University of Reading

Efficient Clustering Techniques for Big Data

Sami Al Ghamdi

Submitted in partial fulfilment of the requirements of
the University of Reading for the degree of
Doctor of Philosophy in Computer Science

Supervisor
Dr. Giuseppe Di Fatta

Department of Computer Science
School of Mathematical, Physical and Computational Sciences
University of Reading

September 2018

Abstract

Clustering is an essential data mining technique that divides observations into groups where each group contains similar observations. K-Means is one of the most popular and widely used clustering algorithms that has been used for over fifty years. The majority of the running time in the original K-Means algorithm (known as Lloyd's algorithm) is spent on computing distances from each data point to all cluster centres to find the closest centre to each data point. Due to the current exponential growth of the data, it became a necessity to improve K-Means even further to cope with large-scale datasets, known as Big Data. Hence, the main aim of this thesis is to improve the efficiency and scalability of Lloyd's K-Means.

One of the most efficient techniques to accelerate K-Means is to use triangle inequality. Implementing such efficient techniques on a reliable distributed model creates a powerful combination. This combination can lead to an efficient and highly scalable parallel version of K-Means that offers a practical solution to the problem of clustering Big Data.

MapReduce, and its popular open-source implementation known as Hadoop, provides a distributed computing framework that efficiently stores, manages, and processes large-scale datasets over a large cluster of commodity machines. Many studies introduced a parallel implementation of Lloyd's K-Means on Hadoop in order to improve the algorithm's scalability. This research examines methods based on triangle inequality to achieve further improvements on the efficiency of the parallel Lloyd's K-Means on Hadoop.

Variants of K-Means that use triangle inequality usually require extra information, such as distance bounds and cluster assignments, from the previous iteration to work efficiently. This is a challenging task to achieve on Hadoop for two reasons: 1) Hadoop does not directly support iterative algorithms; and 2) Hadoop does not allow information to be exchanged between two consecutive iterations. Hence, two techniques are proposed to give Hadoop the ability to pass information from an iteration to the next. The first technique uses a data structure referred to as an Extended Vector (EV), that appends the extra information to the original data vector. The second technique stores the extra information on files where each file is referred to as a Bounds File (BF).

To evaluate the two proposed techniques, two K-Means variants are implemented on Hadoop using the two techniques. Each variant is tested against variable number of clusters, dimensions, data points, and mappers. Furthermore, the performance of various implementations of K-Means on Hadoop and Spark is investigated. The results show a significant improvement on the efficiency of the new implementations compared to the Lloyd's K-Means on Hadoop with real and artificial datasets.

Acknowledgements

First, I would like to express my sincere gratitude to my supervisor Dr Giuseppe Di Fatta. His tremendous academic support, consistent encouragement and concise comments had a huge impact on this work.

I thank my fellow labmates, Hassan, Chris and the nicer Kris, Anas, Salwa, Manal, Mosab and Perusha for their endless support and for all the fun we have had in the last four years. A special thanks to Mark who has a solution for every problem and offers to help before I even ask.

My special regards to Professor Babak Forouraghi who inspired me during my Master's studies and his enthusiasm and integrity left a great impression on me.

My profound gratitude goes to the people who has a special place in my heart, my beloved parents, and my brothers and sisters. My father, Abdulrahman, and my mother, Badryah, your exceptional support and unconditional love have made me who I am today. My extended respect goes to my brothers and sisters, Ahmad, Bandar, Mohammed, Mohannad, Rashed, Bushra and Bashier for standing beside me at all times.

I am deeply grateful to my second half, soon to be, Dr. Sally for her eternal support and patience throughout this journey. Without her non of this would have been accomplished. My sincere thanks go to the joy of my life, Sarah, Fatimah and Yousuf for sacrificing too many weekends and trying to make sense of all of this. Without them, my life is tasteless.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Clustering Big Data

The last two decades witnessed an exponential growth of the data generated by many sources such as, scientific experiments, social media Web sites, government statistics, sensor networks, and many other sources. For example, the Large Hadron Collider project (LHC), which provides more knowledge about the universe by accelerating particles and examining the results from their collisions, is expected to produce around 50 petabytes of data in 2017, and the collected data could reach 10 gigabytes per second [3]. YouTube users eceeded 1 billion users, where 100 hours of videos are uploaded every minute, and 135,000 hours are watched [4]. eBay stores and preocess about 150 billion new records daily [5]. In order to cope with this rapid increase, novel solutions are developed to manage and process large-scale datasets known as *Big Data*.

The term *Big Data* had many definitions since the early 2000s. Most of these defintions summarise the characteristics of Big Data in what is known as the *multi V model*. For example, [6] and [7] define Big Data through the 3V model, where datasets have high *Volume*, generated in high *Velocity*, and with high *Variety* of data types that require new innovative technologies which could improve the decision making. Some works (e.g. [8]) extend the 3V model to a 4V model by including *Value*, which refers to the value gained by an organisation from extracting hidden data. While others (e.g. [9] and [10]) use a 5V model by adding *Veracity*, which refers to the degree of reliability, quality, and trustworthiness of the data source. In an attempt to clear the ambiguity and inconsistency in the various definitions of Big Data, the work in [11] surveys the definition of Big Data in various works and concluded that all definitions mention at least one of the following characteristics: the size of the data, the complexity (e.g. structured, semi-structured, unstructured), and the technologies designed to process such data. The authors introduced their own definition, which states that: *"Big data is a term describing the storage and analysis of large and or complex data sets using a series of techniques including, but not limited to: NoSQL, MapReduce and Machine Learning"*.

Collecting, storing, and managing the data is a crucial process. However, the data itself is worthless unless meaningful knowledge can be extracted from it. For this reason, various innovative techniques were developed over the years dedicated to knowledge discovery. One of the essential approaches to unveil hidden patterns in a given set of observations is to divide these observations into a number of groups (clusters), such that observations in one group have more similarities than observations in other groups. This process is known as clustering or cluster analysis. Clustering algorithms are developed and used in many fields, such as engineering, computer science, life and medical sciences, astronomy and earth sciences, social sciences and economics [12]. Most clustering algorithms, however, are computationally expensive or iterative in nature (e.g. K-Means). This makes the clustering task very challenging, especially when dealing with large and high-dimensional datasets. For this reason, the focus has been shifted lately to parallel clustering solutions on distributed processing models to overcome these challenges. One of the most popular and attractive distributed processing models is known as MapReduce.

Despite the existence of various clustering algorithms, the focus of this project is on the K-Means clustering algorithm for the following reasons. The popularity and the importance of K-Means where it is used as a clustering solution in many fields and for many applications. Furthermore, the target of this project is to cluster Big Data where the volume of the data is very large that only algorithms with sub-linear, linearthmic, and linear time complexity can be used [13]. While most clustering algorithms (e.g. hierarchical clustering algorithm) run in a quadratic time, K-Means is linear in the number of data points, clusters, and dimensions. The linear time complexity makes K-Means more suitable for Big Data. Therefore, making K-Means even faster and highly scalable shall contribute in the development of a powerful clustering algorithm that can cope with the recent exponential growth of data.

1.2 Iterative Clustering Algorithms on MapReduce/Hadoop

The distributed computing framework MapReduce [14], and its open-source implementation known as Hadoop [15], became the de-facto standard for Big Data computing [16]. The popularity of MapReduce comes from its ability to provide a reliable and fault-tolerant parallel programming paradigm without having to deal with the underlying details of the distributed system, such as data distribution and tasks scheduling. However, this programming model faces several limitations when it handles iterative Machine Learning algorithms because the iterative process is not directly supported. For instance, the framework is not loop-aware. In each iteration, map and reduce tasks are not reusable where new tasks must be started and destroyed. Moreover, static data must be loaded repeatedly in each iteration, which wastes network bandwidth and consumes more CPU resources.

Ranked as one of the top ten data mining algorithms [17], K-Means takes the number of clusters as an input and iterates over the input data points until it converges. In each iteration, the standard implementation of K-Means, known as Lloyd's K-Means [18], must compute the distance from each data point to all cluster centroids. This process is considered as a bottleneck in K-Means. Most of these distance calculations, however, are redundant and can be avoided.

One of the most effective methods to reduce the number of distance calculations in K-Means is to apply geometric approaches based on triangle inequality. However, most of these approaches must keep extra information (e.g. distance bounds and cluster assignments) in one iteration and use it in the next. Many works (e.g. [19]) [20] [21], shown how triangle inequality cab be very effective on accelerating the running time of K-Means. Implementing such optimisations on Hadoop should lead to a powerful clustering algorithm that can cluster vast amounts of data in a fast time. However, Hadoop does not offer a mechanism to cache information between MapReduce jobs (each iteration is represented by a MapReduce job) which makes the implementation of such optimisations on Hadoop a complex task. Hence, this work provides solutions that allow Hadoop to pass information from one iteration to the next in order to be able to implement K-Means variants based on triangle inequality. Two methods are introduced to pass information from one iteration to the next in Hadoop: For each input data point, the first method appends the required extra information to the data point vector and creates what we refer to as an *Extended Vector*. Each Extended Vector is read in the iterations to follow along with its appended extra information. The second method writes only the required extra information into files referred to as *Bounds Files*. Bounds Files are stored on the distributed file system in one iteration and read in the following iteration.

To evaluate the effectiveness of the proposed methods, two optimised algorithms are implemented and tested on Hadoop using each method; in addition to an algorithm that employs a basic triangle inequality approach to skip distance computations. The performance of the newly implemented algorithms is compared against the performance of Lloyd's K-Means on Hadoop. This is because Lloyd's algorithm is the straight-forward approach to find the clustering solution where the distance from all data points to all cluster centres is computed. Moreover, the performance of Lloyd's K-Means and the basic triangle inequality K-Means based on a popular distributed framework known as Spark [22] is investigated. The experiments are executed for a predefined number of iterations and not to convergence. This is because the impact of triangle inequality on avoiding distance computations can be observed in the first few iterations. Furthermore, triangle inequality optimisations are guaranteed to converge in the same number of iterations as Lloyd's algorithm while maintaining the exact same output.

1.3 Why MapReduce?

This work implements K-Means optimisations on Hadoop, which adopts the standard MapReduce programming model. Despite the existence of alternative distributed computing systems that are based on MapReduce and support iterative algorithms, these systems usually enhance the iterative process at the expense of other essential components such as fault-tolerance. Twister [23], for example, provides a distributed framework that can efficiently process iterative algorithms by caching the input data in the memory of the worker machines in the first iteration and reuse it in following iterations. However, one of its major limitations is its limited capabilities of handling task failures compared to Hadoop. Furthermore, the stability of the framework was taken into consideration. Hadoop and Spark are adopted by many industrial companies to manage and process their Big Data. In addition the continues contributions from the academic field, Hadoop and Spark are constantly updated which maintain their stability. On the other hand, some iterative MapReduce frameworks (e.g. HaLoop, iMapReduce) were reported to be unstable [24]. A quick look at the release dates of the latest versions for different distributed platforms gives an insight on the scale of support each one has. For example, the latest versions for Hadoop and Spark were released on 2018, while the latest ones for Twister, iMapReduce and HaLoop were on 2011. The focus in this research is on providing parallel clustering solutions on reliable, stable and widely supported distributed platforms, and that is why Hadoop and Spark were chosen.

1.4 Objectives

The following is the list of objectives that are designed to achieve the main aim of this project which is to improve the efficiency and scalability of Lloyd's K-Means on Hadoop:

- Design and implement parallel solutions for K-Means on Hadoop that are more efficient and deterministically produce the same clustering results of Lloyd's algorithm.
- Adopt optimisation techniques based on triangle inequality.
- Provide a mechanism to carry extra information from one iteration to the next on Hadoop.
- Measure the ability of the new solutions to improve the efficiency of Lloyd's K-Means on Hadoop and their ability to scale with an increased number of clusters, dimensions and data points.

Adopt optimisation techniques based on triangle inequality. Provide a mechanism to carry extra information from one iteration to the next on Hadoop. Measure the ability of the new solutions to improve the efficiency of Lloyd's K-Means on Hadoop and their ability to scale with an increased number of clusters, dimensions and data points.

1.5 Contributions

The standard Lloyd's K-Means can achieve significant speedups by using triangle inequality optimisations that reduce the computation time while maintaining the exact output as the standard K-Means. Such optimisations have not been explored on the standard MapReduce distributed framework and its popular implementation known as Hadoop. This is because Hadoop does not support iterative algorithms and does not maintain data between iterations, which is a requirement in K-Means optimisations based on triangle inequality. Although other distributed platforms (e.g. Twister and HaLoop) overcome these limitations in Hadoop by caching the data between iterations, these solutions usually come at the cost of other crucial properties such as fault-tolerance.

The challenge in this research is to be able to achieve considerable improvements on the efficiency of K-Means using triangle inequality optimisations on MapReduce. In Hadoop, the lack of support for iterative algorithms and of providing a mechanism that passes information between iterations is a limiting and challenging factor. The main contribution of this work is that we were able to show that it is possible to speedup the standard K-Means on Hadoop and make it highly scalable while maintaining the exact final results as the original Lloyd's algorithm.

The following contributions are part of the main contribution of this research:

1. The design and the development of two techniques: K-Means on Hadoop using an Extended Vector (EV), and K-Means on Hadoop using a Bounds File (BF). These techniques give Hadoop the ability to pass information from one iteration to the next on iterative algorithms.

2. Parallel implementations of K-Means variants on Hadoop using EVs and BFs to evaluate the effectiveness of the new approaches. The implemented variants accelerate K-Means by using triangle inequality to reduce the number of distance computations. To achieve this purpose, these variants require

extra information, such as cluster assignments, from the previous iteration.

3. An extensive experimental analysis that tests the scalability and efficiency of implementations of K-Means on Hadoop using BFs and EVs with respect to the number of clusters, dimensions, data points, and mappers.

4. An investigation on the impact of the I/O overhead that is created by each proposed approach. Each approach needs to write extra information to Hadoop Distributed File System (HDFS) to use these information in subsequent iterations. This operation could impact the performance and eventually becomes the dominant cost.

5. A comparative analysis of implementations of K-Means on Hadoop using BFs and two K-Means implementations on Apache Spark. The in-memory caching mechanism provided by Apache Spark makes it one of the best distributed computing frameworks designed to process iterative algorithms.

1.6 Thesis Outline

The reminder of this thesis is organised as follows:

Chapter 2 briefly overviews the sequential implementations of K-Means including Lloyd's algorithm and other K-Means optimisations. In addition, the use of triangle inequality to improve the efficiency of K-Means is explained. Finally, an overview of the MapReduce framework, Apache Hadoop, and Apache Spark is presented.

Chapter 3 discusses the related work to parallel implementations of K-Means in general, and to parallel implementations of K-Means on distributed systems, in particular. The chapter also describes the implementations of Lloyd's K-Means on Apache Hadoop and Apache Spark.

Chapter 4 introduces two techniques to pass information from one iteration to the next in parallel K-Means on Hadoop. Furthermore, a detailed description of the implementation of two K-Means variants using each proposed approach is discussed.

The evaluation of the proposed solutions to improve K-Means efficiency and scalability is discussed in Chapter 5 where each algorithm is tested against various parameters and an extensive comparative analysis of all algorithms.

Finally, Chapter 6 concludes this thesis and discusses the improvements this work could benefit from in the future.

Chapter 2

Background

This chapter presents background concepts related to sequential implementations of K-Means and distributed computing frameworks. First, an introduction to cluster analysis and the naive K-Means implementation is presented. Next, an overview of the work related to the sequential variants of K-Means that aim to enhance the clustering quality and the efficiency of K-Means is introduced. Furthermore, optimisations of K-Means based on triangle inequality are discussed. The chapter also explains the workflow and the limitations of the distributed computing framework MapReduce and its open-source implementation Hadoop. Several alternative distributed frameworks are explained at the end of the chapter.

2.1 Cluster Analysis

Clustering [25] [26] [27] is the process of partitioning data points in a given dataset into groups (clusters), where data points in one cluster are more similar than data points in other clusters. Similarity of data points is determined by a similarity or distance measure such as Euclidean distance, which is explained in section 2.1.2.

Clustering is an important technique that has been applied in many areas such as, data mining, pattern classification, Web applications, and text mining [28] [29]. Clustering has been used in many applications such as, analysing gene expression data [30], image segmentation [31] to locate objects' borders in an image, and in market segmentation [32] where markets are broken down into meaningful segments, such as segmenting buyers habits based on age groups.

In many clustering problems, only little prior information about the data is known which makes it difficult to find a meaningful relationship between patterns. For this reason, few assumptions have to be made about the data by the decision maker.

2.1.1 Clustering Techniques

Clustering techniques can be divided into two general approaches [25]:

1. **Hierarchical Clustering:** Methods belong to this technique represent the data hierarchically through a *dendogram*. This hierarchical representation is created in an agglomerative (bottom-up) or a divisive (top-down) fashion. Algorithms that follow the agglomerative approach start with a single data point as a cluster, then recursively merge pairs of clusters together based on their similarities until the algorithm meets a conversion criterion. On the contrary, algorithms that follow the divisive approach start with one cluster that contains all data points. Clusters are then split up in each iteration until each data point is in a single cluster.

   The most popular examples of agglomerative algorithms are single-linkage, all-pairs or complete linkage, centroid-linkage, and sampled-linkage clustering. The single-linkage method uses the shortest distance between any pair of points in two clusters. While the average over all pairs is used in the all-pairs linkage method, the sampled linkage method uses a sample of the data points in the two clusters for calculating the average distance. In centroid-linkage method the distance between the centroids is used. Such clustering techniques require the construction of a similarity matrix of size n x n, where n is the number of data points. This means that the time and memory space complexity for these algorithms is at least quadratic. In the Big Data problem, the number of data points is usually very large. Algorithms with a quadratic time and memory space complexity are not suitable for these targeted problems.

Many hierarchical algorithms were proposed to deal with large-scale datasets. For example, The algorithm Balanced Iterative Reduction and Clustering using Hierarchies (BIRCH) [33] is a well-known clustering algorithm because it is one of the earliest algorithms to scan the data and use data squashing techniques to reduce the I/O cost when the input dataset gets larger than the size of memory. BIRCH creates a height-balanced tree of nodes that summarises the data. Each node in the tree is called a Cluster Feature (CF). Hence, the tree is known as CF-tree. There are two main steps in the BIRCH algorithm. BIRCH scans the input data points and builds a CF-tree by inserting the data points where size of the tree is controlled by a certain threshold T. The default threshold T = 0. The size of the tree, however, could get very large such that it does not fit in memory. In addition to the need to set T, two other parameters (branching factor, maximum of points per leaf) must be set to control the tree structure.

The algorithm known as CURE [34] (Clustering Using REpresentatives) discovers clusters with irregular shapes by using multiple representative data points for each cluster. CURE adopts sampling to speedup the clustering process. However, the time complexity for CURE is quadratic even in two-dimensional space.

2. **Partitional Clustering:** Algorithms that follow the partitional clustering technique divide the input data points into a number of partitions (clusters). Data points are then iteratively reallocated from one cluster to the another based on a certain criterion, such as the *sum of squared errors SSE*, see

section 2.2.4. In doing so they try to discover clusters either by iteratively relocating points between subsets, or by identifying areas heavily populated with data. CLARANS [35] and the mean shift algorithm [36] are examples of partitional clustering algorithms.

Clustering Large Applications based upon RANdomized Search (CLARANS) [35] extends the clustering algorithm called K-Medoid, which cluster data points around medoids instead of cluster centroids as in K-Means. CLARANS uses randomized search to overcome the exponential search space in K-medoid. CLARANS, however, has a quadratic time complexity.

The mean shift algorithm [36] is a simple partitional procedure that iterates over data points and shifts each data point to the average of data points in its neighborhood. Despite the simplicity of the algorithm, it has a quadratic time complexity which make it unsuitable to process large datasets.

Other examples of well-known clustering algorithms is the Density-Based Spatial Clustering of Applications with Noise algorithm (DBSCAN)[37]. DBSCAN is density-based algorithm that aims to discover arbitrarily shaped clusters. DBSCAN can efficiently cluster data points that are packed (where density is high) and mark out outliers. Despite the popularity of DBSCAN, its worst case time complexity $O(n^2)$ where n is the number of data points. It is possible to accelerate the algorithm by constructing a distance matrix between data points. This matrix, however, requires $O(n^2)$ memory space, which is worse than the one required by the original DBSACN ($O(n)$).

The focus in this research is on the poplar partitional algorithm known as K-Means algorithm. Although K-Means has several limitations in terms of its sensitivity to the initial centres and outliers, it is one of the most adopted clustering algorithms in real-life mainly due to its simplicity and efficiency. The next sections describe K-Means in details and review several variants of the algorithm.

2.1.2 Distance Measures

In order to determine if two objects in a dataset are similar, a similarity or distance measure is needed. The distance measure needs to be carefully chosen based on the type of data to be processed [25]. There is a variety of distance measures such as, Euclidian distance or $L_2 - norm$, Manhattan distance or $L_1 - norm$, Jaccard index, and Cosine distance, used for different types of data. However, the focus will be on the Euclidian distance because this work mostly deals with numerical data and Euclidian distance is the most popular distance measure for this type of data [25]. Let $x = (x_1, x_2, \ldots, x_d)$, and $y = (y_1, y_2, \ldots, y_d)$, are two data points in d-dimensional space $\mathbb{R}^d$, the Euclidean distance between x and y is defined as:

$$d(x, y) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2} \tag{2.1}$$

where $d(.,.)$ is the distance between any two points, and i is the i-th attribute of points x and y.

2.2 K-Means

2.2.1 Introduction

Ranked as one of the top ten most influential data mining algorithms [17], K-Means is a well-known clustering algorithm that partitions data into clusters of similar features. Simplicity, efficiency, and straight-forward implementation made K-Means one the most used algorithms in cluster analysis [38]. K-Means was proposed independently in different works [39] [18] [40] [41] targeting different problems.

K-Means is used in many fields to cluster various types of data. Some of the applications that K-Means has been applied to are:

- Colour quantisation where the pixels of an image grouped into clusters [42]

[43] [44].

- Market segmentation [45], where markets are broken down into meaningful segments, such as segmenting buyers habits based on age groups.
- Analysis of gene expression data [46] [47].
- Documents clustering [48] [49], where similar documents are grouped into one cluster while other documents are assigned to other clusters.

2.2.2 Lloyd's Algorithm

The basic K-Means algorithm was independently proposed by Steinhaus [39], Lloyd [18], Ball and Hall [40], and Macqueen [41]. The focus of this reserach is on Lloyd's algorithm which is the most commonly used version of K-Means [50] [21]. Lloyd's algorithm is referred to in the remainder of this thesis as *Naive K-Means*.

As shown in Algorithm 1, given a set X of n data points $X = \{x_1, x_2, ..., x_n\}$ in a d-dimensional space $\mathbb{R}^d$, and an integer k that represents the number of clusters, where $k \ll n$. The initial set of centroids $C = \{c_1, c_2, ..., c_k\}$ is randomly chosen from X. Let i be the i-th point in dataset X, and j be the j-th in the set of centroid C. The inner loop from line 4 to 7 uses a distance measure, in this work Euclidean distance in equation 2.1, to find the minimum distance between each data point x_i and each centroid c_j. Then x_i is assigned to its nearest centroid c_a in line 7. After each point gets assigned to its closest centroid, the mean of the points assigned to each centroid is calculated and this mean represents the location of the new centroid. This process is repeated for all points (the outer loop from line 3 to 8) until convergence, i.e. centroids stop moving, or an early termination condition is met. Figure 2.1 illustrates the clustering stages for a two dimensional dataset into three clusters using K-Means.

While the majority of the Naive K-Means running time is spent on computing the distances in the inner loop (lines 4-6) to find the closest centroid, most of these

Algorithm 1: Sequential Naive K-Means(X, k)

```
1  select $k$ initial cluster centroids randomly from $X$
2  while not converged or an early termination condition is not met do
3      for $i = 1$ to $n$ do
4          for $j = 1$ to $k$ do
5              compute $d(x_i, c_j)$
6              find $c_j$ with minimum distance from $x_i$
7          end
8          assign $x_i$ to its closest $c_a$
9      end
10     for $j = 1$ to $k$ do
11         compute the mean of all $x_i$ assigned to $c_a$
12         move each $c_a$ to its updated mean
13     end
14 end
```

computations are unnecessary and can be avoided. The focus of this work is to investigate and compare the behaviour of different variants of K-Means that attempt to reduce the number of distance computations and introduce implementations of these variants on Hadoop.

