TECHNICAL UNIVERSITY OF CRETE

SCHOOL OF ELECTRICAL & COMPUTER ENGINEERING

DIPLOMA THESIS

Clustering Big Data Streams in Apache Flink

by

Theodoros Bitsakis

A thesis submitted in partial fulfillment of the requirements for the degree of
Diploma in Electrical and Computer Engineering
October 11, 2018

Clustering Big Data Streams in Apache Flink

by Theodoros Bitsakis

Abstract

We live in the era of Big Data where massive amounts of information are generated continuously from numerous types of sources. Today's goal is to apply techniques that take into consideration the volume, the variety and the velocity of the data, in order to gain insight that couldn't be revealed with traditional data processing application software. Cluster analysis is a technique that groups a set of objects such that objects in the same group have similar properties. It is commonly used in the fields of machine learning, data mining, statistical data analysis, pattern recognition and bioinformatics. In this thesis, we propose a parallel implementation for the well-known unsupervised learning algorithm, StreamKM++, for clustering data streams in an online fashion. For the development phase, Apache Flink framework is chosen as a distributed streaming engine with high-throughput, low-latency and fault-tolerant computations over unbounded and bounded data streams. Initially, we introduce the theoretical background of the implemented algorithm and the distributed framework. Afterwards, we propose a parallel implementation which computes the set of cluster centers after the consumption of the input dataset. In addition to that, we propose an alternative implementation which produces periodically requests for the re-evaluation of cluster centers. Finally, we develop a program that exploits the Queryable State feature of Flink, in order to allow the user to query the most up-to-date values of the cluster centers. Experimental evaluation shows that by increasing the level of parallelism the running time droops significantly and at the same time the quality of the clustering gets slightly better.

Acknowledgements

First and foremost, I would like to express my sincere gratitude to Prof. Antonios Deligiannakis, my academic supervisor, who provided me with feedback and support throughout this diploma thesis. Furthermore, I would like to thank the rest of my thesis committee members: Prof. Minos Garofalakis and Prof. Vasilis Samoladas, for their useful comments and their time to evaluate this work. Last but not least, I would like to thank my family and all my friends, for encouraging and supporting me all these years.

Contents

Chapter 1

Introduction

We live in the era of Big Data where massive amounts of information are generated continuously from numerous types of sources (e.g. financial transactions, sensor networks). Today’s goal is to apply techniques that take into consideration the volume, the variety and the velocity of the data, in order to gain insight that couldn’t be revealed with traditional data processing application software. One technique to analyze this kind of data is with streaming algorithms, which they process the input data as a sequence of items. Commonly, algorithms of this category perform few passes over the input data within memory and time constraints.

Cluster Analysis or Clustering is the problem to partition a given set of objects into groups (called clusters) such that objects in the same group are more similar (have same properties) than those that belong in different groups. Therefore, the term “cluster” can be thought of as a new entity that is conceptually composed of a group of objects that share common characteristics. The general purpose of cluster analysis is to organize the collected data into meaningful structures so that new knowledge can be discovered from the underling structure of the data.

Cluster analysis has been an essential asset for the fields of machine learning, exploratory data mining, statistical data analysis, pattern recognition and bioinformatics. Clustering methods have been used in many real applications in Finance & Marketing (stock market analysis, characterization of customers into groups based on their purchasing patterns), in Security & Networks (credit card fraud detection, traffic monitoring of sensor networks), in Biology (Transcriptomics, exploration of plant and animal taxonomies), in Meteorology (identification of spatial and climate changes over the years) and in many others.

StreamKM++ [1] is an unsupervised, online learning algorithm for clustering data streams. It belongs to the category of partition based clustering algorithms, where the initial set of objects is divided to a fixed number of clusters. This algorithm computes incrementally a small weighted sample of the data stream and then solves the problem of clustering on the sample by using the k-means++ algorithm [9]. For the construction of the weighted sample, StreamKM++ uses a hierarchical divisive clustering algorithm called Coreset Tree. For the incremental process of the data stream, the algorithm uses a technique called “merge-and-reduce” [15, 16].

In order to propose a scalable parallel implementation of StreamKM++, we use Apache Flink framework [31]. Apache Flink is an open source framework and distributed processing engine for large scale computations over unbounded and bounded data streams. Flink provides APIs for both Stream and Batch processing, and libraries for relational queries, complex event processing scenarios, graph processing and machine learning. In Flink, programs can be written in Java, Scala, Python and SQL, and can be deployed in local, cluster or cloud mode.

1.1 Thesis Outline

In **Chapter 2,** we give a formal definition of the Partition-based clustering problem, also known as k-means clustering. Then, we analyze the fundamental heuristic algorithm to solve this problem, called k-means [14], and the well-known initialization method of k-means, called k-means++. Finally, we give a detailed description of StreamKM++ along with illustrative diagrams.

In **Chapter 3,** we give a brief description of Apache Flink framework. We describe the process that Flink uses to transform the programs to streaming dataflows. Then, we describe how Flink programs are executed in the distributed runtime environment. Finally, we present the different levels of abstraction that Flink offers to develop streaming and batch applications. We give more emphasis on the DataStream API.

In the first section of **Chapter 4**, we propose a parallel implementation for StreamKM++ which computes the set of cluster centers after the consumption of the entire input dataset. In the second section, we describe the evaluation methodology that we use to rank the quality of the produced clustering, and we propose a parallel implementation to compute it. We claim that with minor modifications, we could use the evaluation methodology to predict the cluster of new incoming data points. In the third section, we propose an alternative parallel implementation for StreamKM++ which produces periodically requests for the re-evaluation of the cluster centers. In the last section, we develop a program that allows the user to query in real-time the cluster centers produced by StreamKM++.

In **Chapter 5,** we conduct several experiments on different datasets to evaluate the performance of the parallel implementation. In the first set of experiments, we discuss the trade-off between the runtime and the clustering cost of StreamKM++, and we compare the quality of our clustering with the original non-parallel algorithm. In the second set of experiments, we conduct experiments with different levels of parallelism to evaluate the runtime, the clustering cost and the throughput of our implementation.

Chapter 6 concludes the thesis by presenting the main contributions, and suggests potential directions for future work.

Chapter 2

K-Means Clustering Algorithms

K-means clustering belongs to the category of Partition based clustering, where the number of clusters is fixed to a discrete value “k”. K-means clustering is a technique of vector quantization in signal processing where clusters are represented by a vector of mean values. The optimization problem in Euclidian space is to partition the set of initial data points into k number of clusters, such that the sum of squared distances between the points and their nearest cluster center is minimized. The center of these clusters is called centroid value and it is calculated as the arithmetic mean of the set of points that each cluster represents. The result of the above procedure is the partitioning of the Euclidian space into Voronoi cells. In literature the term “center” is more frequently used than the term “centroid”, so for the rest of this thesis we will refer to the centroid values as cluster centers.

A more formal definition of k-means clustering problem is the following: Given a set of **n** unlabeled observations $\mathbf{x} = \{\mathrm{x}_1, \mathrm{x}_2, \dots, \mathrm{x}_\mathrm{n}\}$, where each observation is a d-dimensional real vector in Euclidian space, the goal is to partition the **n** observations into **k** clusters $\mathbf{c} = \{\mathrm{c}_1, \mathrm{c}_2, \dots, \mathrm{c}_\mathrm{k} | \mathbf{k} \leq \mathbf{n}\}$, so as the following objective function is solved:

$$\mathbf{arg\,min}_{\mathrm{C}} \sum_{\mathbf{i=1}}^{\mathbf{k}} \sum_{\mathbf{x} \in \mathbf{c}_i} ||\mathbf{x} - \boldsymbol{\mu}_\mathbf{i}||_2^2,$$

where $\mathbf{c_i}$ is the set of points that belong to cluster **i**, $\boldsymbol{\mu}_\mathbf{i}$ is the arithmetic mean (center value) of points in $\mathbf{c_i}$ and $||\mathbf{x} - \boldsymbol{\mu}_\mathbf{i}||_2^2$ is the squared Euclidian norm $\mathbf{L^2}$ between **x** and $\boldsymbol{\mu}_\mathbf{i}$.The objective function aims to minimize the within cluster sum of squares.

It is known that solving k-means clustering in Euclidian space is a NP-hard optimization problem [6, 7], even with just two clusters or even in the 2-dimensional space (for any number of clusters) [8]. To overcome this fact, several heuristic algorithms have been proposed during the past years. In the first sector of this chapter we review Lloyd’s algorithm as one of the most widely used heuristics in the field of k-means clustering. In the second chapter we review k-means++ algorithm, which is used for choosing the initial values of cluster centers for the Lloyd’s algorithm. K-means++ offers specific guarantees for the quality of the clustering. Both of these algorithms play an important role for the understanding of StreamKM++, because they are used as building blocks. In the last sector we give a thorough review of the implemented algorithm StreamKM++, along with illustrative diagrams.

2.1 K-Means

Lloyd's algorithm [14], commonly known as k-means algorithm, is by far the most popular and widely used heuristic algorithm to solve the k-means clustering problem [10]. Historically this algorithm has been published in similar variants by E.W. Forgy in 1965 [2], J. B. MacQueen in 1967 [3] and S. P. Lloyd in 1982. K-means belongs to the category of unsupervised learning in the field of machine learning, where there is no need for explicit assignment of a label-class for each data point. Given a set of initial mean values for the centers of the clusters, this algorithm uses an iterative refinement of two optimization steps to produce a final result:

1. **Cluster assignment step**: each data point is assigned to its nearest cluster, based on the squared Euclidian distance between the point and the center value.
2. **Cluster center update step**: each cluster center is recomputed as the mean value of all points assigned to it.

These two local optimization steps continue repeatedly until no more improvement is possible. K-means algorithm needs in advance the specification of the number of clusters and the initial vector for the mean values. To solve this problem a common practice is to choose these values uniformly at random from the data points. Algorithm 2.1 describes a pseudocode implementation of k-means for better understanding. Lines 2-3 compose the cluster assignment step and it easy to observe that this algorithm aims to minimize the within cluster sum of squares by assigning each point to the cluster with the nearest mean. Lines 4-5 compose the cluster center update step that readjusts the center of mass for all points in each cluster.

ALGORITHM 2.1: k-means

```
Input:   arbitrary "k" mean values for centers: $\boldsymbol{\mu} = \{\mu_1, \mu_2, \dots, \mu_K | k \leq n\}$
         (typically chosen uniformly at random from data points),
         "n" points in Euclidian space: $\mathbf{x} = \{x_1, x_2, \dots, x_n\}$
Output:  "k" clusters: $\mathbf{c} = \{c_1, c_2, \dots, c_k\}$

1 Repeat
2     For each $i \in \{1, 2, \dots, n\}$
3         Assign $x_i$ to $c_j$ if $\|x_i - \mu_j\|_2^2 \leq \|x_i - \mu_p\|_2^2, \forall p \in \{1, 2, \dots, k\}, j \neq p$
4     For each $j \in \{1, 2, \dots, k\}$
5         $\mu_j = \frac{1}{|c_j|} \sum_{x' \in c_j} x'$
6 Until center values (μ vector) has not changed
```

Regarding the time complexity, the worst case running time of the algorithm is exponential in the number of input points $2^{\Omega(\sqrt{n})}$ [11, 12]. This bound stands even in the 2-dimensional space. Nevertheless, in the field of smoothed analysis it has been proven [13] that the running time of the algorithm is polynomially bounded. More precisely the authors of [13] prove that for an arbitrary set of n points in $[0,1]^{d}$, if each point is independently perturbed by a normal distribution with mean 0 and standard deviation σ, then the running time of k-means in the produced set is bounded by $O(n^{34}\log^{4}(n)k^{34}d^{8}/\sigma^{6})$. In practice this algorithm is considered to be very fast as it may converge to a local optimum with a few dozens of iterations, but there is no proof of this allegation. Furthermore, if we set the number of iterations to a fixed number "i", then the overall complexity is $\Theta(nkdi)$.

As it concerns the quality of the clustering, k-means is a heuristic algorithm and there is no guarantee that an optimal solution will be found. However it has been proven that the algorithm converges to a local optimum solution [4], but the result may be affected by the choice of initial clusters. For example, Kanungo et al. [5] construct a situation where k-means algorithm converges to a local minimum that is arbitrary bad compared to the optimal solution.

In figure 2.1, we show some applications of k-means algorithm in different datasets (source of diagrams [21]). In all of these diagrams the initial values of centers were chosen uniformly at random from the points in the plane. Diagram (a) shows an optimal solution of the clustering where clusters fit the actual distribution of the points. Diagram (b) shows a suboptimal solution where the two upper left clusters partition the same group of points into two subgroups. Diagram (c) proves that an inappropriate choice of the number of clusters can lead to poor results that contradict the physical shape of the points in the plane. The last two diagrams, (d) – (e), show that even if the choice of the number of clusters is correct, there is no guarantee that the algorithm may find a clustering that matches the "special" shape of the points. This happens due to the fact that k-means algorithm uses as a distance metric the Euclidian distance which is the only criterion when it comes for a point to be assigned to its nearest cluster.

2.2 K-Means++

In the previous sector we showed that the quality of the clustering solution produced by k-means algorithm depends to a great extent on the initial values of cluster centers. David Arthur and Sergei Vassilvitskii proposed in 2007 an algorithm called k-means++ that chooses the initial values of centers and guarantees a clustering solution with a certain quality. This algorithm is based on the idea that it is a "good" technique to scatter out the initial values of the cluster centers according to some probability. More specifically, this algorithm chooses the first center uniformly at random from the data points. Then each subsequent point is chosen as the next center with a probability proportional to its squared distance from the nearest cluster center.

Finally, this algorithm executes the k-means algorithm with the pre-computed cluster centers.

A description of k-means++ is presented by Algorithm 2.2. Lines 1-3 of the algorithm are often called in the literature as the “seeding procedure of the k-means++”. Moreover, we will refer to line 3 of the code as “sampling a point according to the squared distance”.

Regarding the performance of k-means++, the authors prove that the algorithm is guaranteed to find a solution of k-means clustering problem that is $O(\log k)$ competitive compared to the optimal. Furthermore, the empirical evaluation from this paper showed that k-means++ outperforms k-means in both time and accuracy of the clustering.

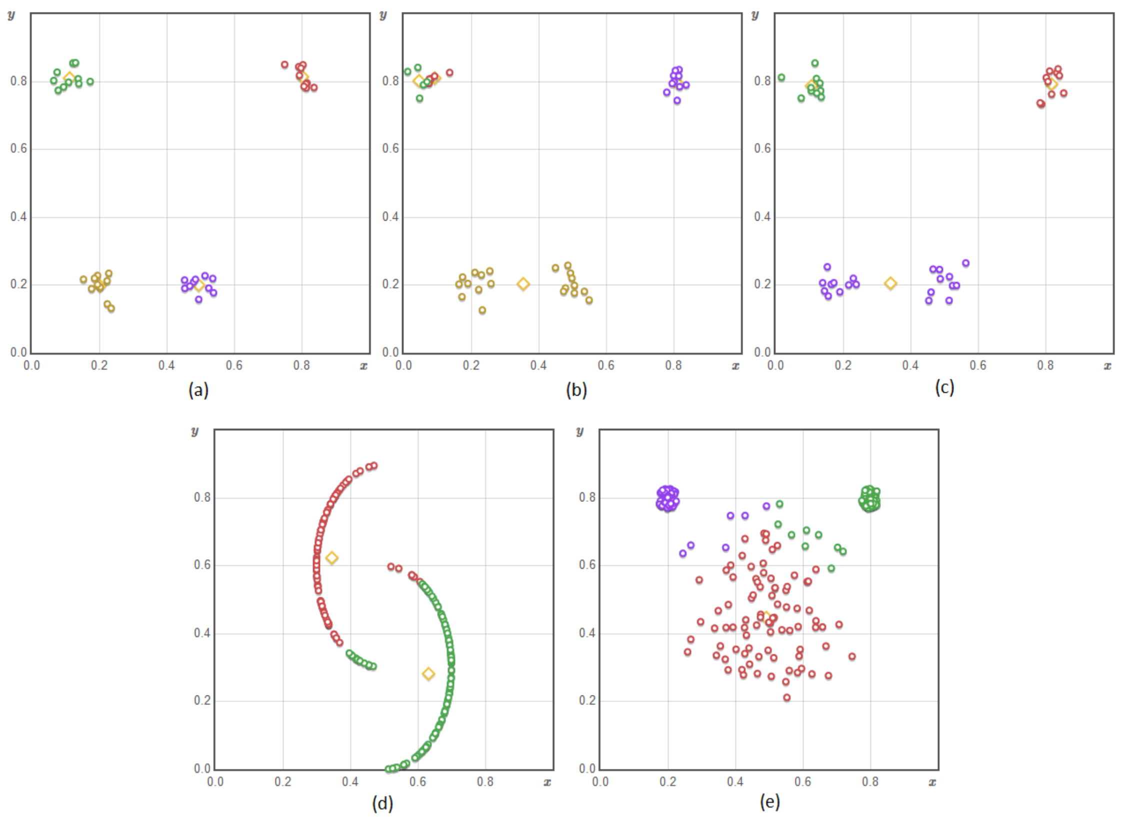


Figure 2.1: K-means paradigms

ALGORITHM 2.2: k-means++

Input: an empty vector for "k" mean values of centers: $\boldsymbol{\mu} = \{\mu_1, \mu_2, \ldots, \mu_K | k \le n\}$
"n" points in Euclidian space: $\mathbf{x} = \{x_1, x_2, \ldots, x_n\}$

Output: "k" clusters: $\mathbf{c} = \{c_1, c_2, \ldots, c_k\}$

```
1 Choose an initial center $\mu_1$ uniformly at random from $\mathbf{x}$
2 For each $j \in \{2, \ldots, k\}$
3     Choose $x_i$ as $\mu_j$ with probability $\frac{\min_{\mu \in \boldsymbol{\mu}} \|x_i - \mu\|_2^2}{\sum_{x \in \mathbf{x}} \min_{\mu \in \boldsymbol{\mu}} \|x - \mu\|_2^2}$ $\forall\, i \in \{1, 2, \ldots, n\}$
4 Execute Algorithm 2.1 with $\boldsymbol{\mu}$ vector
```

2.3 StreamKM++

In real use case scenarios, we want to perform clustering techniques in unbounded sequences of data objects or in large static datasets stored in the hard disk. Classical clustering algorithms, such as k-means and k-means++, fail to perform in the above cases because they need random access to the input data.

StreamKM++ is an on-line algorithm that computes incrementally a small weighted sample of the data stream and then solves the problem of clustering on the sample by using the *k*-means++ algorithm. For the construction of the weighted sample the algorithm uses coreset structures. In general, a coreset is a small set of weighted points that approximates, within an error factor, the original set of points. Typically, the construction process of the coreset depends on the type of optimization problem. The authors propose two types of coresets:

1. A coreset called “Adaptive Coreset”. This coreset is basically the seeding procedure of k-means++ with minor differences.
2. A coreset called “Coreset Tree”. This coreset is underneath a binary tree that is generated with a hierarchical divisive clustering algorithm. The purpose of coreset tree is to speed up the process of sampling that k-mean++ performs.

For the incremental process of the data stream the algorithm uses a technique called “merge-and-reduce” [15, 16]. This technique consumes the data stream with buffers of specific size and at the same time it merges these buffer so that a weighted sample can be found with the use of coreset tree.

The authors compare this algorithm with k-means, k-means++ and two other well-known data stream clustering algorithms, BIRCH [17] and STREAMLS [18]. Briefly, BIRCH is a single-pass algorithm that computes a preclustering of the data stream by summarizing dense regions of points. After the preclustering phase, it uses a hierarchical agglomerative clustering algorithm to calculate the final clusters. This algorithm is super-fast, but it computes clusters with relatively low quality. On the other hand, STREAMLS partitions the input stream into chunks and for each chunk uses a local search algorithm to compute a clustering. Afterwards, the local search algorithm is applied one more time on the union of the previous clusterings. STREAMLS is slower than BIRCH but it computes clusterings with much better quality. The empirical evaluation of StremKM++, shows that the quality of the clustering is comparable (in terms of square error) with STREAMLS, k-means++ and much better than BIRCH. Regarding the time of the execution, StreamKM++ is slower than BIRCH but faster than STREAMLS. More specifically, StreamKM++ seems to scale better with the increment in the number of clusters. Finally, StreamKM++ outperforms k-means in all the experiments. We strongly encourage the reader to take a deeper look in the empirical evaluation from the original paper. In the following sectors we will only focus to the building blocks of StreamKM++.

2.3.1 Preliminaries

In this sector we describe the basic mathematical definitions as these were presented by the authors of StreamKM++. For any two points $x, y \in \mathbb{R}^d$ and any set of points $C \subset \mathbb{R}^d$ we define:

- $D(x,\ y) = \|x - y\|_2$, the Euclidian distance between x and y.
- $D(x,\ C) = \min_{c \in C} D(x,\ c)$, the minimum Euclidian distance between x and any point in set C.
- $D^2(x,\ y) = \|x - y\|_2^2$, the squared Euclidian distance between x and y.
- $D^2(x,\ C) = \min_{c \in C} D^2(x,\ c)$, the minimum squared Euclidian distance between x and any point in set C.

Using the above definitions we give a formal definition of the Euclidian k-means clustering problem:

Definition 2.1 (Euclidian k-means Clustering problem). For a set $P \subset \mathbb{R}^d$ and $k \in \mathbb{N}$, the Euclidean k-means clustering problem is to find a set of centers $C \subset \mathbb{R}^d$ with $|C| = k$, such that

$$\mathrm{cost}(P, C) = \sum_{x \in P} D^2(x, C),$$

is minimized. We define $\mathrm{cost}(P, C)$ as the cost of the k-means clustering for a set of points P with a set of centers C. The minimization of $\mathrm{cost}(P, C)$ satisfies the square error function of Chapter 2. Similarly, for a weighted set of points $S \subset \mathbb{R}^d$ with weight function $w : S \rightarrow \mathbb{R}_{\geq 0}$ and $k \in \mathbb{N}$, the weighted Euclidean k-means clustering problem is to find a set of centers $C \subset \mathbb{R}^d$ with $|C| = k$, such that

$$\mathrm{cost}_w(S, C) = \sum_{y \in S} w(y) D^2(y, C),$$

is minimized. We define $\mathrm{cost}_w(S, C)$ as the cost of the weighted k-means clustering for a set of weighted points S with a set of centers C.

