feldman, Melanie Schmidt and Christian Sohler, in SODA '13, pp. 1434–1453

[4] Streaming k-Means Clustering with Fast Queries, Yu Zhang, Kanat Tangwongsan, Srikanta Tirthapura, ICDE 2017, pp. 449-460

[5] Deterministic Coresets for k-Means of Big Sparse Data Artem Barger and Dan Feldman, in Algorithms 2020, issue 13, pp. 92

[6] D. Arthur and S. Vassilvitskii. k-means++: the advantages of careful seeding ACM-SIAM 2017, pp. 1027-1035.

[7] http://cs.joensuu.fi/sipu/datasets/ birch 1 dataset

[8] Leonard Kaufman; Peter J. Rousseeuw (1990). Finding groups in data : An introduction to cluster analysis. Hoboken, NJ: Wiley-Interscience. p. 87

[9] Ackermann, Lammersen, Märtens, Raupach, Sohler, Swierkot. StreamKM++: A Clustering Algorithm for Data Streams. In Proceedings of the 12th Workshop on Algorithm Engineering and Experiments (ALENEX 2010)

[10] https://github.com/huawei-noah/streamDM

[11] https://github.com/Waikato/moa

[12] https://github.com/yzyuzhang/stream-clustering-cache

[13] Albert Bifet, Geoff Holmes, Richard Kirkby, Bernhard Pfahringer (2010); MOA: Massive Online Analysis; Journal of Machine Learning Research 11: 1601-1604

[14] P.Agarwal, S.Har-Peled and R.Varadarajan. Approximating Extent Measures of Point. Journal of the ACM, 51(4): 606–635, 2004.

[15] M. Badoiu and K. Clarkson. Smaller Core-Sets for Balls. Proceed- ings of the 14th Symposium on Discrete Algorithms (SODA'03), pp. 801–802, 2003.

[16] M. Badoiu, S. Har-Peled, and P. Indyk. Approximate Clustering via Coresets. Proceedings of the 34th Annual ACM Symposium on Theory of Computing (STOC'02), pp. 250–257, 2002.

[17] T.Chan. Faster Core-Set Constructions and Data-Stream Algorithms in Fixed Dimensions. Proceedings of the 20th Annual ACM Sympo- sium on Computational Geometry (SoCG'04), pp. 246–258, 2004.

[18] G. Frahling and C. Sohler. Coresets in Dynamic Geometric Data Streams. Proceedings of the 37th Annual ACM Symposium on Theory of Computing (STOC'05), pp. 209–217, 2005.

[19] S. Har-Peled and A. Kushal. Smaller Coresets for k-Median and k-Means Clustering. Proceedings 21st Annual ACM Symposium on Computational Geometry (SoCG'05), pp. 126–134, 2005.

[20] Palmer, C., Faloutsos, C.: Density biased sampling: An improved method for data mining and clustering. In Proceedings of ACM SIGMOD International Conference on Management of Data (2000) 82-92

[21]. Vitter, J.S.: Random sampling with a reservoir. ACM Transactions on Mathematical Software 11(1985) 37-57

[22]. Li, K.-H.: Reservoir-sampling algorithms of time complexity O(n(1 + log(N/n))). ACM Transactions on Mathematical Software 20(1994) 481-493

[23] Devroye, L.: Non-Uniform Random Variate Generation. Springer-Verlag, New York (1986)

[24] Weighted K-Means for Density-Biased Clustering Kittisak Kerdprasop Nittaya Kerdprasop P. Sattayatham, in DaWaK 2005

[25] https://spark.apache.org

[26] Comparative Study of Apache Spark MLlib Clustering Algorithms. Sasan Harifi, Ebrahim Byagowi, and Madjid Khalilian, DMBD 2017, pp. 61-73