2.2.3 Complexity

Naive K-Means computes the distance from n data points, where each point is in d dimensions, to k number of cluster centroids in order to assign each data point to its closest cluster centroid. The algorithm needs to iterate for e number of iterations until it finds the final clustering solution. Hence, the time complexity for Naive K-Means is *O(nkde)*.

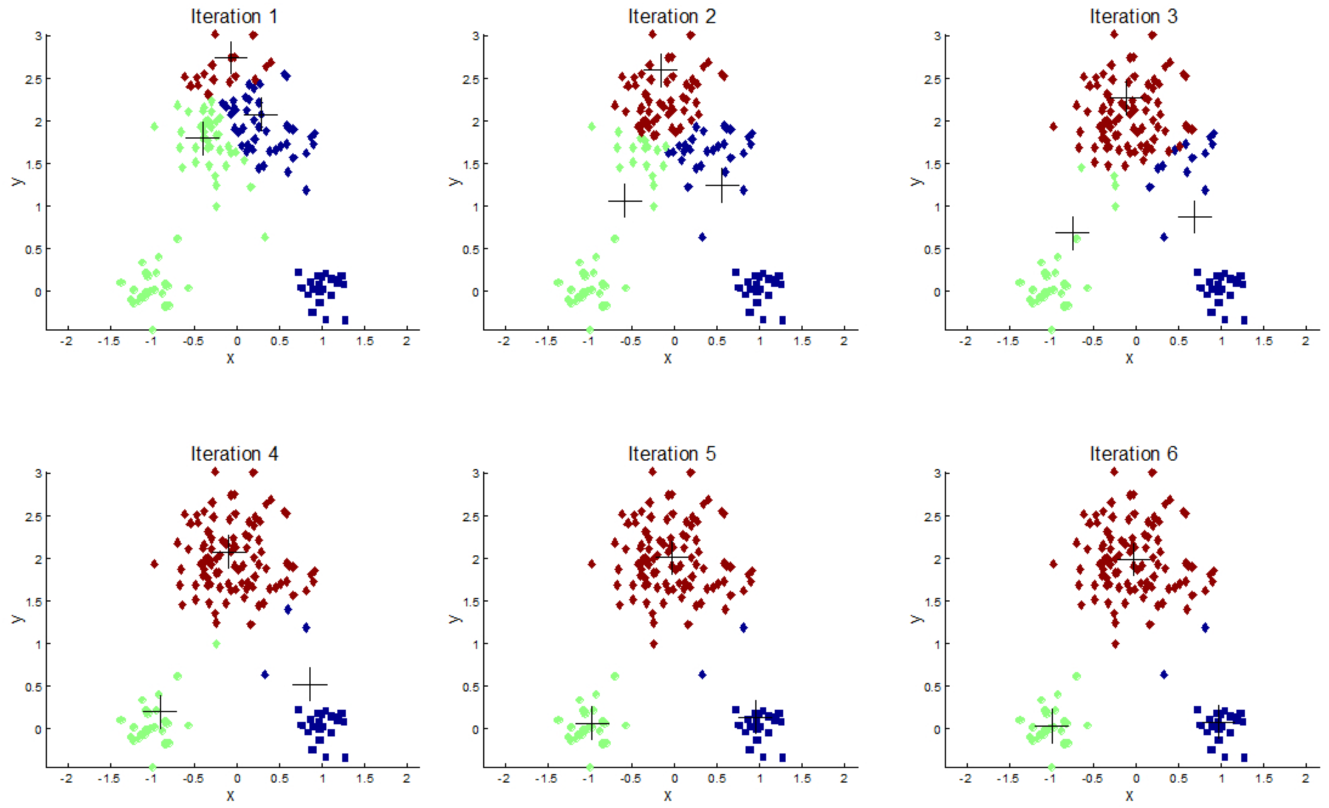


Figure 2.1: Illustration of clustering two dimensional input dataset into three clusters using K-Means algorithm. In iteration 1, the initial seeds are selected. Next, iterations (2-5) the centres are updated and moved in each iteration to their new locations. Iteration 6 shows the final clustering results.

2.2.4 Convergence

The goal of K-Means is to minimise a given criterion function. The most popular criterion function used in partitional clustering in general and K-Means in particular is the *sum of square errors (SSE)* [25]. Given a dataset $X = \{x_1, x_2, ..., x_n\}$ in a d dimensional space, where x is a data point in X and n is the number of data points, and a set of k cluster centroids $C = \{c_1, c_2, ..., c_k\}$, SSE is defined as:

$$SSE = \sum_{j=1}^{k} \sum_{i=1}^{n} \parallel x_i - c_j \parallel^2$$

where i is the i-th data point belongs to the j-th cluster with mean c_j. The function simply squares the distance from each data point to its centroid and then sums up all the squared distances for all clusters. The mean (average) of cluster c_j is:

$$c_j = \frac{1}{|c_j|} \sum_{x_i \in c_j} x_i$$

where $|c_j|$ is the number of points in cluster c_j, and i is the i-th point belongs to cluster c_j.

K-Means achieves full convergence when the SSE does not change, which means data points in each cluster do not change assignments any more. Furthermore, K-Means can be terminated when it meets one or more of the following conditions [20]:

- The algorithm reaches a predefined number of iterations,
- The total value of criterion function, SSE, is below a certain threshold.

2.2.5 Limitations

Despite its linear compleixty in n, d, and k, which makes it faster than most clustering algorithms [50], K-Means has some notable drawbacks that many works has tackled since its emergence about fifty years ago. K-Means drawbacks [51] can be summarized as:

- The number of clusters must be given as an input, which is a hard task to determine the best number of clusters in a given dataset.
- The quality of the final results relies heavily on the initial choice of centroids.
- The algorithm converges to local minimum.
- It is sensitive to outliers.
- It does not scale well with large datasets because all data points must be loaded to memory in each iteration.

This project aims to enhance the last drawback by proposing solutions to improve the K-Means efficiency and scalability while maintaining the clustering quality. The next section reviews some of the proposed approaches to solve these drawbacks.

2.3 Sequential K-Means Optimisations

One of the advantages of K-Means is that it can be optimised almost in every aspect [50]. These aspects include, enhancing centroids initialisation, determining the number of clusters k, and accelerating distance computations in each iteration. This section overviews some variants of K-Means that firstly, improve the method of the centroids' initialisation step. Secondly, determine the number of clusters k. Finally, accelerate the distance computation per iteration, which is the main focus of this work.

Finding the best initial centres

The choice of initial centroids affects the quality and efficiency of K-Means [52]. A good set of initial centroids produces more accurate clusters, and can lead to a reduction in the number of iterations which, as a result, improves the overall running time. This aspect of K-Means has been extensively investigated and many approaches have been proposed to improve it.

MacQueen [41] proposed two approaches to select the initial centroids. The first approach selects the first k points in dataset X as initial centroid, which is sensitive to the order of the data. The second approach selects k number of centroids randomly from dataset X, which is much like Lloyd's algorithm. To determine which set of initial centroids is the best, the algorithm is run many times, each run is with a different initial sets of centroids and the run with the lowest SSE is chosen. The difference between MacQueen's and Lloyd's algorithms is in the way new centroids are updated. While Lloyd's algorithm updates the cluster centroid once per pass, MacQueen's algorithm updates the centroid each time a point is assigned or removed from the cluster. This process makes MacQueen's algorithm more like an online algorithm, while Lloyd's algorithm is more like a batch algorithm.

In [53], the authors presented the furthest-first algorithm to provide an approximate solution to the k-center problem. The algorithm was used as an initialisation method to K-Means, where a point is randomly selected as the first centre. The next centre is selected as the furthest point from the current chosen centre. This process is repeated until k centres are collected. The main drawback of this method is that it is susceptible to choose outliers as the initial centres.

Arthur and Vassilvitskii [54] proposed K-Means++, which carefully selects the initial set of centroids. Consider $D(x)$ the shortest distance from a data point $x \in X$, where X is the input dataset, to the nearest cluster centroid that was already chosen. K-Means++ proceeds as follows:

1. pick one centroid c randomly from dataset X.

2. Select the next centroid c_j with probability $\frac{D(x)^2}{\sum_{x \in X} D(x)^2}$, where x is a data point that belongs to dataset X and *D(x)* is the minimum squared distance from x to its nearest centroid.

3. Repeat step 1 until k number of centroids is collected.

4. Run Naive K-Means on the collected set of centroids.

Bradley and Fayyad [55] proposed a refined K-Means algorithm that proceeds by partitioning the original dataset randomly into P subsets. Then each subset is clustered using K-Means and produces P sets of solutions (centroids) each with K points. Those sets are then combined together into one set and clustered by K-Means P times. Each run of K-Means is initialised by a different set of centroids. The solution with the lowest SSE is picked as the final set of centroids.

Finding the best *k*

The number of clusters (k) is an essential factor that affects the quality and efficiency of K-Means. The classic approach for picking the number of k is to run K-Means a number of times, each time with a different number of k and choose the number of k that produces the least SSE. Elkan and Hamerly [56] introduced a variant of K-Means called *G-Means* that automatically determines k in K-Means. G-Means proceeds with a small number of k. Then, in each iteration, clusters with data points that do not fit a Gaussian distribution are split up. The current solution is refined by applying K-Means between each round of the splitting.

Another variant called *X-Means* [57] was developed to improve the efficiency of K-Means and to provide a good estimation of the number of clusters in an input dataset. After each run of K-Means, X-Means makes local decisions about which subset of the current centroids should split themselves in order to better fit the data. The splitting decision between the children of each centre and itself is done by computing the Bayesian Information Criterion (BIC). BIC is a statistical method that selects models among a finite set of models where the model with the lowest BIC is selected [58]. The authors developed X-Means based on a prior work [59] that accelerates K-Means by storing the input data points on a binary search tree known as kd-tree. In kd-tree, the points are stored at the nodes of the tree along with sufficient statistics (the number of points and their vector sum). In the same work, a technique the authors call blacklisting, which is an additional geometric computation, is used to quickly estimate the number of clusters required for a dataset. The technique keeps a list of only those centroids that need to be considered for a given region. X-Means can be a potential candidate to overcome the problem of choosing the best number of k in K-Means. However, X-Means was not considered to be parallelised because it involves the construction of a kd-tree at the beginning of each iteration which require $nd+nlog(n)$ time, considering n as the number of data points in d dimensions. Another reason is that it was reported in [59] and [43] that kd-trees are not efficient with high demensional datasets (i.e $d > 20$) which is not the case when dealing with large-scale datests. Furthermore, kd-tree approachers depend heavily on the structure of the data where the tree could suffer from load imbalance when implemented on a parallel environment [60]. For these reasons X-Means is not a suitable clustering solution when the target is to cluster large-scale datasets.

Accelerating the iteration time

Assigning data points to their closest cluster centroids requires calculating the distance from each data point to each centroid. While this operation consumes the majority of the K-Means running time, most of these distance computations are redundant and can be avoided because after the few first iterations the centroids move slightly and most points do not change assignment [19]. Therefore, the aspect of reducing the number of distance calculations per iteration to gain more speed has been extensively studied and various approaches have been proposed regarding this matter.

One approach is based on using *tree data structures* to store data points where nearest data points to some query point (e.g. centroid) are in the same sub-tree. kd-trees [61] were proposed independently by Pellge and Moore [59], Alsabti *et al.* [62], and Kanungo *et al.* [43], to accelerate K-Means by reducing the number of distance calculations from each data point to each centroid.

A kd-tree is a binary search tree that is used for storing data points in a multi-dimensional space [61]. kd-trees have been generally used to accelerate nearest-neighbour queries. Therefore, they can be applied to K-Means to find the nearest cluster centroid for a given data point and reduce the expensive computational cost of distance calculations per iteration. However, [59] and [43] have reported that kd-trees performance degrades as the number of dimensions gets larger ($d > 8$ in [59], and $d > 20$ in [43]). Furthermore, Pettinger and Di Fatta [60] presented a parallel implementation of kd-trees and reported that kd-trees may suffer from load imbalance when it is implemented in a parallel fashion depending on the structure of the input data. Moore [63] introduced the anchor hierarchy (tree-like) data structure to accelerate the clustering process in high-dimensional datasets.

Optimisations based on tree-structured approaches can be effective in speeding up K-Means running time. However, Elkan [19] showed that using triangle inequality can be more effective at pruning large numbers of redundant distance computations, which is the focus of this research and will be discussed in the following sections.

2.3.1 K-Means and Triangle Inequality

Despite the simplicity of triangle inequality, it is considered as a powerful geometric tool that can be useful in reducing the number of distance computations in

K-Means. First, the basic triangle inequality principle is explained, then an explanation of how can it be applied to K-Means is presented, and finally, an overview of popular optimised algorithms is provided. For a given three points x, y, $z \in \mathbb{R}^d$, and $d(.,.)$ is the distance between two points, the triangle inequality states that:

$$d(x,z) \leq d(x,y) + d(y,z) \tag{2.2}$$

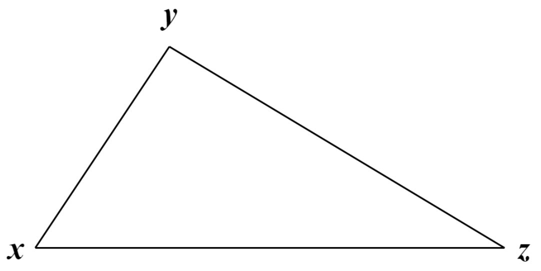


Figure 2.2: Triangle Inequality Property

In other words, triangle inequality means that the sum of the lengths of two sides of a triangle must be larger then the length of the remaining side. The next triangle inequality properties can be inferred from $\triangle xyz$ in Figure 2.2:

$$d(x,y) + d(y,z) > d(x,z)$$
$$d(x,y) + d(x,z) > d(y,z)$$
$$d(x,z) + d(y,z) > d(x,y)$$

The next section shows the benefits of using triangle inequality and how these properties can be applied to K-Means in order to speedup the algorithm's running-time.

2.3.1.1 How Can Triangle Inequality Accelerate K-Means?

The most expensive operation in K-Means is computing the distance from each data point to all centres to find the centre with the minimum distance. One of the most important remarks in K-Means is that after a few number of iterations, most data points do not change their cluster assignment, especially with well-clustered datasets. The reason behind this is that after a few number of iterations the movement of cluster centroids is insignificant[19] [21]. Thus, most of the distance calculations from points to centroids are redundant, and this is where triangle inequality excels.

Generally, the main goal of using triangle inequality with K-means is to prove that a given point in the input dataset is closer to a certain centroid without the need to calculate the distance to other centroids. Given three points $x, a, b \in \mathbb{R}^d$ where x is the input data point, and a and b are two centroids. Triangle inequality aims to prove that $d(x, b) \geq d(x, a)$ without computing the exact distance from x to b. However, achieving this goal usually comes with extra computational and space costs. These costs varies from one algorithm to another depending on the requiments of the algorithm. In some algorithms like Elkan's [19] and Hamerly's, the computational cost comes from the need to compute k^2 (k is the number cluster centroids) centre-centre distances at the beginning of each iteration which also requires k^2 memory space. Moreover, some algorithms need to cache distance bounds from points to centres in one iteration for them to be used in following iterations.

Triangle inequality was used in different ways to prune distance calculations. For a point x and two cluster centroids a and b, the following are some of the cases that triangle inequality can be applied to K-Means [19] [21]:

1. Show that x is closer to a than b, with calculating only $d(x, a)$ and $d(a, b)$.

2. Form an upper-bound from x to its closest centroid.

3. Form a lower-bound from x to one or more centroid.

The following Lemma helps finding the closest centroid from a given point by using pre-calculated centre-centre distances and the distance from the point to its previously assigned centroid.

Lemma 2.3.1. *[19] Let x be a point, and p and q be two centroids,*

$$\textit{if } d(p,q) \geq 2d(x,p) \textit{ then } d(x,q) \geq d(x,p)$$

Proof. From 2.2, it is known that

$$d(p,q) \leq d(x,p) + d(x,q)$$
$$d(p,q) - d(x,q) \leq d(x,p).$$

The left hand side can be written as:

$$d(p,q) - d(x,q) \geq 2d(x,p) - d(x,p) = d(x,p).$$

Hence:

$$d(x,p) \leq d(x,q).$$

□

The usage of triangle inequality to avoid unnecessary distance calculations was investigated by many researchers. The usage of Lemma 2.3.1 was proposed by Hodgson in [64]. Hodgson's approach compared a given centroid c with only its closest centroid c', that is, if $d(x,c) < d(c,c')$ then the distance calculation to only c' is avoided.

In [65], triangle inequality was used to improve the search of the nearest-neighbor. For a given point x and a candidate nearest-neighbor y, the author showed that another point z cannot be closer to x if Lemma 2.3.1 holds. The same approach was applied to K-Means in [66] by Phillips on an algorithm he called Compare-means.

Judd *et al.* [20] noticed that in a square-error clustering algorithm they call CLUSTER (similar to K-Means), a change in cluster assignments occurs mostly (70–80%) at the first two iterations. Therefore, most distance calculations are redundant and can be avoided. In an attempt to reduce the number of distance calculations the authors proposed an approach called *computing spheres of guaranteed assignment for cluster centroids.* This approach computes and caches the

Notation	Description
X	The input dataset of size n
x_i	A data point, where $x_i \in X$, with $1 \leq i \leq n$
k	Number of clusters
C	The set of cluster centroids of size k
c_j	Cluster centroid, where $c_j \in C$, with $1 \leq j \leq k$
c'_j	New location for centroid c_j
c_a	Closest centroid to data point x, where $1 \leq a \leq k$
h_j	Half minimum distance from c_j to its closest centroid
u_i	An upper-bound from data point x_i to its closest centroid c_a
$l_{i,j}$	A lower-bound from data point x_i to centroid c_j

Table 2.1: Description of notations used in implementations of sequential algorithms based on triangle inequality

centre-centre distances along with the cluster assignments for all points from the previous iteration. Given a point x and two centroids c and c', where c is the closest centroid to x in the past iteration and c' is the closest centroid to c, from Lemma 2.3.1, if $d(x, c) < \frac{1}{2}d(c, c')$, then $d(x, c') \geq d(x, c)$. This means c is closer to x than c' and distance computations to all other centroids can be skipped.

To fully understand the different approaches that triangle inequality was used as an optimisation technique to accelerate K-Means, the following sections explain in detail some of the popular algorithms that apply triangle inequality. The reasons behind choosing these particular algorithms is because triangle inequality approaches almost always come with an extra cost (e.g. memory overhead, centre-centre distance calculations), and these algorithms cover most of these extra costs ranging from high-cost (e.g. Elkan's algorithm) to low-cost (e.g. Compare-means).

2.3.1.2 Clustering Quality

Given the same input dataset and the same initial centroids, multiple runs of the standard K-Means algorithm (Lloyd's K-Means in this work) always produce the same final clustering results. Most of the studies that aim to improve the efficiency of Lloyd's K-Means try to accelerate the algorithm without affecting the final clustering results. Algorithms with this nature are aslo known as *exact algorithms* [21]. In terms of K-Means variants based on triangle inequality, most of these variants are exact algorithm where the quality of the final clustering output is equivalent to the output in Lloyd's K-Means.

Most data points do not change cluster assignments after the first few iterations [20] [19]. As a result, many distance computations from points to cluster centroids are redundant. Triangle inequality avoids computing these redundant distances. Theoretically, this optimisation does not affect the final clustering outcome. In practice, to ensure that the quality of the clustering results of the optimised algorithms is intact, the cluster centroids produced by an optimised algorithm in each iteration are compared with the cluster centroids produced by Lloyd's K-Means for the same iteration. If the two sets of centroids match, this indicates that the results are intact and the optimised algorithm is an exact algorithm. More details on how the clustering quality is investigated in this work are explained in later in section 5.3 .

2.3.1.3 Compare-means and Sort-means

In [66], Phillips presented two approaches, Compare-means and Sort-means, to accelerate K-Means using triangle inequality in order to avoid unnecessary distance computations. Given a point $x \in X$, and two centroids c and $c' \in C$, it is known from the inequalities in (2.2) that $d(c, c') \leq d(x, c) + d(x, c')$, hence: $d(x, c') \geq d(c, c') - d(x, c)$. If it is already known from Lemma 2.3.1 that $d(c, c') \geq 2d(x, c)$, it can be concluded that $d(x, c') \geq d(x, c)$, which means that centroid c is closer to x than centroid c' without the need to compute the exact distance from x to c'.

Compare-means starts by computing k^2 matrix of centre-centre distances at the beginning of each iteration each time the centres move and keeps the cluster assignment a_i from the previous iteration for each point x. Next, the test $d(c_j, c_{a_i}) \geq 2d(x_i, c_{a_i})$ is performed before computing $d(x_i, c_j)$, if the test holds, $d(x_i, c_j)$ calculation is skipped.

Sort-means uses the inequality in Lemma 2.3.1 as well, but after computing k^2 matrix of centre-centre distances, a new $k \times k$ matrix is constructed where row j holds the sorted indices of k number of centres in increasing order of their distances from centre c_j. Then, before $d(x_i, c_j)$ is calculated, the algorithm compares the distance from the previously assigned centroid to the currently processed point with the centroid distances sorted in increasing order. If the distance of the compared centroid is larger or equals to $2d(x_i, c_{a_i})$, the algorithm prunes all the remaining centroids and proceeds to the next point.

2.3.1.4 Elkan's Algorithm

Elkan [19] introduced an optimised version of K-Means that efficiently prunes unnecessary distance calculations by applying triangle inequality along with a set of updated upper and lower bounds. Elkan's algorithm caches extra information in one iteration and use it in the next. For n number of points in d dimensions, and k number of clusters, the next information required by Elkan's algorithms:

- n upper-bounds on the distance from each point to its closest centroid.
- n cluster assignments for the points from the previous iteration.
- nk lower-bounds on the distances from each point to each centroid.
- k^2 centre-centre distances.

Unlike our implementation of Elkan's algorithm on Hadoop, which will be explained Chapter 4, the sequential implementation caches the extra information in memory.

Let c and c' be two centroids, where c is the closest centroid to point x, and c' is any other centroid, then from Lemma 2.3.1:

$$\text{if } \tfrac{1}{2}d(c,c') \geq d(x,c)$$
$$\text{then } d(x,c') \geq d(x,c),$$

which means that x is closer to c than c' and there is no need to calculate $d(x,c')$.

Upper Bound: Elkan's algorithm sets an upper-bound (u) on the distance between x_i and its assigned closest centroid c_{a_i}, such that $u_i \geq d(x_i, c_{a_i})$. The pseudo-code in Algorithm 2 describes the sequential implementation of Elkan's algorithm. At the beginning of each iteration, the centre-centre distances, and the values of h_j are computed. Each value in h_j corresponds the half the distance from centroid c_j to its closest other centroid in the set of centroids. Half the distance from each centroid c_j to its closest other centroid is computed in line 7 and stored in a collection denoted as h_j. If the upper-bound u_i for point x_i is less or equal to half the distance from its owner centroid c_j to its other closest centroid, then according to Lemma 2.3.1 the distance from x_i to all centroids can be avoided.

In lines 15-18 of Algorithm 2, the upper-bound u_i is tested to see if it is out-of-date (i.e. it is possible that $u_i \neq d(x_i, c_{a_i})$. To do this, the boolean variable r is initialised to *true* in line 12. If r is true, this means the distance from x_i to its owner centroid c_{a_i} must be computed and u_i is set to this distance (line 16). Then, r can be set to false.