2.3.2 Coreset Definition

A key feature of StreamKM++ is the use of coreset structures for the computation of the small weighted sample of the data stream. In this sector we give a formal definition of coresets in the scope of k-means clustering problem.

A coreset extraction from an initial set of points P, corresponds to a small weighted set of points S, in which the weighted cost of k-means clustering approximates the clustering cost of the original set P within a small relative error, for any set of cluster centers C. The significance of coreset structures in k-means clustering problem is that we can apply any k-means clustering algorithm of our choice and obtain an approximate clustering result in less time. A more formal definition from the authors is presented:

Definition 2.2 (Coreset for k-means Clustering problem). Let $\mathrm{k} \in \mathbb{N}$ and ε with $0 < \varepsilon \leq 1$ be a precision parameter. A weighted multiset $\mathrm{S} \subset \mathbb{R}^{\mathrm{d}}$ with positive weight function $\mathrm{w} : \mathrm{S} \rightarrow \mathbb{R}_{\geq 0}$ is called (κ, ε)-coreset of P for the k-means clustering problem if, for each $\mathrm{C} \subset \mathbb{R}^{\mathrm{d}}$ of size $|\mathrm{C}| = \mathrm{k}$, we have

$$(1-\varepsilon)\mathrm{cost}(\mathrm{P},\mathrm{C}) \leq \mathrm{cost}_{\mathrm{w}}(\mathrm{S},\mathrm{C}) \leq (1+\varepsilon)\mathrm{cost}(\mathrm{P},\mathrm{C}).$$

2.3.3 Coreset Construction

For the extraction of the weighted sample of the data stream, the authors propose two novel coreset structures. The first one is called “adaptive coreset” and the second one “coreset tree”. The construction of both of these coresets is based on the k-means++ seeding procedure which works well for high dimensional datasets.

Adaptive Coreset

This coreset is quite easy to implement (see Algorithm 2.3) and the authors provide a formal proof that this structure is indeed a (κ, ε)-coreset (Theorem 2.1).

More specifically, given a set of points $\mathrm{P} \subset \mathbb{R}^{\mathrm{d}}$ with size $|\mathrm{P}| = \mathrm{n}$, the algorithm initially chooses a set of points $\mathrm{S} = \{\mathrm{q}_1, \mathrm{q}_2, \ldots, \mathrm{q}_\mathrm{m}\}$ of size m $(\mathrm{m} \leq \mathrm{n})$ at random according D^2, with the same methodology as k-means++ does. Afterwards, it assigns every point of the set P to its nearest point q_i, denoting a new set called Q_i that contains these assignments. Finally, by using the weight function $\mathrm{w} : \mathrm{S} \rightarrow \mathbb{R}_{\geq 0}$ with $\mathrm{w}(\mathrm{q}_\mathrm{i}) = |\mathrm{Q}_\mathrm{i}|$, it obtains the weighted set S as the resulted coreset. The basic differences between this algorithm and k-means++ are that this algorithms uses a larger number of initial centers $(\mathrm{k} \leq \mathrm{m} \leq \mathrm{n})$ and that it assigns to every center a weight attribute that is computed from the set of points that are nearest to it.

Regarding the performance of the algorithm, we have to mention that the size bound of the coreset proven by the theorem 2.1 is not tight, as it depends on the dimensionality of the points. Nevertheless, the authors claim that in practice a size of $\mathrm{m} = 200\mathrm{k}$ is sufficient, based on the experimental evaluation of the algorithm. The major drawback of this algorithm is that is still iterates many times over the initial point set S. More precisely, the overall computational complexity is $\Theta(\mathrm{dnm})$.

```
ALGORITHM 2.3: AdaptiveCoreset(P, m)
1   choose an initial coreset point $q_1$ uniformly at random from P
2   $w(q_1) \leftarrow 0$
3   $S \leftarrow \{q_1\}$
4   for $i \leftarrow 2$ to $m$
5       choose $q_i$ at random according to $D^2$ from P
6       $w(q_i) \leftarrow 0$
7       $S \leftarrow S \cup \{q_i\}$
8   for each $p \in P$
9       let $q_i \in S, 1 \leq i \leq m$, be the nearest coreset point to p
10      $w(q_i) \leftarrow w(q_i) + 1$
```

Therorem 2.1 Let $k \in \mathbb{N}$, let ε with $0 < \varepsilon \leq 1$ be a precision parameter and let δ with $0 < \delta < 1$ be an error probability. Given a point set $P \subset \mathbb{R}^d$ of size $|P| = n$ and a size parameter

$$m = \left(\frac{d}{\delta\varepsilon}\right)^{O(d)} \cdot k \cdot \log(n) \cdot \log^{\frac{d}{2}}\left(\frac{k\log(n)}{\delta\varepsilon}\right),$$

algorithm AdaptiveCoreset computes a weighted multiset S with size m that is a $(\kappa,\ 6\varepsilon)$-coreset of P with probability at least $1 - \delta$ (log(.) denotes the binary logarithm to the base 2).

Coreset Tree

This type of coreset was proposed by the authors as a solution to reduce the computational complexity of the coreset constructed by AdaptiveCoreset algorithm, from $\Theta(dnm)$ to $O(dm^2)$. The coreset tree structure is basically a binary tree that is constructed with a hierarchical divisive clustering algorithm applied to the initial set of points P. When the construction of the binary tree is completed, the union of its leaves constitutes the set of points of the coreset. The significant advantage of this technique is that it allows as to perform fast sampling of the input data, using subsets of points from the initial set P. Figure 2.2 illustrates a possible instance of a coreset tree for a set of 10 points in the plane.

A brief description of the hierarchical divisive clustering algorithm is the following: At the beginning, the algorithm starts with a single cluster that contains the whole point set P. Afterwards, it successively partitions the existing clusters into two subclusters, such that the points in one subcluster are far from the points in the other subcluster. The division step is repeated until the number of clusters corresponds to the desired number of points of the coreset.

Definition of the Coreset Tree

In this paragraph we define the basic properties of a coreset tree (T) and the attributes that each node of the tree stores. A coreset tree that is constructed with the previous hierarchical divisive technique must satisfy the following properties:

- Each node of *T* is associated with a cluster in the hierarchical divisive clustering.
- The root of *T* is associated with the single cluster that contains the whole point set *P*.
- The nodes associated with the two subclusters of a cluster *C* are the child nodes of the node associated with *C*.

Each node "v" of the coreset tree (T), stores the following attributes:

- A point set $\mathrm{P_v}$: This attribute contains the points of the cluster associated with node v. It has only to be stored explicitly in the leaf nodes of the tree. For an inner node, the set $\mathrm{P_v}$ is defined by the union of the point sets of its children.
- A representative point $\mathrm{q_v}$ from $\mathrm{P_v}$: This attribute is stored only in the leaf nodes of the tree. At any point of time the union of these points represent the points that have been chosen so far to be the points of the final coreset.
- A weight value $\mathrm{weight(v)}$: For a leaf node, this attribute represents the sum of squared distances over all points in $\mathrm{P_v}$ to $\mathrm{q_v}$, that is $\mathrm{cost(P_v, q_v)}$. For an inner node, $\mathrm{weight(v)}$ is computed as the sum of the weights of its children.

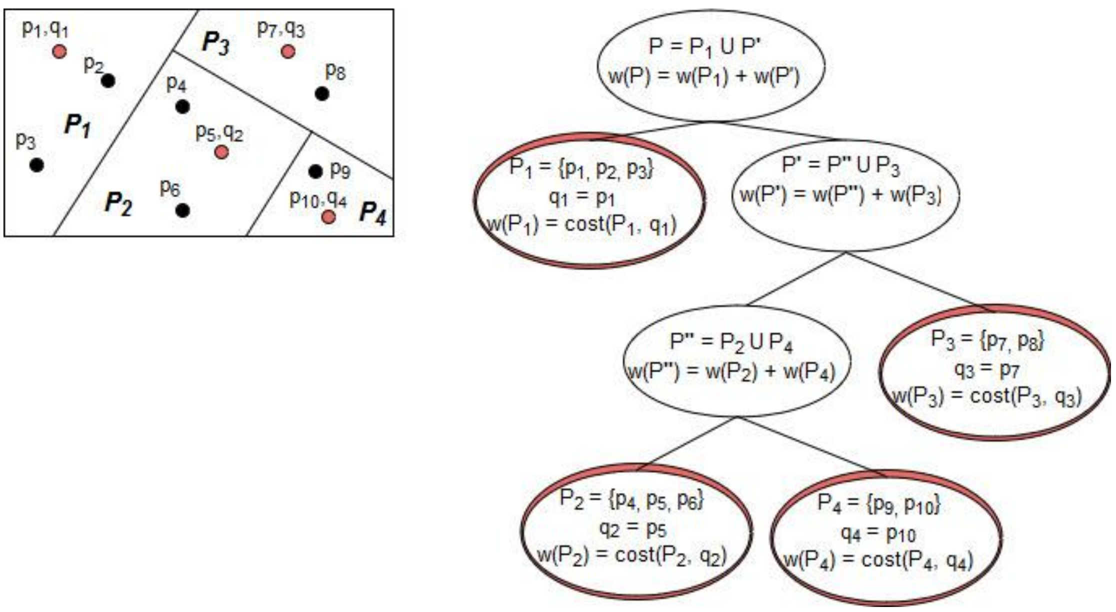


Figure 2.2: Example of a coreset tree for a set of 10 points in the plane.
The representative points are chosen in the order $\mathrm{q_1, q_2, q_3, q_4}$.

Construction of the Coreset Tree

We move on the methodology for the construction of the coreset tree (see algorithm 2.4). As we mentioned before, the purpose of the coreset tree is to construct a coreset $S = \{q_1, q_2, \dots, q_m\}$, given a set of points $P = \{p_1, p_2, \dots, p_n\}$ with $m \leq n$. The construction of the tree (T) starts with a single node, the root, which contains the whole point set P. The first representative point q_1 is chosen from the root, uniformly at random from the point set P. Then, assuming that the tree contains already "i" leaf nodes, with $1 < i \leq m$, and therefore has already computed the representative points $q_1, q_2, \dots, q_i$ and the corresponding point sets $P_1, P_2, \dots, P_i$, the next representative point q_{i+1} is sampled by performing the following three steps:

1. We choose a leaf node ℓ at random with a probability proportional to $\mathrm{cost}(P_\ell, q_\ell)$: More specifically, we start from the root of the tree and then we iteratively select the inner nodes until we reach a leaf node. In this process, a child node u is chosen from the current node v, with probability $\mathrm{weight}(u)/\mathrm{weight}(v)$. Following this procedure, a leaf node is chosen among the others with probability $\mathrm{cost}(P_\ell, q_\ell)/\sum_{j=1}^{i} \mathrm{cost}(P_j, q_j)$.
2. We chose the next representative point q_{i+1} from the subset P_ℓ with a probability proportional to its squared distance to the representative point q_ℓ. In this step, each point p of the set P_ℓ is chosen among the other points with probability $D^2(p, q_\ell)/\mathrm{cost}(P_\ell, q_\ell)$.
3. We split the current node based on q_ℓ and q_{i+1}, in order to create two new child nodes ℓ_1 and ℓ_2: At first, we store at node ℓ_1 the point q_ℓ and at node ℓ_2 the new representative point q_{i+1}. Then, we assign to each node the points of P_ℓ that are nearest to its representative point. More formally, $P_{\ell_1} = \{p \in P_\ell | D(p, q_\ell) < D(p, q_{i+1})\}$ and $P_{\ell_2} = \{p \in P_\ell | D(p, q_{i+1}) \leq D(p, q_\ell)\}$. Finally, we compute the new weight attribute of every child and we propagate this update until we reach the root of the tree.

The previous three steps are executed until the number of the leaf nodes matches the desired number of the coreset points (i.e. m points). When the construction of the tree is completed, we obtain the point set $S = \{q_1, q_2, \dots, q_m\}$, from the representative points of leaf nodes. We denote the weight of each $q \in S$ as the total number of points in each subset P of the corresponding leaf node. Figure 2.3 illustrates the construction process of the coreset tree presented by figure 2.2.

ALGORITHM 2.4: TreeCoreset(P, m)

```
1  choose $q_1$ uniformly at random from P
2  construct a root node with $q_{root} = q_1$ and $weight(root) = cost(P, q_l)$
3  $S \leftarrow \{q_1\}$
4  for i ← 2 to m
5      start at root, iteratively select one of the two child nodes at random
       according to their weights, until a leaf $\ell$ is chosen
6      choose $q_i$ according to $D^2$ from $P_\ell$
7      $S \leftarrow S \cup \{q_i\}$
8      create two child nodes, $\ell_1$ and $\ell_2$ from $\ell$, and update $weight(\ell)$
9      propagate the update of weight attribute upwards to the root node
```

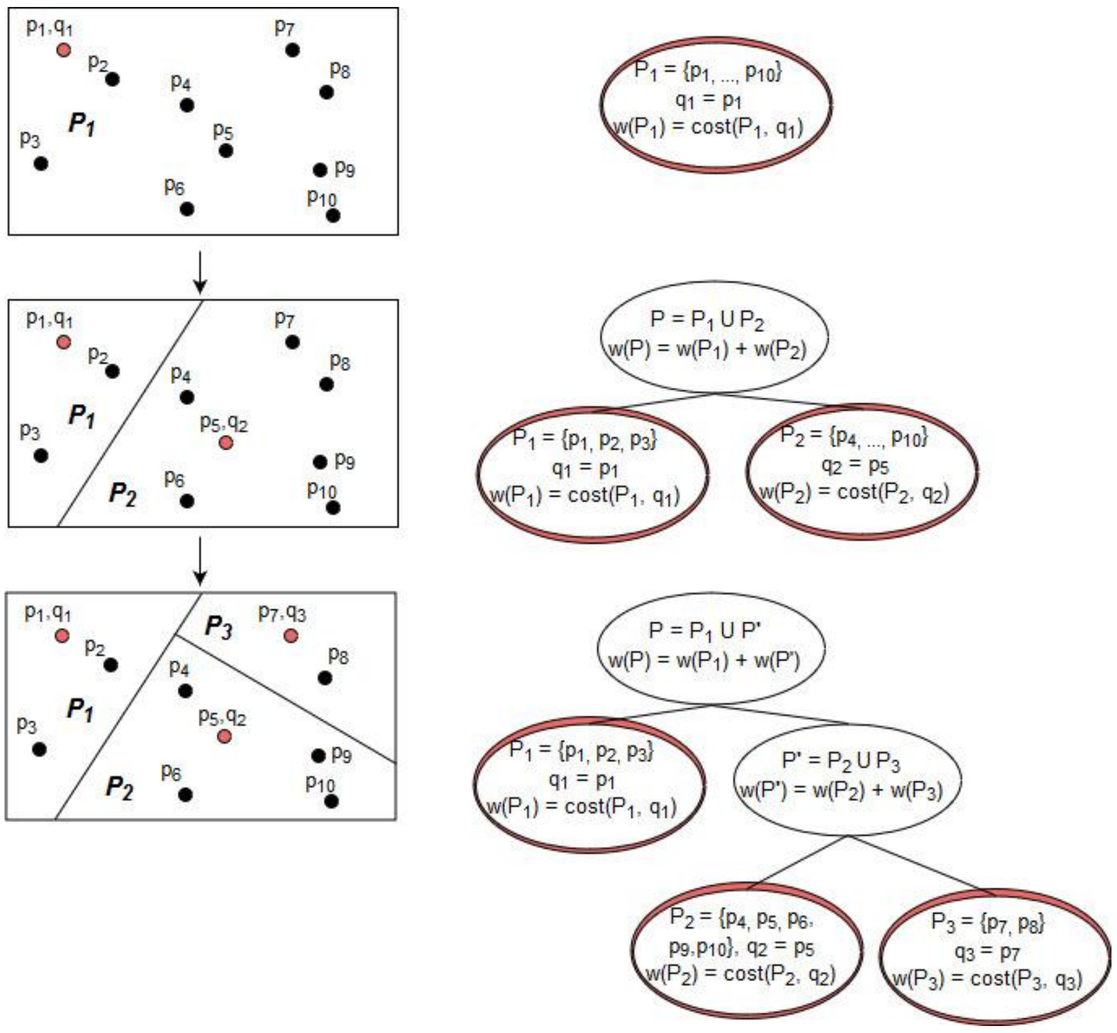


Figure 2.3: Construction process of the coreset tree presented by fig. 2.2

2.3.4 The Streaming Algorithm

In this section we describe the clustering algorithm of StreamKM++. The algorithm uses a technique called merge-and-reduce, which consumes the input stream of points with buffers of specific size. At any point of time that two buffers represent the same number of consumed points, this technique takes the union of these buffers (merge step) and then it applies to the unified set the coreset tree algorithm in order to maintain a small set of representative points. When the stream of points is consumed, or generally when there is a need to find a clustering from the points that have already been consumed, we can extract the final coreset and apply to it any k-means clustering algorithm of our choice.

The Merge and Reduce Technique

We proceed to the formal description of the technique (see algorithm 2.5). Given the size $|S| = m$ of the coreset as a fixed size parameter and the size $|P| = n$ of the data stream, the algorithm maintains $L = \lceil \log_2(n/m) + 2 \rceil$ buckets $B_0, B_1, \dots, B_{L-1}$. For these buckets the following properties must be satisfied:

1. Bucket B_0 can store any number of points between 0 and m, whereas bucket B_i with $i > 0$ must be either empty or have exactly m points.
2. At any point of time if bucket B_i with $i > 0$ is full, then it contains a coreset of size m that represents $2^{i-1}m$ points of the data stream. Bucket B_0 is used as an input buffer for the incoming points of the stream.

Given the above properties, the algorithm consumes the data stream with the following procedure: New points from the data stream are always inserted into bucket B_0. If bucket B_0 is full, then the points are moved from B_0 to B_1. If B_1 is empty, then the data stream continues to flow into bucket B_0. In the case where B_1 is full, then a new coreset is constructed from the union of B_0 and B_1. The computed coreset is stored in bucket B_2 as it represents $2m$ points. If bucket B_2 is also full, then the previous procedure of the coreset construction is repeated until an empty bucket of the appropriate size is found.

In order to extract the final weighted sample of the data stream, we can obtain a final coreset from the union of all buckets $B_0, B_1, \dots, B_{L-1}$. The unified set will contain at most $m \cdot \lceil \log_2(n/m) + 2 \rceil$ weighted points. The technique of merging and reducing coreset structures is based on the following observations:

1. If S_1 and S_2 are (k, ε)-coresets for disjoint sets P_1 and P_2, respectively, then $S_1 \cup S_2$ is a (k, ε)-coreset for $P_1 \cup P_2$.
2. If S_1 is a (k, ε)-coreset for S_2 and S_2 is a (k, ε')-coreset for S_3, then S_1 is a $(k, (1+\varepsilon)(1+\varepsilon') - 1)$-coreset for S_3.

The StreamKM++ Algorithm

The algorithm performs two steps: In the first step, it applies the algorithm 2.5 to every input point of the data stream. For the coreset construction, the coreset tree structure is used (algorithm 2.4). In the second step, it computes a final coreset from the union of all non-empty buckets created by the merge-and-reduce technique. Then, it applies five times independently the k-means++ algorithm on the final coreset, and obtains the best clustering as the final result. Some important clarifications must be noted:

1. The overall computational complexity of the algorithm is $O(dnm)$ and the maximum number of memory units is $\Theta(dm\log(n/m))$. The choice for the appropriate size of the coreset implies the trade-off between the time and the accuracy of the clustering. From the experimental evaluation of StreamKM++, a sufficient size of the coreset and the buckets is considered to be $m = 200k$, where k is the desired number of clusters.
2. The algorithm is not only designed for clustering data streams of specific size. The size parameter $|P| = n$, is used for the exact estimation of the total number of buckets. Even if we don't know a priori the size of the stream, the growth in the numbers of buckets is logarithmically bounded (i.e. $L = \lceil \log_2(n/m) + 2 \rceil$).
3. The authors do not prove that the coreset tree structure is indeed a (k, ε)-coreset. However, they are optimistic that a mathematical proof can be found, as the creation methodology is based on the k-means++ seeding procedure.

ALGORITHM 2.5: InsertPoint(p)

```
1   put p into $B_0$
2   **if** $B_0$ is full
3       create an empty bucket S
4       move points from $B_0$ to S
5       empty $B_0$
6       $i \leftarrow 1$
7       **while** $B_i$ is not empty
8           create coreset from the union of $B_i$ and S
9           store coreset in S
10          empty $B_i$
11          $i \leftarrow i + 1$
12      move points from S to $B_i$
```

Chapter 3

Apache Flink

Apache Flink is an open source framework and distributed processing engine for large scale computations over unbounded and bounded data streams. Flink was formerly known as “Stratosphere”, a research project conducted by three universities in Berlin. In December 2014, it was accepted as an Apache top-level project. The core of Apache Flink is a distributed streaming dataflow engine written in Java and scala. Batch processing is built on top of the streaming engine as a special case of stream processing. In Flink, programs can be written in Java, Scala, Python and SQL, and can be deployed in local, cluster or cloud mode. Flink also provides connectors to thirdly-party systems for data sources and sinks, such as Apache Kafka, Apache Cassandra, Apache Kinesis, HDFS, and Elasticsearch.

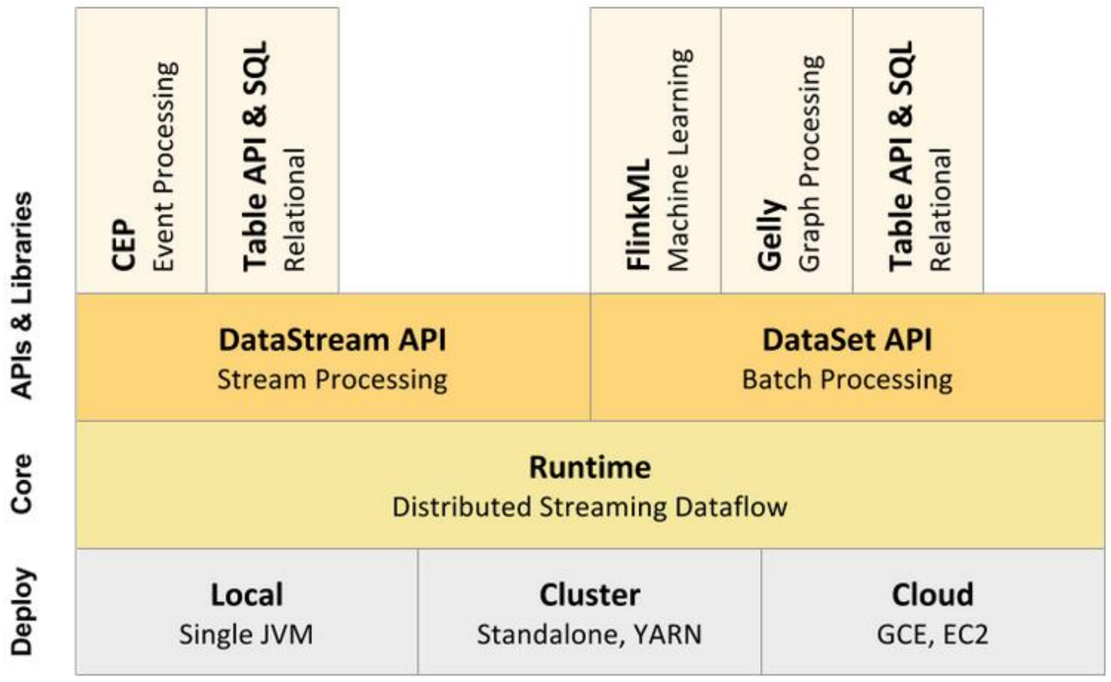


Figure 3.1: Component Stack of Apache Flink

We briefly describe some of the features that let Flink to have a wide acceptance in real-time analytics and applications:

- **Continuous Streaming Processing:** In many real-time use case scenarios, data is generated in the form of streams. Flink’s engine, process data streams as true streams because each record is processed immediately and independently as soon as it arrives. Furthermore, Flink’s expressive APIs and specific performance guarantees allow applications to run 24/7.

- **Low Latency & High Troughput:** Benchmarks have proven that Flink can compete with other well-known distributed Big Data platforms and that it can process millions of records per second [19]. Users of Flink have reported impressive performance numbers, such as applications running on thousands of nodes that process multiple trillions of events per day.
- **Fault Tolerance:** Streaming applications often require some custom state to maintain intermediate results of their computations. Flink uses an asynchronous lightweight incremental checkpoint mechanism [20] that guaranties extacly-once state consistency in case of a failure.
- **Event Time Handling:** Apache Flink embraces the notion of event time in stream processing, guaranteeing that out of order events are handled correctly and that results are accurate.

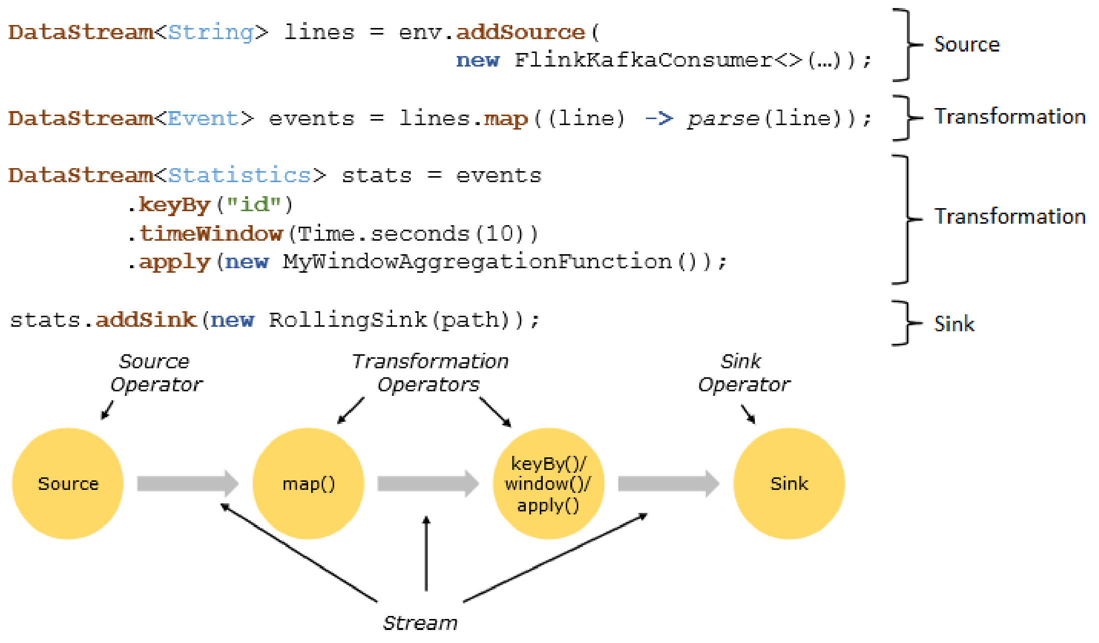
In the following sectors, we describe briefly the core of Apache Flink, the distributed runtime environment and the different levels of abstraction of the programming API. In this analysis we are more concerned about handling streaming data.

3.1 Dataflow Programming Model

The core of Apache Flink is consisted of two building blocks, streams and transformations. A stream is a (potentially never-ending) flow of data records and a transformation is an operation that takes one or more streams as input and produces one or more output streams as a result. Flink programs are mapped to streaming dataflows and each one of them is constituted by the following four features:

1. One or more data sources: The source defines where the input data comes from (e.g Apache Kafka or HDFS)
2. One or more data sinks: The sink defines where the output result is stored (e.g Apache Cassandra or Elasticsearch)
3. One or more operators: An operator applies transformations into streams (e.g Map, Filter, KeyBy, Aggregations, Join)
4. Intermediate streams: The resulted streams produced by data sources, sinks or operators.

Flink represents the streaming dataflows as directed acyclic graphs (DAGs). However, special forms of cycles are permitted through iteration operators. Figure 3.2 shows an example of a Flink program written in the DataStream API, along with the DAG of the streaming dataflow. In the beginning, the program uses a data source connector to consume data from Apache Kafka in the form of a stream of string records. Then, a Map operator transforms the initial data stream of strings to events, with a parse function that process each string record individually. The next operator groups the data stream of events according to some key “id,” and then applies every 10 seconds an aggregation function to the events with the same key. Finally, a data sink is used to store the results of the aggregation function to rolling files in the system.



```
DataStream<String> lines = env.addSource(
                                   new FlinkKafkaConsumer<>(…));

DataStream<Event> events = lines.map((line) -> parse(line));

DataStream<Statistics> stats = events
        .keyBy("id")
        .timeWindow(Time.seconds(10))
        .apply(new MyWindowAggregationFunction());

stats.addSink(new RollingSink(path));
```

Figure 3.2: Streaming Dataflow

When programs are executed in parallel, each stream has one or more stream partitions and each operator has one or more operator subtasks. The operator subtasks are independent from each other and execute in different threads of machines. Furthermore, different operators in the same streaming dataflow may have different levels of parallelism, but the parallelism of stream partitions is always the same of its producing operator.

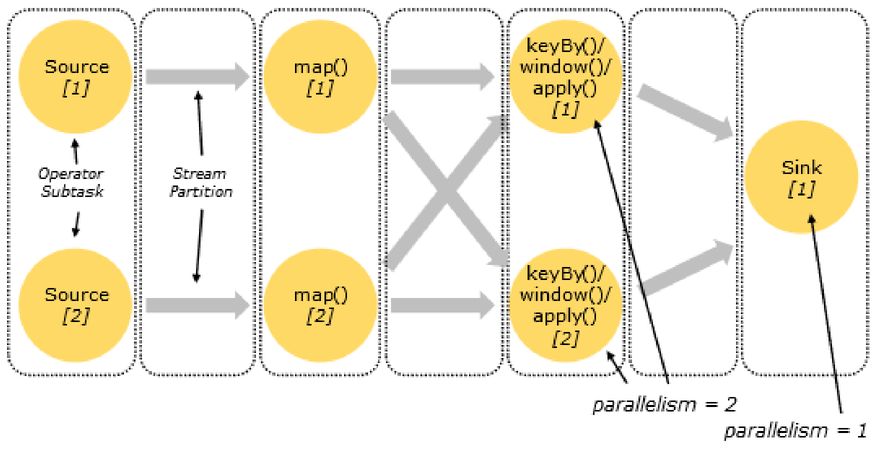


Figure 3.3: Parallel Dataflow

Figure 3.3 shows an arbitrary parallel dataflow of the previous example. In this figure we observe that there are two different types of streams, one-to-one and redistributing streams. The first type of stream, preserves the partitioning and the ordering of the records (e.g stream between Source[1] and Map[1]). In the second type, the ordering of records is only preserved within each pair of sending and receiving subtasks (e.g stream between Map[1] and KeyBy/window[2]).

Stateful Operators & Fault Tolerance

Streaming applications often require data structures to store intermediate results of their computations (e.g current version of a machine learning model). Operators that remember information across the processing of individual data records are called stateful and the information that each operator remembers is called state. Flink provides in-core data structures for stateful operations, that are scoped per parallel subtask (e.g figure 3.3 Map[1]) or per key attributes from the data records (e.g figure 3.2 keyBy("id")).

Flink uses a checkpoint mechanism to ensure that in the presence of failures the set of states will reflect every record from the input data stream exactly once. A checkpoint is a global asynchronous snapshot of the set of states, taken in a regular basis. In case of a failure, Flink restarts the application using the most recently completed checkpoint.

Notion of Time

An important aspect of streaming applications is the measurement of time. Flink supports three different notions of time:

1. Processing time refers to the system time of the machine that is executing the respective time-based operation.
2. Ingestion time is the time when an event enters the Flink streaming dataflow at the source operator.
3. Event time is the time when an event was created and it is usually described by a timestamp in the events.

The mechanism of Flink to measure progress in time-based operations is called watermarks. Watermarks carry a timestamp and flow as part of the parallel streaming dataflow along with stream events. Events that follow a watermark's timestamp should have a timestamp with greater value. When a subtask of an operator receives a watermark, it advances its internal clock according to the watermark's timestamp. This mechanism is crucial for streaming operators that handle out-of-order events.

3.2 Distributed Runtime Environment

In the distributed execution, Flink chains different operators into tasks. For example, the DAG presented by figure 3.2 is executed with three tasks. The first task is composed of the source and map operators, the second and third task contains the KeyBy/window and sink operators respectively. The same idea applies when we execute DAGs with parallelism higher than one (e.g figure 3.3 has five subtasks). The procedure of chaining operators increases the overall throughput of the program.

Figure 3.4 shows the various processes that take part in the distributed execution of a Flink program:

- The **Job Client** is the starting point of the program execution and it is not a part of the runtime. The job client creates and sends the streaming dataflow to the Job Manager for the execution in the distributed environment. During the execution, the client receives statistics and results from the Job Manager.
- The **Job Managers**, also known as master nodes, coordinate and manage the distributed execution of the program. They assign the parallel subtasks of the dataflow to the available Task Managers, they coordinate the checkpoint mechanism and the recovery upon failures, and more.
- The **Task Managers**, also known as worker nodes, execute the parallel subtasks of the dataflow and they exchange data streams with other Task Managers. Each Task Manager is a separate JVM process and it is composed by a number of task slots.

Each task slot represents a fixed subset of resources of the Task Manager. A Task Manager with three slots will execute three parallel subtasks of a task, with 1/3 of the managed memory dedicated to each slot. By default, slots are allowed to contain at the same time subtasks of different tasks, as long as they are from the same job. In this way, a task slot may hold the entire parallel pipeline of the streaming dataflow. A general practice is to set the total number of slots as the total number of CPU cores in each Task Manager. If the CPU supports hyper-threading then each task slot will use two virtual threads. The maximum number of task slots in the distributed environment denotes the maximum parallelism of the streaming job. Figure 3.5 shows a Task Manager with three slots, where each slot contains a parallel pipeline of the streaming dataflow presented by figure 3.2.

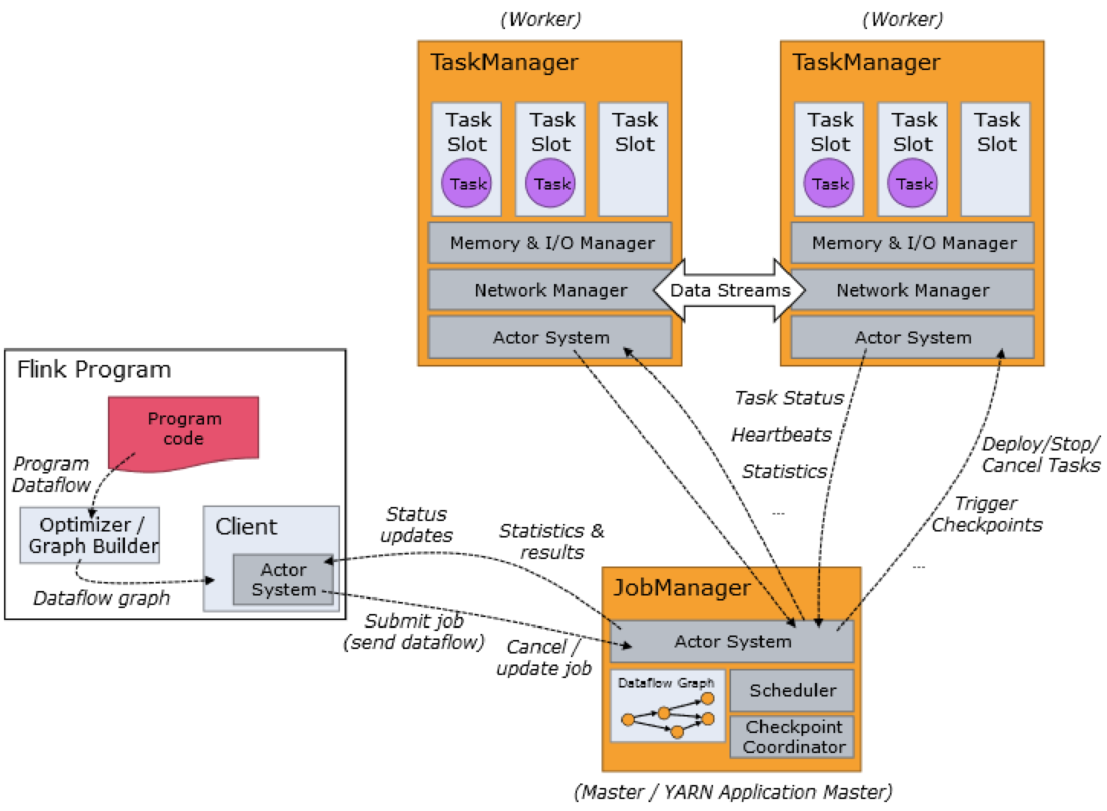


Figure 3.4: Distributed Architecture

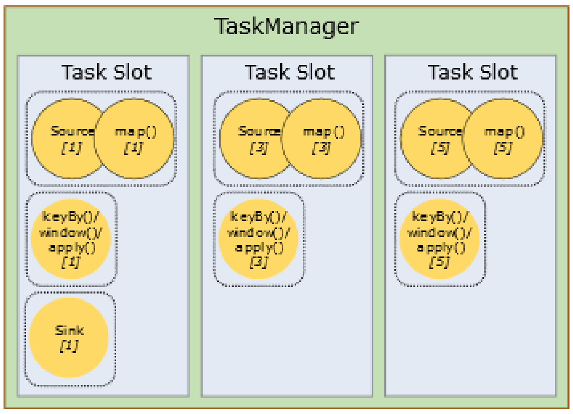


Figure 3.5: Task Slots of the Task Manager

3.3 Programming APIs & Libraries

- **DataStream** is the core API for handling unbounded and bounded streams. This API provides many common stream processing operators, such as map, filter, keyBy, reduce, aggregations, time & count windows, iterations, window-based join transformations and more. The window mechanism of Flink, offers various types of windows (e.g global, sliding, session, tumbling) and the ability to handle out of order elements. DataStream API supports two types of broadcast streams, streams that are broadcasted to the downstream parallel subtasks of an operator and streams that are available among the parallel subtasks. All of the above operators can be stateful and fault tolerant with the appropriate use of state data structures. Flink has a feature called Queryable State that allows the user to query the state from outside of the distributed environment. Furthermore, DataStream API is compatible with Apache Storm and therefore allows the reuse of Storm code (e.g Storm topologies, Spouts & Bolts).

- **DataSet** API enables transformations on bounded datasets (e.g., filtering, mapping, joining, grouping) with batch processing. Flink offers broadcast variables and distributed cache to make datasets and local files available to all parallel instances of an operation. DataSet API supports a type of iterate operator called Delta, which allows partial updates on the solution set of every iteration. Flink is compatible with Apache Hadoop and therefore allows implemented Hadoop code to be reused. Batch processing is treated as special case of stream processing, however there are a number of differences: DataSet programs use a query optimizer to generate the optimal execution plan, DataSet operators are blocking and in the event of failure recovery happens by replaying failed partitions.
- **Table API & SQL** are integrated in a joint relational API. Table API is a declarative domain specific language that allows the composition of queries from relational operators such as selection, projection and joins on tables. Tables have a schema attached, similar to other relational databases. SQL API has the same semantics and expressiveness with the Table API, but it executes queries with SQL syntax. Apache Calcite is used for parsing, validation, and query optimization. Tables can be generated from and converted to DataSream/DataSet and vice versa, allowing the user to run relational queries to unbounded and bounded datasets.
- **CEP** is the complex event processing library of Flink, which allows the user to detect event patterns of the input data. The CEP library is available through the DataStream API, and therefore patterns are evaluated on data streams.
- **GELLY** is a library for scalable graph processing and analysis, which is available through the DataSet API. It provides data structures to store and represent graph data and it supports methods to create, transform and modify the graphs. Moreover, Gelly contains a library of graph algorithms, such as label propagation, triangle enumeration, and page rank.
- **FlinkML** is the machine learning library developed by Flink, which is built upon the DataSet API. It supports supervised learning (e.g Support Vector Machines, Multiple linear regression) and unsupervised learning algorithms (e.g k-Nearest neighbors join, Principal Components Analysis), recommendation algorithms (e.g Alternating Least Squares), data preprocessing techniques (e.g Polynomial Features, Standard/MinMax Scaler) and more. FlinkML has a feature called ML-pipelines, which provides the ability to chain different transformers and predictors in a type-safe manner.

3.3.1 DataStream API

In this section we describe the anatomy of DataStream API programs along with specific operators that will help us understand the distributed implementation of StreamKM++. DataStream programs implement transformations on data streams. Data streams are represented by special classes (e.g DataStream<T>, KeyedStream<T, KEY>), which are immutable collections of data. Each DataStream program consists of the following five stages:

1. Obtain a streaming execution environment: The execution environment defines the context in which a program is executed (e.g local or remote environment). We can automatically obtain the execution environment from the getExecutionEnvironment() function.

```
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
```

2. Load or create input data from data sources: Data sources are attached to the execution environment. We can implement a custom source function or use one of the pre-implemented. To read a text file as a sequence of lines, we can use readTextFile("path") function. This function is treated by Flink's distributed runtime as two subtasks, called directory monitoring and data reading. The role of the monitoring subtask is to split the files of the directory into splits and then assign these splits to the downstream readers (this is a not-parallel subtask). The second subtask is performed in parallel by the readers who read the actual data into multiple splits one-by-one.

```
DataStream<String> text = env.readTextFile("file:///path/to/file");
```

3. Perform transformations on this data: DataStream API provides a variety of stream operators. Here, we describe a few of them.

- FlatMap: Receives one element from the input DataStream and produces zero, one, or more elements into a new DataStream.

```
DataStream<Integer>  intNumbers = text.flatMap( new FlatMapFunction<String, Integer>() {
        @Override
        public void flatMap(String value, Collector<Integer> out) throws Exception {
                for (String str: value.split(" ")){
                        out.collect(Integer.parseInt(str));
                }
        }
});
```

- Connect: Merges two input DataStreams and retains their types. The new ConnectedStream allows the process of the connected streams from a shared context.

```
DataStream<Double> doubleNumbers = [...]
ConnectedStreams<Integer, Double> connectedStreams = intNumbers.connect(doubleNumbers);
```

- CoFlatMap: Applies a flatMap function individually to the elements of the connected stream, and produces a new DataStream.

```
DataStream<String> stringStream =
connectedStreams.flatMap( new CoFlatMapFunction<Integer, Double, String>() {
        @Override
        public void flatMap1(Integer value, Collector<String> out) {
                out.collect("Integer");
         }
        @Override
        public void flatMap2(Double value, Collector<String> out) {
                out.collect("Double");
         }
});
```

- Transform: Is a method for transforming a DataStream with a user defined operator. This function gives us the ability to define low level operators that can handle basic mechanisms of the DataStream API, such as the Watermark mechanism.

- KeyBy: Partitions a DataStream into disjoint partitions according to some key. All elements with the same key are assigned to the same partition, so that the number of partitions is equal to the number of distinct keys. Internally, KeyBy is implemented with hash partitioning. In the distributed runtime each task slot will preserve zero or more entire partitions.

```
DataStream<Tuple2<Integer,String>> inputStream = [...]
KeyedStream<Tuple2<Integer,String>,Integer> keyedStream = inputStream.keyBy(0);
```

- Process: Transforms a DataStream or a KeyedStream given a ProcccessFunction. The ProcessFunction can be thought of as a FlatMapFunction with access to keyed state and timers.