Lower Bounds: Furthermore, a lower-bound (l) is set on the distance between each point x_i to each centroid c_j. Let $c_{j'}$ and c_j be the positions of centroid c_j at the previous and current iterations, respectively. A lower-bound l can be created on $d(x_i, c_j)$ by assuming that $c_{j'}$ has moved towards point x_i a distance of $d(c_{j'}, c_j)$, that is:

$$d(x_i, c_j) \geq d(x_i, c_{j'}) - d(c_{j'}, c_j) \tag{2.3}$$

The right side of (2.3) forms a lower-bound on $d(x_i, c_j)$. This step can be seen in line 38 in Algorithm 2. Lines 20-24 show that the distance from x_i to c_j is

Algorithm 2: Sequential Elkan K-Means(X, k) [19]

```
1  select $k$ initial cluster centroids from $X$
   /* upper and lower bounds initialisation */
2  $l_{i,j} \leftarrow d(x_i, c_j)$
3  $u_i \leftarrow min_j d(x_i, c_j)$
4  $a_i \leftarrow argmin_j d(x_i, c_j)$
5  while not converged or an early termination condition is not met do
6      compute $d(c_j, c_{j'}))$, for all $1 \leq j, j' \leq k$
7      compute $h_j \leftarrow \frac{1}{2} min_{j \neq j'} d(c_j, c_{j'})$, for all $1 \leq j, j' \leq k$
8      for $i = 1$ to $n$ do
9          if $u_i \leq h_{a_i}$ then
10             continue
11         end
12         $r \leftarrow true$ //a boolean flag to check if $u_i$ is out-of-date
13         for $j = 1$ to $k$ do
14             if $j \neq a_i$ AND $u_i > l_{i,j}$ AND $u_i > \frac{1}{2} d(c_j, c_{a_i})$ then
                   /* check if $u_i$ needs to be updated */
15                 if $r == true$ then
16                     $u_i \leftarrow d(x_i, c_{a_i})$
17                     $r \leftarrow false$
18                 end
19                 else
20                     $d(x_i, c_{a_i}) \leftarrow u_i$
21                 end
22                 if $u_i > l_{i,j}$ or $u_i > \frac{1}{2} d(c_j, c_{a_i})$ then
23                     $l_{i,j} \leftarrow d(x_i, c_j)$
24                     if $d(x_i, c_j) < d(x_i, c_{a_i})$ then
25                         $c_{a_i} \leftarrow c_j$
26                     end
27                 end
28             end
29         end
30     end
31     for $j = 1$ to $k$ do
32         $c'_j \leftarrow \frac{1}{|c_j|} \sum_{x_i \in c_j} x_i$ //mean of points belong to $c_j$
33         reposition cluster $c_j$ to $c'_j$
34         $m_j \leftarrow d(c_j, c'_j)$ //the distance centroid $c_j$ has moved
35     end
       /* update upper and lower bounds */
36     for $i = 1$ to $n$ do
37         $u_i \leftarrow u_i + m_{a_i}$
38         for $j = 1$ to $k$ do
39             $l_{i,j} \leftarrow l_{i,j} - m_j$
40         end
41     end
42 end
```

explicitly calculated only if $u_i > l_{i,j}$ or $u_i > \frac{1}{2}d(c_{a_i}, c_j)$, otherwise, all distance computations associated to point x_i can be avoided.

The algorithm then finds the locations of the new centroids, lines 30-34, by computing the mean of points assigned to each cluster (c'_j in Algorithm 2). In addition, the distance each centroid has moved (m_j in Algorithm 2) is cached to be used in updating the distance bounds.

Finally, the algorithm updates the bounds in lines 34-40, where each movement of each centroid is added to each upper-bound, and subtracted from each lower-bound.

As explained previously, while upper-bound u_i stores the distance from data point x_i to its closest centroid, lower-bound $l_{i,j}$ stores the distance from data point x_i to another centroid c_j. The idea of employing these distance bounds in Elkan's algorithm is to avoid computing the exact point-centre distances. For example, the test $u_i \leq l_{i,j}$ compares the distance from data point x_i to its previously assigned cluster centroid with the distance from the same point to another centroid c_j. If this test holds, this means that the cluster centroid that x_i was assigned to in the previous iteration is closer to x_i than centroid c_j. In this case, the distance computation from x_i to centroid c_j can be eliminated.

In the best case scenario where data points in the input dataset are distributed on well-separated clusters, Elkan's algorithm reduces the complexity of computing the point-centre distances from $O(nk)$ to closer to $O(n)$ per iteration. However, updating bounds at the end of the algorithm takes $O(nk)$ time. This means the time complexity of the algorithm remains at least $O(nkd)$ per iteration adding to it the time to compute centre-center distances which is $O(dk^2)$.

Elkan's approach can be a promising candidate to deal with large-scale datasets. However, the large memory overhead of $O(nk + k^2)$ makes it susceptible to an out-of-memory exception. For this reason variants of Elkan's algorithm were introduced with the aim to prune the most possible number of distance calculations with the least possible number of distance bounds. The following section overviews one example of these variants that is called Hamerly's algorithm.

2.3.1.5 Hamerly's Algorithm

In [67], Hamerly proposed a fast K-Means algorithm using triangle inequality properties along with upper and lower bounds to avoid looping over all cluster centres to find the closest centre for a certain point (the loop responsible for computing distances is called the *innermost loop* in Hamerly's paper). Hamerly's algorithm uses one upper-bound, as in Elkan's algorithm, on the distance from each point to its closest centre, and instead of using k lower-bounds as in Elkan's algorithm, it applies one lower-bound on the distance from each point to its second closest centre.

For a given point x_i, the algorithm tracks the movement of each centre from one iteration to the next. Each time the centres move, the bounds get updated. The lower-bound l_i is updated by subtracting it from the distance moved by the furthest moved centre from x_i. The upper-bound u_i is updated by incrementing its value by the distance moved by the centre c_{a_i} that point x_i is assigned to. The algorithm applies the following two tests to avoid the innermost loop:

1. $u_i \leq l_i$
2. $u_i \leq min_{a_i \neq j} d(c_{a_i}, c_j)/2$

If one of the above conditions is true, the assignment of x_i does not change and the algorithm avoids the innermost loop which calculates the distance from x_i to all k centres.

Hamerly's approach avoids a number of distance computations that is higher than the one in Elkan's algoithm when it operates on data in small to medium dimensions ($d \leq 50$) [67]. Elkan's algorithm, on the other hand, skips a large number of distance computations as the number of dimensions increases. To elaborate more on this, we discuss the "*curse of dimensionality*". Generally, the curse of dimensionality [68] refers to problems in the field of data analysis that are caused by data with a large number of attributes [69]. The problem with processing high-dimensional data is that, as the number of dimensions increases, the distribution of data points becomes more sparse. The challenge in clustering high-dimensional data is that the curse of dimensionality makes the distances less discriminative and, in some cases, the distribution of the data becomes relatively uniform. In such cases, finding the closest centroid to a point becomes very challenging [69]. So, why does Elkan's algorithm works better than Hamerly's algorithm with high-dimensional data? Because Elkan's method uses multiple lower-bounds each of which is specialised in one cluster, while Hamerly's algorithm uses only one lower-bound for all clusters. In higher dimensions, using multiple bounds allows more accurate tests to skip distance computations.

An obvious advantage in Hamerly's algorithm as compared to Elkan's is the low memory overhead. The memory overhead created by Hamerly's algorithm is $O(3n + 2k)$ compared to Elkan's memory overhead of $O(nk + k^2)$. Hamerly's memory overhead comes from keeping the following information:

- n upper-bounds,
- n lower-bounds,
- n cluster assignments,
- k distances moved by centres in the previous iteration, and
- k distances from each centre to its closest other centre.

The time per iteration for Hamerly's algorithm is $O(ndk + dk^2)$, where ndk is the time for the Naive K-Means and dk^2 is the time to calculate the distance from each centroid to its other closest centroid.

The previous sections reviewed several sequential variants of K-Means that used triangle inequality to improve the efficiency of the Naive K-Means. The following sections presents a background about distributed computing frameworks that can be used as parallel environments to improve the scalability of K-Means.

2.4 MapReduce

Dean and Ghemawat [70], introduced MapReduce in a Google white paper in 2004. MapReduce is a programming paradigm that is designed to store and process large-scale datasets efficiently and reliably on large clusters of commodity machines. MapReduce is designed to provide a high performance parallel execution of programs without dealing with underlying details of the distributed system such as scheduling, distribution and fault-tolerance.

Fault-tolerance in MapReduce is achieved by dividing each job into many small tasks where each task is processed by an independent machine. The input of a job is replicated over the distributed file system. In case of a failed task, a replica can be loaded to that task to continue the work without the need to resubmit the whole job.

In addition, a system that deploys MapReduce can scale its performance up and down as the computation requirements change. Currently, many cloud computing service providers offer MapReduce as a Web service (e.g. Amazon Elastic MapReduce or Amazon EMR [71]) where one can use the ”pay-as-you-go” service model to run data-intensive applications on an expandable and low-spec. cluster that implements the MapReduce framework.

2.4.1 MapReduce Dataflow

In the MapReduce paradigm, the input data is stored on a distributed file system such as Google File System (GFS) [72], or Hadoop Distributed File System

(HDFS) [73]. The input and output data are in the form of *key-value* pairs. The computation process is expressed by implementing two functions: *map* and *reduce* from the MapReduce library, which are typically implemented by the user. The dataflow and main phases in the MapReduce model are illustrated in Figure 2.3. The three main phases in this framework are: the *Map Phase*, the *Shuffle Phase*, and the *Reduce phase*. In addition, there is an optional phase called the *Combine Phase* that implements a function called *combine*. The MapReduce phases are explained as follows:

Map Phase: the map phase consists of three main operations: *map*, *partition*, and *sort*. When the input dataset is loaded to HDFS, it is split into what is known as *input-splits*. The number of mappers equals the number of input-splits and the size of each input-split can be modified (default 128 MB). Each mapper processes one input-split independently on a separate node. The *map* function takes as an input the records in each input-split in the form of *key* (K) and *value* (V) pairs. The *map* function processes one <K1,V1> pair at a time. Once processed, the *map* function outputs a new <K2,V2> pair that is written to a circular buffer. The buffer spills its contents to spill files on the local-disk of the mapper's node when it reaches a certain threshold (default is 512 MB). Before data is written to disk, a *partitioner* is invoked to partition data by reducer. The data within each partition gets sorted in-memory by key (K2). If a combiner is defined, it runs on the output of the sorted data. However, it is not guaranteed that the combiner will run at all, since it will only run if the number of spill files is at least three. Finally, these partitioned sorted <K2,V2> pairs are spilled to the disk of the machine the mapper is running on.

Shuffle Phase: The shuffle phase makes sure that each partition is transferred to the right reducer. Each reducer uses HTTP protocol to fetch its own partition from the mappers' output files that reside on the mappers' nodes. The shuffling phase can be the performance bottleneck in many cases where the MapReduce job performance is limited by the available network bandwidth. That is why many studies attempt to refine and improve this phase [16] [74]. The shuffle process starts as a predefined percentage (default is 5%) of mappers complete their work. This makes this phase overlap with the Map phase where a portion of the intermediate data is shuffled while some mappers are still running.

Reduce Phase: The reduce phase starts after each reducer fetches its own partition from the mapper's output files. Before invoking the *reduce* method, the reducer merges and sorts the mappers' output files fetched from different mappers and then the *reduce* method is invoked. The *reduce* method receives the input key associated with the list of the values that belong to the input key. Each reducer processes an individual input key with its associated list of values. After processing all the data, each reducer outputs the resulted <K3,V3> pairs to the distributed file system *DFS*, in our case Hadoop Distributed File System *HDFS*.

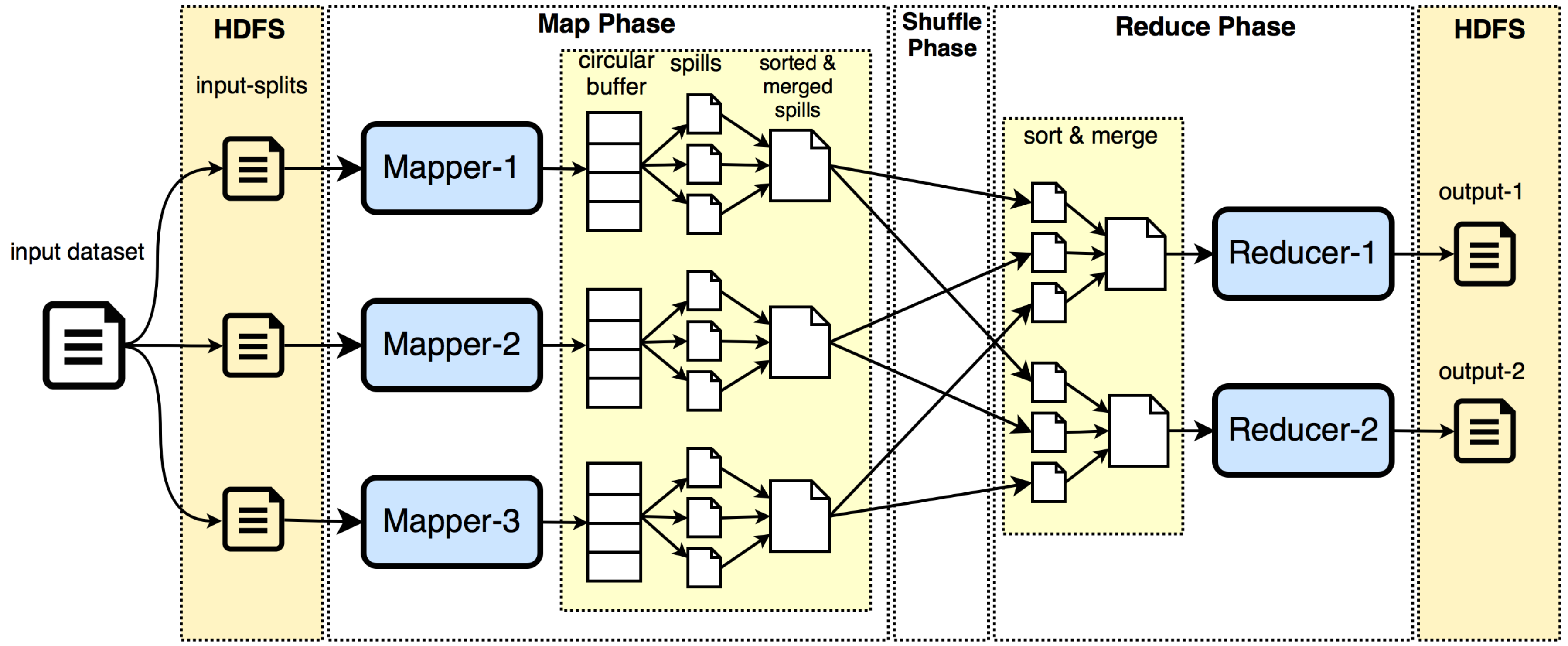


Figure 2.3: MapReduce framework dataflow

Combine Phase: The goal of the *combiner* is to minimise the amount of intermediate data transferred from mappers to reducers across the cluster. The combiner works as follows, the *combine* function takes the mapper output as an input and makes it more compact by doing a local aggregation over the intermediate data on the map phase. Simply, instead of outputting a single pair of <K2,V2>, it groups the <K2,V2> pairs by (K2) and outputs a pair of <K2,list<V2>> to the reducer. The combiner can be thought of as a ”mini reducer” [5], because in many cases the implantation of both is the same. However, the reduce function has to be associative and commutative [16], meaning that the order of the operations and operands does not matter (e.g. addition and multiplication). There are some cases where these conditions are not met and a combiner can still be implemented (e.g. calculate the average). However, the combiner has to be implemented separately.

As mentioned earlier in the Map Phase, the combiner runs only if the number of spilled files from the circular buffer is more than three. The reason behind that is that if there is only one or two spill files, the generated overhead from invoking the combiner is not worth the potential benefit from reducing the size of the mapper output. This means that the combiner might not be invoked at all, which is the reason why the reducer's implementation must outputs the same results as if there is no combiner is defined [16].

The combiner does not have a predefined interface. Therefore, it must implement the *reduce* method that belongs to the reducer interface.

2.4.2 Iterative Process on MapReduce

This section explains the general approach of managing the iterative process on MapReduce. It will also present an overview of some tools provided by Hadoop to declare global counters, and distribute files over all the nodes of a cluster.

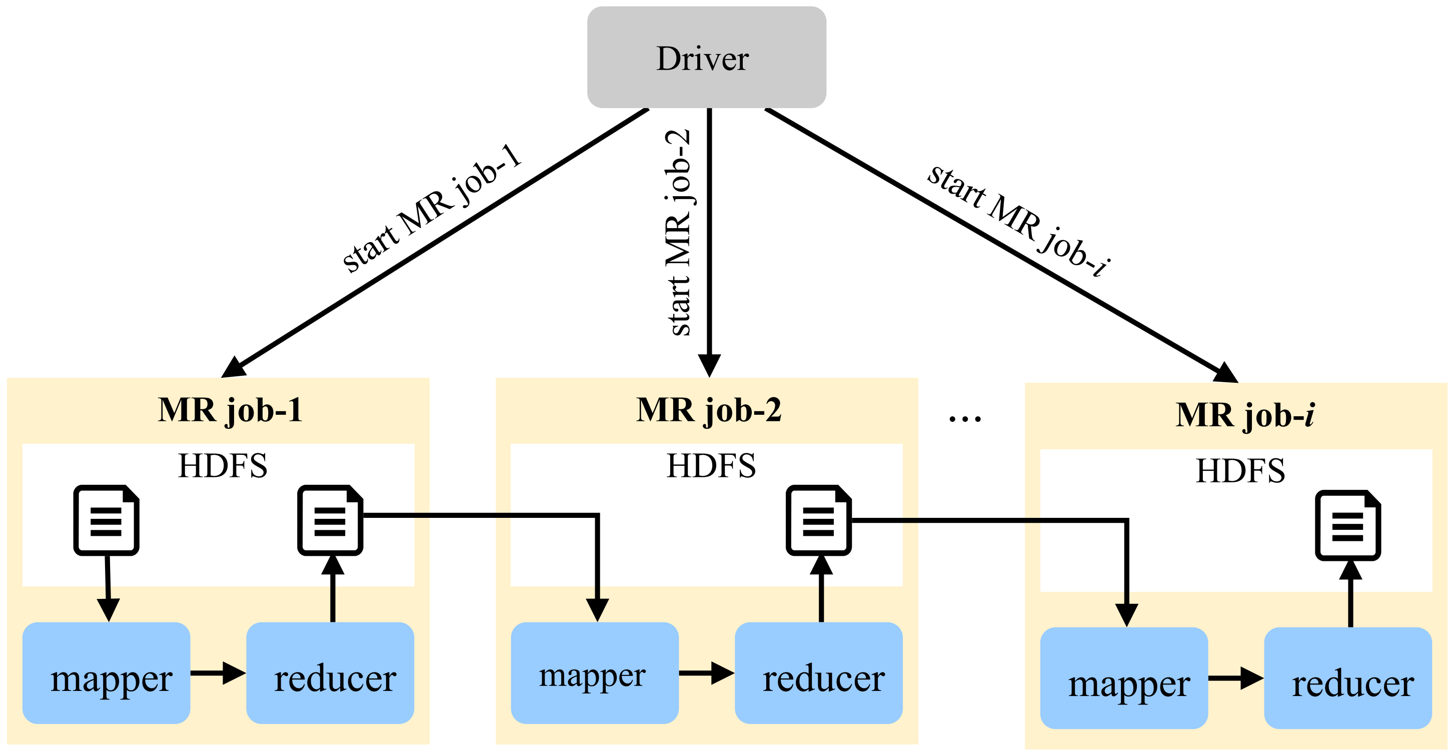


Figure 2.4: Iterative process on MapReduce. Each MapReduce job represents one iteration and the input/output files are read and written from/to HDFS. In this specific case, the reducer's output in one iteration is the mapper's input in the next.

2.4.3 Limitations of MapReduce

Despite the advantages that MapReduce offers to store, manage, and process large-scale datasets, several limitations were addressed in many works [9] [10] [75] [76] [77] [78] [79] [74] [80] [81] [24] [82]. Since the focus of this work is on iterative clustering algorithms on MapReduce, the following limitations are related to processing such algorithms on MapReduce:

First, iterative machine learning algorithms, such as K-Means, PageRank [83], and logistic regression [84], require caching some information from former iterations in order to process the data in the current iteration. To achieve this in a MapReduce framework, first, a *Driver* program must be implemented to control the iterative process. The Driver sets up and triggers a MapReduce job for each iteration in the algorithm. In addition, the Driver manages the input and output files for each iteration. Figure 2.4 illustrates a particular case where the output of the reducer in one iteration is used as an input for the next iteration. To maintain fault tolerance, the output of each iteration should be stored on the distributed file system, or HDFS as in Hadoop, to be replicated.

- *Absence of loop-aware task scheduling*: For each iteration in an iterative algorithm, MapReduce creates a new MapReduce job. This means a considerable amount of time is wasted in repeated operations such as, the initialisation and termination of map and reduce tasks.

- *Reload and reshuffle static data*: Some iterative algorithms such as K-Means have two types of data, static, and state data. Unlike state data, static data is the data that does not change during iterations. MapReduce reloads and reshuffles static data in each iteration which creates an unnecessary I/O and communication overheads.

- *Synchronous execution of map and reduce tasks*: This means reduce tasks cannot start until all map tasks are finished. In some algorithms, reduce tasks can process data as it is omitted from map tasks without the need to wait for all map tasks to finish. In this case, asynchronous execution of map and reduce tasks would accelerate the iterative process. However, K-Means algorithm would not benefit from asynchronous execution because reducers needs all values (points) related to each key (centre) to be present to be able to calculate their mean and produce new centres.

- *Extra one iteration to terminate*: MapReduce may need to perform one extra MapReduce job to check for the termination condition to terminate the iterative process. This process creates an overhead that is caused by scheduling tasks, reloading data, and managing nodes' communications.
- *Retrieve information from previous iterations*: Since each iteration in Hadoop is an independent MapReduce job, and memory is not shared between those jobs, Hadoop does not have the ability to retrieve any information from previous iterations. This limitation imposes extra complexities on iterative algorithms that require information from previous iterations in order to proceed their work efficiently. This project investigates this limitation in particular.

2.5 Apache Hadoop

Apache Hadoop [15] is an open-source platform that implements the MapReduce programming paradigm to process large-scale datasets across large clusters of commodity machines. Hadoop delivers a reliable, high performance, low cost, and fault tolerant distributed computing framework where each machine has the ability to compute and store data independently. Hadoop has been widely used by many companies such as Yahoo!, Facebook, Twitter, and IBM to manage and analyse massive amounts of daily generated data [16].

To understand the dataflow in Hadoop, first, an introduction to Hadoop Distributed File System (HDFS), and the main components of Hadoop's architecture is presented in the following sections.

2.5.1 Hadoop Distributed File System

The design of Hadoop Distributed File System (HDFS) was inspired by the Google File System (GFS) [72]. It was designed to store massive datasets across hundreds or thousands of commodity servers, and transfer those data to user applications at a high bandwidth. More commodity servers can be easily added based on the demand of storage and computation while maintaining the cost of expansion. The main characteristics of HDFS are its ability to partition, replicate, and store input files on different locations in the cluster. To store a file, HDFS partitions the file into m partitions and stores them on blocks distributed across DataNodes. Each block is of size 64MB or 128MB (can be modified). These blocks are replicated (default is 3 replicas) based on a block placement policy that stores the first replica on a DataNode on the same rack of the original copy, the second and third replicas are stored in two different nodes on different racks [73].

2.5.2 Hadoop Generations

Hadoop 1.x: Hadoop 1.x [15] is the first generation of Hadoop. The architecture of Hadoop 1.x consists of four components: JobTracker, NameNode, TaskTracker, and DataNode.

The *JobTracker* is the service within Hadoop 1.x that is responsible for resource management, tasks scheduling and monitoring. The JobTracker farms out MapReduce tasks to specific nodes in the cluster, ideally the nodes where data resides.

The *NameNode* keeps the directory tree of all files in the file system, and tracks where across the cluster the data is located. Moreover, the NameNode manages the HDFS namespace.

The *TaskTracker* is a process exists in each node and controlled by the JobTracker to manage map, reduce and shuffle tasks on these nodes.

The *DataNode* stores data on HDFS where client applications can talk directly to a DataNode once the NameNode has provided the location of the data. DataNode instances can also talk to each other, which is what they do when they are replicating data.

Data Flow: When a MapReduce job is submitted, the JobTracker schedules map and reduce tasks to run on TaskTrackers. TaskTrackers send reports to the JobTracker with the status of the task and the progress of the job in general. If a task fails, the JobTracker reschedule this task to run on a different TaskTracker.

Each map task starts by taking as input one partition of the data stored on HDFS called input-split, or split. Hadoop is designed to run map tasks on the same node where input data are stored. This technique is referred to as *data locality optimization*. Map tasks write their intermediate data to local disks instead of HDFS and then this data get sorted and shuffled and sent to reducers. The number of reducers can be specified by the user (default is 1). The output of the reduce tasks written on HDFS and replicated afterwards. The Data Flow in both versions of Hadoop is illustrated in Figure 2.5.