All of the above operators can be stateful with the use of Flink's state data structures. There are two basic kinds of states, Keyed State and Operator State.

- Keyed State is used only on a KeyedStream and it denotes the state of each partition (or distinct key). We can think of KeyedState as the state for each combination of <parallel operator instance, key>. Flink supports different types of data structures that are scoped per key, such as ValueState<T>. This state keeps a value which can be updated or retrieved for each different key.

- Operator State, is the state of each parallel operator instance. Flink provides only one data structure for this kind of state, called ListState<T>. This state keeps a list of elements. We can append elements or retrieve an Iterable over all currently stored elements.

4. Store output results: DataStream API has a variety of data sink functions. For example, the following function writes the elements of a string stream to the output file, line by line.

```
stringStream.writeAsText("file:///path/to/outputFile");
```

5. Trigger the program execution: Flink programs are executed lazily, and therefore we have to trigger the execution of the transformations by calling the execute() method from the execution environment.

```
env.execute(“Job name”);
```

Chapter 4

Algorithm Implementation

In this chapter we describe the distributed implementation of StreamKM++ in Apache Flink. For the development phase, we use the DataStream API in Java to apply transformations on bounded data streams. For data sources and sinks, we use Apache Hadoop Distributed File System (HDFS) [22]. During our implementation, we took into consideration the original source code of StreamKM++, which was developed by the authors in C [23].

In the first section, we propose a parallel implementation which computes the set of cluster centers after the consumption of the entire input dataset. In the second section, we describe the evaluation methodology that we use to rank the quality of the produced clustering, and then we propose a parallel implementation to compute the quality. We claim that with minor modifications, we could use the evaluation methodology to predict the cluster of new incoming data points. In the third section, we propose an alternative parallel implementation of StreamKM++ which produces periodically requests for the re-evaluation of the cluster centers. Additionally, we modify the stage that computes the clustering, to produce simultaneously clusterings with different number of centers. In the last section, we develop a program that allows the user to query in real-time the cluster centers produced by StreamKM++.

4.1 Distributed StreamKM++

In this section, we describe our distributed implementation of StreamKM++ for clustering bounded data streams (for unbounded data streams see section 4.3). First, we read line by line in parallel the input file from HDFS which contains the data points. Then, we transform each line to a d-dimensional point in Euclidean Space. Afterwards, we consume each data point according to algorithm 2.5 and therefore we maintain a list of buckets for every level of the parallelism. Once the input file is consumed, we take a union of the buckets from each one of the lists and then we extract a coreset using the coreset tree structure. We name these parallel coresets as “partial coresets”. We forward the partial coresets to a non-parallel task that computes incrementally a new coreset for each incoming partial coreset, until the total number of partial coresets is consumed. The result of the above procedure is the construction of a final coreset that represents the entire input. Finally, we apply five times the k-means++ algorithm on the final coreset and we extract the best clustering (StreamKM++ performs the same technique on the final coreset). At this point we describe the implementation details of each operator in the streaming dataflow (figures 4.1 & 4.2).

- **HdfsSource**: The input file that contains the data points is stored in HDFS in order to be accessible from the TaskManagers of the Job. Each line of the file represents a single point in the d-dimensional space. To access the contents of the file line by line, we use the pre-implemented data source function readTextFile("path") to generate a DataStream<String>. As mentioned previously, this function is composed of two subtasks, directory monitoring (figure 4.2 Custom File Source) and data reading (figure 4.2 HdfsSource). HdfsSource is executed in parallel and forwards the lines of the file to the downstream operator.

- **HandleWatermark**: The purpose of this operator is to detect the end of file. Generally, when a source terminates its execution, the corresponding source function emits a watermark to the parallel streaming dataflow with value "Long.MAX_VALUE". The HandleWatermark operator extends the base class of all stream operators and thus can access the watermark mechanism of Flink. Internally, this operator sends continuously the input lines to the downstream operator, and when it detects the final watermark it produces a new line with value "EOF". This technique guarantees that the downstrem operator will receive the "EOF" line after the last data point. The result of this operation is the generation of a new DataStream<String> that contains the set of initial points and the string "EOF".

- **FlatMapToPoint**: This operator transforms the points from the DataStream<String> to the DataStream<Point>. A Point is a class that holds the attributes of each data point (e.g vector with coordinates, weight & id). We choose to implement this operation with a flatMap function so that we can handle strings that represent incorrect points. The flatMap function parses the input string to extract the coordinates of the point. If the number of coordinates is equal to the number of dimensionality then a new Point is formed with the corresponding attributes. If the input string contains the "EOF" word, we generate a new marked Point. In any other case, the input string is aborted.

- **FlatMapToPartialCoresets**: Each one of the parallel subtasks of this operator consumes the input points according to algorithm 2.5 (merge-and-reduce technique). At any point of time, these subtasks hold in memory the list of buckets that represent the current number of consumed points. The size of buckets is predefined at the start of the program execution. In our implementation we do not need the total number of input points, because we maintain only the necessary buckets (e.g B_0 and any number of full buckets $B_{i>0}$). The execution of the algorithm 2.5 continues until the parallel subtasks receive the marked point that denotes the end of file. In that case, we rely on the first observation of the merge-and-reduce technique, and we take the union of the buckets for each parallel list. Then, we extract a new coreset for each unified set by using the coreset tree. We name the resulting coresets as "partial coresets". This operator produces a DataStream<Point[]> that contains the partial coresets of its parallel subtasks.

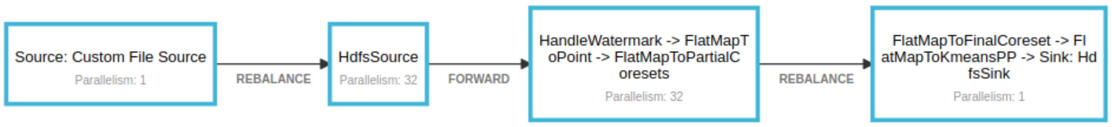
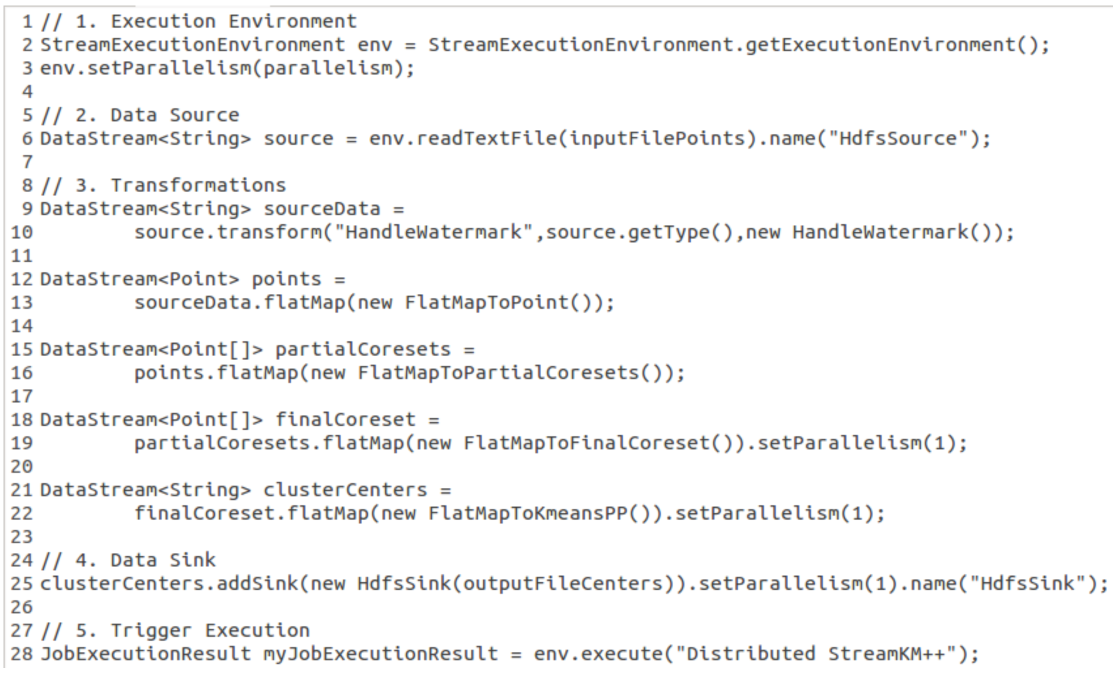

Source: Custom File Source
Parallelism: 1
REBALANCE
HdfsSource
Parallelism: 32
FORWARD
HandleWatermark -> FlatMapToPoint -> FlatMapToPartialCoresets
Parallelism: 32
REBALANCE
FlatMapToFinalCoreset -> FlatMapToKmeansPP -> Sink: HdfsSink
Parallelism: 1

Figure 4.1: Streaming dataflow of Distributed StreamKM++



```
// 1. Execution Environment
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setParallelism(parallelism);

// 2. Data Source
DataStream<String> source = env.readTextFile(inputFilePoints).name("HdfsSource");

// 3. Transformations
DataStream<String> sourceData =
        source.transform("HandleWatermark",source.getType(),new HandleWatermark());

DataStream<Point> points =
        sourceData.flatMap(new FlatMapToPoint());

DataStream<Point[]> partialCoresets =
        points.flatMap(new FlatMapToPartialCoresets());

DataStream<Point[]> finalCoreset =
        partialCoresets.flatMap(new FlatMapToFinalCoreset()).setParallelism(1);

DataStream<String> clusterCenters =
        finalCoreset.flatMap(new FlatMapToKmeansPP()).setParallelism(1);

// 4. Data Sink
clusterCenters.addSink(new HdfsSink(outputFileCenters)).setParallelism(1).name("HdfsSink");

// 5. Trigger Execution
JobExecutionResult myJobExecutionResult = env.execute("Distributed StreamKM++");
```

Figure 4.2: Source code of Distributed StreamKM++

- **FlatMapToFinalCoreset**: This is a non-parallel operator, which receives the partial coresets one by one. Initially, this operator saves the first partial coreset as the current solution. Then, for each one of the following, it calculates a new coreset from the union of the current partial coreset and the current solution, and saves the new coreset as the current solution. The above procedure continues until the last partial coreset is consumed. The final solution constitutes the final coreset, which represents the entire data set. The operator produces a DataStream<Point[]> that contains only the final coreset.

- **FlatMapToKmeansPP**: At this stage, the parallel consumption of the input data stream is completed. We can perform any k-means clustering algorithm of our choice on the final coreset. We choose to follow the same technique that was proposed by the authors of StreamKM++, and therefore we apply five times the k-means++ algorithm on the final coreset with a convergence criterion. We obtain the clustering with the minimum cost. This operator produces a DataStream<String> that contains the cluster centers in a string format.

- **HdfsSink**: This is a custom implementation of a sink function that stores the text file with the cluster centers inside the HDFS, with a specific format.

Flink provides a Web Dashboard to monitor the current state of running Jobs (e.g memory usage, back pressure, current state of checkpoints and more). Figure 4.1 was taken from the Web Dashboard and represents the streaming dataflow of the Distributed StreamKM++. We observe the chaining mechanism of Flink's distributed runtime that groups different operators into tasks with the same parallelism. Furthermore, we notice the different types of streams among the tasks, one-to-one (FORWARD) and redistributing streams (REBALANCE).

4.2 Evaluation Methodology

In this section we describe the methodology that we used to evaluate the quality of the clustering produced by the distributed StreamKM++. We rank these clusterings according to their $\mathrm{cost}(\mathrm{P}, \mathrm{C}) = \sum_{\mathrm{x} \epsilon \mathrm{P}} \mathrm{D}^2(\mathrm{x}, \mathrm{C})$, where P is the set of data points from the input file and C is the set of cluster centers produced by StreamKM++. In order to compute $\mathrm{cost}(\mathrm{P}, \mathrm{C})$, we have to read the file from the start and then for each data point we have to find the cluster center that has the minimum squared distance to the corresponding point. This is a trivial procedure which can be implemented in any programming language. However, we decided to develop a Flink streaming program to process the input file in parallel. The first reason to develop this program is that the computation of the clustering cost may be time-consuming, based on the size of the input file. The second reason is that with minor modifications the proposed streaming dataflow could be used to predict the cluster of new incoming data points.

At this point we give a description of the program. We consume data from two different sources. In the first source, we read line by line in parallel the file with the data points from HDFS. Then, we transform each line to a d-dimensional point and we forward these points to the downstream operator. In the second source, we read line by line the file with the cluster centers from HDFS, which was produced by the distributed StreamKM++. We transform each one of these centers to a d-dimensional point and we assign to that point the weight of the corresponding center. Then, we broadcast the cluster centers to the downstream operator, so that each one of the parallel subtasks receives the entire set of centers. In the following operator, we receive input from both cluster centers an ordinary data points. We wait until the entire set of centers is collected, and then we sum in parallel the cost of the incoming data points (the cost of each point is equal to the minimum squared distance to its nearest center). We name these parallel sums as “partial costs”. Finally, we calculate the total cost of the clustering from the sum of partial costs.

Figures 4.3 & 4.4 represent the different operators of the streaming dataflow. We described some of them in the previous section and therefore we will refer only to specific operators.

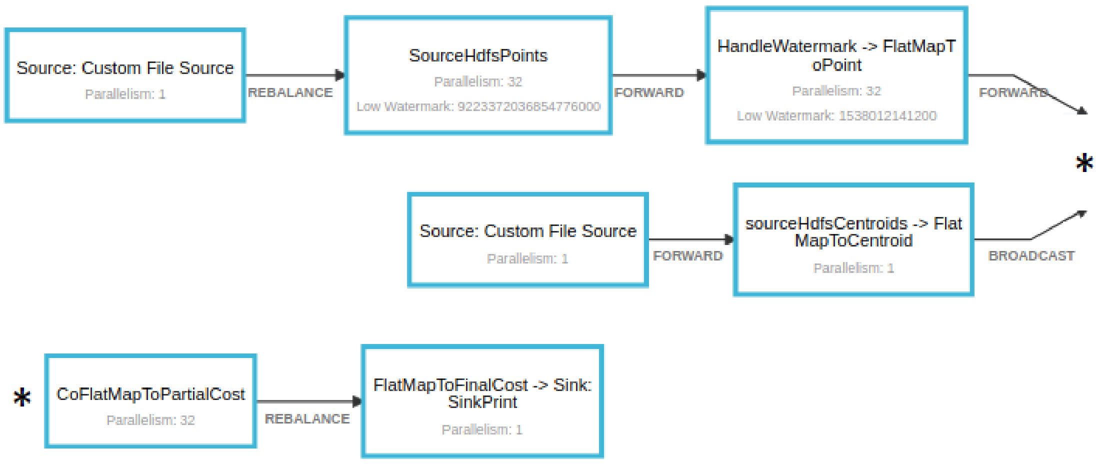
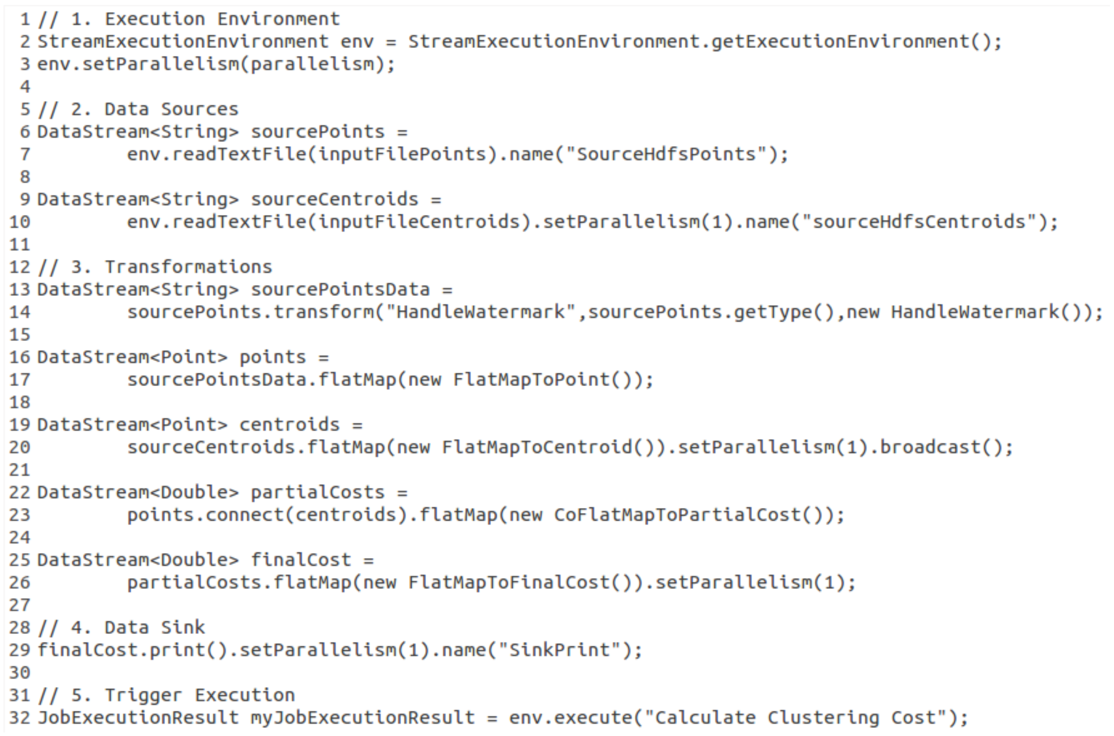

Source: Custom File Source
Parallelism: 1
REBALANCE
SourceHdfsPoints
Parallelism: 32
Low Watermark: 9223372036854776000
FORWARD
HandleWatermark -> FlatMapToPoint
Parallelism: 32
Low Watermark: 1538012141200
FORWARD
*
Source: Custom File Source
Parallelism: 1
FORWARD
sourceHdfsCentroids -> FlatMapToCentroid
Parallelism: 1
BROADCAST
*
CoFlatMapToPartialCost
Parallelism: 32
REBALANCE
FlatMapToFinalCost -> Sink: SinkPrint
Parallelism: 1

Figure 4.3: Streaming dataflow of the Evaluation Methodology



```
// 1. Execution Environment
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setParallelism(parallelism);

// 2. Data Sources
DataStream<String> sourcePoints =
        env.readTextFile(inputFilePoints).name("SourceHdfsPoints");

DataStream<String> sourceCentroids =
        env.readTextFile(inputFileCentroids).setParallelism(1).name("sourceHdfsCentroids");

// 3. Transformations
DataStream<String> sourcePointsData =
        sourcePoints.transform("HandleWatermark",sourcePoints.getType(),new HandleWatermark());

DataStream<Point> points =
        sourcePointsData.flatMap(new FlatMapToPoint());

DataStream<Point> centroids =
        sourceCentroids.flatMap(new FlatMapToCentroid()).setParallelism(1).broadcast();

DataStream<Double> partialCosts =
        points.connect(centroids).flatMap(new CoFlatMapToPartialCost());

DataStream<Double> finalCost =
        partialCosts.flatMap(new FlatMapToFinalCost()).setParallelism(1);

// 4. Data Sink
finalCost.print().setParallelism(1).name("SinkPrint");

// 5. Trigger Execution
JobExecutionResult myJobExecutionResult = env.execute("Calculate Clustering Cost");
```

Figure 4.4: Source code of the Evaluation Methodology

- **FlatMapToCentroid**: We use this operator to parse the lines of the source function in order to extract the coordinates and the weight of each cluster center. We execute this operator with parallelism-1 because the file that contains the centers has a small size. Finally, we broadcast each one of the centers to the subtasks of the downstream operator.

- **CoFlatMapToPartialCost**: The purpose of this operator is to compute in parallel the cost of each incoming data point. In order to achieve that, each parallel subtask of this operator must preserve the entire set of centers. Therefore, we create a connected stream which is composed from the union of the broadcasted cluster centers and the incoming data points. The coFlatMap function provides us the ability to process individually in the same operator, each one of the elements of the connected streams. Internally, coFlatMap contains two flatMap functions. We use the second flatMap function to save the cluster centers into a vector. The first function, initially buffers the incoming data points until the vector with the cluster centers is filled. Then, it computes the cost for each one of the buffer points. When the buffer is emptied, the function computes the cost of the new incoming data points until the input file ends. During the execution of this operation, we maintain the current sum of costs. We name that sum as "partialCost". When the file ends, each parallel subtask emits its partialCost to the downstream operator.

- **CoFlatMapToFinalCost**: We use this operator to calculate the total cost of the clustering from the sum of the partial costs.

In many real applications, it is often desirable to compute the initial values of the cluster centers from historical data, and then use these centers to predict the cluster in which the new data falls into. It is possible to accomplish this scenario by connecting the streaming dataflow of the figure 4.1 with the one from figure 4.3. More specifically, we could replace the operator "sourceHdfsCentroids" of the figure 4.3 with "FlatMapToKmeansPP" of the figure 4.1. In that way, we could provide the initial cluster centers from historical data. Moreover, we could replace the operator "sourceHdfsPoints" of figure 4.3 with Flink's Apache Kafka connector to consume real-time data. Finally, we could modify the output of "CoFlatMapToPartialCost" and "CoFlatMapToFinalCost" operators to collect the specific cluster center that the incoming points fall into.

4.3 Distributed Solution with Requests

The Distributed SreamKM++ of section 4.1 produces the final coreset, and therefore the cluster centers, after the consumption of the entire input dataset. However, when the input data is generated from an unbounded stream, we have to compute the cluster centers periodically in order to reflect the evolution of the input stream. In this section, we propose an alternative parallel dataflow based on the Distributed StreamKM++, that produces periodically requests for the re-evaluation of the cluster centers. Additionally, we modify the stage that computes the clustering, to produce simultaneously clusterings with different number of centers. In section 4.3.1 we develop a program to access these clusterings from outside the Flink's runtime enviroment. We note that the alternative dataflow still consumes data from HDFS but we create the conditions for this implementation to be generalized for unbounded input streams.