Despite the fact that Hadoop 1.x provides a reliable, scalable and fault tolerant MapReduce framework, it has several limitations [16] [1] that can be summarised as follows:

- Slave nodes could only scale up to $\approx$4000 nodes because the JobTracker and NameNode become a bottleneck if more nodes are added.
- JobTracker and NameNode are a single point of failure (SPOF).
- NameNode is not highly available and could not be scaled horizontally.
- It supports only MapReduce applications.

Hadoop community was motivated to upgrade the system to overcome these limitations. As a result, YARN or Hadoop 2.x was introduced.

YARN/Hadoop 2.x: YARN, which stands for Yet Another Resource Negotiator, addresses Hadoop 1.x drawbacks that were mentioned above by applying the following improvements [1]:

- As shown in Figure 2.5-b, YARN breaks the two essential tasks of the JobTracker, resource management, and tasks scheduling and monitoring, into two separate daemons: a global *ResourceManager* that orchestrates the compute resources assignment to applications but does not interfere with per-application state management, and an *ApplicationMaster* that is responsible for negotiating resources, referred to as Containers, from the ResourceManager. A Container holds a collection of resources such CPU and memory that the application requires. It is created by the ApplicationMaster, monitored by NodeManager, and scheduled by the ResourceManager. The NodeManager is a per-node agent that monitors application containers in a single node and reports node status to the ResourceManager.
- NameNode can scale-up horizontally and it is Highly-Available because it is not a single point of failure (SPOF) any more.
- YARN supports Non-MapReduce applications like MPI.
- YARN can scale-up to ≈10000 nodes.

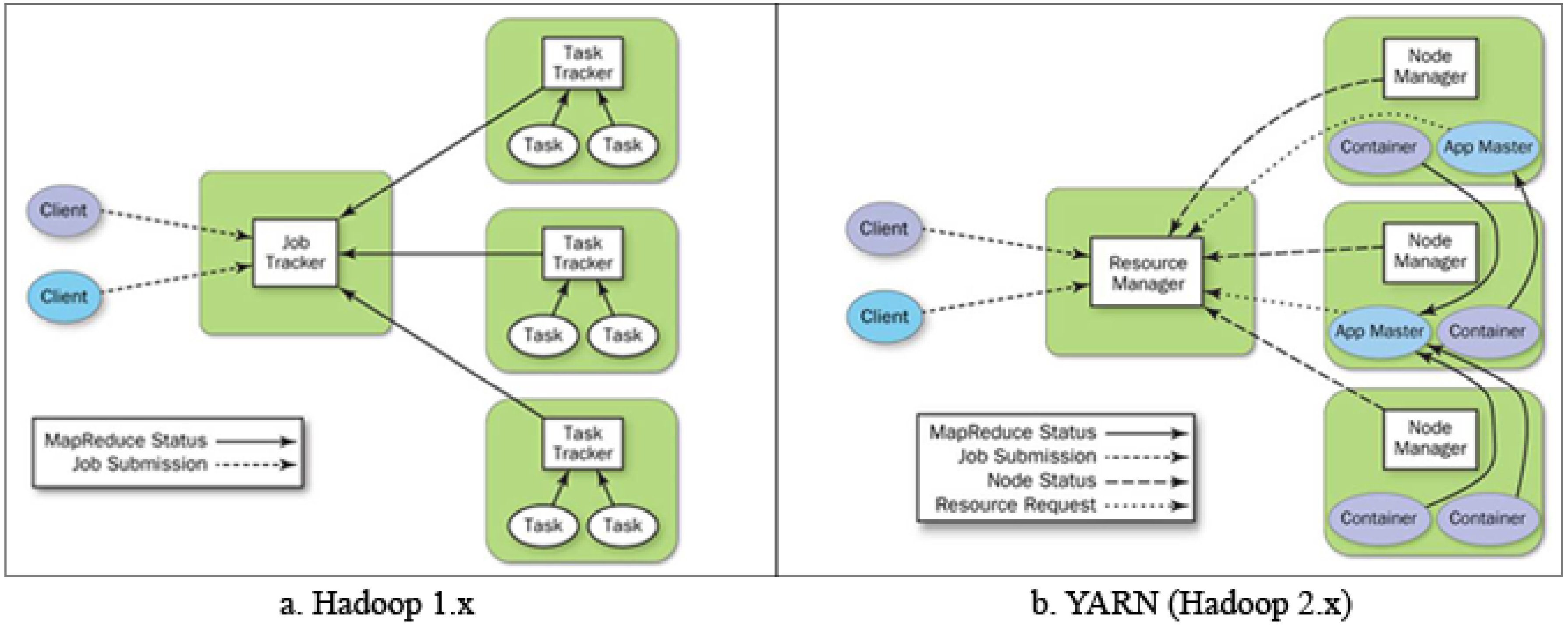


Figure 2.5: The Components of Hadoop 1.x and YARN (Hadoop 2.x). [1]

2.5.3 Hadoop Counters

Hadoop provides two types of counters, built-in counters, and user-defined counters.

Built-in counters are counters that collect statistics about each MapReduce job and report different metrics for the job. Built-in counters are divided into several groups based on the information the counters' report. For example, the counters: map skipped records, map output records, and map output bytes, belongs to a group of counter called MapReduce Task Counters. Furthermore, the counters: filesystem bytes read, and filesystem bytes written belong to a group of counters called Filesystem Counters.

User-defined counters are counters that are defined by the user and can be incremented in mappers or reducers. These counters are global, which means they are aggregated across all map and reduce tasks and their total is produced at the end of the job. These counters can be useful to iterative algorithms in terms of using them to track the condition of the algorithm's convergence it will be explained in later sections when K-Means on MapReduce is implemented on MapReduce.

2.5.4 Distributed Cache

Distributed Cache is a feature provided by Hadoop to allow applications and tasks to have a local access to files that are stored on some data nodes on the distributed system. When a file is cached, it is copied to the local-disk (the term *cache* is not accurate in this case, because the files are not stored in-memory but on the local-disks of data nodes) of each and every data node. Mappers and reducers can then read the files locally. In order to make room for new files, cached files are deleted when their size reaches a predefined threshold (default is 10 GB). The cached files are read-only files that are loaded to the Distributed Cache in the driver program and they cannot be modified while the job is running.

To ensure that the cached files are consistence, the Distributed Cache tracks the modification timestamps of cache files. To determine on which node a particular key-value pair resides, the cache engine uses a hashing algorithm. The files are guaranteed to be always consistence since there is always a single state of the cache cluster.

In the case of processing iterative algorithms on Hadoop, each iteration is represented by a new MapReduce job. In this case, map and reduce tasks and Distributed Cache components are not persistent between jobs where each job must configure new tasks and Distributed Cache. This means intermediate data that is cached in the Distributed Cache in one iteration will not exist in the following iteration. The distributed Cache is used in this work to cache centroid files into mappers and reducers.

2.6 Apache Spark

Apache Spark [22] is a distributed framework that is designed to process large-scale working sets that are reused over multiple parallel operations in-memory. The goal of Spark is to process iterative machine learning algorithms and interactive analytics problems faster than Hadoop MapReduce while maintaining MapReduces' fault tolerance and scalability. Spark can operate on Hadoop YARN, Apache Mesos [85], a cluster resource manager that allows different distributed frameworks to share the same resources on one cluster, Amazon Elastic Cloud (EC2) [86], or as a standalone system.

2.6.1 Main Abstractions

Two main abstractions are provided by Spark to process parallel applications:

1. *Resilient Distributed Datasets (RDD).* An RDD is a collection of immutable (read-only) objects partitioned among cluster nodes that can be rebuilt in

case a partition is lost. RDDs can be cached in-memory once across worker nodes (*executors*) and reused by applications that run on multiple parallel operations. This process aims to reduce the I/O and communication overheads that Hadoop encounters. In case of a node failure, RDDs can be rebuilt and reconstructed which makes Spark fault-tolerant. Constructing RDDs can be done by:

- Loading a file that is stored on a distributed file system such as HDFS.
- Partitioning and parallelising a collection data structure, such as an array, and distributing these partitions over multiple nodes.
- Performing a transformation on an existing RDD to produce a new RDD.
- Persisting existing RDDs in-memory. After the first action is performed on an RDD, Spark discards this RDD from memory, unless there is a hint (by using the *cache* action) that this RDD needs to be persisted in memory for future operations. In this case, the RDD will resides in-memory (if there is a sufficient memory space) for later operations.

2. *Parallel operations.* Parallel operations can be divided into two types:

    - *Transformations*, where an RDD can be transformed from a file on stable storage, or from another existing RDD. Transformation operations are *lazy operations*. That is, a transformed RDD does not get evaluated until an action (e.g. *count*) is triggered. Some examples of main transformation operations are: *map*, *reduce*, *join*, *groupBy*, and *filter*.
    - *Actions*, where a value is returned to the application driver, or stored on a data storage. Examples of action operations are: *count*, *collect*, *take*.

Spark provides two types of shared variables:

1. *Accumulators*: accumulators are variables that can aggregate values across multiple tasks that run on different executers, and return their aggregated value to the driver.

2. *Broadcast variables*: broadcast variables allow Spark applications to send read-only variables to all working nodes and keep these variables in-memory.

2.6.2 Spark Architecture

Spark core is the engine for processing parallel data. The responsibilities of Spark core include: managing jobs' scheduling and distribution on a cluster, managing the memory and recovering failed RDDs, communicating with storage file systems.

To run a Spark job, a *driver* program must be written by the developer in a Scala, Python, or Java API. The driver creates and distributes RDDs, controls parallel operations (transformations and actions), and retrieves the results returned by action operations. As shown in Figure 2.6, the driver creates a *SparkContext* object, which establishes a connection with the computing cluster to schedule the execution of the jobs. SparkContext also creates RDDs, accumulators, and broadcast variables. As the execution of the driver starts, a logical *directed acyclic graph* (DAG) is created. The created DAG represents the order of operations in the driver. This graph is then converted into a physical execution plan. Spark negotiates the resources needed for each job through a *cluster manager*. Spark can run on top of different cluster managers such as, YARN, Mesos, or Spark's standalone cluster manager.

Spark is shipped with a library containing machine learning algorithms called *MLlib* [87]. This library provides many machine learning algorithms such as, K-Means, PageRank, and logistic regression.

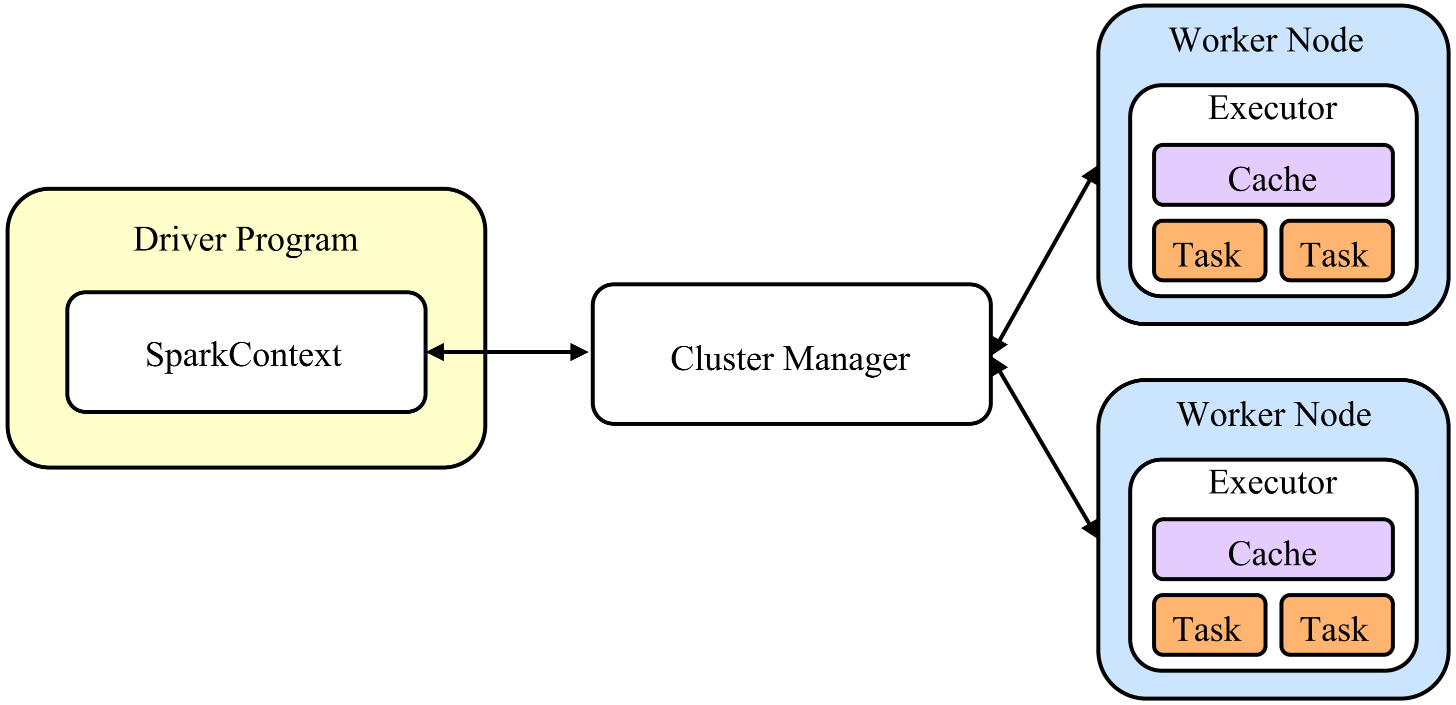


Figure 2.6: Apache Spark basic architecture.

There are several solutions, in addition to Spark, that aim to overcome the limitations of the MapReduce model on processing iterative algorithms. The next section will overview some of these solutions.

2.7 Iterative MapReduce Implementations

Although the MapReduce programming model is a good solution for processing large-scale datasets, it does not support data analytics algorithms that are iterative in nature [80]. This section reviews three solutions, HaLdoop, iMapReduce, and Twister, that aim to overcome the drawbacks (explained in section 2.4.3) of the MapReduce framework on supporting iterative algorithms.

HaLoop

HaLoop [74] extends Hadoop MapReduce framework with the aim to support the execution of iterative algorithms by providing the next properties:

1. Cache the mappers input on the local disks of the worker machines to avoid reloading and reading invariant data in each iteration.

2. Avoid the extra MapReduce job that checks for the termination condition by caching the reducer output and perform the test on a distributed fashion.

3. Cache reducers input on the local disks of the worker machines in order to access invariant data without shuffling them from mappers to reducers. However, it is required for the cached reducer's input to be constant over all iterations which is not the case in K-Means. This is because K-Means needs to shuffle all data points and cluster centroids from mappers to reducers to compute the average.

Moreover, HaLoop introduces a new API to automatically control the iterative process without the need for the programmer to write a detailed driver program to do this. In Addition, HaLoop modifies Hadoop's task scheduler that assigns map and reduce tasks that access the same data but are executed in different iterations to the same physical machine.

In [74], the performance of HaLoop was compared against Hadoop's. The results show improvements in terms of speedup and the reduced amount of shuffled data across the cluster nodes. K-Means performance is *"marginally better"* ($\approx$ 5% improvement in non-local reads).

As discussed in section 2.3.1.1, K-Means optimisations based on triangle inequality require caching extra information in one iteration in order to read this information in the next and skip distance computations. To implement such algorithms on HaLoop, the extra information could be cached, along with data points, in the map phase in a given iteration. The following iteration will read the data points with the required extra information at the beginning of the map phase. HaLoop can be a promising MapReduce framework that handles iterative algorithms efficiently. However, the latest version of HaLoop (released in 2012) works only on top of old Hadoop releases (Hadoop 1.x) which is not supported by the cluster that this project bases the experimental work on.

iMapReduce

iMapRedue [80] is a Hadoop extension that supports the processing of iterative algorithms. The main goals for iMapReduce are:

1. To reduce the resulting overhead from creating a new job in each iteration.
2. To eliminate the static data shuffling between map and reduce tasks.
3. To execute iterations asynchronously, that is, an iteration can start before all previous tasks have finished.

To achieve the first goal, the authors introduced *persistent tasks concept* which keeps the map and reduce tasks alive during the iterative process until a termination condition is satisfied. Specifically, after map and reduce tasks process the input data and output results, they stay idle waiting for input. The map tasks wait for the reduce tasks output, and the reduce tasks wait for the input from the map tasks.

The second goal is achieved by loading static data to the map tasks only once and does not shuffle the static data to the reduce tasks. The state data that is outputted from the reduce tasks is passed to map tasks without writing it to the distributed file system. Before injecting data to map tasks, a join operation is performed between state and static data to update state data then send it to map tasks. This technique aims to significantly reduce the amount of shuffled data between mappers and reducers. However this only applies to graph-based algorithms because they only need state data in the reduce phase, while K-Means and K-Means-like algorithms need both state and static data in both map and reduce tasks. Furthermore, since a local connection is preferred, the task scheduler always schedules a pair of a map and reduce tasks to the same machine to maintain a local connection. This leads to a drawback where the number of map and reduce tasks must be identical.

The third goal is achieved by making map and reduce tasks execute asynchronously. That is, a map task can start as soon as its correspondent reduce task sends its output to this map task as an input. This approach aims to speed up the process of the framework. However, in K-Means this cannot be applied because map tasks in K-Means need the centroids list that is calculated by all reduce tasks and, hence, map tasks must wait for all reduce tasks to finish.

iMapReduce handles failures the same way as Hadoop. However, the state data from a previous iteration must be all written to the distributed file system in order to re-execute a failure iteration. The tasks in iMapReduce are persistent which conflicts with the task scheduling mechanism followed in the standard MapReduce. Therefore, iMapReduce cannot benefit from MapReduce mechanism for load balancing.

In the same work [80], it was reported that iMapReduce improved K-Means performance with a speedup factor of 1.2x compared to the standard MapReduce programming model. The experimental work in this thesis shows that accelerating the standard K-Means on Hadoop by using triangle inequality could achieve significant speedups (up to 34x). The benefit from implementing optimisations of K-Means presented in this work on iMapReduce is not expected to be significant. This is because the required extra information would be written/read to/from the distributed file system and this is the major bottleneck in implementing such optimisations on MapReduce.

Twister

Twister [23] is an enhanced MapReduce runtime that was designed to handle iterative algorithms more efficiently than Hadoop. Twister has substantial differences from Hadoop in terms of handling input/output data, transferring intermediate data via a communication infrastructure, and scheduling map/reduce tasks. To support iterative computations based on MapReduce, Twister introduces long-running map/reduce tasks which are configured one and reused by subsequent iterations. In addition, static data is cached into the memory of the worker nodes to be reused in multiple iterations. Twister's architecture and main differences from Hadoop is discussed next.

The following sections describe the architecture of Twister. Moreover, the differences between Twister and Hadoop are explained in general, and in particular to the implementation of K-Means on both systems.

Architecture

The architecture of Twister consists of three major components:

1. Twister *driver* that controls the entire MapReduce job,

2. Twister *daemon* that runs on worker nodes, and

3. The *broker network* that is based on publish/subscribe messaging infrastructure.

A daemon process is started in each worker node as Twister starts running. Each daemon maintains a worker pool to execute the map and reduce tasks assigned to it, checks the status of the tasks, and responds to the control events. The driver handles the entire MapReduce job by converting Twister API calls to control commands and sends the input data messages to the daemons that manages map and reduce tasks through the broker network. The resulted outputs from all reducers are then collected (combined) and returned to the driver to decide on proceeding with a new iteration or not.

Twister vs. Hadoop

The main differences between Twister and Hadoop can be explained as follows.

- *Task scheduling:* While Hadoop must configure and terminate independent map/reduce tasks for each iteration, Twister runs long-running map/reduce tasks that are configured once and used many times. To guarantee the reusability of map/reduce tasks over many iterations, Twister schedules the tasks statically (tasks scheduled to the same nodes over multiple iterations) as opposed to dynamically scheduled tasks in Hadoop. Static scheduling, however, could limit the resources of the computing infrastructure to a specific job even if they remain idle. This scheduling approach assumes that the computing infrastructure can accommodate all persistent tasks simultaneously, which limits the number of MapReduce jobs that can run concurrently [24].

- *Cacheable tasks:* Twister distinguishes between static and variable data. While variable data can be modified (e.g. centroids in K-Means), static data does not change over multiple iterations (e.g. input data points in K-Means). Twister loads (caches) static data in the memory of worker nodes once in the first iteration to eliminate the need of reloading it in each iteration. Hadoop, on the other hand, reloads static data from HDFS in each iteration which imposes an extra I/O cost when processing iterative algorithms. Although Twister takes advantage of caching input and intermediate data, it can be a limitation if the data does not fit in the memory of the worker nodes.

- *Input data:* Twister does not store input data on a distributed file system. Instead, it assumes that the input dataset is manually partitioned by the user and the partitions are stored as native files on the local disks of worker nodes. These file are then cached to the memory of each worker node. This caching mechanism can be a drawback in Twister because it assumes that the partitions will fit in the memory of worker nodes, which cannot be guaranteed when dealing with very large datasets [9]. This approach was adopted to simplify the implementation of the platform and to be able to pass the data files to any map/reduce task as a command line argument [23]. Hadoop, on the other hand, automatically partitions the input dataset and replicates the partitions over several nodes on HDFS to maintain fault tolerance and to try to run map/reduce tasks on local file.

- *Checking convergence:* To decide whether to start a new iteration or stop at the current one, Hadoop uses a controlling program (driver) that merges the output files written by multiple reducers on HDFS into one file. The driver then reads the merged file to check for the convergence or a stopping condition and decides whether to start a new iteration or stop at the current one. Twister, however, introduces a new phase called *combine* phase where the outputs from multiple reducers are combined and passed directly to the driver without writing/reading them to/from HDFS as in Hadoop. If the stopping condition is not satisfied, a new iteration is started and variable data is broadcasted to all worker nodes. However, this technique is not beneficial if the iterative algorithm decided to stop at a predefined number of iterations [24].

- *Fault tolerance:* Twister assumes that master node failures are rare; the broker network is reliable; and the data is replicated over the worker nodes. In case of a failure, Twister only guarantees restoring input data that can be reloaded from the file system or static parameters inherited from the driver. Any intermediate data processed by map and reduce tasks will be lost. Hence, Twister supports only intra-iteration fault-tolerance. On the contrary, Hadoop assumes that a failure could occur at any stage of the MapReduce job. In case of a node failure, Hadoop can restore intermediate data and re-assigns failure tasks to healthy nodes.

K-Means on Twister

K-Means is implemented on Twister as follows. First, the initial cluster centroids are picked from the input dataset and broadcasted to all the worker nodes. Then, the input dataset is manually partitioned into a number of files and loaded to the local file system of the worker nodes. Since the input data points in each partitioned file are considered as static data, each file is cached into the memory of each worker node. Next, the iterative process starts where each mapper finds the closest centroids from each data point and outputs the data point with the cluster index of its closest centroid. The mappers send their results to the reducers via the broker network where each reducer computes the new centroids. The new centroids are sent to a user program (driver) that tests the convergence criterion and decides whether to stop the iteration or start a new one. In the following iteration only the centroids that will be broadcasted to the nodes while data points are reused from the previous iteration. This technique reduces the amount of data transferred in the network compared as opposed to Hadoop which must reloads the same data in each iteration. Another advantage in this implementation is the long running map and reduces tasks which eliminates the overhead from the need to generate new tasks in each iteration in Hadoop. On the other hand, Twister assumes that the input data will fit in the memory of the worker nodes which is not always the case when the target is to cluster Big Data.

K-Means optimisations based on triangle inequality can use the caching mechanism in Twister to store the needed extra information (cluster assignments and distance bounds) in-memory. The extra information can be associated with data points and updated in the map phase. However, this could be a challenging task to achieve since the data is expected to be very large and the memory of the worker nodes might not fit the data and the extra information together. Therefore, we expect that the performance of the optimised algorithms presented in this project to excel if implemented on Twister as long as the memory can fit all data points with extra information.

2.8 Summary

This chapter presented a background of several sequential implementations of K-Means and showed how the efficiency of K-Means can be improved based on triangle inequality. It was discussed how triangle inequality approaches improves the efficiency of K-Means but with an extra cost from the need to maintain extra information such as cluster assignments and distance bounds. Various K-Means optimisations based on triangle inequality were reviewed and two of them (Elkan and Compare-means) were chosen to be implemented on MapReduce.

The chapter also explained the main phases in the MapReduce programming paradigm and introduced the architectures of two widely used distributed computing platforms known as Hadoop and Spark. Furthermore, the chapter reviewed some distributed MapReduce platforms including, HaLoop, iMapReduce and Twister. A detailed comparison between these platforms described the features each platform offers to overcome the lack of support to iterative algorithms in Hadoop. Table 2.2 summarises the features of each platform. The reviewed MapReduce platforms can be categorised into two categories: disk-based and memory-based, where the former store the data (static or variant) on the file system (local or distributed), and the latter store the data in the memory of the worker nodes. While Hadoop, HaLoop and iMapReduce are disk-based platforms, Twister and Spark are memory-based.

Although iterative MapReduce platforms can outperform Hadoop when processing iterative algorithms, some of these solutions (e.g. Spark and Twister) require the dataset to be small enough to fit into the main memory of the worker nodes. Otherwise, data will be spilled into the local disks of the worker nodes and read/write operation will be performed from/to local disks which increases the I/O overhead. Moreover, other platforms (e.g. HaLoop and iMapReduce) are not stable and mature enough because they were basically built as research prototypes [77]. Twister handles iterative algorithms efficiently by caching static data in-memory but at the expense of fault-tolerance. Therefore, such platforms could face serious challenges while operating on real-world settings [13].

Table 2.3 compares the implementations of K-Means based on the reviewed iterative MapReduce platforms (HaLoop, iMapReduce, Spark and Twister) with the optimised implementations of K-Means that will be presented later in Chapter 4. The first column in the table shows the name of the algorithm and the platform it was implemented on. For example KM-Twister denotes the implementation of K-Means on Twister. Since the new implementations of K-Means on Hadoop using triangle inequality have not been explained yet, we call our implementation in the table KM-Hadoop (our work). This method of naming K-Means implementations is for the purpose of this particular table only. The comparison is based on the method each implementation follows to load input data (data points) and cluster centroids to worker nodes. In addition, the method each approach follows to check for convergence is compared. Finally, a brief description of the advantages and disadvantages of each implementation is provided.

Other solutions including, Piccolo [88], Pregel [89], MapReduce online [90] and PrIter [91], proposed various techniques to support iterative algorithms on the MapReduce programming model.

	In-memory System	Iterative Process Techniques	File System	Fault Tolerance	Intermediate Data Transfer
Hadoop	No	Iterative algorithms not supported	HDFS	Strong	File
HaLoop	No	Loop-aware task scheduling, caching	HDFS	Strong	File
iMapReduce	No	Persistence tasks, asynchronous iterations	HDFS	Strong	File
Spark	Yes	RDDs, DAG, caching	HDFS, Cassandra, Amazon S3	Strong	RDDs
Twister	Yes	Long running tasks, caching	Local disks	Weak	Publish/Subscribe messaging

Table 2.2: Summary of features of the distributed computing frameworks Hadoop, HaLoop, iMapReduce, Spark and Twister.

	Input Data	**Centroids**	**Convergence**	**Advantages**	**Disadvantages**
KM-Hadoop (*our work*)	Re-loaded in each iteration	Broadcasted to workers' local-disks in each iteration via DistributedCache.	The driver compares old and new centroids and decides whether or not to start a new iteration.	Computational complexity is reduced. Final output is intact. Strong fault-tolerance.	Large I/O and network overheads. Sensitive to the structure of the dataset.
KM-HaLoop	Loaded once to local disks, reused over next iterations	Broadcasted to each mapper's local-disk in each iteration.	Reducers keep centroids from the previous iteration and compares them with centroids from the current iteration.	Reusable input data over iterations. Convergence is checked in the reduce phase	K-Means does not benefit from reducer input cache.
KM-iMapReduce	Loaded from HDFS to local disks once, reused in following iterations	Reducers load centroids directly to mappers' local-disks without writing them to HDFS.	Same as KM-Hadoop	Long-running tasks. Low I/O and communication overheads.	K-Means does not benefit from asynchronous execution of map tasks. Number of map and reduce tasks must be identical.

Continued on next page

	Input Data	**Centroids**	**Convergence**	**Advantages**	**Disadvantages**
KM-Spark	Cached in-memory in 1st iteration, reused over next iterations.	Cached in the memory of each mapper in each iteration.	Same as KM-Hadoop	Data points cached in-memory. Low I/O and communication overheads. Long running tasks	Not efficient when input data does not fit into memory.
KM-Twister	Cached in-memory in 1st iteration, reused over next iterations	Same as Spark	New operation *Combine* aggregates output from all reducers and checks for convergence.	Data points cached in-memory. Long running tasks. Efficient in checking convergence.	Weak fault-tolerance. Static and intermediate data must fit in the distributed memory.

Table 2.3: A comparison between the optimised K-Means implementation on Hadoop using triangle inequality (this research) and the standard K-Means implementations based on iterative MapReduce platforms including HaLoop, iMapReduce, Spark and Twister.

Chapter 3

Parallel and Distributed K-Means

This chapter aims to provide a general overview over various parallel implementations of K-Means on different parallel environments. Furthermore, the chapter covers the related work to K-Means on Hadoop and Spark. To have a full understanding of how K-Means works on Hadoop and Spark, a detailed description of the implementation steps of Naive K-Means on Hadoop and Spark will be introduced. Understanding these implementations is of importance as the next chapter describes the proposed approaches to improve the efficiency and scalability of Naive K-Means on Hadoop and Spark.

The remainder sections in this chapter are organised as follows. The first section overviews the related work to implementations of Naive K-Means on different parallel environments. The following section presents the related work to parallel K-Means implementations based on distributed computing platforms, such as Hadoop and Spark. A detailed explanation of the parallel implementation of Naive K-Means on Hadoop and Spark is presented in the following section. The final section summarises the work that has been presented in this chapter.

Abbreviations

Note that the names of the implemented algorithms on Hadoop and Spark in this chapter and the remainder chapters are abbreviated, where each abbreviation consists of the following parts:

[K-Means version]-[distributed framework]-[approach to pass information]

For example, CMP-H-BF denotes the algorithm Compare-means on Hadoop using a Bounds File. If the last part (approach to pass information to the next iteration) is not present, this means the algorithm does not pass any information. For example, the implementation of Naive K-Means on Spark is denoted as NKM-S.

3.1 Related Work to Parallel K-Means Implementations

As one of the most popular and influential data mining algorithms, K-Means has been extensively researched and extended to cope with the rapid growth of data. As previously mentioned, one of the drawbacks of K-Means is its poor scalability as the data grows larger in terms of the number of data points n, dimensions d, and clusters k. This section reviews the previous studies that implemented K-Means on various parallel models.

Several approaches [92] [20] [93] parallelised K-Means based on Message Passing Interface MPI [94]. For example, Dhillon and Modha [92] introduced a parallel implementation of K-Means on distributed memory multiprocessors based on MPI. Their method partitions the original dataset into a number of subsets. Then, each processor processes an independent subset where distance calculations are performed between points and centroids and each point is assigned to its closest centroid. Then, partial sums and partial SSEs are collected and new centroids are calculated. This process is repeated until the algorithm converges. In [93], the authors also implemented K-Means based on MPI using Erlang language which communicates through hundreds of active processes via MPI and adopts concurrent functional paradigm.

In [20], the authors presented a parallel version of a square-error clustering algorithm called *P-CLUSTER* (P is for parallel) on a *network of workstations (NOW).* The implementation of P-CLUSTER is based on the client-server network infrastructure. The server partitions the input dataset into blocks where each block is assigned to a client process. The algorithm proceeds by sending the initial set of centroids to each client process. Then, each client assigns each data point in the block to its nearest centroid. After the assignment phase, each client calculates the partial sum of points assigned to each cluster (block partial sum) and sends it to the server where the new centroids are computed and returned to the clients starting a new iteration of assignments. This process is repeated until convergence or an early termination condition is met. Three approaches were proposed to prune unnecessary distance calculations: 1) computing spheres of guaranteed assignment for cluster centroids, 2) computing the maximum movement effect for patterns across iterations, 3) and maintenance of partial sums for centroids. Since these optimisations are related to triangle inequality, they were already discussed in section 2.3.1.1. The experimental work shows improvement in speed while increasing the size of data in terms of the number of data points, dimensions, and the number of clusters.

The work presented in [21] is realted to the work presented in this project in terms of testing the scalability of variants of K-Means that uses triangle inequality to reduce the number of distance computataions on a parallel environment. The authors presented Annular K-Means and Heap K-Means as two optimised versions of K-Means using triangle inequality. Annular K-Means and Heap K-Means along with other variants, including Lloyd's K-Means [18], Compare-means, Sort-means [66], Elkan's [19], Hamerly's [67], and adaptive K-Means algorithm [95], were parallelised on a *multithreaded* fashion to test the scalability of each algorithm. A uniformly distributed dataset was used as an input in the experiments with variable number of clusters (k), and constant number of data points ($n = 10^6$), and dimensions ($d = 8$). The speedup of each algorithm was measured as a function of the number of threads t, where speedup for t number of threads is defined as the ratio of the running time on a single thread and the running time on t threads. The results show that Naive K-Means is the most algorithm that benefits from increasing the number of threads where its speedup is approximately linear to the number of threads. The authors attributes this result to two reasons:

1. Synchronisation between threads, where Naive K-Means requires a small amount of synchronisation.
2. The work per-thread in Naive K-Means is the most predictable, while in the optimised versions it varies from one thread to another.

This section reviewed some of the works that are related to the work in this project with respect to implementations of K-Means on various parallel processing models. The following section presents the related work to this project with a concentration on implementations of K-Means based on MapReduce and distributed computing platforms that are similar to MapReduce.

3.2 Related Work to Parallel K-Means Based on Distributed Computing Frameworks

This section reviews the work related to this project with an emphasis on implementations of K-Means on MapReduce and other distributed computing frameworks such as Spark and Twister.

Zhao *et al.* [96] implemented K-Means on MapReduce with a combiner. In their experimental work, the authors show that K-Means performed well in terms of speedup, scaleup, and sizeup with datasets that varies in size (1-8 GB) on variable number of nodes (1-4 nodes). However, no further details were provided regarding the nature of the datasets, e.g. the distribution of the data, the number of data points n, or the number of dimensions d. Furthermore, the number of clusters k was not mentioned. In [97] [98] [99], K-Means was implemented based on the approach used in [96]. In [98], the created clusters are resized in the reduce phase based on the number of data points in each cluster. That is, if the number of points in a cluster is bigger than a certain threshold, a predefined number of data points are reassigned from this cluster to a cluster that has a low number of data points. This contradicts the main concept of clustering which is grouping similar objects together, without trying to create balanced clusters. In this work we take these straightforward implementations one step further and enhance the efficiency of K-Means by applying triangle inequality optimisations.

Esteves *et al.* [100], evaluated the performance of parallel naive K-Means on Apache Mahout [101], a library on Hadoop to support machine learning algorithms. The authors reported that K-Means scales well on Mahout if the process involves large-scale datasets. On the other hand, when datasets are small in size, the overhead created by writing replicas, job initialisation, and shuffling intermediate data, becomes the dominant cost.

In [102], the authors presented an implementation of parallel K-Means on Twister [23]. Twister is an optimised MapReduce framework that supports iterative algorithms based on publish/subscribe messaging infrastructure and caches static data in-memory of compute nodes, explained in detail in section 2.7. The aim of their work was to efficiently cluster high dimensional social images. Triangle inequality was used to reduce the number of distance computations based on Elkan's [19] work. Except that instead of keeping a number of lower-bounds that is equal to the number of clusters (k) for each point as in Elkan's algorithm, fewer number of lower-bounds was chosen. For example, in one test where the$k = 3200$, the algorithm was tested with 400 and 800 lower-bounds instead of 3200 to investigate the impact of keeping a number of lower-bounds that is smaller than k. The experimental results showed notable reductions in the number of distance computations. The paper, however, does not describe how the extra information (cluster assignments and distance bounds) was cached. It is expected, however, that the extra information was cached in-memory at the end of the map phase. It can be noticed from this paper how the in-memory caching in Twister can be beneficial to K-Means. The challenge in this paper would be to cache all k lower-bounds instead of a just a small portion and show how Twister would react in case the bounds does not fit in memory. Note that the idea of choosing a number of lower bounds less than k was presented in other studies (e.g. [67] and [95]) and some of them swere reviewed in section 2.3.1. The focus of this paper is on the effect of using triangle inequality optimisations on the amount of shuffled intermediate data between mappers and reducers. Limited information was introduced about the impact of these optimisations on the running time. Moreover, the experimental analysis does not investigate the effect of the increase in dimensionality on the performance. The work presented in this research focuses on the impact of the number of clusters, dimensions, data points, and mappers on the performance of several K-Means implementations Hadoop and Spark.

K-means|| or Scalable K-mean++ [103] and Competitive K-Means [104], are two approaches that implements the popular K-Means++ (explained in section 2.3) algorithm on MapReduce. The two approaches address one of K-Means++ downsides which is its inherently sequential nature. The approach of K-means|| is to sample $O(k)$ points in each round instead of sampling one point as in K-means++. This process is repeated for approximately $O(logn)$ rounds. At the final round, the algorithm produces $O(klogn)$ points. These points are then clustered again into k initial centroids for the Naive K-Means iteration. Both K-means|| and Competitive K-Means can gain significant benefits from our work. Since the process of computing distances from points to centres is independent from the centroids initialisation methods presented in these algorithms, our work can be integrated with these techniques to improve the algorithm's accuracy in addition to efficiency and scalability.

The work in [105] proposed a K-Means implementation on MapReduce that attempts to eliminate the iteration dependence and uses only three MapReduce jobs. In the first job, the input dataset is sampled using k as the number of clusters and probability $p_x = \frac{1}{\varepsilon^2}N$, where N is the number of data points, and $\varepsilon \in (0, 1)$ controls the sampling size. The sampling phase produces $2k$ samples. The mappers of the second job perform the clustering on these samples using k centres and generate $2k^2$ centres in total. These centres are sent to a single reducer in order to be merged into k final centres. The analysis shows that this algorithm outperforms traditional K-means, and Kmeans|| [103] in terms of the total running time. However, it is not clear how to choose the value of ε. One of the advantages of the work presented in this thesis is that there is no need to present new parameters, such as ε in this paper. Although some optimisations that present new input parameters could achieve significant efficiency and clustering quality improvements, these approaches are not practical in real-world settings where the focus is to simplify the existing solutions and not vice versa.

In [106], K-Means was implemented on MapReduce and its efficiency was improved by using locality sensitive hashing (LSH) to divide points into buckets where the original points are transformed into the weighted representative points. This method is used to prune unnecessary distance computations by computing the distance of a given point to only a small number of centres that exist in the same bucket as the point. The algorithm was tested with real datasets and showed improvement in speed by 67% and 76% when k was 1500 and 3000 respectively, compared to scalable K-Means++. However, the algorithm was tested with datasets with 26 and 41 dimensions which does not give an insight on the algorithm's behaviour with data in higher dimensions. While LSH approaches find approximate nearest neighbours rather than exact neighbours, triangle inequality approaches compute the exact distance from points to centroids which is leads to more accurate solutions. This is one of the reasons triangle inequality approaches are adopted in this work.

Lee *et al.* [24], presented a comparison between four iterative algorithms, including K-Means, on five different distributed frameworks including three disk-based systems: Hadoop, HaLoop [74] and iMapReduce [80]; and two memory-based systems: Twister [23] and Spark [22]. In the experimental work, the total elapsed time, the total HDFS read, and the reduce shuffle I/O were measured for each pair of system-algorithm. In addition, the normalised time is measured while varying the size of datasets, and the number of iterations. Finally, the impact of data skew on the total elapsed time was measured. K-Means algorithm was executed on a real-world dataset where $n = 147,251,521$, $d = 3$, and the number of k was not mentioned. In the results section, the normalised time of K-Means on Hadoop is better than Spark when the data size was increased. The reason was not explicitly mentioned but general observations were reported and the reason can be derived from these observations. One remark with regards to testing the effect of the data size was that as the data size increased and the memory-based systems start using disk (because the dataset does not fit in the main memory), the elapsed time of memory-based systems in some cases increased rapidly as a consequence. Another observation was that the *garbage collection* in memory-based systems can significantly affect the execution time when it is triggered. It was reported that, in general, Spark performed better than the other four systems (i.e. Hadoop, HaLoop, iMapReduce, and Twister). Another interesting remark is that some experimental results on iMapReduce were missing. The authors attribute this to the instability of the iMapReduce framework where some tests stopped while execution with no obvious reason. This paper reinforces our decision to adopt Hadoop and

Spark as the distributed systems in our work because of their advantageous features (particularly, their excellent stability and strong fault-tolerance) over other systems.

The study in [107] compares MapReduce and Spark in terms of three major architectural components: shuffle, execution model, and caching. On both frameworks, five algorithms were tested: Word Count, Sort, K-Means, linear regression, and PageRank. In K-Means, three artificially generated datasets were used as input where each point has 20 dimensions and the number of data points for each dataset are: 1 million, 200 million, and 1 billion. The results showed that K-Means on Spark was 1.5x faster than K-Means on MapReduce in the first iteration, and 5x faster in subsequent iterations. This is because of RDD caching in Spark (explained in section 2.6), where input data is transformed into RDDs and cached into memory in the first iteration. Then, subsequent iterations read input data directly from memory which eliminates disk I/O that Hadoop suffers from. In our work, the performance of K-Means on Hadoop and on Spark is also compared. By optimising K-Means using triangle inequality we were able to speedup the standard K-Means on Hadoop to the point where the average iterations time is almost equal to the average iteration time of the standard K-Means on Spark.

In [108], the implementation of K-means-based clustering algorithms (e.g. fuzzy c-means) on Spark was described. Two ways were provided for loading input data: 1) if each instance of the data is represented by all features, data points are loaded to RDDs as dense vectors; and 2) if each instance is represented by a $< AttrID, Val >$ tuple ($AttrID$ is the ID of the attribute, and Val is the corresponding attribute value), data points are transformed into sparse vectors. The experimental work compared the clustering quality of the new implementation with CLUTO [109], a software package that runs clustering algorithms on a single machine, and the reported clustering quality was described as satisfactory. The new implementation achieved 1.5x speedup compared to MLlib's implementation of K-Means on Spark. Furthermore, testing the new implementation while varying the number of tasks (RDD partitions) showed that the decrease in the running time of the new implementation is approximately *log* linear with the increase of the number of tasks. This project accelerates K-Means on Spark using a simple triangle inequality approach described in section 4.6. The experimental work shows that this approach is 7x faster than standard K-Means while maintaining the exact output as the standard K-Means.

The work in [110] compares the performance of K-Means on Hadoop using Mahout [101], and Spark using MLlib [87]. The experiments are run on one and two nodes with two datasets of sizes 64MB and 1240MB. The results show that K-Means on Spark performs faster than K-Means on Hadoop because of the RDD caching mechanism on Spark. However, the experimental work is very limited and does not give the reader enough information about the behaviour of K-Means on both frameworks with respect to various important parameters such as, number of clusters, number of data points, number of mappers, and number of reducers. The work presented in this thesis tests standard and optimised K-Means algorithms with various parameters and provides a detailed analysis of the overhead generated by each operation in each algorithm.

The literature shows that most of the works have studied the behaviour of the Naive K-Means on Hadoop and compared its performance with implementations of K-Means on other parallel models such as Spark and Twister. Furthermore, some works have attempted to improve the clustering quality of K-Means on Hadoop by implementing several techniques that enhance the choice of the initial set of centroids. Other studies have tried to speedup the running time of K-Means on Hadoop by implementing heuristic methods that reduce the number of iterations. To our knowledge, no attempts have addressed the issue of improving the efficiency of K-Means on Hadoop by using triangle inequality approaches to remove redundant distance computations between points and cluster centroids. Since opti-

Notation	Description
X	Input dataset of size n
C	Set of cluster centroids of size k
c_j	Cluster centroid, where $c_j \in C$, with $1 \leq j \leq k$
c'_j	New location of centroid c_j
c_a	Closest centroid to data point x, where $1 \leq a \leq k$

Table 3.1: Description of notations used in implementations of NKM-H.

misations based on triangle inequality produce promising results on the sequential versions of K-Means, this project takes these optimisations one step further and proposes different techniques to implement them on Hadoop.

The following sections will describe the implementation of Naive K-Means on Hadoop using different approaches.

3.3 Implementation of Naive K-Means on Hadoop (NKM-H)

This section provides a detailed description of the implementation of Naive K-Means on Hadoop (*NKM-H*). Three types of implementations of NKM-H are covered:

1. NKM-H using the standard MapReduce model.
2. NKM-H using a combiner.
3. NKM-H using in-mapper-combiner.

The combiner and in-mapper-combiner are two techniques that aim to enhance the communication overhead by reducing the amount of intermediate data that is transferred from mappers to reducers.

Table 3.1 describes the notations that are used in NKM-H implementations.

3.3.1 Implementation of NKM-H with Basic MapReduce Model

This section describes the implementation of K-Means on Hadoop using the basic or standard MapReduce programming model. The standard programming model in MapReduce means that *map* and *reduce* methods are implemented without any extensions or optimisations on these basic operations. The following section will explain the implementation of the driver, mapper, and reducer in K-Means.

Driver

The driver takes the number of clusters k and the path to the input dataset as input parameters. The pseudocode in Algorithm 3 describes the main steps in the driver. The driver starts by randomly selecting the initial set of centroids from the input dataset in line 1. Then, in line 5 the centroids file is sent to all computing nodes in the cluster through Hadoop's *DistributedCache*. The DistributedCache is a Hadoop component that copies required files by applications to the local-disk of each data nodes in the cluster (explained in section 2.5.4). Once a file is broadcasted through DistributedCache, it is copied to the local disk of each node that processes the MapReduce job. This process is performed before any map or reduce task starts. As explained in section 2.5.4, the Distributed Cache tracks the modification timestamps of cache files to ensure that the cached files are consistence. The mapper, combiner, and reducer classes are then set in lines 6-8. Next, the MapReduce job is triggered in line 9. If the job contains more than one reducer, the centroids' files that were produced by all reducers are merged into one file in line 11. The centroids' files are merged to facilitate caching and reading the centroids in the next iteration.

The convergence status is checked in line 13 by retrieving the value of a user-defined counter from the reducer (Counters are explained in section 2.5.3). In the reducer, a user-defined counter called *centresCounter* is defined to count the

Algorithm 3: Driver(X, k)

```
1  $C \leftarrow$ select $k$ initial cluster centroids from $X$ randomly
2  $iteration \leftarrow 1$
3  $centresCounter \leftarrow 1$ //stores the count of converged centroids
4  while $centresCounter > 0$ or an early termination condition is not met do
5      send the set of centroids $C$ to all computing nodes through
         DistributedCache
6      set mapper to NKM–H-Mapper //Algorithm 4
7      set combiner to NKM-H-Combiner (if applied)// Algorithm 6
8      set reducer to NKM-H-Reducer //Algorithm 5
9      run the MapReduce job
10     if $numberofreducers > 1$ then
11         merge reducers output into one file
12     end
13     $centresCounter \leftarrow$ get the value of $centresCounter$ for the current
         iteration
14     $iteration \leftarrow iteration + 1$
15 end
```

number of converged centroids. That is, each reducer increments the counter's value by one if the centroid does not converge, else the counter's value is set to 0. Thus, the value of *centresCounter* that is retrieved in the driver represents the number of converged centroids across all reducers. If the counter's value is zero, this means all the centroids have converged and the loop stops; otherwise a new iteration is started.

Mapper

The MapReduce framework assigns each input-split received from the HDFS to an individual mapper. The size of each input-split can be predefined by the user through using the following two Java methods:

- `TextInputFormat.setMinInputSplitSize(job, size)`, and
- `TextInputFormat.setMaxInputSplitSize(job, size)`,

where job is the job to modify, and size is the desired input-split size.