The program receives data from two sources. In the first source, we read the input file from HDFS and then we transform the input lines to data points. In the second source, we generate periodically request elements and then we broadcast these elements to the downstream operator. The operator which receives the data points and the requests, produces the parallel partial coresets per request. Once we receive the partial coresets, we compute incrementally the final coreset that represents the current number of consumed data points. Afterwards, we create a number of copies of the final coreset that is equal to the number of different clusterings that we want to perform. Finally, we compute these clusterings and we store them to the state of that operator. We continue with the implementation details of specific operators (figures 4.5 & 4.6).

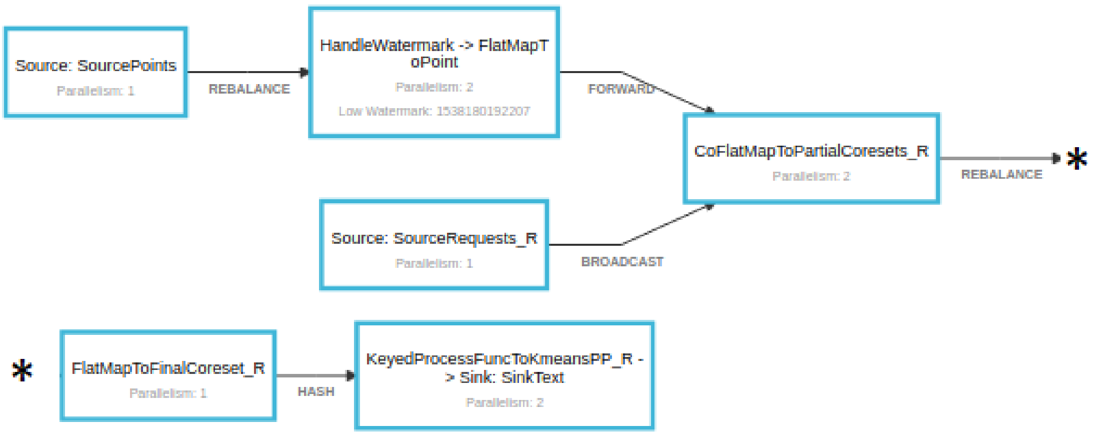
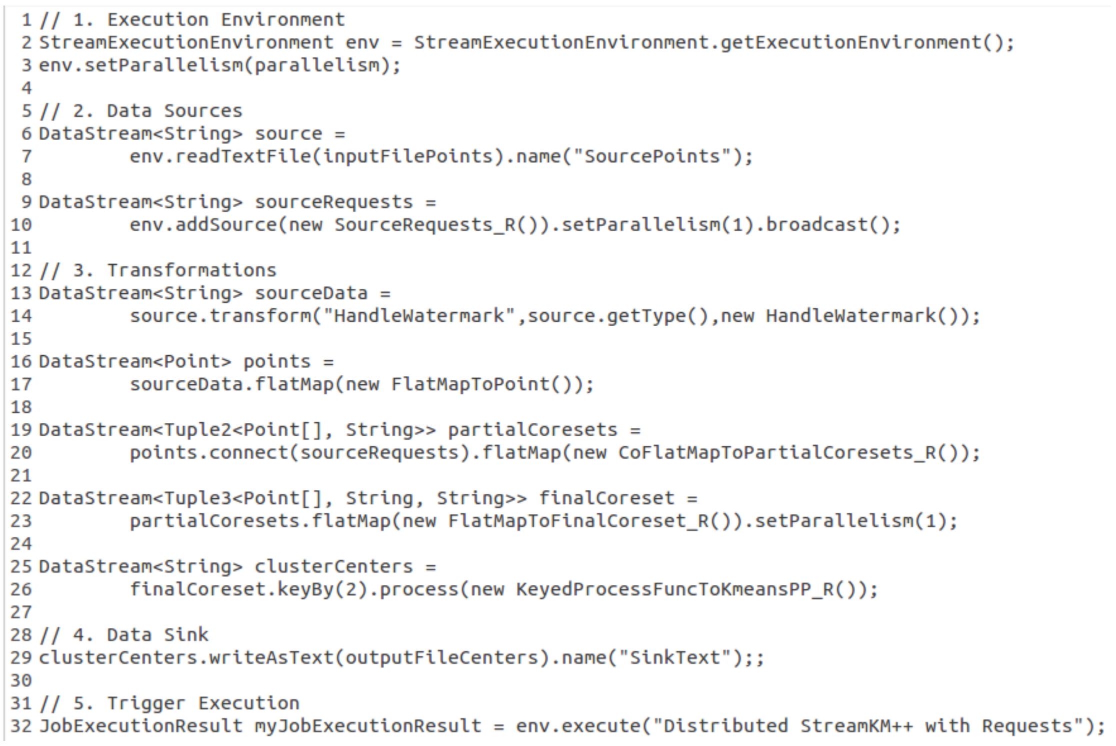


Figure 4.5: Streaming dataflow of the Distributed StreamKM++ with requests


```
// 1. Execution Environment
StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
env.setParallelism(parallelism);

// 2. Data Sources
DataStream<String> source =
        env.readTextFile(inputFilePoints).name("SourcePoints");

DataStream<String> sourceRequests =
        env.addSource(new SourceRequests_R()).setParallelism(1).broadcast();

// 3. Transformations
DataStream<String> sourceData =
        source.transform("HandleWatermark",source.getType(),new HandleWatermark());

DataStream<Point> points =
        sourceData.flatMap(new FlatMapToPoint());

DataStream<Tuple2<Point[], String>> partialCoresets =
        points.connect(sourceRequests).flatMap(new CoFlatMapToPartialCoresets_R());

DataStream<Tuple3<Point[], String, String>> finalCoreset =
        partialCoresets.flatMap(new FlatMapToFinalCoreset_R()).setParallelism(1);

DataStream<String> clusterCenters =
        finalCoreset.keyBy(2).process(new KeyedProcessFuncToKmeansPP_R());

// 4. Data Sink
clusterCenters.writeAsText(outputFileCenters).name("SinkText");;

// 5. Trigger Execution
JobExecutionResult myJobExecutionResult = env.execute("Distributed StreamKM++ with Requests");
```
Figure 4.6: Source code of the Distributed StreamKM++ with requests

- **SourceRequests_R**: In this operator we produce the periodic requests. A request is composed by a string that holds the request's id. We broadcast these requests so that each parallel instance of the following operator receives them.

- **CoFlatMapToPartialCoresets_R**: We use this operator to produce the partial coresets per request. In order to handle both the requests and the data points inside the same operator, we connect the streams that were produced by the two source functions. When this operator receives a data point, it executes the algorithm 2.5 to update the list of buckets. When it receives a request or the special data point that denotes the end of file, it takes the union of the buckets and extracts the partial coresets. The output of each parallel instance of this operator is composed by the corresponding partial coreset along with the request id.

- **FlatMapToFinalCoreset_R**: In this operator we compute the final coreset per request, from the partial coresets. We use the same incremental technique that we applied in the FlatMapToFinalCoreset operator of section 4.1. Once the final coreset is produced, we create a number of copies that is equal to the numbers of the different clusterings that we want to perform. The term “different clusterings” refers to clusterings with different number of cluster centers. For example, if we desire to compute clusterings with 10 and 20 centers on the final coreset, then this operator produces two tuples for each request, <final coreset, request id, 10> and <final coreset, request id, 20>. We note that the number of different clusterings is predefined at the start of the program.

- **KeyedProcessFuncToKmeansPP_R**: Initially, we use the KeyBy operator to group the output of FlatMapToFinalCoreset_R according to the number of the cluster centers. By using the KeyBy operator, we construct a new KeyedStream that each key represents the different number of cluster centers. For each one of the keys, we apply the KeyedProcessFuncToKmeansPP_R function. This function executes five times the k-means++ algorithm on the final coreset and then stores the best clustering to the Keyed State of Flink. We decided to use the ValueState to store the best clustering for each key. We update the clustering of the ValueState every time we receive a new request. In this way, we store to the Keyed State of Flink the most up-to-date clustering for each different number of cluster centers.

4.3.1 Real-Time Cluster Queries

In the previous section, we proposed an implementation that uses requests to keep up-to-date the final coreset of StreamKM++ algorithm. During that process, every time we updated the final coreset, we performed simultaneously multiple clusterings with different number of cluster centers. Finally, we stored these clusterings to Flink's Keyed State, with the key being the number of cluster centers. In this section, we exploit the Queryable State Feature of Flink to allow the user to query these keys from outside Flink's runtime environment. The Queryable State of Flink, exposes the Keyed State to the outside world and allows the user to query the available keys of a specific operator.

Architecture of Queryable State

The Queryable State is composed by three processes:

- The QueryableStateClient, which runs outside from Flink's distributed runtime environment. The user submits to the client queries for the values of the keys.

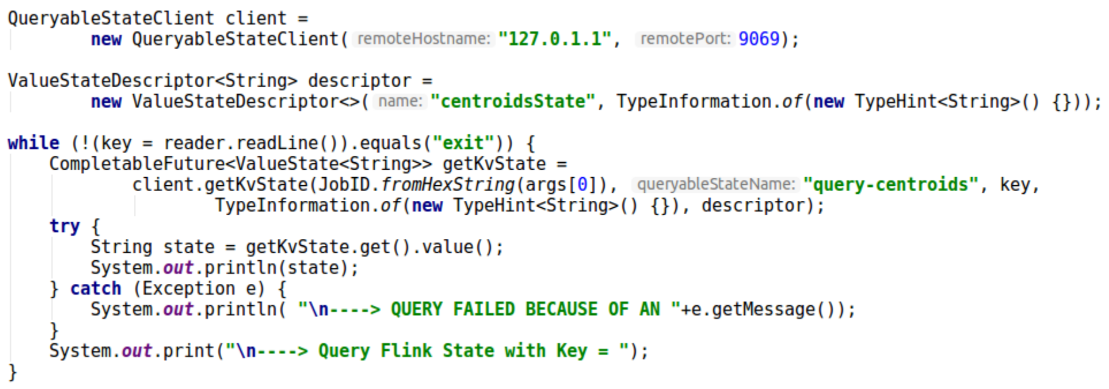
- The QueryableStateClientProxy, which runs on each TaskManager and is responsible to receive the client’s queries, to obtain the value of the requested key from the corresponding Task Manager, and to return the value to the client.

- The QueryableStateServer, which runs on each TaskManager and is responsible for serving the key values of its locally stored state.

Queryable State in Action

The Queryable State feature is capable of performing queries only in states maintained by KeyedStreams. In our use case, the keyed state is stored inside the “KeyedProcessFuncToKmeansPP_R” function, to the ValueState data structure. In order to query the keys of the ValueState, we have to specify the hostname and the port of the TaskManager. Then, we have to specify the name of the data structure that holds the state (e.g “centroidsState”). Afterwards, we submit the query to the Task Manager with a specific key. Internally, the QueryableStateClientProxy of the Task Manager receives the request, and then asks the Job Manager which one of Task Managers holds the value of the query-ed key. Based on that answer, the proxy will retrieve the value from the QueryableStateServer of the corresponding Task Manager. This value is then returned from the proxy to the client.

Figure 4.7 represents a sample of the code that we used to query the state of the “KeyedProcessFuncToKmeansPP_R” function. The value of the key corresponds to the number of cluster centers. In Figure 4.8, we show the results from queries with different keys. In figure 4.8 (a), we show the result from a query with key = 50. We observe that the cluster centers are computed from the first evaluation of the final coreset (request Id = 0). The field “subtask id = 1” denotes that the key belongs to the second parallel instance of that operator. In figure 4.8 (b), we show the result of the same query which was performed by (a), only in this case we observe that the cluster centers are computed upon the third evaluation of the final coreset (request Id = 2). Figures 4.8 (c-d) represent the results of queries with keys 60 and 70 respectively. In figure 4.8 (d), we observe that key = 70 is stored in the first parallel instance of that operator (subtask Id = 0).

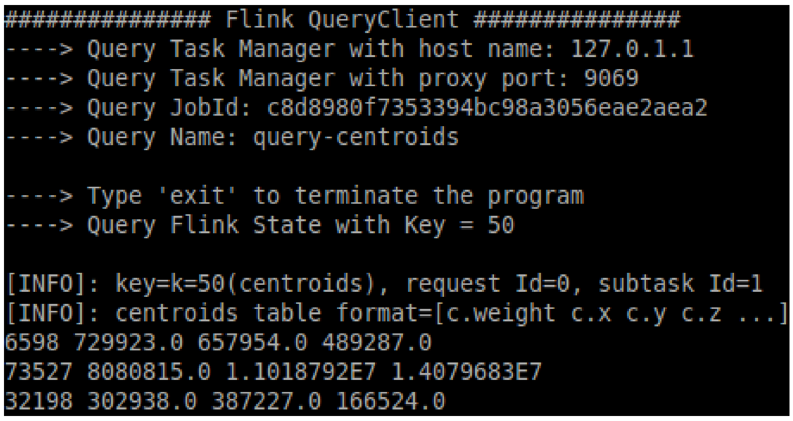


```
QueryableStateClient client =
        new QueryableStateClient( remoteHostname: "127.0.1.1",  remotePort: 9069);

ValueStateDescriptor<String> descriptor =
        new ValueStateDescriptor<>( name: "centroidsState", TypeInformation.of(new TypeHint<String>() {}));

while (!(key = reader.readLine()).equals("exit")) {
    CompletableFuture<ValueState<String>> getKvState =
            client.getKvState(JobID.fromHexString(args[0]),  queryableStateName: "query-centroids", key,
                    TypeInformation.of(new TypeHint<String>() {}), descriptor);
    try {
        String state = getKvState.get().value();
        System.out.println(state);
    } catch (Exception e) {
        System.out.println( "\n----> QUERY FAILED BECAUSE OF AN "+e.getMessage());
    }
    System.out.print("\n----> Query Flink State with Key = ");
}
```

Figure 4.7: Sample code to query the cluster centers

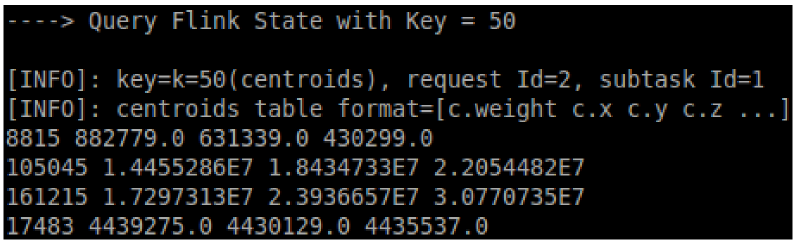


```
############### Flink QueryClient ###############
----> Query Task Manager with host name: 127.0.1.1
----> Query Task Manager with proxy port: 9069
----> Query JobId: c8d8980f7353394bc98a3056eae2aea2
----> Query Name: query-centroids

----> Type 'exit' to terminate the program
----> Query Flink State with Key = 50

[INFO]: key=k=50(centroids), request Id=0, subtask Id=1
[INFO]: centroids table format=[c.weight c.x c.y c.z ...]
6598 729923.0 657954.0 489287.0
73527 8080815.0 1.1018792E7 1.4079683E7
32198 302938.0 387227.0 166524.0
```

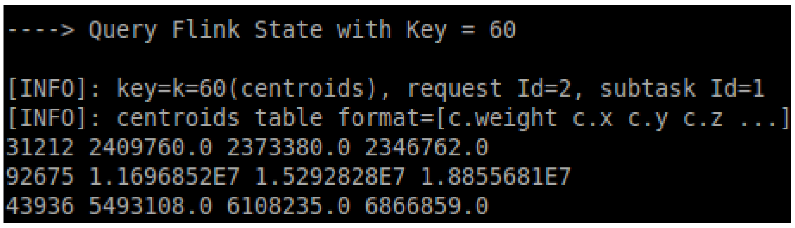
(a)



```
----> Query Flink State with Key = 50

[INFO]: key=k=50(centroids), request Id=2, subtask Id=1
[INFO]: centroids table format=[c.weight c.x c.y c.z ...]
8815 882779.0 631339.0 430299.0
105045 1.4455286E7 1.8434733E7 2.2054482E7
161215 1.7297313E7 2.3936657E7 3.0770735E7
17483 4439275.0 4430129.0 4435537.0
```

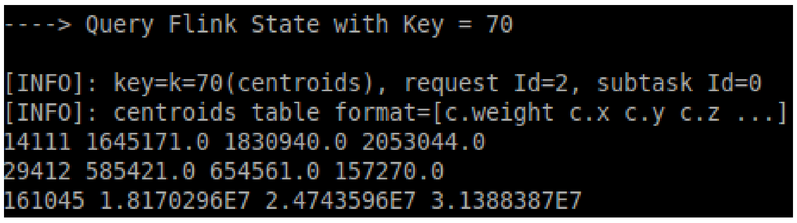
(b)



```
----> Query Flink State with Key = 60

[INFO]: key=k=60(centroids), request Id=2, subtask Id=1
[INFO]: centroids table format=[c.weight c.x c.y c.z ...]
31212 2409760.0 2373380.0 2346762.0
92675 1.1696852E7 1.5292828E7 1.8855681E7
43936 5493108.0 6108235.0 6866859.0
```

(c)



```
----> Query Flink State with Key = 70

[INFO]: key=k=70(centroids), request Id=2, subtask Id=0
[INFO]: centroids table format=[c.weight c.x c.y c.z ...]
14111 1645171.0 1830940.0 2053044.0
29412 585421.0 654561.0 157270.0
161045 1.8170296E7 2.4743596E7 3.1388387E7
```

(d)

Figure 4.8: Results from queries performed with different keys

Chapter 5

Experimental Evaluation

We conducted several experiments on different datasets to evaluate the performance of the distributed implementation of StreamKM++. In the first set of experiments, we discuss the trade-off between the runtime and the clustering cost of StreamKM++. Moreover, we compare the quality of the clustering with the original non-parallel StreamKM++. In the second set of experiments, we conduct experiments with different levels of parallelism to evaluate the runtime, the clustering cost and the throughput of our implementation. The highest number of input elements that we tested our implementation was 176 million (approximately 124 GB input file). We note that we conducted the following experiments by using the first version of our distributed implementation (Distributed StreamKM++ of section 4.1). Furthermore, we measure the quality of the clustering by using the cost function (sum of squared distances for each point to its nearest cluster center, see section 4.2).

5.1 Flink Cluster Setup

In order to run our experiments to the multi-node cluster of our university, we deployed Flink by using the Standalone Cluster setup. This setup includes a single Job Manager (master node) and at least one Task Manager (worker nodes). In our setup, we used 11 Task Managers with maximum number of parallel task slots 44 (i.e. 44 physical cores). During our experiments, the maximum Job parallelism that we used was 32, so we let Flink's runtime to make the choice of the specific task slots. Table 5.1 presents the system specifications of the Job and Task managers.

Node	CPU	Cores	Ram GB
1 Job Manager	Intel Xeon E5-2430 v2	6 (4 used)	32 (2 used)
11 Task Managers	Intel Xeon X3323	4 (4 used)	8 (2 used)

Table 5.1: Cluster Specifications

5.2 Non-Parallel Experiments

In this section we conduct experiments to compare the performance between our distributed implementation and the original StreamKM++. In order to accomplish a fair comparison, we used the same datasets from the original paper. The main source of these datasets is the UCI Machine Learning Repository [24]. At this point we give a brief description for the content of all the datasets that we use in the following experiments (table 5.2):

- Spambase [25] is a collection of spam and non-spam emails from work and personal emails. Each record is a vector that contains frequencies of specific words or characters which appear in the email. Each record is classified as spam or non-spam email. The total number of records is 4,601, and the number of attributes is 57 without the classification attribute.

- Intrusion [26] is a dataset that contains information about TCP transmissions in a simulated network environment. This simulation produced both normal and intrusion connections. We used the 10% unlabeled subset of the whole dataset, which contains 311,078 TCP transmissions with 34 attributes.

- Covertype [27] contains measurements of cartographic variables obtained from four areas in the Roosevelt National Forest of northern Colorado. The analysis of this dataset aims to classify the forest cover type of specific regions. Without the classification attribute this dataset contains 581,012 points is 54 dimensions.

- Tower [28] contains the RGB values from a 2,560 by 1,920 tower image. The 4,915,200 pixels are mapped to a 3-dimensional vector.

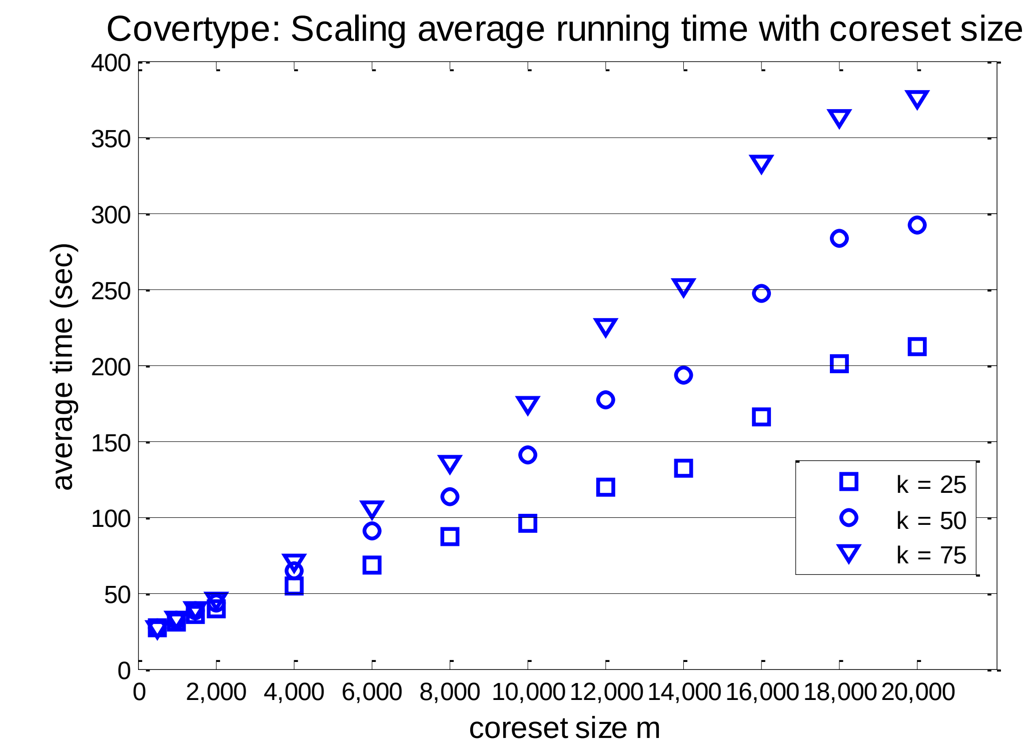
- Census [29] contains a one percentage sample of the Public Use Microdata (PUMS) person records from the 1990 U.S. census. The dataset was contributed by the U.S. Department of Commerce Census Bureau. It consists of 2,458,285 points in 68 dimensions.

Datasets	Data Points	Dimension	Type
Spambase	4,601	57	Integer, Real
Intrusion	311,078	34	Integer, Real
Covertype	581,012	54	Integer
Tower	4,915,200	3	Integer
Census	2,458,285	68	Integer

Table 5.2: Datasets Characteristics

5.2.1 Runtime vs. Quality

StreamKM++ produces clusterings which their quality depends on the size m of the coresets. A choice of a small size leads to clusterings with higher cost, because the cluster centers are computed from a smaller set of representative points. However, a coreset with a small size leads to less computation time, because the buckets of the merge-and-reduce technique consume the input stream more often. In order to choose an appropriate size for the coresets, we followed the same experimental procedure from the original paper by using our implementation of StreamKM++. We conducted several experiments for different values of k and m on the datasets Covertype and Tower. Figures 5.1 and 5.2 present the average running times and cost of the clusterings from 10 runs for each fixed k and m. Regarding the cost of the clustering, we observe that for coresets with size close to the number of centers, the quality can be improved significantly by increasing the size m. However, for coresets with size higher than 10,000 the quality of the clustering is marginally improved. As it concerns the running time of the clusterings, we observe that the growth of time is almost linear to the size of the coresets. Based on the above observations, we come to the same conclusion as the authors that the choice of m = 200k implies is a good trade-off between runtime and quality.



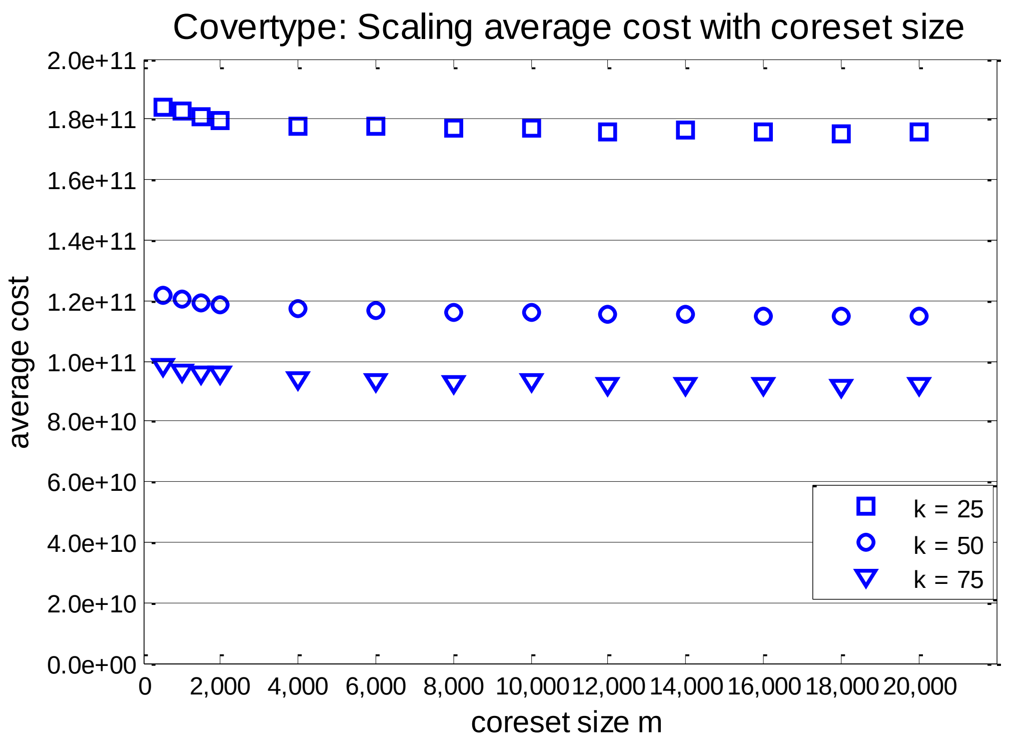
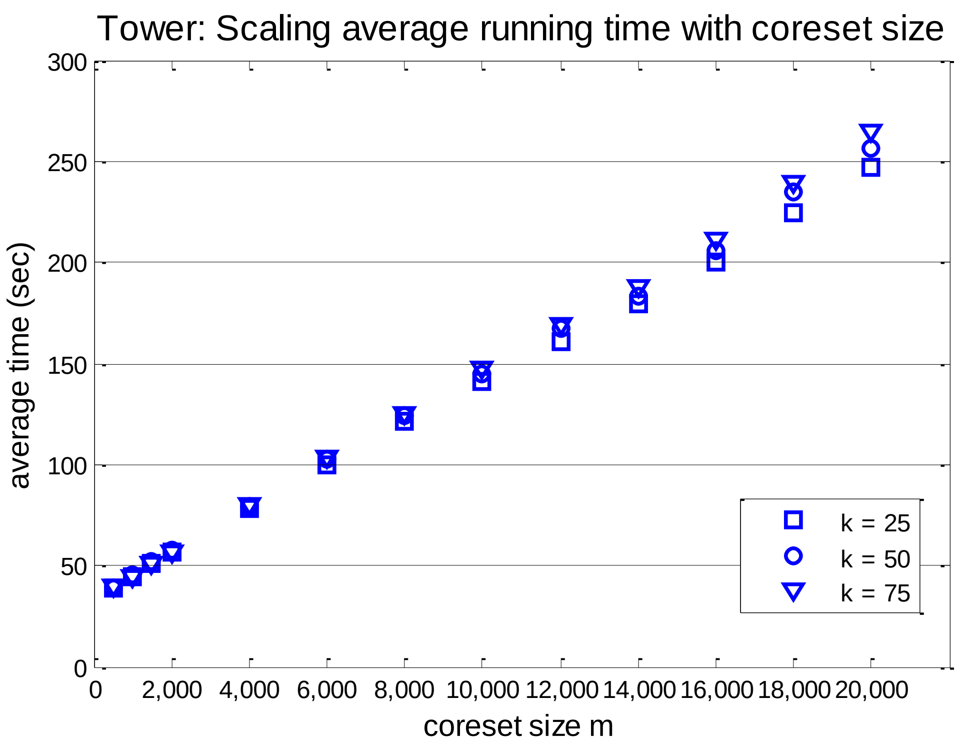


Figure 5.1: Scaling running time and average cost for Covertype



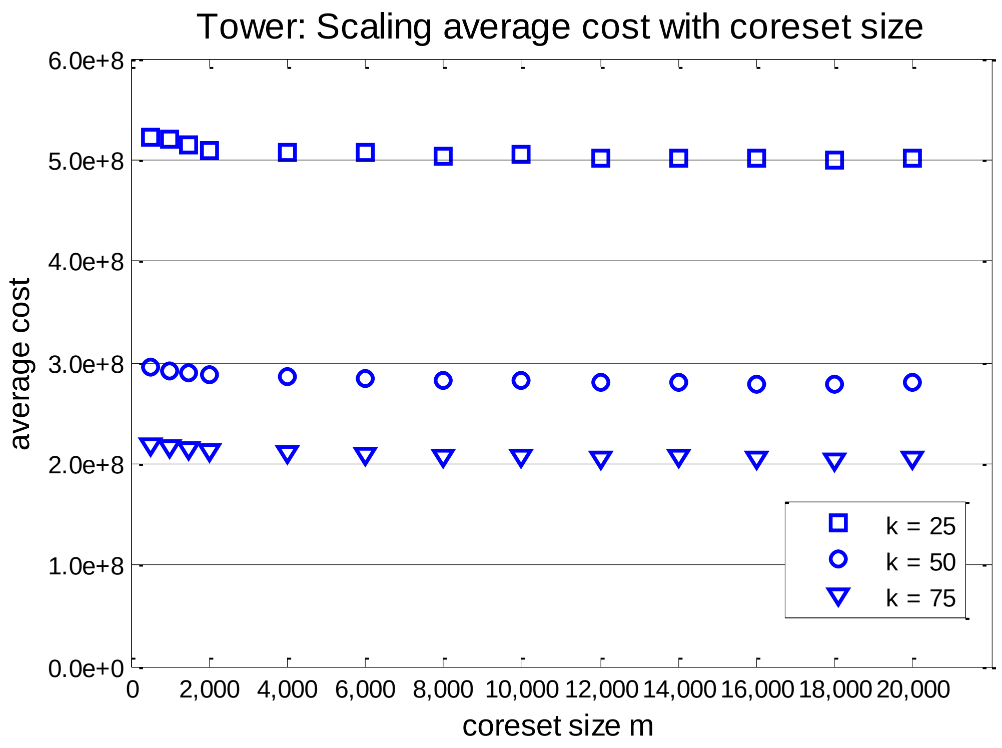


Figure 5.2: Scaling running time and average cost for Tower

5.2.2 Comparison with original StreamKM++

In this section we compare the clustering cost of our implementation with the original StreamKM++. We conducted the same experiments for all the datasets of table 5.2. We took the clustering costs of the original StreamKM++ from the table VII in the appendix section of that paper. In each of these experiments we set the same coreset size m = 200k and we conducted 10 experiments for each fixed k. We note that for some values of k, the Spambase dataset has less data points than 200k. We decided to use the original C-code to re-run the experiments of Spambase with coreset size m = 20k, because the authors do not mention the size of the coreset which they used. In figures 5.3-5.7, we observe that the clustering cost of our implementation is almost identical to the original. Now that we have established that our implementation produces accurate clusterings, we can move on to the experimental evaluation of the parallel execution.

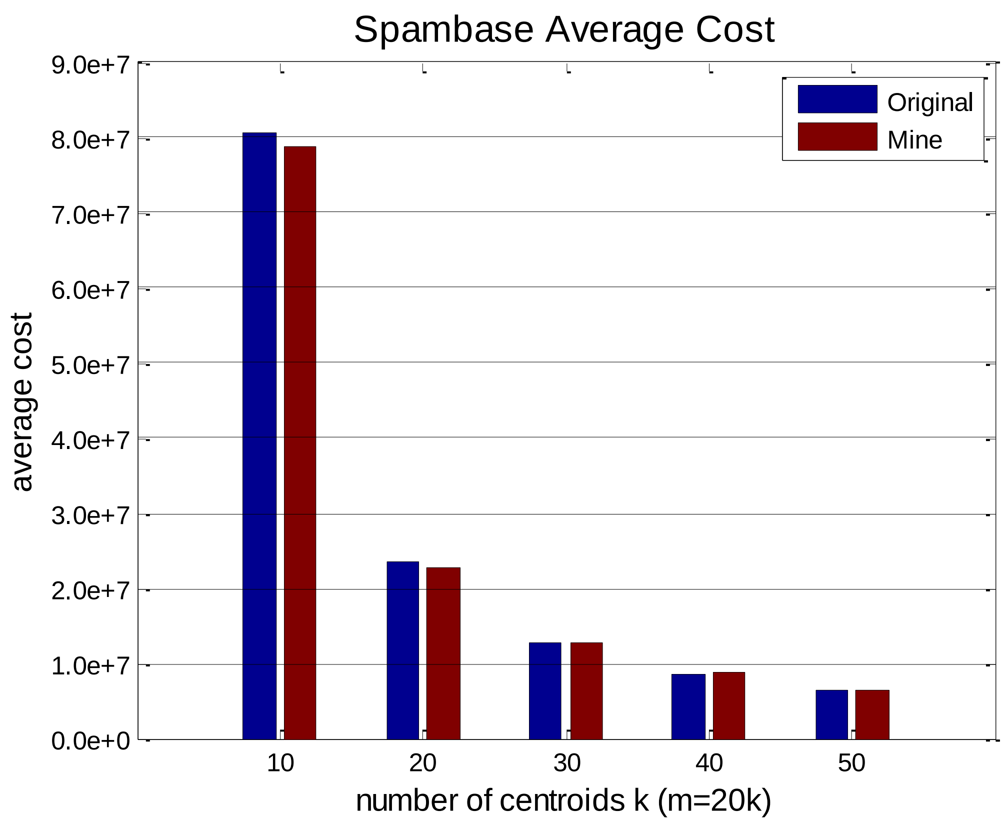


Figure 5.3: Average cost comparison for Spambase

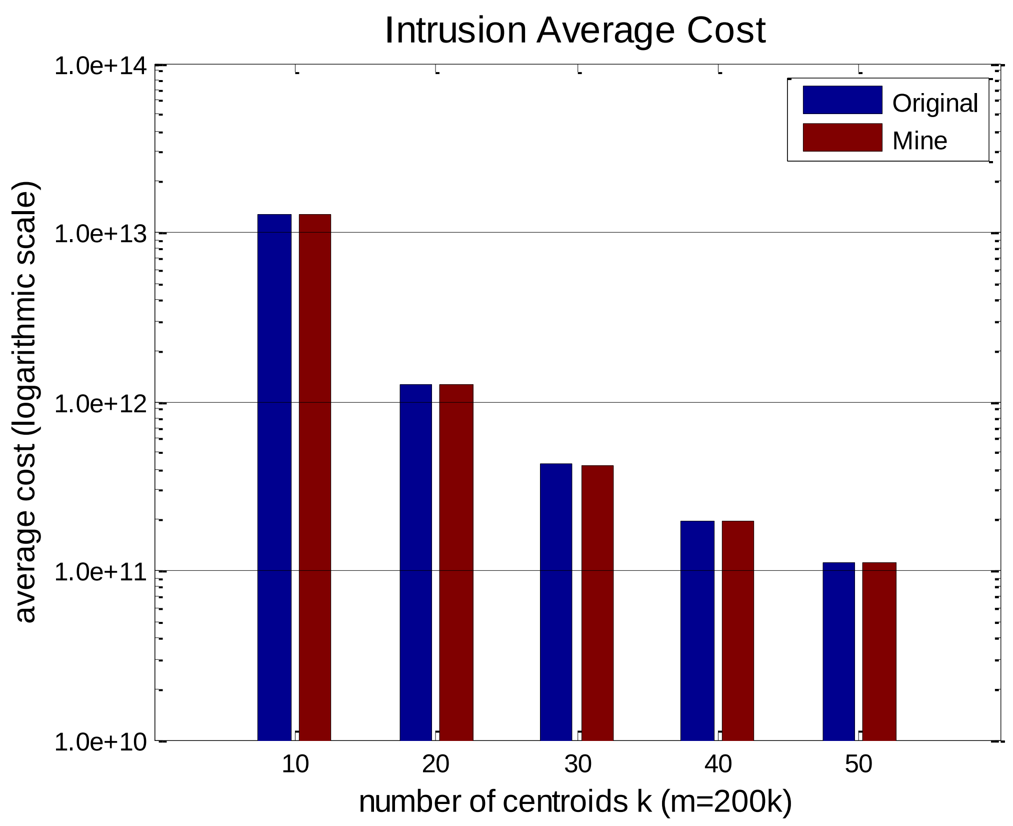


Figure 5.4: Average cost comparison for Intrusion

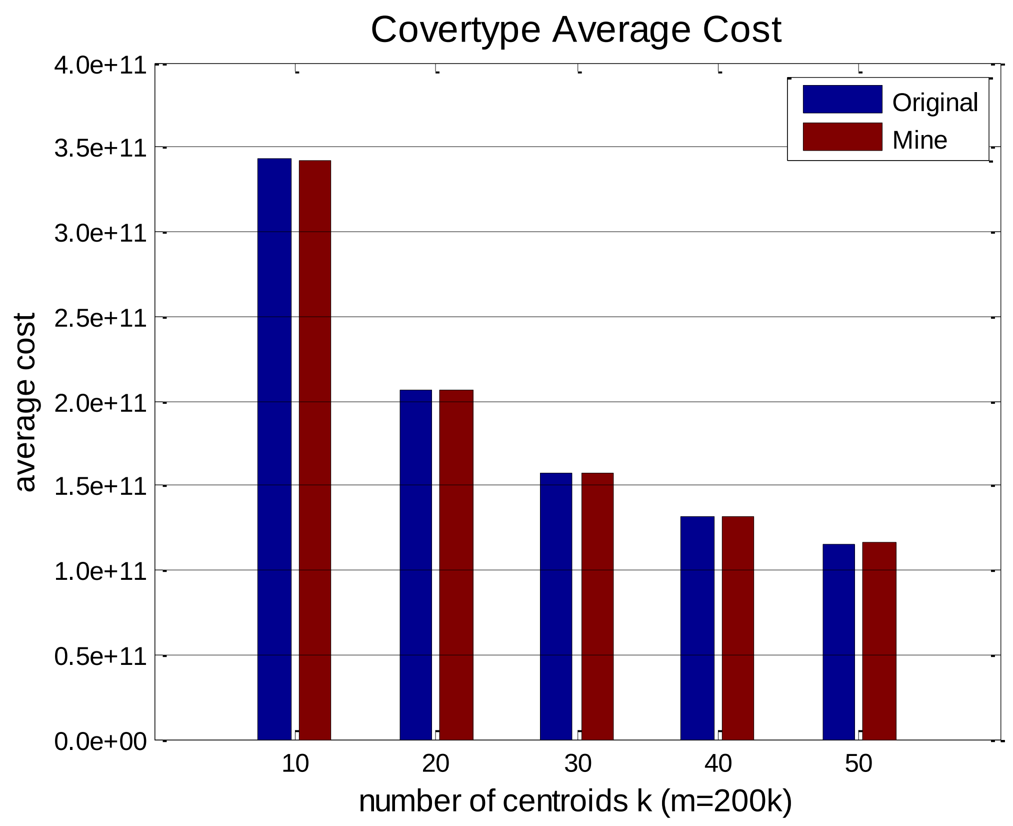


Figure 5.5: Average cost comparison for Covertype

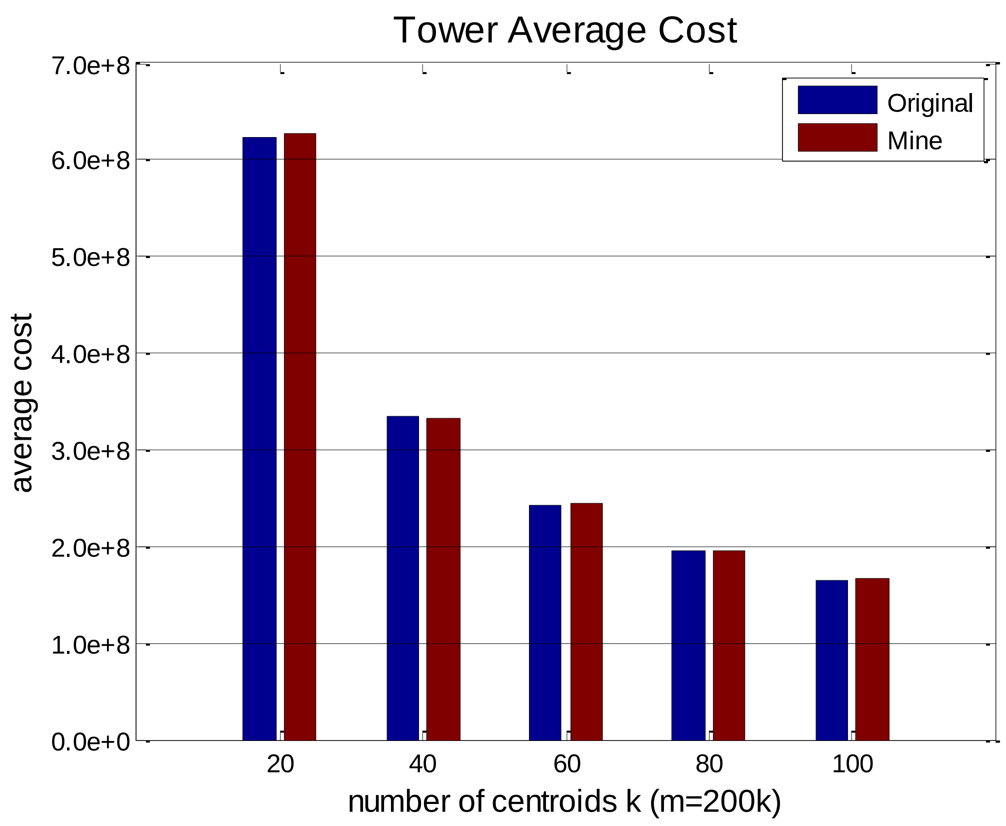


Figure 5.6: Average cost comparison for Tower

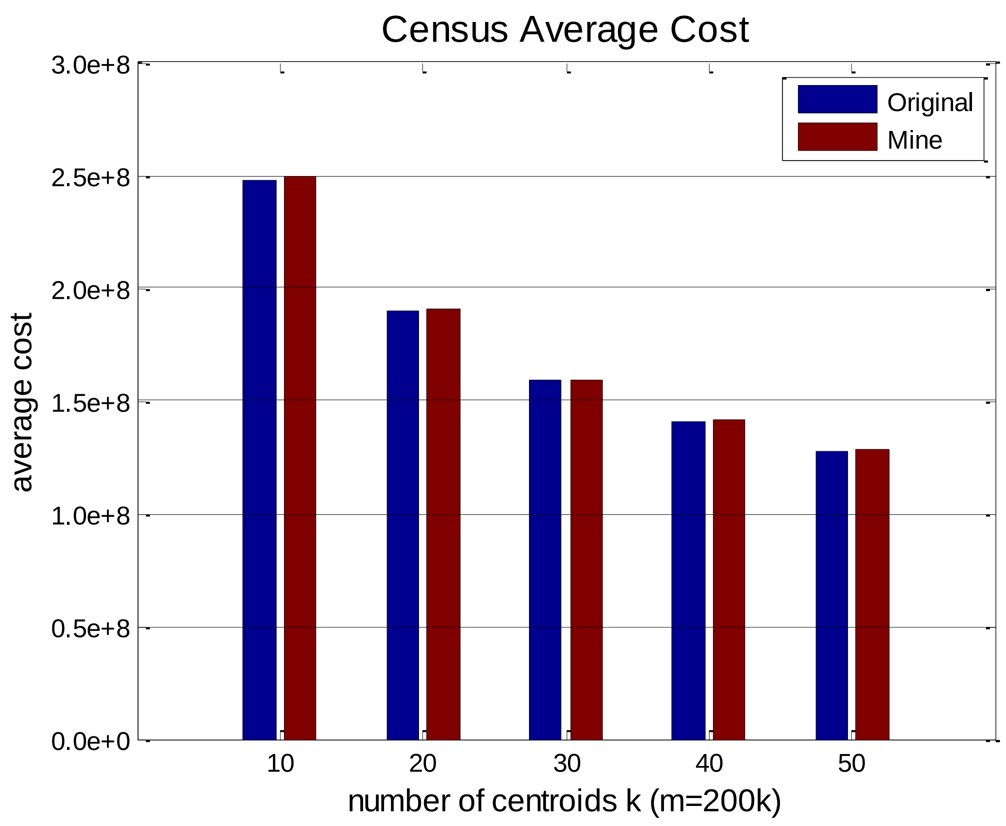


Figure 5.7: Average cost comparison for Census

5.3 Parallel Experiments

In this section, we evaluate the runtime, the clustering cost and the throughput of our distributed implementation, for different levels of parallelism. We use the HIGGS dataset [30] from the UCI Machine Learning Repository to run our experiments. This dataset contains events from simulated proton-proton collisions produced by the ATLAS experiments at CERN. It was published for the classification problem to distinguish between a signal process which produces Higgs bosons and a background process which does not. The dataset is composed of 11 million events (7.76 GB file size) and 28 real-valued attributes.