Algorithm 4: NKM-H-Mapper(k)

```
Function setup():
    load centroids from DistributedCache to $C$
Function map(offset, value)
    $x \leftarrow value$
    $minDistance \leftarrow \infty$
    $a \leftarrow -1$
    for $j \leftarrow 1$ to $k$ do
        $d \leftarrow \mathrm{d}(x, c_j)$
        if $d < minDistance$ then
            $minDistance \leftarrow d$
            $a \leftarrow j$
        end
    end
    output$(a, x)$
```

Each mapper contains three functions, *setup*, *map*, and *cleanup*. While the *map* function is invoked for each record in the input-split, *setup* and *cleanup* are executed only once on each run of the mapper class. As shown in Algorithm 4, *setup* reads the set of centres from DistributedCache and loads them to the data structure c. Then, the *map* function takes as an input, key-value pairs where the key is the offset of the data point in the input file, and the value is the data point itself. Subsequently, the *map* function, in lines 6-12, iterates over C to find the centroid with the minimum distance from the input data point. In line 13, the index of the closest centroid (a) is emitted to the reducers with its assigned data point as a key-value pair.

Reducer

After each mapper outputs a key-value pair, these pairs are grouped by key and sent to the reducer in the form of (*key*, *list(values)*) pairs, where *key* is the cluster index j and *values* are the data points that were assigned to this centroid c_j by the mappers.

The number of reducers can be determined by the user. As in the mapper, the

Algorithm 5: NKM-H-Reducer(k)

```
1  Function setup():
2      load centroids from DistributedCache to $C$
3      $centresCounter \leftarrow 0$
4  Function reduce($j$, $values$):
5      $pointsCounter \leftarrow 0$
6      sum ← (0,0,...,0)
7      foreach $x \in values$ do
8          sum ← sum + $x$ //vector sum
9          $pointsCounter \leftarrow pointsCount + 1$
10     end
11     $c'_j \leftarrow$ sum/$pointsCounter$
12     load $c'$ to $C'$
13     if $c'_j \neq c_j$ then
14         //not converged yet
15         increment $centresCounter$ by 1
16     end
17 Function cleanup():
18     write all new centroids in $C'$ to HDFS
```

reducer also contains three functions: *setup*, *reduce*, and *cleanup*. In Algorithm 5, the *setup* function loads the set of centroids C, and initialises C', which holds the set of updated centroids. In the loop from lines 7 to 10, the vector sum of all the points in the list is calculated and stored in **sum**. Then, the updated centroid, which is represented by the mean of the data points in each cluster, is calculated in line 11 by dividing the **sum** over the count of the points in each cluster. The test in lines 13 compares the new and old values of the centroid. If the values are the same, this means the centroid has not move and the value of the counter *centresCounter* remains zero; otherwise, the centroid has moved and *centresCounter*'s value is incremented by one. Since each centroid is processed by an individual reducer, each reducer will set the value of *centresCounter* to 0 or 1. The driver will then aggregate the produced values of *centresCounter* that are set by each reducer. If the aggregated value of *centresCounter* is zero, this means that all the centroids have converged and the algorithm stops, otherwise, at least one centroid has not converged yet and a new iteration is started.

Algorithm 6: NKM-H-Combiner(k)

```
1  Method setup():
2      load centroids from DistributedCache to $C$
3  Method reduce($j$, $list(values)$)
4      $pCount \leftarrow 0$ //holds the partial count of data points in each cluster
5      pSum $\leftarrow$ (0,0,0,...,0) //holds the partial vector sum of data points in
         each cluster
6      foreach $x \in values$ do
7          pSum $\leftarrow$ pSum $+ x$
8          $pCount \leftarrow pCount + 1$
9      end
10     output($j$, ($pSum$,$pCount$))
```

3.3.2 Implementation of NKM-H with a Combiner

The combiner is an optional component in Hadoop that aims to reduce the amount of intermediate data shuffled from mappers to the reducers across the cluster. The combiner achieve this aim by performing partial aggregations on the intermediate data the mapper has just processed. If the operations on intermediate data are associative (grouping of numbers is not important) and commutative (order of numbers is not important), then the reducer can work as a combiner. In the case of K-Means, these two properties do not hold when the mean value of data points associated to each cluster is computed. Therefore, a separate combiner class must be implemented.

As explained in section 2.4, there is no guarantee on how many times the combiner will run as it might not run at all. Therefore, this issue must be considered on the reducer's implementation in a way that the reducer must be able to run and produce the correct output even if the combiner does not run. Algorithm 6 shows the pseudocode of the combiner class.

The implementation of the combiner class is almost the same as the reducer. However, to make the reducer works properly with or without the presence of the combiner, small changes have to be made to the mapper and reducer classes in Algorithms 4 and 5, respectively. In fact, the implementation of both, mapper and reducer, using a combiner and with a in-mapper-combiner is almost the same as their implementation in NKM-H. For this reason the parts of mapper and reducer algorithms that need to be changed will be highlighted.

The mapper proceeds as the mapper in Algorithm 4. The only change is in the mapper's output, where the *value* is a compound object that consists of the point and integer one that represents the count of each point:

$$\texttt{output(}a\texttt{, (}point\texttt{,1))}$$

Each combiner receives a pair of *key*-list(*values*), where *key* is the cluster index (a) and each $value \in values$ consists of (*point*,1). The *reduce* function aggregates the partial sum (*pSum*), and the partial count ($pCount$) of the data points in each cluster. In Algorithm 6, the loop from line 6-8 shows the aggregation process on data points in the input *list*(*values*) for each centroid's index j. Line 10 outputs key-value pairs where key is the cluster index j, and value consists of the *pSum* and *pCount* of the data points assigned to j.

After the execution of the combiner, the reducer receives *key*-list(*values*) pairs, where *key* is the centroid's index j and each $value \in values$ is composed of the *pSum* and $pCount$. The reducer always expects to receive the *pSum* and $pCount$ for each value. However, if the combiner is not invoked, the reducer's input comes directly from the mapper and each value in the list of *values* is composed of the data point and integer one, i.e (x,1). In this case, the reducer's would not be affected by the absence of the combiner because *point* replaces *pSum* and integer one replaces *pCount*. Therefore, the changes in reducer Algorithm 5 start from line 7, where the *for* loop iterates over each $value \in values$ instead of $x \in values$, and lines 8 and 9 become:

$$count \leftarrow count + value.get(pCount)$$

$$\mathbf{sum} \leftarrow \mathbf{sum} + value.get(\mathbf{pSum}),$$

where the values of *count* and *sum* will be extracted from each *value* in the received list(*values*.

By sending partial sums and counts, the amount of intermediate data sent from mappers to reducers is reduced. In addition, the reducer spends less time aggregating those partial sums and counts.

3.3.3 Implementation of NKM-H with in-mapper-combiner

An alternative technique of the combiner is called *in-mapper-combiner* [5], where the process of computing partial sums and counts for data points that belong to each cluster centroid can be done inside the mapper itself. As can be seen in Algorithm 7, the idea is to define two data structure (e.g. lists or arrays) of size k in the mapper. The first data structure is called $pSum$ and stores the partial sums of data points, and the second is called $pCount$ and stores the partial counts of points assigned to each cluster. Each time the *map* function assigns a data point to a cluster centroid, this data point is summed up with the data points that were assigned to the same cluster centroid in previous *map* calls, and the count of the assigned data points to this cluster is incremented by one. Instead of emitting each point with its assigned cluster centroid to the reducer at the end of the *map* function, the contents of $pSum$ and $pCount$ are emitted at the *cleanup* function, which is invoked once on each run of the mapper class.

The implementation of the reducer is identical to the reducer in NKM-H using a combiner in the previous section. The reducer operates on the partial sums and partial counts of points.

Unlike the combiner, the programming style in the in-mapper-combiner does not need to implement a separate reduce function. In addition, it is guaranteed that intermediate data is aggregated before it is shuffled. However, a scalability bottleneck can be caused. If the number of k is too large, the node that runs the mapper must have a sufficient memory size to hold partial sums and counts [5], or an out-of-memory exception is expected.

Algorithm 7: NKM-H-InMapperCombiner(k))

```
Function setup():
    load centroids from DistributedCache to $C$
    consider $pSum$ a list that holds the partial sums of points in each
      cluster
    consider $pCount$ a list that holds the partial counts of points in each
      cluster
Function map(offset, value)
    $x \leftarrow value$
    $minDistance \leftarrow \infty$
    $a \leftarrow -1$
    for $j \leftarrow 1$ to $k$ do
        $d \leftarrow \mathrm{d}(x, c_j)$
        if $d < minDistance$ then
            $minDistance \leftarrow d$
            $a \leftarrow j$
        end
    end
    $pSum_a \leftarrow pSum_a + point$
    $pCount_a \leftarrow pCount_a + 1$
Function cleanup():
    for $j \leftarrow 1$ to $k$ do
        output($j$, ($pSum_j$,$pCount_j$))
    end
```

3.4 Implementation of Naive K-Means on Spark (NKM-S)

Apache Spark is considered as one of the most important distributed computing frameworks that gained huge popularity during the last five years. One of the motivations behind Spark's design is to overcome the limitation on Hadoop as iterative machine learning algorithms are not directly supported on Hadoop. Spark provides an efficient abstraction for in-memory distributed computing called Resilient Distributed Dataset (RDD). RDDs can be transformed to new RDDs and actions can be performed on each RDD. This feature in Spark could be useful when running K-Means because K-Means needs to read all the input data in each iteration in order to preform clustering. This section explains the implementation

Naive K-Means on Spark (NKM-S) and its performance will be compared with multiple implementations of K-Means on Hadoop. See section 2.6 for more details about Spark's main abstractions and architecture.

An implementation of K-Means is provided in MLlib [87], which is library shipped with Spark that provides various types of machine learning algorithms, including K-Means. However, the provided version of K-Means does not show the full implementation of the algorithm where the user needs to provide only the path to the input dataset, number of clusters, and maximum number of iterations. Therefore, an implementation of K-Means on Spark is provided to explore how Spark operates. Note that since all algorithms are implemented with Java, the description of NKM-S implementation and TIKM-S (will be explained in the next chapter) is from the Java prospective, as other programming languages may differ in some implementation details.

In general, each Spark application must have a driver program that configures Spark's job parameters and performs various parallel operations. The *SparkContext* object allows the application to connect to the Spark computing cluster, and can be used to build RDDs. Each RDD is divided into multiple partitions and can be processed by multiple nodes across the distributed computing cluster. Once an RDD is created, it can be *transformed* to a new RDD, or an *action* can be run on it (transformations and actions are discussed in section 2.6.1).

To let Spark perform a specific computation, functions can be passed to Spark after implementing one of Spark's function interface from a specific package provided in Java. Spark's API provides multiple functions to transform RDDs. As a basic example, consider an RDD called *rdd*, the transformation *rdd.map*() applies a function to each element in *rdd* and the result of the function is the new value of each element in the returned RDD. In addition, many actions can be applied to each RDD in order to return a result to the driver or output data to the distributed file system. For example, the action *reduce*(), takes a function of two input values

Algorithm 8: NKM-S-Driver(X, k)

```
1  $C \leftarrow$ select $k$ initial cluster centroids from $X$ randomly
   /* transform $X$ to an RDD and cache it */
2  JavaRDD<String> $dataRDD \leftarrow$ read input dataset $X$ and cache it
3  De-serialise String values in dataRDD into Vectors
4  $converged \leftarrow false$
5  while $converged == false$ or an early termination condition is not met do
6      JavaPairedRDD<Integer,Vector> $mapPairRDD \leftarrow$
         $dataRDD.mapToPair$(FindClosest$(C, k, point)$)
       /* Count points in each cluster in $mapRDD$ */
7      Map<Integer,Long> $pointsCount \leftarrow mapRDD.countByKey()$
         /* Calculate the vector sum of data points in
         each cluster */
8      Map<Integer,Vector> $pointsSum \leftarrow mapRDD.reduceByKey()$
         /* compute new centroids */
9      for $j \leftarrow 1$ to $k$ do
10         $c'_j \leftarrow pointsSum_j / pointsCount_j$
           /* check for convergence */
11         if all centroids has converged then
12             $converged \leftarrow true$
13         end
14     end
       /* update the centroids list */
15     for $j \leftarrow 1$ to $k$ do
16         $c_j \leftarrow c'_j$
17     end
18 end
```

Algorithm 9: FindClosest(C, k, $point$)

```
1  $x \leftarrow point$
2  $minDistance \leftarrow \infty$
3  for $j \leftarrow 1$ to $k$ do
4      $d \leftarrow$ d$(x, c_j)$
5      if $d < minDistance$ then
6          $minDistance \leftarrow d$
7          $a \leftarrow j$
8      end
9  end
10 return($a$,$x$)
```

of one type and returns a new value of the same type.

Algorithm: The pseudocode in Algorithm 8 illustrates the basic steps to implement K-Means on Spark. In line 3, the input dataset file is read from HDFS. Each data point in the file is represented as a String. Therefore, the data points are transformed into a new RDD called *dataRDD* as Strings. Each data point in *dataRDD* is de-serialised into a Vector. *dataRDD* is partitioned and cached in the memory of the worker nodes so that it could be re-used in subsequent iterations. This step reduces the communication overhead compared to Hadoop's implementation of K-Means. In line 6, each data point in *dataRDD* is assigned to its closest centroid and each cluster index with its associated data point are returned to *mapPairRDD* as a pair of type <Integer,Vector>. Inside the function *mapPairRDD*, the function $FindClosest()$, illustrated in Algorithm 9, is called where the list of centroids C, the number of clusters k, and the data point *point* are passed as parameters. $FindClosest()$ finds the closest centroid from the passed data point and returns the index of the centroid with the data point as a pair of type <Integer,Vector>. At this point each record in *mapPairRDD* is a pair of $< a, x >$ where a is the cluster index of the closest centroid from point x. What is left is to find the count of points in each cluster and the vector sum of these points in order to be able to compute the new centroids. Line 7 counts the points in each cluster and returns the cluster index and the count of points in this cluster to *pointsCount* as key-value pairs. To compute the vector sum of data points in each cluster, line 8 uses the function $reduceByKey()$ which groups the points in *mapRDD* by key and the vector sum of points in each cluster is calculated. The results are returned to *pointsSum* as key-value pairs, where key is the cluster index and value is the vector sum of points in this cluster. At this point the algorithm has acquired the count and the vector sum of data points in each cluster and can proceed with computing the new centroids by dividing the vector sum of points in each cluster over the count (lines 10-15). These steps are then repeated in the following iterations until convergence.

Section 4.6 in the next chapter will present the implementation of Triangle Inequality K-Means on Spark, which is similar to NKM-S with a simple optimisation using a basic triangle inequality approach. The implementation of the driver will be identical to the NKM-S-Driver in Algorithm 8. Therefore, some sections in the next chapter will refer to Algorithm 8.

3.5 Summary

In this chapter, the early sections reviewed the related work to this project in terms of parallel implementations of K-Means on parallel settings in general, and on the standard MapReduce and other distributed systems based on MapReduce in specific. It can be observed from the literature the lack of studies on parallel implementations of K-Means using triangle inequality on the standard MapReduce model. The focus of most studies was on methods that enhance the choice of the initial cluster centroids in order to improve the clustering quality which usually leads to better efficiency. Since the operation of computing point-centre distance is independent, in most cases, from the one that chooses the initial centroids, our work can be integrated with algorithms such as K-means|| and Competitive K-Means. This integration is expected to produce a version of K-Means that is not only efficient and highly scalable, but with better clustering quality.

Furthermore, a detail description of the implementation of Naive K-Means on Hadoop with three different settings was presented. These settings include the standard MapReduce model which uses only map and reduce tasks; a combiner which is a functionality offered by Hadoop to reduce the amount of intermediate data transferred from mappers to reducers; and the in-mapper-combiner which shares the same concept as the combiner except that the intermediate data is combined inside the map stage instead of being implemented in a separate stage. This thesis considers the implementation of K-Means on Hadoop using the standard MapReduce model as the base-line for the accelerated version of K-Means on

Hadoop and Spark. This is because the Naive K-Means and the standard MapReduce model are the straightforward and basic forms of K-Means and MapReduce, respectively.

Although the implementation of Naive K-Means on Hadoop is straightforward, it is a challenging task to implement variants of the same algorithm based on triangle inequality. The reason is because such approaches require passing data from one iteration to the next which Hadoop does not support. The next chapter presents new methods that implement such approaches on Hadoop and explains the deferences between these methods.

Chapter 4

Efficient Parallel K-Means using Triangle Inequality

4.1 Introduction

As it was explained in section 2.4.3, one of Hadoop's limitations is its lack of support to cache intermediate data between two consecutive MapReduce jobs. Several K-Means variants, such as Elkan's algorithm [19], Hamerly's algorithm [67], Drake's algorithm[95], and Compare-means algorithm [66], require information from the previous iteration to use them in the current iteration to eliminate unnecessary distance computations between points and centres. Therefore, this chapter introduces two approaches: K-Means on Hadoop using an Extended Vector, and K-Means on Hadoop using a Bounds File. These approaches aim to allow Hadoop to pass information from one iteration to the next to efficiently accelerate the K-Means algorithm.

In section 2.7, several iterative MapReduce approaches were discussed. Some of these approaches (e.g. Twister and HaLoop) are based on caching the static data in-memory and the reuse of the same data over iterations. One problem with such solutions is that the worker nodes need to have a large memory space to be able to fit the input data. Approaches presented in this chapter, however, require much less memory space since input data and extra information are stored on the distributed file system. In addition, some platforms, such as Twister, has a limited support to fault-tolerance compared to Hadoop because the data are stored on local disks rather than a distributed file system. In general, algorithms implemented on iterative MapReduce platforms could outperform those implemented on the standard Hadoop's MapReduce. However, this improvement in efficiency is usually at the expense of other crucial features such as fault-tolerance.

To evaluate the proposed approaches, Elkan's algorihtm [19], and Compare-means algorithm [66] are implemented on Hadoop using each approach. Since the steps of eliminating unnecessary distance computations for both, Elkan's and Compare-means algorithms, were explained in sections 2.3.1.4 and 2.3.1.3 respectively, the focus in this chapter will be on how these algorithms can be implemented on Hadoop using the proposed approaches.

In general, the assignment of data points to their closest centres in K-Means on Hadoop is the responsibility of the mappers, while reducers are responsible for aggregating points belonging to each centroid and producing the new set of centroids. Therefore, the optimisation steps occur in the map phase and, as a consequence, several mapper algorithms will be discussed in the next sections. On the other hand, the implementation of only one reducer will be discussed because the implementation of the reducer is identical in all of the proposed solutions.

4.2 K-Means on Hadoop using an Extended Vector (EV)

This section explains the use of a data structure called Extended Vector (EV) to pass extra information from one iteration to the next. The idea of the Extended Vector is to append any required information in the current iteration to the original input data vector to form an EV. This EV will be the input in the next iteration, where the input data along with any extra information associated with it, can be read together. Therefore, the Extended Vector can be defined as: a data structure that stores the input data vector and any extra information related to this data vector in a given iteration, in order to be used in subsequent iterations. This can be considered as the straight-forward solution to the problem of passing information between iterations in Hadoop. Two K-Means variants are implemented using this approach, Elkan's algorithm [19] and Compare-means [66]. The following sections will explain the implementation steps for each algorithm on Hadoop using an EV.

4.2.1 Elkan's Algorithm on Hadoop using an Extended Vector (ELK-H-EV)

The implementation of Elkan's algorithm on Hadoop using an Extended Vector is referred to as *ELK-H-EV*. As it was explained in section 2.3.1.4, Elkan's algorithm efficiently eliminates large number of unnecessary distance computations while maintaining the same output as the Naive K-Means. In addition to the need of computing the k^2 centre-centre distances at the beginning of each iteration, the algorithm needs to cache the following information in one iteration and use them in the next:

1. n upper-bounds on the distance between each data point and its assigned centroid.
2. nk lower-bounds on the distance between each data point and each centroid.
3. n cluster assignments for each data point from the previous iteration.

EV Size: Since extra information is associated with each data point, the required information will be appended to the data point, which forms the Extended Vector (EV). Figure 4.1 illustrates the structure of an EV in ELK-H-EV. Each EV in ELK-H-EV consists of:

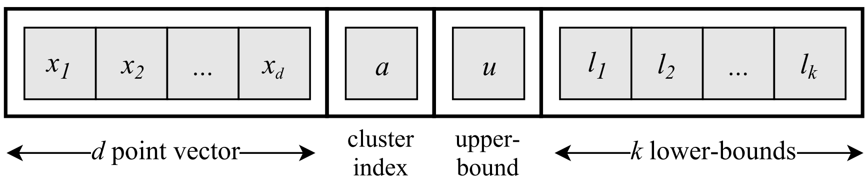


Figure 4.1: Structure of an Extended Vector in ELK-H-EV.

- Data point vector in d dimensions.
- One upper-bound for the distance from the point to its closest centroid.
- One cluster assignment index from the previous iteration.
- k lower-bounds for the distances from the point to each centre.

Therefore, the size of each EV in ELK-H-EV is $d+k+2$. This means that each mapper writes $\frac{n}{p}(d+k+2)$ EVs to HDFS per iteration, where p is the number of mappers.

Algorithm: The implementation of ELK-H-EV can be divided into three major phases:

1. A *driver* that initiates the MapReduce jobs and controls the iterative process,
2. A *map* phase that assigns each point to its closest centroid (distance computation elimination steps occur in this phase), and
3. A *reduce* phase that computes the means of points assigned to each cluster centroid and produces new set of centroids.

Table 4.1 explains the notations that will be used in the pseudo-codes that will be explained in the following sections.

Driver: Algorithm 10 shows the pseudo-code that describes the driver's implementation, where it is mostly similar to the NKM-H's driver (Algorithm 3 illustrated in section 3.3.1). As explained in section 3.3.1 the driver starts by randomly picking the initial k centroids. Then, in the while loop the centroids file is

Notation	Description
X	Input dataset of size n
C	The set of cluster centroids of size k
k	Number of clusters
c_j	Cluster centroid, where $c_j \in C$, with $1 \leq j \leq k$
c'_j	New location for centroid c_j
c_a	Closest centroid to data point x, where $1 \leq a \leq k$
$s_{i,j}$	Distance between centroids c_i and c_i, where $1 \leq i, j \leq k$ and $i \neq j$
h_j	Half minimum distance from c_j to its closest centroid
m_j	The distance that centroid c_j has moved in the last iteration, i.e. d(c_j,c'_j)
u	An upper-bound from data point $x \in X$ to its closest centroid c_a
l_j	A lower-bound from data point $x \in X$ to centroid c_j
w	An Extended Vector class object, which stores the data vector $w.x$ ($x \in X$) and required extra information (e.g. $w.a$, $w.u$)
b	A collection (e.g. array, list) of all distance bounds and cluster assignments associated to each data point.
p	Number of mappers

Table 4.1: Description of notations and data structures used by ELK-H, CMP-H, and TIKM.

sent to all computing nodes in the cluster through *DistributedCache*, which is a component in Hadoop that copies desired files to all the computing nodes (section 2.5.4 explains *DistributedCache* in detail). The driver then sets the mapper to ELK-H-EV-Mapper-1 in the first iteration, and to ELK-H-EV-Mapper-2 in later iterations. Following that, the reducer is set to the Reducer in Algorithm 13. Next the MapReduce job is submitted. Before starting a new iteration, the output from each reducer (newly computed centroids) is merged with the output from all other reducers into one file in case of multiple reducers. This file becomes the input centroids file for the next iteration. The final step is to check the convergence

Algorithm 10: Driver(X, k)

```
1  select $k$ initial cluster centroids randomly
2  $iteration \leftarrow 1$
3  $centresCounter \leftarrow 1$
4  while $centresCounter > 0$ or an early termination condition is not met do
5      send the centroids' file to all computing nodes through DistributedCache
6      if $iteration == 1$ then
7          set mapper to ELK-H-EV-Mapper-1 //Algorithm 11
8      else
9          set mapper to ELK-H-EV-Mapper-2 //Algorithm 12
10     end
11     set reducer to Reducer //Algorithm 13
12     run the MapReduce job
13     if $numberOfReducers > 1$ then
14         merge reducers output into one file
15     end
16     $centresCounter \leftarrow$ get the value of $centresCounter$ for the current
         iteration
17     $iteration \leftarrow iteration + 1$
18 end
```

status through checking the value of *centresCounter*. The iteration runs until convergence or an early termination condition is met (section 3.3.1 gives more details about checking the convergence status).

Note that only minor changes will occur in the driver's implementation for algorithms presented later in this chapter in comparison with the implementation of this driver (Algorithm 10). Therefore, the implementation of this driver will be referenced in the explanation of later algorithms and the changes will be pointed out.