Dataset	Data Points	Dimension	Type	Size
HIGGS	11,000,000	28	Real	7.76 GB

Table 5.3: HIGGS Dataset

5.3.1 Runtime & Cost

In order to evaluate the runtime and the clustering cost of the distributed StreamKM++, we conducted several experiments on the Flink Cluster for different values of cluster centers and job parallelism. More specifically, we tested our implementation for the set of k = [25, 50, 75, 100] cluster centers and for job parallelism [1, 2, 4, 8, 16, 32]. We conducted 4 experiments for each fixed value of centers and parallelism, and we calculated the average running time and clustering cost. Furthermore, we tested our implementation with two different settings. In the first setting, we execute 5 times the k-means++ algorithm on the final coreset and we keep the clustering with the minimum cost. In the second setting, we execute only one time the k-means++ on the final coreset. We use these settings because k-means++ is executed by a non-parallel operator in the streaming dataflow, and therefore we want to observe the impact of the number of applications on the running time and cost.

In figure 5.8 we observe that the average running time for each fixed value k, droops rapidly with the increase of the job parallelism. This result was expected because we split the input file according to the parallelism, and we use parallel operators for the consumption of the data points and for the construction of the partial coresets. However, for parallelism higher than 8 the running time droops with lower rates. The reason for this is that the non-parallel part of our implementation (5 times execution of k-means++ on the final coreset) takes constant time, that is independent of both the size of the input file and the job parallelism. Thus, the influence of this constant time is greater, when the running time of the clustering decreases. Table 5.4 shows for each fixed value k the percentage reduction in the running time that we accomplished, by increasing the parallelism from 1 to 32. The overall mean percentage reduction from all the different values of k is 86.2%.

In figure 5.9 we observe that the average cost for each fixed value k, decreases with the increase of the job parallelism. This behavior is interesting because it gives us the advantage to increase the job parallelism without sacrificing the cost of the clustering. In the contrary, with the increase of the parallelism we get slightly better results. We believe that this behavior comes from fact that with parallelism higher than one, we maintain more parallel lists and thus more total buckets. Therefore, the partial coresets represent the input data points with more precision.

Cluster centers	Running time (sec) with parallelism 1	Running time (sec) with parallelism 32	Percentage reduction
k = 25	859.0	53.2	93.8%
k = 50	1063.8	119.1	88.8%
k = 75	1228.0	207.6	83.0%
k = 100	1530.4	322.5	78.9%
Mean percentage reduction = 86.2%			

Table 5.4: Percentage reduction in average running time (5 k-means++)

Figures 5.10, 5.11 represent the same experiments with the corresponding figures 5.8, 5.9, but with only one application of k-means++ on the final coreset. In figure 5.10 we observe that for each fixed value of cluster centers, the overall running time decreases. Moreover, we observe the running time becomes less dependent on the value of the cluster centers. For example, when the parallelism is 32 the running time for each value of k is below 200 seconds. This is also noticeable from the fact that the overall mean percentage reduction is increased from 86.2% to 91.9%. Figure 5.11 shows that by using one application of k-means++, the average cost slightly increases form the first setting which applies 5 times the k-means++, but it still gets better with the increment of the job parallelism.

Cluster centers	Running time (sec) with parallelism 1	Running time (sec) with parallelism 32	Percentage reduction
k = 25	854.2	49.0	94.3%
k = 50	1014.6	67.8	93.3%
k = 75	1132.2	100.3	91.1%
k = 100	1282.9	140.6	89.0%
Mean percentage reduction = 91.9%			

Table 5.5: Percentage reduction in average running time (1 k-means++)

To get a better picture for the comparison between these two settings, we constructed the bar graphs of figures 5.12, 5.13, 5.14 and 5.15. Each one of these figures compares the average running time and clustering cost between these two settings, for a fixed value of cluster centers. We observe that the benefit we get from the running time, by reducing the number of applications of k-means++ from five to one, is greater when the number of cluster centers increases. At the same time, we observe that by reducing the number of applications, the clustering cost slightly increases. However, the accuracy in the clustering cost that we sacrifice is two orders of magnitude smaller than the total cost. For the above reasons, we believe that the setting which applies one time the k-means++ on the final coreset, is better.