Map phase: The map phase is responsible of assigning each point to its closest centroid. ELK-H requires two mapper implementations, the first mapper is executed in the first iteration, and the second mapper is executed in subsequent iterations. This is because in the first iteration distance bounds and cluster assignments are not initialised yet. Therefore, the first mapper runs in the first iteration and initialises distance bounds and cluster assignments, and the second mapper runs in subsequent iterations and eliminates unnecessary distance computations.

Algorithm 11: ELK-H-EV-Mapper-1(k)

```
1  Function setup():
2      load centroids from DistributedCache to $C$
3      compute $s_{i,j} \leftarrow d(c_i, c_j)$ //for all $1 \leq i, j \leq k$
4  Function map(offset, point)
5      let $w$ be an Extended Vector
6      let $t$ be a boolean list of size $k$
7      $w.x \leftarrow point$
8      for $j \leftarrow 1$ to $k$ do
9          $t_j \leftarrow false$
10     end
11     $minDistance \leftarrow \infty$
12     for $j \leftarrow 1$ to $k$ do
13         if $t_j$ then continue
14         $d \leftarrow$ d($w.x$, $c_j$)
15         $w.l_j \leftarrow d$
16         if $d < minDistance$ then
17             $minDistance \leftarrow d$
18             $w.u \leftarrow minDistance$
19             $w.a \leftarrow j$
20             for $z \leftarrow j + 1$ to $k$ do
21                 if $s_{j,z} \geq 2 * d$ then
22                     $t_z \leftarrow true$
23                 end
24             end
25         end
26     end
27     write $w$ to HDFS
28     output($w.a$, $w.x$)
```

Note that the detailed explanation of Elkan's method to eliminate unnecessary distance computations is presented in section 2.3.1.4, where the sequential version of Elkan's algorithm is also presented. For this reason this section briefly explains Elkan's approach to prune distance computations, and the main focus will be on how the algorithm is implemented on Hadoop using the proposed methods.

- **First Mapper:** The pseudo-code in Algorithm 11 shows how upper and lower bounds associated with each input data point x are initialised in ELK-H-EV, where w represents an ExtendedVector (EV) class object, with the index of the assigned cluster centroid (a), the upper-bound (u), the lower-bound (l), and the data point (x), as members of w. First, a new Extended Vector (w) is declared in line 5, then, the input data point is assigned to $w.x$.

As previously mentioned, the first mapper initialises the distance bounds and the cluster assignments. This means the first mapper would not eliminate any distance computations and it would be much slower than subsequent iterations. However, a modest number of distance computations, compared to Elkan's method, can be eliminated using Lemma 2.3.1 in section 2.3.1.1. Lemma 2.3.1 simply states that: given two centres p and a, and a point x, if $d(p, a) \geq 2d(x, p)$ then $d(x, a) \geq d(x, p)$. This Lemma can be used to skip the distance computation from $v.x$ to the next centroid in the centroids list. To achieve this, t holds the skip status of each centroid, that is, if the distance computation from $v.x$ to centroid c_j can be skipped, c_j's status in t_j will be *true*, otherwise, it is *false*. First, all values in t are set to *false* in line 9. Line 13 tests the status of the currently processed centroid. The distance computation to this centroid is avoided if its status is true. Lines 14-19 find the closest centroid from $w.x$. Then in line 20, the distance from the current centroid to the next centroid is extracted from structure s, and line 20 tests Lemma 2.3.1 to check if the distance to the next centroid can be eliminated. If the test holds, the skip status of the next centroid is set to *true* and the distance computation to it is skipped.

Each time the distance from point $w.x$ to any centroid c_j is computed, the lower-bound $w.l_j$ that corresponds to c_j is set to this distance. While the upper-bound $w.u$ is set to the distance from $w.x$ to its closest centroid c_a.

In line 27, w is written to HDFS. EVs that are written by each mapper will be the input for the mappers in the next iteration and the *map* function will read the data point with all extra information associated to this point. Finally, the mapper outputs data point ($w.x$) and its assigned cluster index

$(w.a)$ to reducers as a key-value pair.

Note that the input files for the next iteration will be larger in size than the original input dataset the first mapper received because of the size of EVs. This can cause an increase in the number of mappers in the next iteration, which consequently increases the communication time.

- **Second Mapper**: Algorithm 12 illustrates the pseudo-code of the second mapper in ELK-H-EV, which is executed on iterations > 1. The second mapper takes as input key-value pairs, where each value represents an EV that was stored by a mapper in the previous iteration. In lines 9-12, the lower and upper bounds are updated. The distance (m_j) centroid c_j has moved in the previous iteration is added to the upper-bound and subtracted from each lower-bound. The centroid's movement is part of the data structure that holds the centroid's vector and is computed and stored at the end of the reduce stage. If the test in line 15 holds, all distance calculations associated with the currently processed point are skipped. Furthermore, if any of the three tests in lines 14-16 does not hold, the distance computation to currently processed centroid is avoided. The distance from the point $w.x$ to any centroid other than the one assigned to it does not get calculated until line 29, while the tests at line 28 repeats the tests at lines 18 and 19 but with an updated upper-bound $w.u$. At this point w acquires updated values for the assigned cluster index a, the upper-bound u, and the lower-bounds l_j $(1 \leq j \leq k)$ and can be written to HDFS at line 39. Finally, the mapper outputs the point $w.x$ with the index of its closest centroid $w.a$ to the reducers.

Reduce phase: The reduce phase is responsible for computing the new centroids and writes these centroids to HDFS. In section 3.3.1, the pseudo-code for the NKM-H-Reducer (Algorithm 5) is explained in detail. Since the implementa-

Algorithm 12: ELK-H-EV-Mapper-2(k)

```
Function setup():
    load centroids from DistributedCache to $C$
    compute $s_{i,j} \leftarrow d(c_i, c_j)$, for all $1 \leq i, j \leq k$
    compute $h_j \leftarrow min_{j \neq j'} d(c_j, c_{j'}) * 0.5$, for all $1 \leq j, j' \leq k$
Function map(offset, value):
    let $w$ be an Extended Vector
    $w \leftarrow value$
    //update $k$ lower-bounds
    for $j \leftarrow 1$ to $k$ do
        $w.l_j \leftarrow \max[w.l_j - m_j, 0]$
    end
    $w.u \leftarrow w.u + m_{w.a}$ //update upper-bound
    $g \leftarrow true$ //flag to check if $u$ is updated
    $d1$, $d2 \leftarrow 0$
    if $w.u \leq h_{w.a}$ then continue
    for $j \leftarrow 1$ to $k$ do
        if $(j \neq w.a)$
        $\&(w.u > w.l_j)$
        $\&(w.u > s_{w.a,j} * 0.5)$ then
            if $g$ then
                $d1 \leftarrow$ d($w.x$, $c_{w.a}$)
                $w.u \leftarrow d1$
                $w.l_{w.a} \leftarrow d1$
                $g \leftarrow false$
            else
                $d1 \leftarrow w.u$
            end
            if $d1 > w.l_j$ or $d1 > s_{w.a,j} * 0.5$ then
                $d2 \leftarrow d(w.x, c_j)$
                $w.l_j \leftarrow d2$
                if $d2 < d1$ then
                    $w.a \leftarrow j$
                    $w.u \leftarrow d2$
                    $g \leftarrow false$
                end
            end
        end
    end
    write $w$ to HDFS
    output($w.a$, $w.x$)
```

tion of the reducer is identical in all algorithms including NKM-H, this section will briefly explain the implementation of the reducer, and more details are in section

Algorithm 13: Reducer(k)

```
Function setup():
    load centroids from DistributedCache to $C$
    let $C'$ be a list that stores the new centroids
    $centresCounter \leftarrow 0$
Function reduce(a, points):
    $pointsCounter \leftarrow 0$
    sum $\leftarrow$ (0,0,...,0)
    foreach $x \in points$ do
        sum $\leftarrow$ sum $+ x$ //vector sum
        $pointsCounter \leftarrow pointsCount + 1$
    end
    $c'_j \leftarrow$ sum$/pointsCounter$
    load $c'_j$ to $C'$
    if $c'_j \neq c_j$ then
        //not converged yet
        increment $centresCounter$ by 1
    end
Function cleanup():
    write all centroids in $C'$ to HDFS
```

3.3.1.

In Algorithm 5, the reducer receives the index (a) of the cluster as the key and the list of points that were assigned to a as a list of values. Each reducer processes each a with its associated points independently. The reducer iterates over the points to compute the vector sum. After that, the average is computed by dividing the vector sum over the number of points to produce the new centroid. The old and new centroids are compared, if they are not equal or results of the comparison is not under a certain threshold, the *centresCounter* is incremented by one, which consequently makes the driver runs one more iteration.

4.2.2 Compare-means on Hadoop using an Extended Vector (CMP-H-EV)

Compare-means [66] is a variant of K-Means that also uses triangle inequality to skip redundant distance computations. While Elkan's algorithm uses a combination of distance bounds and triangle inequality to eliminate unnecessary distance computations, Compare-means uses only triangle inequality without any distance bounds. The only required information from the previous iteration is the cluster assignment for each data point. The method Compare-means used to reduce the number of distance computations is presented in section 2.3.1.3. The implementation of Compare-means on Hadoop using an Extended Vector is referred to as CMP-H-EV.

As in ELK-H-EV, CMP-H-EV needs to compute k^2 centre-centre distances at the beginning of each mapper. In addition, the algorithm needs to cache one cluster assignment for each data point from last iteration.

Each EV in CMP-H-EV consists of:

- Data point vector of size d dimensions.
- One cluster assignment index from the previous iteration.

Therefore, the size of each EV in CMP-H-EV is $d + 1$. This means that each mapper writes $\frac{n}{p}(d + 1)$ EVs to HDFS per iteration.

Algorithm: The implementation of CMP-H-EV can be also divided into three major phases:

1. A *driver* that initiates the MapReduce jobs and controls the iterative process,
2. A *map* phase that assigns each point to its closest centroid (distance computation elimination steps occur in this phase), and
3. A *reduce* phase that computes the means of points assigned to each cluster centroid and produces new set of centroids.

driver: The driver's implementation is similar to the driver in Algorithm 10, section 4.2.1, but with minor changes. For example, CMP-H-EV has only one mapper, therefore, there is no need to have an *if* statement to invoke two separate mappers' implementations.

Algorithm 14: CMP-H-EV-Mapper(k)

```
1  Function setup():
2      load centroids from DistributedCache to $C$
3      compute $s_{i,j} \leftarrow d(c_i, c_j)$ //for all $1 \leq i, j \leq k$
4  Function map(offset, value)
5      let $w$ be an Extended Vector
6      if $iteration == 1$ then
7          $w.x \leftarrow value$
8          $w.a \leftarrow 1$
9      end
10     $minDistance \leftarrow \mathrm{d}(w.x,\ c_{w.a})$
11     $d \leftarrow 0$
12     for $j \leftarrow 1$ to $k$ do
13         if $s_{j,w.a} \geq 2 * minDistance$ or $j == w.a$ then
14             continue
15         end
16         $d \leftarrow \mathrm{d}(w.x,\ c_j)$
17         if $d < minDistance$ then
18             $minDistance \leftarrow d$
19             $w.a \leftarrow j$
20         end
21     end
22     write $w$ to HDFS
23     output($w.a$, $w.x$)
```

Map phase: Unlike ELK-H-EV, CMP-H-EV has only one mapper because it does not need to initialise any distance bounds. As mentioned previously in this section, the only extra information CMP-H-EV needs from the previous iteration is the index of the assigned cluster to each data point (a), which needs to be initialised in the first iteration. In this situation, a is initialised to 1 in the first iteration for all data points, which is the index of the first centroid in the centroids list C.

The pseudo-code in Algorithm 14 describes the steps of the mapper in CMP-H-EV. First, it can be observed that CMP-H-EV's algorithm is simpler than ELK-H-EV with regards to the method each algorithm eliminates distance computations. This simplicity makes the algorithm lighter than ELK-H-EV in terms of I/O overhead, but this come at the cost of the amount of skipped distance computation.

In the first iteration, the *map* function receives the byte-offset of the input record and the data point vector as a key-value pair. A new Extended Vector (w) is declared in line 5 and the received value (data point) is assigned to $w.x$. The index for the cluster centroid that was assigned to $w.x$ in the previous iteration is initialised to one for all data points, which is the index of the first centroids in the centroids list. Consequently, $minDistance$ in line 10 will be the distance from $w.x$ to the first centroid in the centroids list. Distance computations are avoided if the test in line 13 holds. The test in line 13 uses Lemma 2.3.1 from section 2.3.1.1, which states that: for two centres c_1 and c_2, and a data point x, if we know that $d(c_1, c_2) \geq 2d(x, c_1)$ then $d(x, c_2) \geq d(x, c_1)$, and $d(x, c_2)$ can be avoided. CMP-H-EV performs this test at line 13 using the last centroid that point $w.x$ was assigned to in the previous iteration ($w.a$). If the test does not hold, the distance to the centroid is computed as in NKM-H. Finally, w is written to HDFS, and the pair ($w.a$,$w.x$) is emitted to the reducers.

In iterations > 1, the *map* function receives the value as an EV that contains the data point $w.x$ and cluster index for the centroids that point $w.x$ was assigned to in the previous iteration. The algorithm then attempts to skip distance computations at line 13 as explained earlier.

Note that $minDistance$ is initialised to the distance of the centroid that $w.x$ was assigned to in the previous iteration. This is because in many datasets where data points are distributed into clusters, after the first few iterations, cluster centroids do not move much. Meaning that the last centroid that was assigned to a given data point has more potential to be the closest centroid in later iterations than other centroids.

Reduce phase: The implementation of the reducer is identical to the reducer in section 3.3.1, Algorithm 5.

4.3 K-Means on Hadoop using a Bounds File (BF)

After presenting the first approach which passes information from one iteration to the next in K-Means on Hadoop using EVs, this section introduces the second approach called K-Means on Hadoop using a Bounds File (BF). The idea behind this approach is motivated by the large overhead EVs create when processing large number of clusters and dimensions. Thus, BFs attempt to reduce the overhead from writing EVs to HDFS in each iteration.

A *Bounds File (BF)* can be defined as a flat file that is written to HDFS in each mapper, where each record in this file represents extra information that is associated to a data point in the input dataset. In other words, in a given iteration, each mapper stores the desired extra information related to each input data point on a file on HDFS, this file is called a Bounds File. Unlike implementations that use EVs, each record in a BF stores only the extra information without the data point. These files can then be read by the mappers in subsequent iterations and each point is joined with its corresponding extra information. Figure 4.2 illustrates the dataflow in one iteration of K-Means on Hadoop using BFs.

The following sections explain the implementations of two K-Means variants: Elkan's algorithm, and Compare-means on Hadoop using BFs. The sequential implementations of these variants are discussed in sections 2.3.1.4, 2.3.1.3. In addition, sections 4.2.1 and 4.2.2 explained the implementation steps of both algorithms on Hadoop using EVs (ELK-H-EV and CMP-H-EV) with an explanation of the method each algorithm follows to eliminate distance computations. Therefore, the following sections focus on how to store extra information in one iteration and retrieve it in the next using BFs.

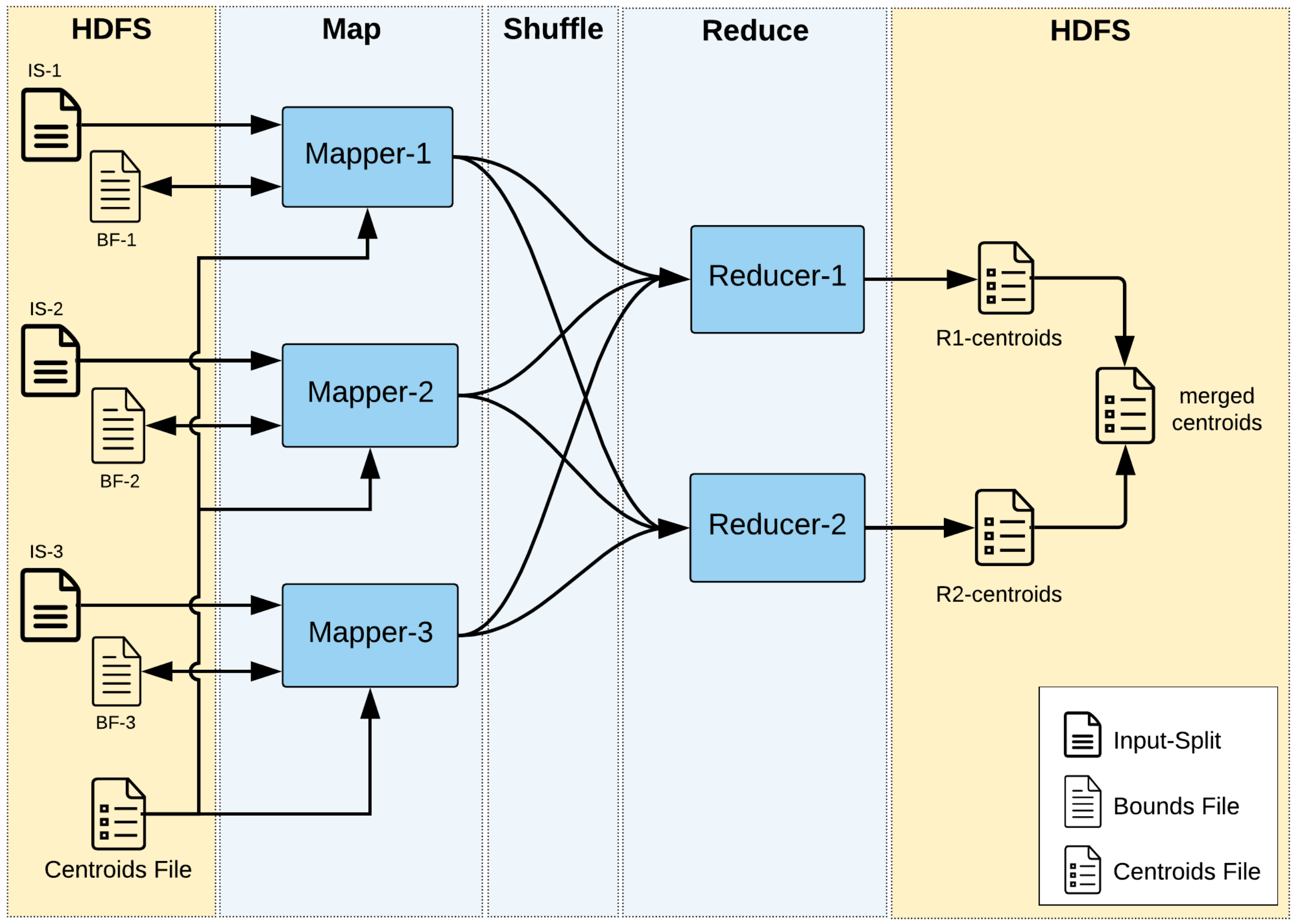


Figure 4.2: Dataflow in one iteration of K-Means on Hadoop using Bounds Files.

4.3.1 Elkan's Algorithm on Hadoop using a Bounds File (ELK-H-BF)

Elkan's algorithm uses a combination of triangle inequality and distance bounds to reduce the number of distance computations. Elkan's algorithm needs to maintain the following information in one iteration and use them in the next:

1. n upper-bounds on the distance between each data point and its assigned centroid.

2. nk lower-bounds on the distance between each data point and each centroid.

3. n cluster assignments for each data point from last iteration.

In a given iteration, each mapper in Elkan's algorithm on Hadoop using a Bounds File (ELK-H-BF) writes one upper-bound, k lower-bounds, and one cluster assignment, that are associated to each data point to a BF on HDFS. In the next iteration, each mapper finds the BF that corresponds the input-split that was assigned to this mapper and loads all the extra information in this BF to memory. At this point, each mapper acquired the extra information that each data point needs to proceed with the elimination process.

How to identify which BF corresponds to which input-split? Hadoop splits the original input dataset into a number of input-splits where each mapper processes an individual input-split. The splitting mechanism does not change from one iteration to another, that is, each input-split contains the same data points in the same order from one iteration to the next. However, the input-split processed by a given mapper in one iteration could be processed by a different mapper on a different node in the next iteration. This issue causes a difficulty in associating each BF to its corresponding input-split. To solve this issue, the BF's name is set to be the *starting byte offset* of the currently processed input-split. Hence, in given iteration, the mapper searches HDFS for the BF with the name that matches the *starting byte offset* of the input-split assigned to this mapper in the current iteration. The contents of the BF are then loaded the memory of the mapper's node. Since the order of the records in the input-split does not change from one iteration to another, the order of the records on the input-split will match the order of records in the corresponding BF.

BF Size: In a given iteration, each mapper in ELK-H-BF writes the following extra information for each data point to a BF:

- One upper-bound for the distance from the point to its closest centroid.
- One cluster assignment index from the previous iteration.
- k lower-bounds for the distances from the point to each centre.

Therefore, each record in a BF in ELK-H-BF is of size: $K + 2$, which makes the size of each BF: $\frac{n}{p}(k + 2)$ per iteration, where n is the total number of data points, and p is the number of mappers.

Algorithm: Similar to ELK-H-EV, ELK-H-BF consists of three major phases:

1. A *driver* that initiates the MapReduce jobs and controls the iterative process,
2. A *map* phase that assigns each point to its closest centroid (distance computation elimination steps occur in this phase), and
3. A *reduce* phase that computes the means of points assigned to each cluster centroid and produces new set of centroids.

Driver: The driver's implementation is similar to the one described in section 4.2.1, Algorithm 10.

Map phase: Similar to ELK-H-EV (described in section 4.2.1), ELK-H-BF requires two mappers' implementations, the first mapper runs in the first iteration and initialises the distance bounds and cluster assignments, while the second mapper runs in subsequent iterations and performs the techniques for eliminating unnecessary distance computations.

Algorithm 15: ELK-H-BF-Mapper-1(k)

```
Function setup():
    load centroids from DistributedCache to $C$
    compute $s_{i,j} \leftarrow d(c_i, c_j)$ //for all $1 \leq i, j \leq k$
Function map(offset, value)
    $x \leftarrow value$
    let $b$ be a collection that stores the extra information for $x$
    for $j \leftarrow 1$ to $k$ do
        $t_j \leftarrow false$
    end
    $minDistance \leftarrow \infty$
    for $j \leftarrow 1$ to $k$ do
        if $t_j$ then continue
        $d \leftarrow \mathrm{d}(x, c_j)$
        $b.l_j \leftarrow d$
        if $d < minDistance$ then
            $minDistance \leftarrow d$
            $b.u \leftarrow minDistance$
            $b.a \leftarrow j$
            for $z \leftarrow j + 1$ to $k$ do
                if $s_{j,z} \geq 2 * d$ then
                    $t_z \leftarrow true$
                end
            end
        end
    end
    write $b$ to a BF on HDFS
    output($b.a$, $x$)
```

- **First mapper:** Algorithm 15 shows the pseudo-code of the first mapper in ELK-H-BF, where most of the steps are similar to the steps in Algorithm 11, except that ELK-H-BF stores and reads extra information to/from BFs.

  The following data structures are introduced in Algorithms 15 and 16:

  1. b: is a collection of all the distance bounds and cluster assignments associated to each data point. In ELK-H-BF, each b is of size $k + 2$ (k lower-bounds, one upper-bound, and one cluster assignment). Note that in CMP-H-BF, the only required information is the index for the assigned cluster from last iteration. This could be assigned to a variable.

However, in the pseudo-code, b is used to avoid confusion, maintain consistency, and to consider this as a general approach to store any information from previous iteration.

2. f: is a list that stores all b's objects for all the points that are processed by a certain mapper (in our implementation an *ArrayList* is used for f). The size of f in ELK-H-BF $\frac{n}{p}(k+2)$, and in CMP-H-BF is $\frac{n}{p}$, where n is the number of data points, and p is the number of mappers. The variable $pointsCounter$ represents the index of b in f, where the order of data points in f matches the order of data points in the input-split.

Each time the distance from the input data point to a given centroid c_j is calculated (line 13), the lower-bound $b.l_j$ is set to that distance in line 14. Additionally, when the distance to the closest centroid is determined, the upper-bound ($b.u$) is set to that distance in line 17, and the index of this closets centroid is assigned to $b.a$ in line 18. At this point all the extra information for point x are acquired and can be written to a BF in line 26.