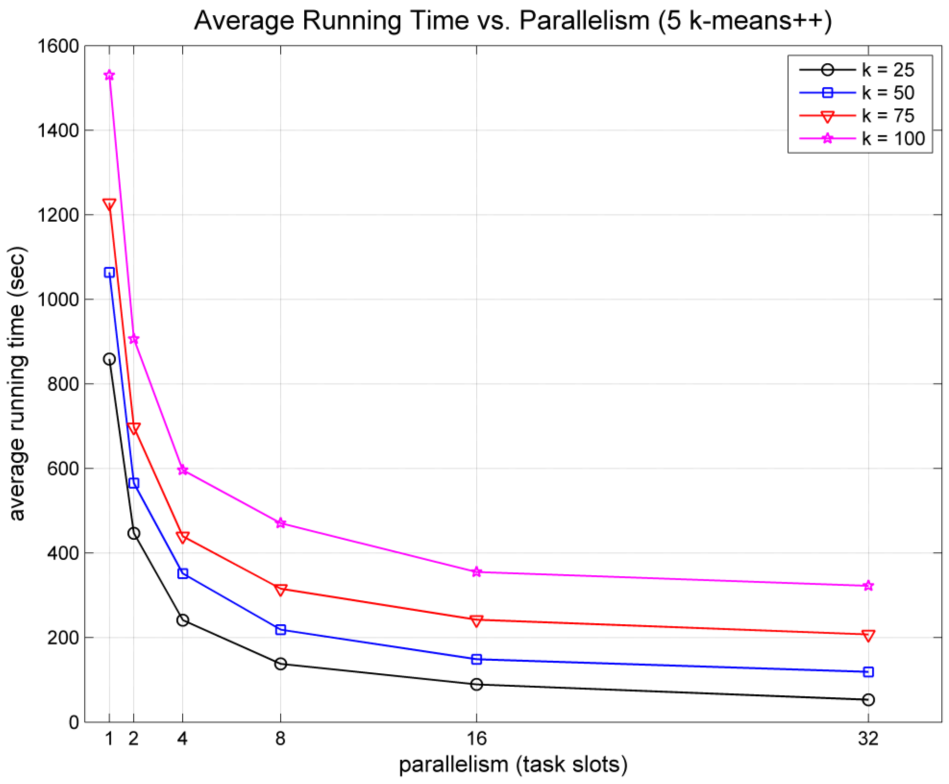


Figure 5.8: Average running time for 5 applications of k-means++

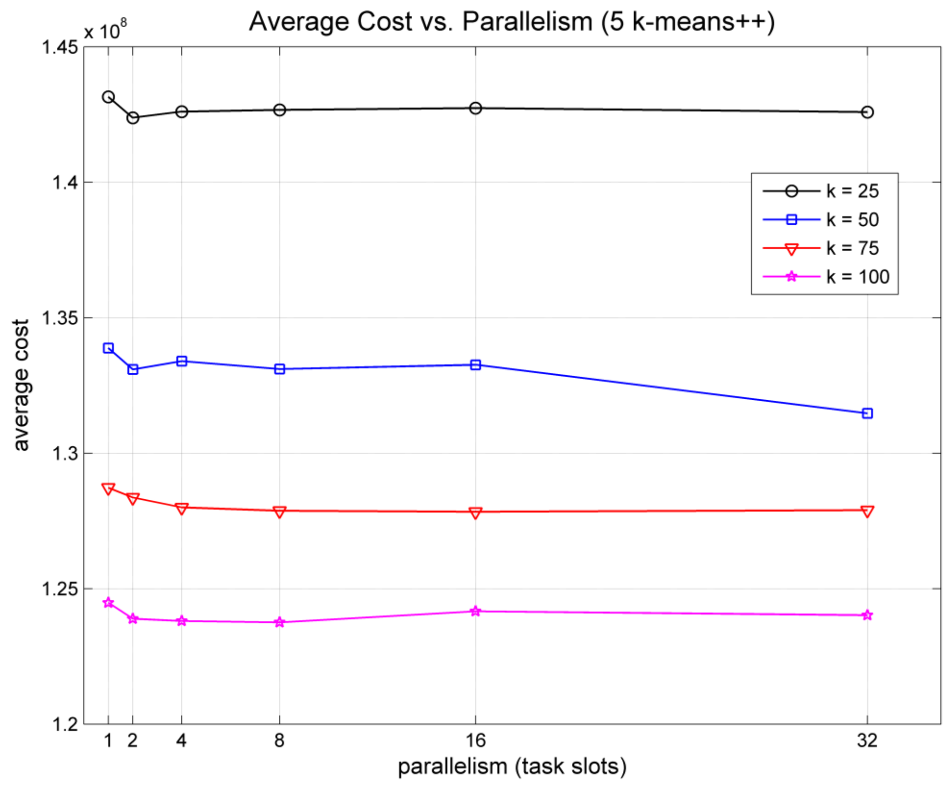


Figure 5.9: Average cost for 5 applications of k-means++

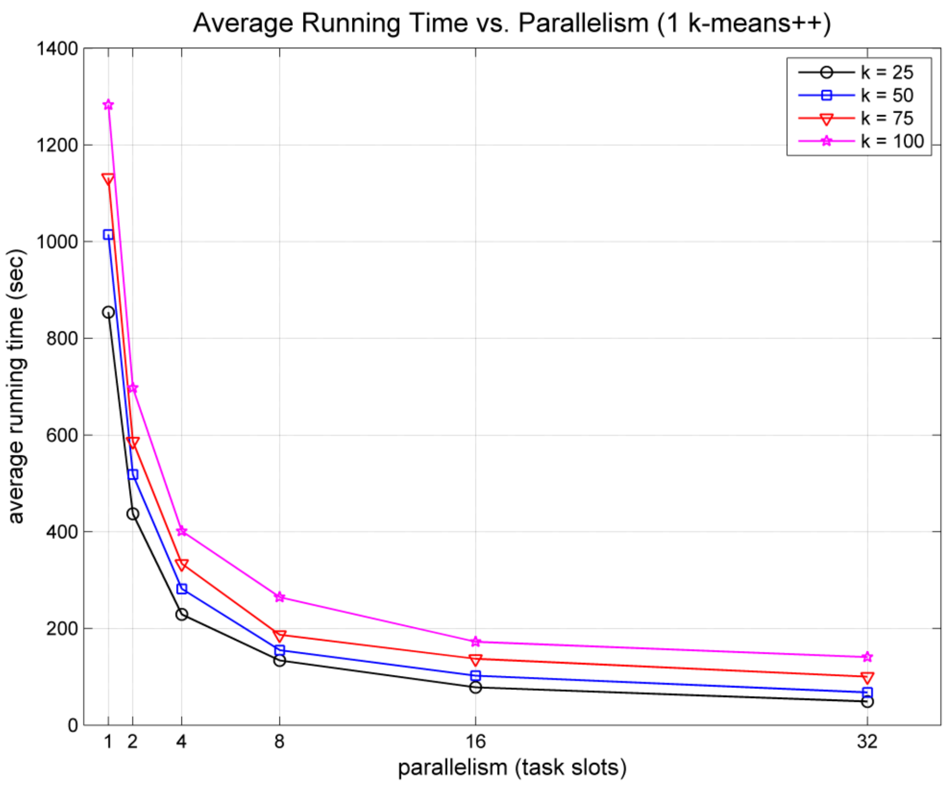


Figure 5.10: Average running time for 1 application of k-means++

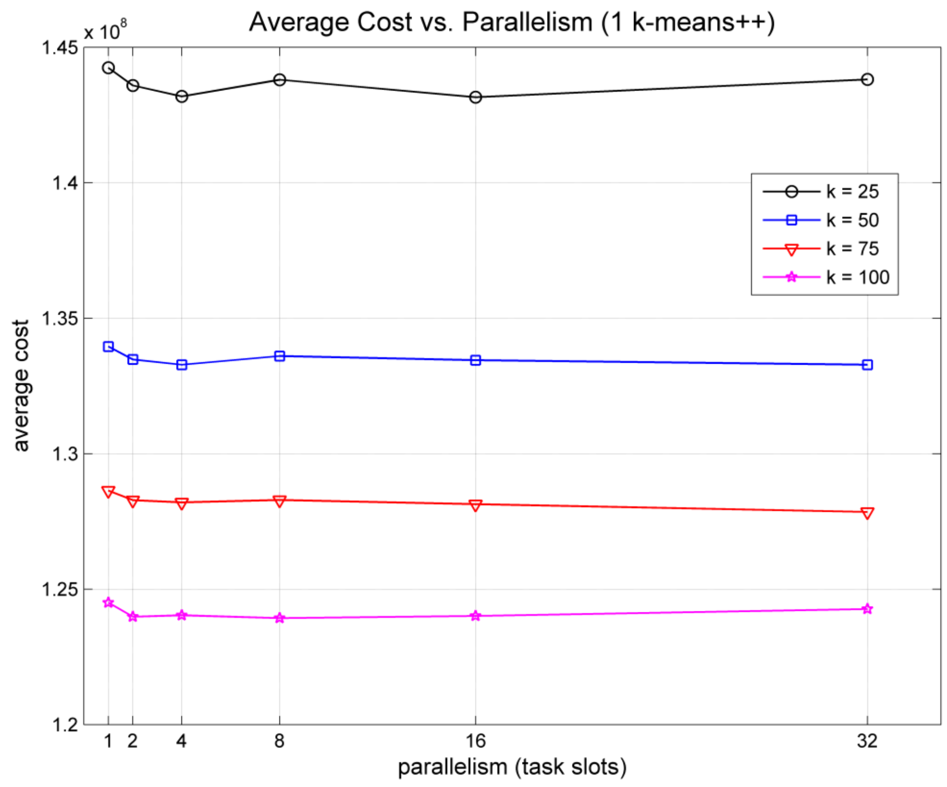


Figure 5.11: Average cost for 1 application of k-means++

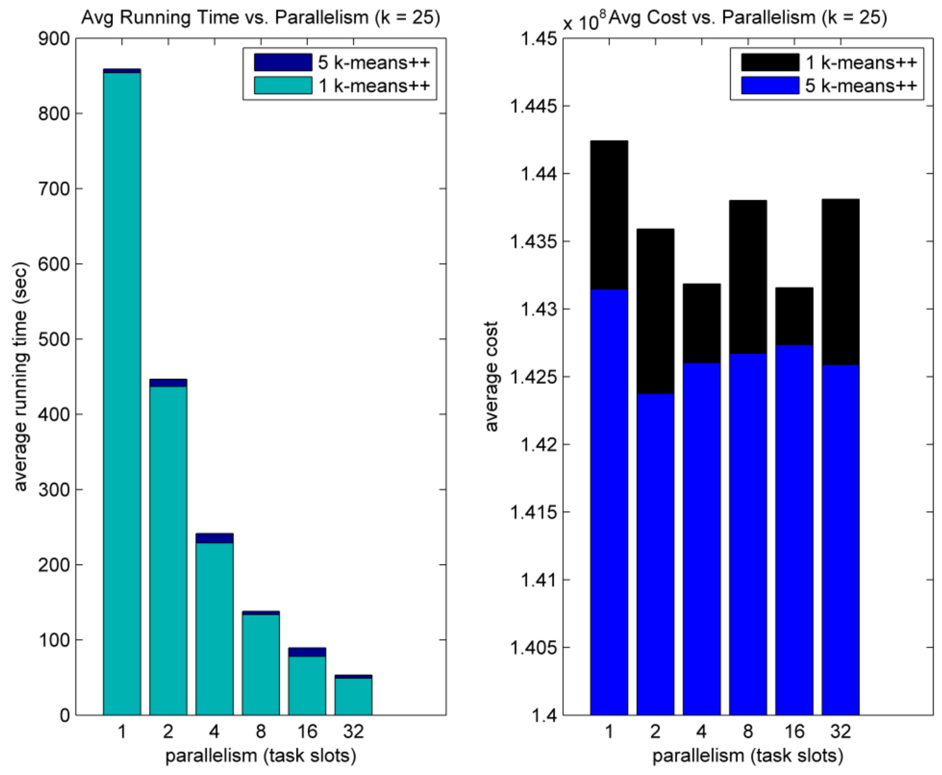


Figure 5.12: Comparison between 1 & 5 applications of k-means++ for k = 25

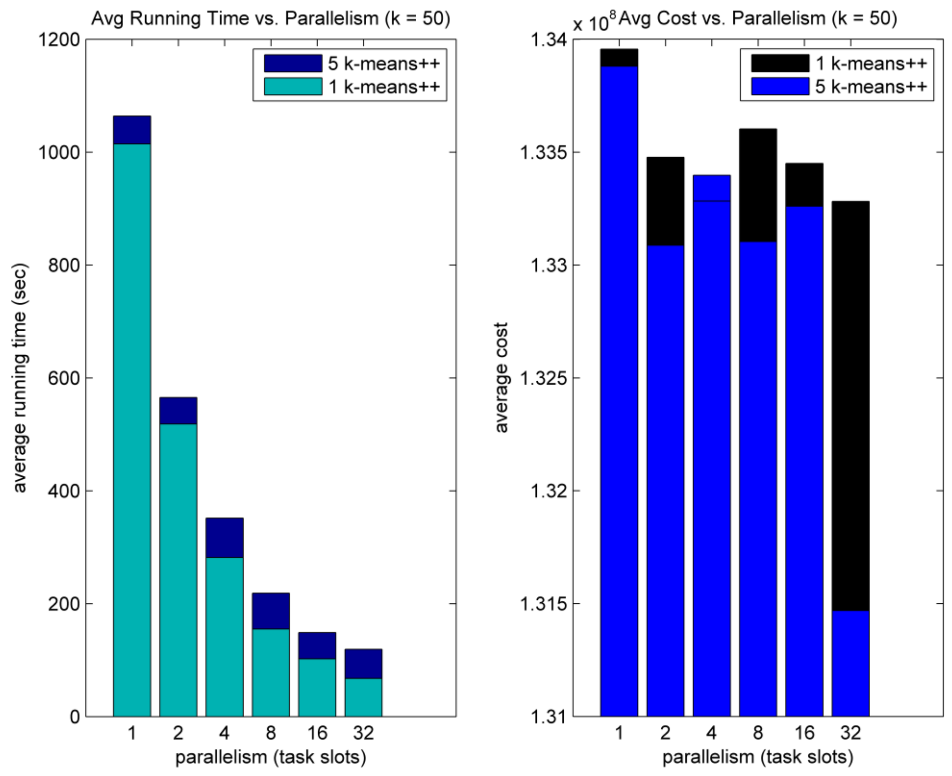


Figure 5.13: Comparison between 1 & 5 applications of k-means++ for k = 50

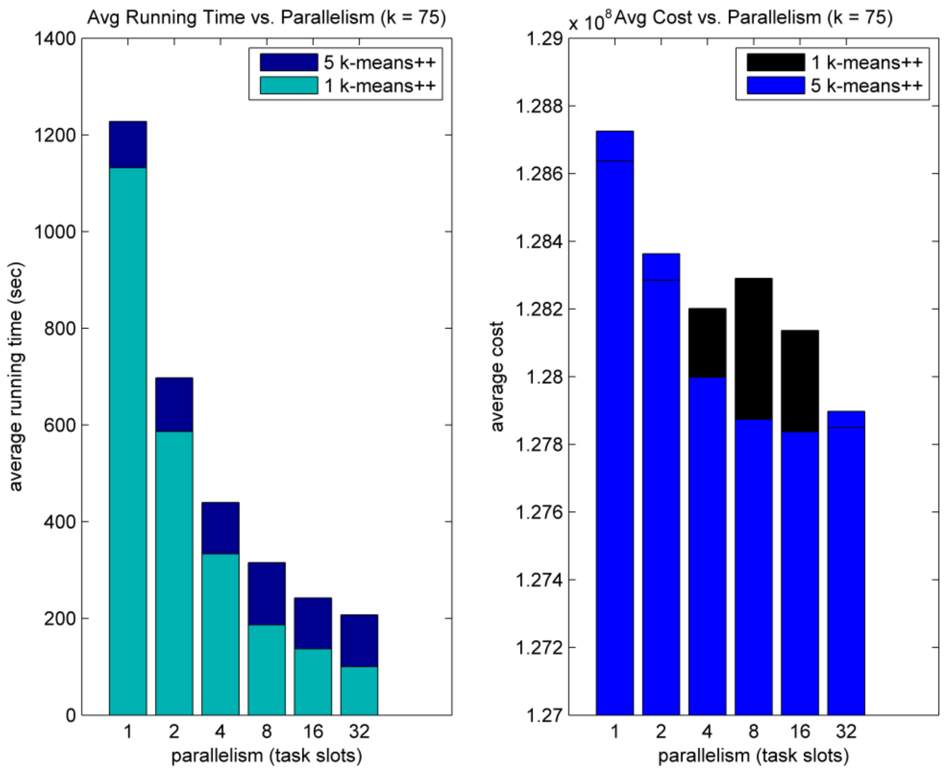


Figure 5.14: Comparison between 1 & 5 applications of k-means++ for k = 75

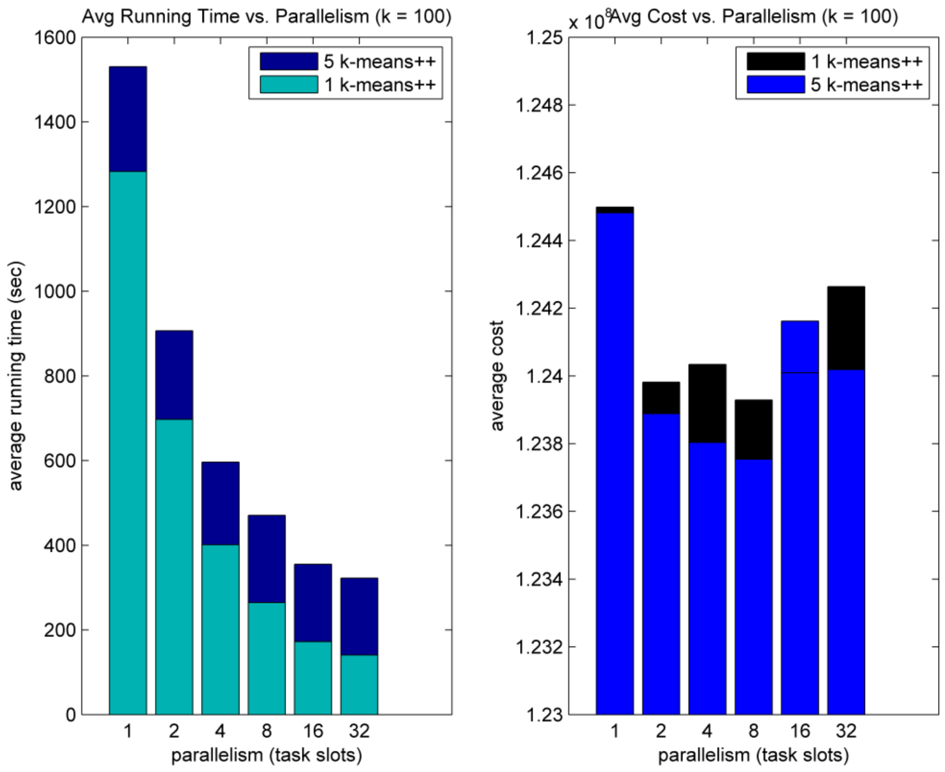


Figure 5.15: Comparison between 1 & 5 applications of k-means++ for k = 100

5.3.2 Throughput

In this section, we conduct experiments to evaluate the throughput (tuples/sec) of our implementation for increasing number of data points. In order to simulate the increasement on the size of the input, we used multiple times the HIGGS dataset. Table 5.6 shows the different numbers of data points along with the size of the input file that we used in our experiments. We note that for the following experiments we used the second setting of our implementation, which applies only one time the k-means++ algorithm on the final coreset. We conducted 4 experiments for each fixed value of centers, parallelism and data points, and we took the average rate.

Data points (million)	Input file size (GB)
1 x 11 = 11	1 x 7.76 = 7.76
2 x 11 = 22	2 x 7.48 = 15.52
4 x 11 = 44	4 x 7.48 = 31.04
8 x 11 = 88	8 x 7.48 = 62.08
16 x 11 = 176	16 x 7.48 = 124.17

Table 5.6: Different sizes of input file used for throughput experiments

Each one of the figures 5.16 and 5.17 compares the throughput rate that our implementation achieves for different levels of parallelism. In figure 5.16 we observe that for the fixed number of 25 centers, by increasing the parallelism from 8 to 16 we succeed 46% better average throughput. In figure 5.17 we observe that for the fixed number of 50 centers, the same increment leads to 37% better average throughput. In both of these figures we observe that by increasing the number of data points the throughput rate increases significantly. As we mentioned previously, our implementation contains a non-parallel part (execution of k-means++ on the final coreset) that takes constant time and it is independent of both the size of the input file and the job parallelism. Therefore, when we increase the number of data points the constant time is divided to the number of points, and thus throughput rate increases.

Figures 5.18 and 5.19 represent the same experiments but for fixed levels of parallelism. In figure 5.18, the increasement of the cluster centers from 25 to 50 leads to 13% lower average throughput with parallelism 8. In figure 5.19, the same increasement leads to 22% lower average throughput with parallelism 16.

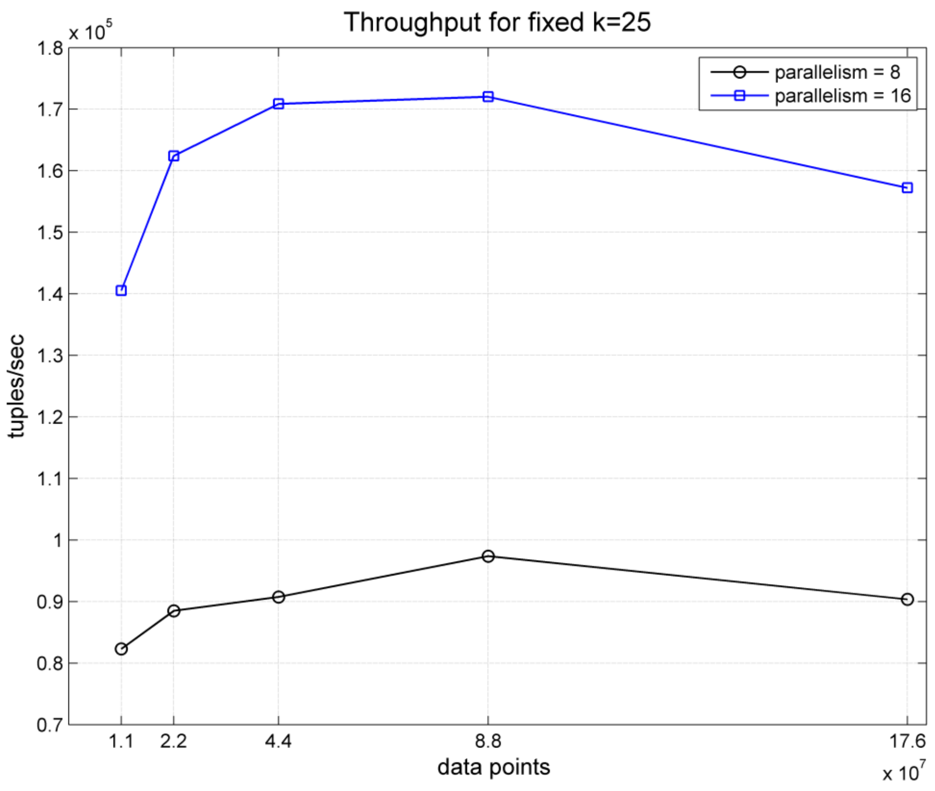


Figure 5.16: Throughput rate for 25 cluster centers

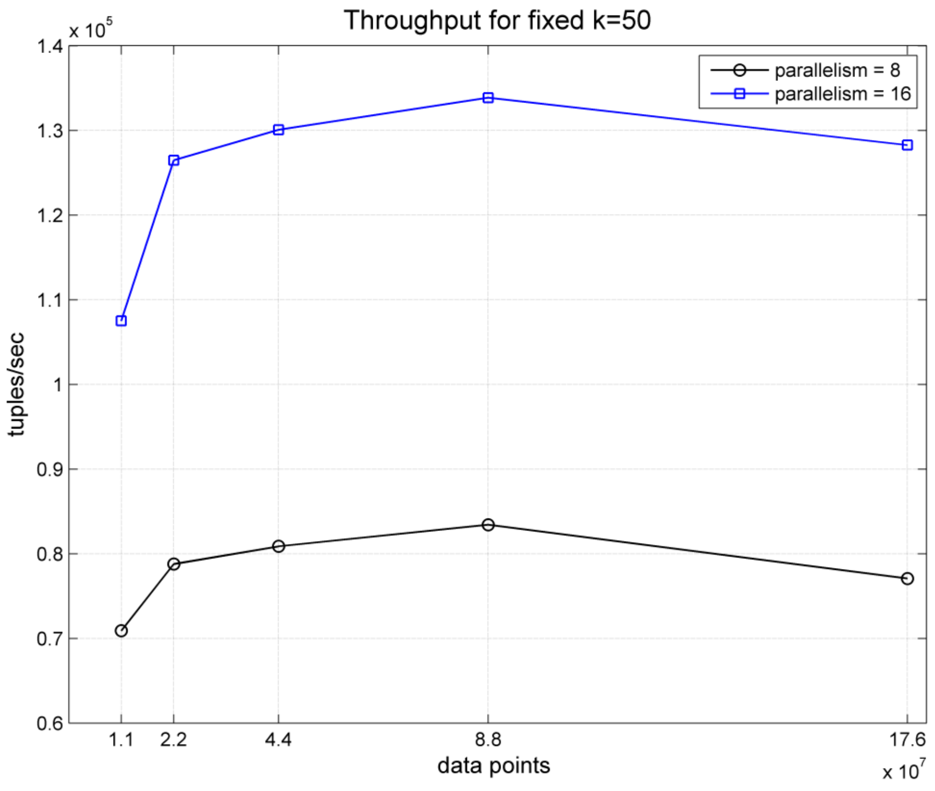


Figure 5.17: Throughput rate for 50 cluster centers

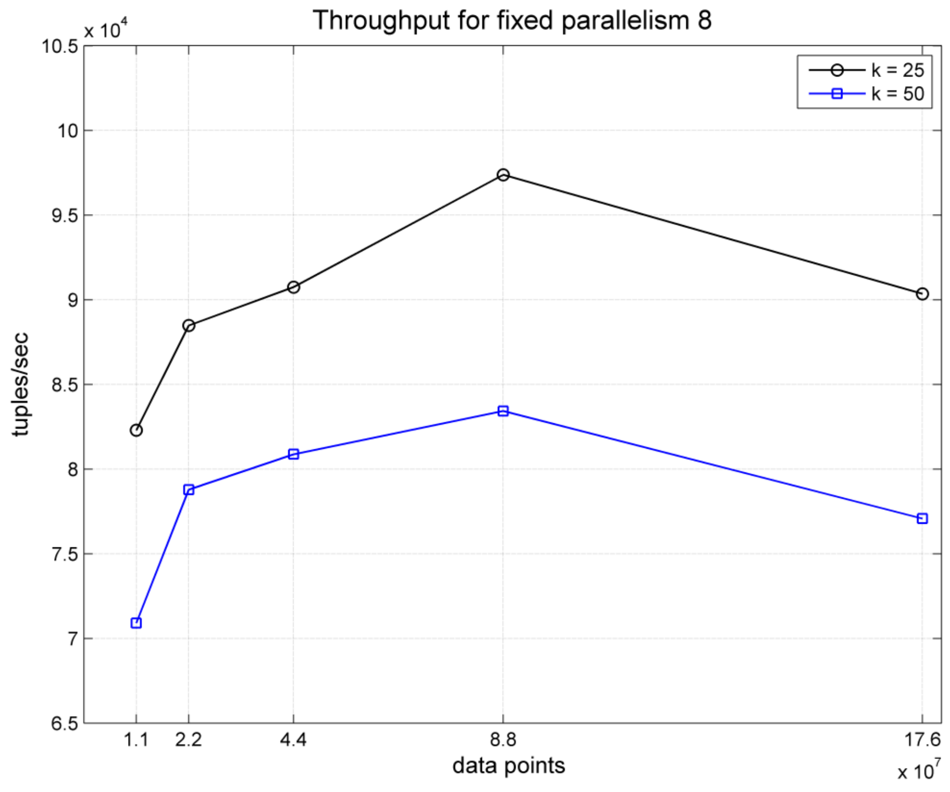


Figure 5.18: Throughput rate for job parallelism 8

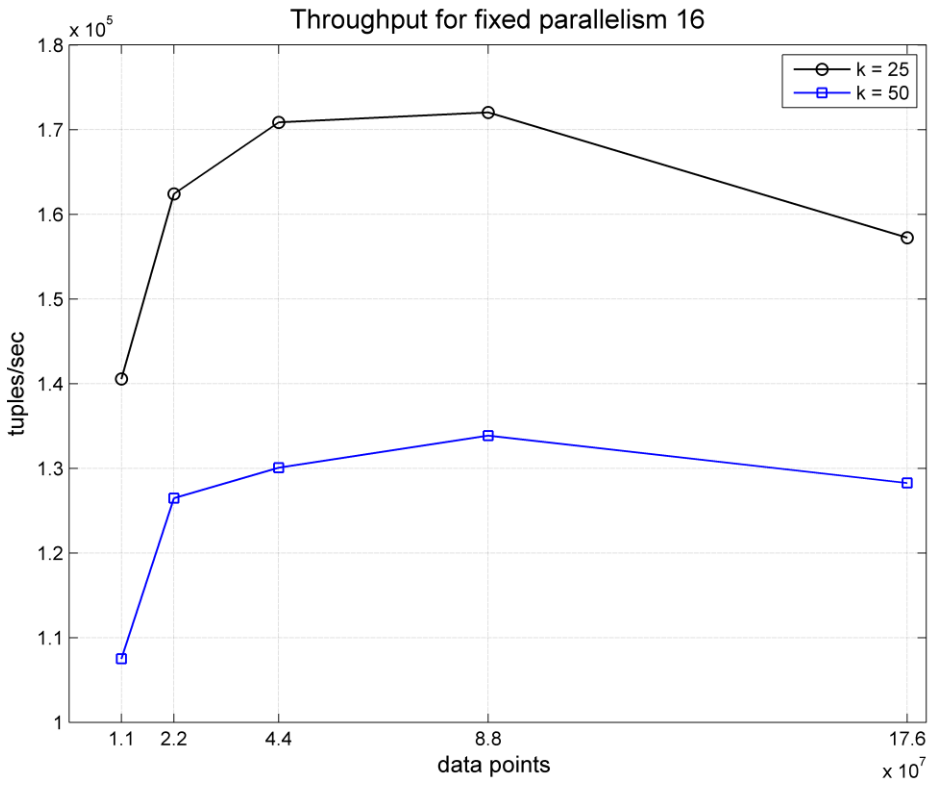


Figure 5.19: Throughput rate for job parallelism 16

Chapter 6

Conclusions & Future Work

In this diploma thesis, we proposed a distributed implementation of the well-known StreamKM++ algorithm for clustering data streams. For the development phase, we used Apache Flink framework which is a state-of-the-art distributed processing engine for large scale computations over unbounded and bounded data streams. We proposed two different distributed implementations for the StreamKM++. In the first one, we computed the set of cluster centers after the consumption of the entire input dataset. In the second one, we proposed an alternative distributed model that produces periodically requests for the re-evaluation of the cluster centers. In addition to that, we modified the stage that computes the clustering, to produce simultaneously clusterings with different number of centers. Moreover, we developed a program that exploits the Queryable State feature of Flink, in order to allow the user to query the most up-to-date values of the cluster centers.

We conducted several experiments on different datasets to evaluate the performance of our implementation. In the first set of experiments, we proved that our implementation produces cluster centers with the same quality of the original StreamKM++. In the second set of experiments, we tested the running time, the clustering cost and the throughput of our implementation, for different levels of parallelism. The experimental results proved that by increasing the job parallelism, the running time droops significantly and at the same time the quality of the clustering gets slightly better. Additionally, we showed that by reducing the number of applications of k-means++ on the final sample, we get even better results in the running time by sacrificing only a very small fraction of the clustering cost. Finally, throughput experiments showed that our implementation handles efficiently the growth in the size of the input.

In future work it is important to evaluate the performance of the parallel implementation that produces requests. We could measure the throughput rate for different rates of requests and for different numbers of clusterings that we produce in the final stage of the dataflow. Furthermore, we could develop a prediction system based on section 4.2 that periodically re-evaluates the cluster centers and then uses these centers to predict the cluster in which the new data falls into. Finally, we could modify the StreamKM++ algorithm to handle concept drift. This could be done by using only a small percentage of the buckets that represent “old points”.

References

[1] ACKERMANN, M. R.; MÄRTENS, M.; RAUPACH, C.; SWIERKOT, K.; LAMMERSEN, C.; and SOHLER, C. 2012. StreamKM++: A clustering algorithm for data streams. Journal of Experimental Algorithmics 17:2–4.

[2] FORGY, E. W. 1965. Cluster analysis of multivariate data: Efficiency versus interpretability of classifications. Biometrics 21, 768–780.

[3] MACQUEEN, J. B. 1967. Some methods for classification and analysis of multivariate observations. In Proceedings of the 5th Berkeley Symposium on Mathematical Statistics and Probability. Vol. 1, University of California Press, 281–297.

[4] SELIM, S. Z. AND ISMAIL, M. A. 1984. k-means-type algorithms: A generalized convergence theorem and characterization of local optimality. IEEE Trans. Pattern Anal. Mach. Intell. 6, 1, 81–87.

[5] KANUNGO, T.,MOUNT, D. M., NETANYAHU, N. S., PIATKO, C. D., SILVERMAN, R., AND WU, A. Y. 2004. A local search approximation algorithm for k-means clustering. Computat. Geometry 28, 2-3, 89–112.

[6] ALOISE, D.,DESHPANDE, A.,HANSEN, P., AND POPAT, P. 2009. NP-hardness of Euclidean sum-of-squares clustering. Machine Learn. 75, 2, 245–248.

[7] DASGUPTA, S. 2008. The hardness of k-means clustering. Tech. rep. CS2008-0916, University of California.

[8] MAHAJAN, M.; NIMBHORKAR, P.; VARADARAJAN, K. (2009). "The Planar k-Means Problem is NP-Hard". Lecture Notes in Computer Science. Lecture Notes in Computer Science. 5431: 274–285.

[9] ARTHUR, D. AND VASSILVITSKII, S. 2007. k-means++: the advantages of careful seeding. In Proceedings of the 18th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA ’07). SIAM, 1027–1035.

[10] PAVEL BERKHIN. A survey of clustering data mining techniques. In Grouping multidimensional data, pages 25–71. Springer, 2006.

[11] VATTANI, A. 2009. k-means requires exponentially many iterations even in the plane. In Proceedings of the 25th ACM Symposium on Computational Geometry (SoCG ’09). ACM, 324–332.

[12] ARTHUR, DAVID; VASSILVITSKII, SERGEI (2006-01-01). "How Slow is the K-means Method?". Proceedings of the Twenty-second Annual Symposium on Computational Geometry. SCG '06. New York, NY, USA: ACM: 144–153.

[13] ARTHUR, D.,MANTHEY, B., AND R¨OGLIN, H. 2009. k-means has polynomial smoothed complexity. In Proceedings of the 50th Annual IEEE Symposium on Foundations of Computer Science (FOCS ’09). IEEE Computer Society, 405–414.

[14] LLOYD, S. P. 1982. Least squares quantization in PCM. IEEE Trans. Infor. Theory 28, 2, 129–137.

[15] BENTLEY, J. L. AND SAXE, J. B. 1980. Decomposable searching problems I: Static-to dynamic transformation. J. Algor. 1, 4, 301–358.

[16] HAR-PELED, S. AND MAZUMDAR, S. 2004. On coresets for k-means and k-median clustering. In Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC ’04). ACM, 291–300.

[17] ZHANG, T., RAMAKRISHNAN, R., AND LIVNY, M. 1996. BIRCH: An efficient data clustering method for very large databases. In Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD ’96). ACM, 103–114.

[18] GUHA, S.,MEYERSON, A.,MISHRA, N.,MOTWANI, R., AND O’CALLAGHAN, L. 2003. Clustering data streams: Theory and practice. IEEE Trans. Knowl. Data Engin. 15, 3, 515–528.

[19] https://data-artisans.com/blog/curious-case-broken-benchmark-revisiting-apache-flink-vs-databricks-runtime

[20] P. Carbone, G. F´ora, S. Ewen, S. Haridi, and K. Tzoumas. Lightweight asynchronous snapshots for distributed dataflows. arXiv:1506.08603, 2015.

[21] http://www.onmyphd.com/?p=k-means.clustering

[22] https://hadoop.apache.org/

[23] https://cs.uni-paderborn.de/en/cuk/research/completed-projects/dfg-priority-programme-1307/streamkm/

[24] https://archive.ics.uci.edu/ml/index.php

[25] https://archive.ics.uci.edu/ml/datasets/spambase

[26] http://archive.ics.uci.edu/ml/machine-learning-databases/kddcup99-mld/, file: kddcup.newtest _10_percent_unlabeled.gz

[27] https://archive.ics.uci.edu/ml/datasets/covertype

[28] http://homepages.uni-paderborn.de/frahling/coremeans.html

[29] https://archive.ics.uci.edu/ml/datasets/US%2BCensus%2BData%2B(1990)

[30] https://archive.ics.uci.edu/ml/datasets/HIGGS#

[31] https://flink.apache.org/