Algorithm 16: ELK-H-BF-Mapper-2(k)

```
Function setup():
    load centroids from DistributedCache to $C$
    compute $s_{i,j} \leftarrow d(c_i, c_j)$, for all $1 \leq i, j \leq k$
    compute $h_j \leftarrow min_{j \neq j'} d(c_j, c_{j'}) * 0.5$, for all $j \in k$
    let $f$ be a list that stores the cluster assignments for all data points
    find the BF that corresponds to the input-split assigned to this mapper
      and load its records to $f$
    $pointsCounter \leftarrow 1$
Function map(offset, value):
    $x \leftarrow value$
    let $b$ be a collection that stores the cluster index assigned to $x$
    $b \leftarrow f(pointsCounter)$
    //update $k$ lower-bounds
    for $j \leftarrow 1$ to $k$ do
        $b.l_j \leftarrow \max[b.l_j - m_j, 0]$
    end
    $b.u \leftarrow b.u + m_{b.a}$ //update upper-bound
    $g \leftarrow true$ //flag to check if $u$ is updated
    $d1$, $d2 \leftarrow 0$
    if $b.u \leqslant h_{b.a}$ then continue
    for $j \leftarrow 1$ to $k$ do
        if $(j \neq b.a)$ &$(b.u > b.l_j)$ &$(b.u > s_{b.a,j} * 0.5)$ then
            if $g$ then
                $d1 \leftarrow$ d($x$, $c_{b.a}$)
                $b.u \leftarrow d1$
                $b.l_{b.a} \leftarrow d1$
                $g \leftarrow false$
            else
                $d1 \leftarrow b.u$
            end
            if $d1 > b.l_j$ or $d1 > s_{b.a,j} * 0.5$ then
                $d2 \leftarrow d(x, c_j)$
                $b.l_j \leftarrow d2$
                if $d2 < d1$ then
                    $b.a \leftarrow j$
                    $b.u \leftarrow d2$
                    $g \leftarrow false$
                end
            end
        end
    end
    $pointsCounter \leftarrow pointsCounter + 1$
    write $b$ to a BF on HDFS
    output($b.a$, $x$)
```

- **Second mapper:** The pseudo-code of ELK-H-BF's second mapper is shown in Algorithm 16. ELK-H-BF follows the same method that ELK-H-EV uses on eliminating distance computations, which was illustrated in Algorithm 12. The two algorithms differ in the method of reading and writing cluster assignments and distance bounds from/to HDFS. The second mapper assumes that the extra information was stored to a BF by a mapper in the previous iteration. Therefore, each mapper searches HDFS for the BF that corresponds to the input-split that is assigned to this mapper (line 6). When the BF is located, each record in the BF is parsed to a collection structure called b in the algorithm, where the size of b is $k + 2$ (k lower-bounds, one upper-bound, and one cluster assignment). All b's are then loaded to the list f. The *map* function reads each b from f that corresponds to each data point and uses the information in b to eliminate distance computations. Before sending the output to the reducers, each b is written to a BF on HDFS in line 41. This BF is then read by a mapper in the following iteration.

Reduce phase: The implementation of the reducer is identical to the reducer in section 3.3.1, Algorithm 5.

4.3.2 Compare-means on Hadoop using a Bounds File (CMP-H-BF)

This section highlights the differences between implementations of Compare-means on Hadoop using EVs and BFs. The method CMP-H follows to eliminate distance computations is explained in section 2.3.1.3. Therefore, the focus in this section is on how Compare-means on Hadoop writes and reads the cluster assignment for each data point from the previous iteration using Bounds Files.

BF Size: In a given iteration, each mapper in CMP-H-BF writes the index for the cluster assigned to each data point in the previous iteration to a BF. Therefore, each mapper writes a BF of size: $\frac{n}{p}$ per iteration, where n is the total number of data points, and p is the number of mappers.

Algorithm: CMP-H-BF consists of three major phases:

1. A *driver* that initiates the MapReduce jobs and controls the iterative process,
2. A *map* phase that assigns each point to its closest centroid (distance computation elimination steps occur in this phase), and
3. A *reduce* phase that computes the means of points assigned to each cluster centroid and produces new set of centroids.

Driver: The Driver's implementation is similar to the Driver in Algorithm 10, section 4.2.1, but with minor changes. For example, CMP-H-EV has only one mapper, therefore, there is no need to have an *if* statement to invoke two separate mappers' implementations.

Map phase: The pseudo-code in Algorithm 17 illustrates the implementation steps of the mapper in CMP-H-BF. In the first iteration, the index of the assigned cluster to point x from previous iteration is initialised to one, which is the first centroid in the centroids list C. If the test at line 20 holds, the distance computation to centroid c_j is skipped. After assigning x to its closest centroids c_j, index j is assigned to $b.a$ which is then written to a BF on HDFS. This process is repeated on subsequent iterations where previous cluster assignments can be read from BFs. Therefore, in the *setup* function, the records of the BF that corresponds the input-split that is assigned to the mapper is loaded to f. The *map* function can read updated cluster assignments (line 15) from the previous iteration for each data point.

Reduce phase: The implementation of the reducer is identical to the reducer in section 4.2.1, Algorithm 13.

Algorithm 17: CMP-H-BF-Mapper(k)

```
Function setup():
    load centroids from DistributedCache to $C$
    compute $s_{i,j} \leftarrow d(c_i, c_j)$ //for all $1 \leq i, j \leq k$
    if $iteration > 1$ then
        let $f$ be a list that stores the cluster assignments for all data point
        locate the BF that corresponds to the input-split assigned to this
          mapper and load its records to $f$
        $pointsCounter \leftarrow 1$
    end
Function map(offset, value)
    $x \leftarrow value$
    let $b$ be a collection that stores the cluster index assigned to $x$
    if $iteration == 1$ then
        $b.a \leftarrow 1$ //initialise cluster assignment
    else
        $b \leftarrow f(pointsCounter)$
    end
    $minDistance \leftarrow \mathrm{d}(x, c_{b.a})$
    $d \leftarrow 0$
    for $j \leftarrow 1$ to $k$ do
        if $s_{j,b.a} \geq 2 * minDistance$ or $j == b.a$ then
            continue
        end
        $d \leftarrow \mathrm{d}(x, c_j)$
        if $d < minDistance$ then
            $minDistance \leftarrow d$
            $b.a \leftarrow j$
        end
    end
    if $iteration > 1$ then
        $pointsCounter \leftarrow pointsCounter + 1$
    end
    write $b$ to a BF on HDFS
    output($b.a$, $x$)
```

4.4 Extended Vectors vs. Bounds Files

To distinguishes between EVs and BFs, we take a closer look at the type of information each approach maintains between iterations and which of this information is algorithm specific and which is generic.

In general, both methods (EVs and BFs) write/read the extra information to/from files on HDFS. The type and size of extra information that each approach maintains is algorithm specific. To show the deference, consider the two algorithms ELK-H and CMP-H where each algorithm is implemented on Hadoop using EVs and BFs and run on the same input dataset. In each iteration, for n number of data points in d dimensions, k number of cluster centroids, and p number of mappers, ELK-H-EV writes $\frac{n}{p}$ number of EVs where the size of each EV is $(d+k+2)$ consists of the data point, k lower-bounds, one upper-bound and one cluster assignment. CMP-H-EV writes the same number of EVs $(\frac{n}{p})$ but each EV is of size $(d+1)$ because the algorithms only needs to maintain the cluster assignments between iterations. Note that d is fixed in both algorithms and what follows d depends on the extra information each algorithm needs to maintain. On the other hand, the size of extra information in algorithms implemented with BFs is independent from d. ELK-H-BF, for example, writes $\frac{n}{p}$ records in each BF where the size of each record is $(k+2)$. CMP-H-BF writes the same number of records in each BF but each record consists of only the index of the assigned cluster.

4.5 Triangle Inequality K-Means on Hadoop (TIKM-H)

This section explains the implementation of Triangle Inequality K-Means on Hadoop (TIKM-H). TIKM-H uses the most basic form of triangle inequality to skip redundant distance computations from points to cluster centroids. That is why it was named after triangle inequality. By the *most basic form of triangle inequality* we mean that this approach does not require any information from the previous iteration to skip distance computations. This approach only needs to compute intra centre distances at the start of each mapper. In fact, the method TIKM-H follows to skip distance computations is the same as the one used in the first mapper of ELK-H-EV (Algorithm 11), and ELK-H-BF (Algorithm 15), where the centre-

Algorithm 18: TIKM-H-Mapper(k)

```
1  Function setup():
2      load centroids from DistributedCache to $C$
3      compute $s_{i,j} \leftarrow d(c_i, c_j)$ //for all $1 \leq i, j \leq k$
4  Function map(offset, value)
5      $x \leftarrow value$
6      //initialise all values in $t$
7      for $j \leftarrow 1$ to $k$ do
8          $t_j \leftarrow false$
9      end
10     $minDistance \leftarrow \infty$
11     for $j \leftarrow 1$ to $k$ do
12         if $t_j$ then
13             continue
14         end
15         $d \leftarrow \mathrm{d}(x, c_j)$
16         if $d < minDistance$ then
17             $minDistance \leftarrow d$
18             $a \leftarrow j$
19             for $z \leftarrow j + 1$ to $k$ do
20                 if $s_{j,z} \geq 2 * d$ then
21                     $t_z \leftarrow true$
22                 end
23             end
24         end
25     end
26     output($a$, $x$)
```

centre distances are computed at the *setup* function of each mapper and the map function tests the inequality in Lemma 2.3.1 in section 2.3.1.1 to see if the distance to the next centroids in the list can be skipped.

This approach does not have the potential to prune lots of distance computations compared to ELK-H and CMP-H. However, its very small overhead could make it a potential competitor to ELK-H and CMP-H on situations where the overhead becomes the dominant cost.

Algorithm 18 shows how TIKM-H eliminates distance computations by simply maintaining the skipping status of each centroid in the boolean list t, where the distance to centroid c_j is skipped if its status in the list t_j is *true*. The status of the centroids is set to *true* if the test in line 20 holds. The test basically checks if the distance from the current centroid c_j to the next centroid c_{j+1} (which were precomputed in line 3 and stored in s) is larger or equal to double the distance from the point x to centroid c_j, which is the same inequality in Lemma 2.3.1 in section 2.3.1.1.

4.6 Triangle Inequality K-Means on Spark (TIKM-S)

Triangle Inequality K-Means on Spark (TIKM-S) uses the same approach used in TIKM-H in the previous section. Basic triangle inequality optimisation is used to eliminate unnecessary distance computations where the only required information is the centre-centre distances before computing the distance from each point to each centroid. The implementation of Naive K-Means on Spark is explained in section 3.4. The implementation of TIKM-S-Driver is identical to NKM-S-Driver in Algorithm 8 in section 3.4. The difference between NKM-S and TIKM-S is on the implementation of the function $FindClosest()$, which finds the closest centroid from each point by computing the distance between them. Therefore, to avoid redundancy, only the implementation of $FindClosest()$ will be presented in this section.

As Algorithm 19 shows, centre-centre distances are computed at the beginning of the function, and as in NKM-H, the skipping status of each centroid in the boolean list t, where the distance to centroid c_j is skipped if its status in the list t_j is *true*. The status of the centroids is set to *true* if the test in line 17 holds. The test basically checks if the distance from the current centroid c_j to the next centroid c_{j+1} (which are precomputed in line 1 and stored in s) is larger or equal to double the distance from the point x to centroid c_j, which is the same inequality in Lemma 2.3.1 in section 2.3.1.1. $FindClosest$ returns each cluster index j with

Algorithm 19: FindClosest(C, k, $point$)

```
compute $s_{i,j} \leftarrow d(c_i, c_j)$ //for all $1 \leq i, j \leq k$
$x \leftarrow point$
//initialise all values in $t$
for $j \leftarrow 1$ to $k$ do
    $t_j \leftarrow false$
end
$minDistance \leftarrow \infty$
for $j \leftarrow 1$ to $k$ do
    if $t_j$ then
        continue
    end
    $d \leftarrow$ d($x$, $c_j$)
    if $d < minDistance$ then
        $minDistance \leftarrow d$
        $a \leftarrow j$
        for $z \leftarrow j + 1$ to $k$ do
            if $s_{j,z} \geq 2 * d$ then
                $t_z \leftarrow true$
            end
        end
    end
end
return($a$,$x$)
```

its assigned data point x as a pair.

4.7 Overhead Analysis

All the optimised algorithms generate extra overhead in order to gain speed relative to NKM-H. This overhead can be in the form of extra memory space, computation, or time to write/read information to/from HDFS. The cost is considered as an overhead if it does not exist in NKM-H. Each type of overhead cost is examined per iteration. In addition, the analysis will include only the overhead that occur at the map phase, since most of the extra costs occur in this phase.

NKM-H requires a memory space of $O(kd)$ to load k cluster centroids in d dimensions, in each mapper. Furthermore, each mapper needs to compute $O(\frac{n}{p}(kd))$

distances from each data point to each centroid, where n is the number of data points, and p is the number of mappers.

First, the overhead is examined in the simplest two algorithms; TIKM-H and TIKM-S. Both algorithms have a memory overhead of $O(k^2)$, and a computational overhead of $O(k^2)$, from computing centre-centre distance once in each mapper/executer. In fact, these two overheads can be applied to all the optimised algorithms as all of them perform the same operation. Table 4.2 shows the asymptotic overhead cost for each overhead type in each algorithm.

Algorithm	Memory Space	Write to HDFS Time
TIKM-H	$O(k^2)$	-
TIKM-S	$O(k^2)$	-
ELK-H-EV	$O(k^2)$	$O(\frac{n}{p}(d+k+2))$
CMP-H-EV	$O(k^2)$	$O(\frac{n}{p}(d+1))$
ELK-H-BF	$O(\frac{n}{p}(k+2)+k^2)$	$O(\frac{n}{p}(k+2))$
CMP-H-BF	$O(\frac{n}{p}+k^2)$	$O(\frac{n}{p})$

Table 4.2: The asymptotic overhead (where NKM-H is the baseline) for each algorithm in terms of: memory space, and time to write extra information to HDFS. All algorithms takes k^2 extra time to compute centre-centre distances. The examined overhead is for each mapper per one iteration. (n: No. of data points, d: No. of dimensions, k: No. of clusters, p: No. of mappers).

For the rest of algorithms, the overhead is as follow:

- **ELK-H-EV**: In addition to the k^2 extra memory space and k^2 extra time that was mentioned earlier, each mapper in ELK-H-EV needs to write $\frac{n}{p}(d+k+2)$ EVs to HDFS in each iteration. This increases the size of the input dataset from nd to $ndk+2$, which consequently increases the number of mappers on iterations that follow the first iteration.

- **CMP-H-EV**: The time to write EVs to HDFS in CMP-H-EV is small relative to ELK-H-EV, because each EV in CMP-H-EV is of size $d+1$. This makes the time to write EVs to HDFS per mapper: $O(\frac{n}{p}(d+1))$.

- **ELK-H-BF**: In general, algorithms implemented using BFs require more memory space, and spend less time in writing extra information to HDFS compared to algorithms that use EVs. In a given iteration, each mapper takes $O(\frac{n}{p}(k+2))$ time to write k lower bounds, one upper-bound and one cluster assignment. In addition, each mapper in ELK-H-BF needs to load $\frac{n}{p}$ records from each BF, where each record consists of k lower-bounds, one upper-bound, and one cluster assignment. This makes the extra memory space required by each mapper (adding to it the space for centre-centre distances) $O(\frac{n}{p}(k+2)+k^2)$.

- **CMP-H-BF**: Each mapper in CMP-H-BF takes $O(\frac{n}{p})$ time to write cluster assignments to HDFS. Each mapper needs extra memory space of $O(\frac{n}{p}+k^2)$ to store cluster assignments and centre-centre distances.

- **Average time to write extra information to HDFS** is the time each mapper takes to write the required information using BFs or EVs.

4.8 Summary

This chapter introduced two proposed solutions: Extended Vectors (EVs) and Bounds Files (BFs) to pass information from one iteration to the next in K-Mean on Hadoop. These solutions were developed to facilitate the use of triangle inequality approaches with K-Means. Each EV consists of the original data point vector and the extra information related to it. EVs are written and read to and from HDFS in the map phase. BFs, on the other hand, are files that consist of the extra information related to each data point. These files are written to HDFS in one iteration, and joined with their corresponding input-splits in the next.

The discussion about the overhead generated by each approach showed that EVs are better than BFs in terms of memory space consumption where the records in each BF must fit in the memory of the worker node that runs the mapper.

Conversely, the extra information in EVs are read as input in the map tasks along with the data point. BFs, however, reduces the overhead generated by EVs from rewriting data points of d dimensions to HDFS in each iteration. For a large number of dimensions, algorithms that use BFs are expected to outperform those that use EVs.

Earlier in this chapter, we explained that the main problem on implementing K-Means variants based on triangle inequality is in caching the required extra information in one iteration and read it in the next. An interesting idea to approach this problem is to extend the Distributed Cache by adding a new functionality that allows it to cache files in the map phase. This way we would be able to load the extra information, stored as BFs or EVs, to Distributed Cache in one iteration and read them in the next iteration. However, this approach is expected to face the following challenges. The Distribute Cache is not persistence. This means that in each iteration the Distributed Cache will have new configurations and old files wont exist any more. Furthermore, in the implementations of K-Means using BFs, each mapper needs to read only one BF that corresponds to the input split the mapper is going to operate on. Distributed Cache, however, copies each cached file to all the worker nodes in the cluster which is not the target in our case. Moreover, the benefit from using Distributed Cache is not expected to be significant since it will still writes/reads the files to/from disks and not to memory. This could be a promising approach if the extra files are cached in memory and if we could find a way to cache only the extra files that correspond to input splits processed by each mapper without broadcasting each file to all the nodes in the cluster.

The next chapter will present a detailed analyses of the experimental work for each algorithm.

Chapter 5

Experimental Results

This chapter presents and discusses the results of the experimental work that has been carried out to evaluate the efficiency and scalability of the proposed implementations of parallel K-Means on Hadoop and Spark.

The previous two chapters discussed the implementation steps for several K-Means algorithms on Hadoop and Spark. These algorithms include: NKM-H, ELK-H-EV, ELK-H-BF, CMP-H-EV, CMP-H-BF, TIKM-H, NKM-S, and TIKM-S, which are eight algorithms in total. The algorithms are tested against various parameters that affect the performance of K-Means in general, and the parallel K-Means on Hadoop and Spark in particular.

The remaining sections in this chapter are organised as follows. First, a description of the datasets that are used in the experimental work is provided in section 5.1. Section 5.2 describes the hardware and software specifications that are used in the experiments. Section 5.3 explains the design of the experimental work in terms of the evaluation metrics that have been chosen to measure the performance of each algorithm, and the reason behind choosing certain values for various parameters. A comparative analysis of optimised algorithms implemented using EVs and BFs is discussed in section 5.4, which involves testing each algorithm against variable number of clusters and dimensions, and a detailed analysis of the overhead. Section 5.5.1 presents the results of comparing the performance of algorithms implemented using a BF and tested against a variable number of clusters, dimensions, data points, and mappers. Section 5.6 presents a comparative analysis of algorithms implemented on Hadoop using a BF and two implementations of K-Means on Spark (NKM-S and TIKM-S). The final section summarises the discussed issues in this chapter.

5.1 Datasets

The datasets used in the experimental work are either artificially generated datasets or real-world datasets.

Artificial datasets: The reason behind generating artificial datasets is to be able to study the performance of each algorithm within a range of parameters, including the number of data points, number of clusters, and number of dimensions. Furthermore, the effect of different distributions of data points is investigated by running the algorithms on datasets with well-separated clusters as well as datasets with uniformly (randomly) distributed data points where there are no clusters.

Table 5.1 describes the characteristics of each artificial dataset in terms of its number of data points (n), number of dimensions (d), and the size in megabytes (MB). The data points in all datasets in Table 5.1, except dataset DS7, are normally distributed around 128 centres to form 128 well-separated clusters.

A Java program was developed to generate the data points. The program takes the following parameters as input: number of data points (n), number of dimensions in each data point (d), number of clusters (k) and the standard deviation (SD) where SD determines the density of the generated data points around each cluster centroid. For each dataset configuration in Table 5.1, the main steps for generating the data points are as follows:

1. A set of k centre vectors was generated with a uniform distribution with real

Name	No. of points (n)	No. of dimensions (d)	Size (MB)
DS1	100,000	8	15
DS2		32	28
DS3		128	235
DS4		512	941
DS5		1024	1884
DS6		2048	3788
DS7		512	947
DS8	1,000,000	128	1638
DS9	3,000,000		3584
DS10	5,000,000		5836
DS11	7,000,000		8192
DS12	9,000,000		10588

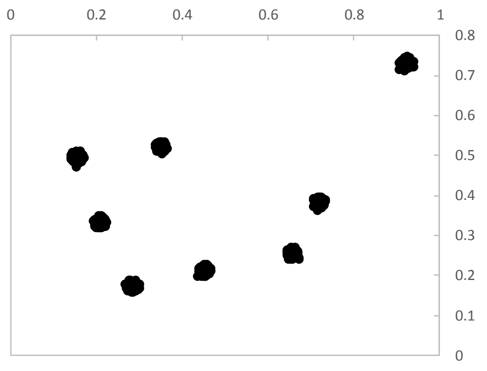
Table 5.1: Characteristics of artificially generated datasets. (MB=Megabyte)

numbers in $\mathbb{R}^d$.

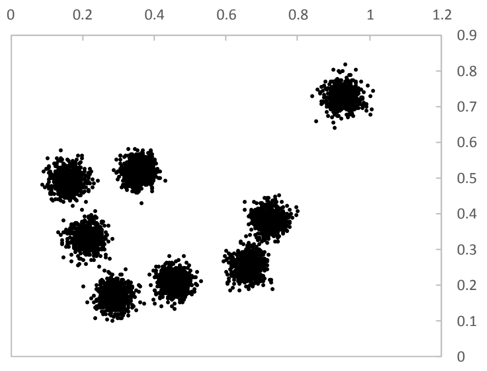
2. For each centre, $\frac{n}{k}$ number of data points was generated with an independent univariate Gaussian distribution for each dimension, and with a constant standard deviation *SD*. Except for dataset DS7 (which will be described later), datasets DS[1-12] have a constant standard deviation $SD = 0.02$.

The standard deviation determines the density of data points around each cluster. The lower the standard deviation, the higher the density of data points around the cluster centres. Generating data points with standard deviation $SD = 0.02$ creates well-separated clusters where the density of data points in each cluster is high.

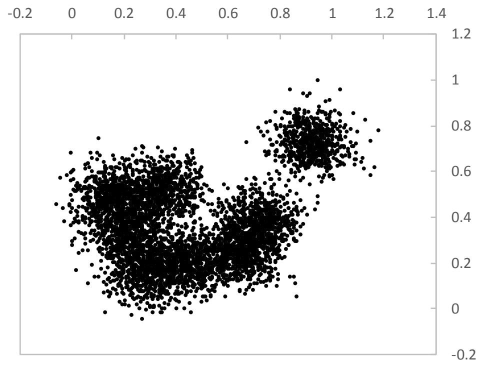
The data points in dataset DS7 are generated with a uniformly random distribution where there is no underlying structure in the data. The attempts to accelerate the performance on this type of data, especially in high dimensionality, gets very challenging because there are no meaningful clusters to be found [63]. Therefore, dataset DS7 is used to test the worst scenario for the optimised



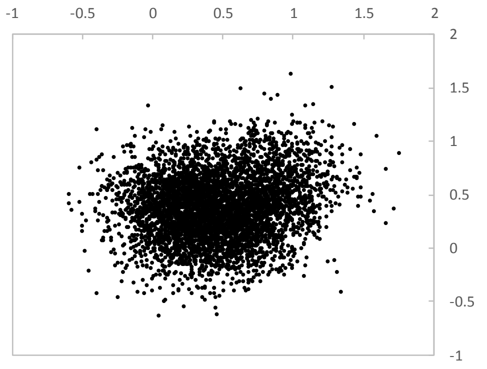
(a) $SD = 0.01$



(b) $SD = 0.05$



(c) $SD = 0.15$



(d) $SD = 0.5$

Figure 5.1: 2D representation of data points in four datasets with variable standard deviation (SD) and constant values $n = 5000$, $d = 2$, and $k = 8$. All datasets are generated with the same initial prototype vectors (centres). SD value varies in the range [0.01,0.5], where clusters in dataset with $SD = 0.01$ are well-separated (Figure 5.1a), while clusters in dataset with $SD = 0.5$ are heavily overlapped (Figure 5.1d).

algorithms in section 5.5.1, where algorithms implemented using Bounds Files are tested against a variable number of clusters and dimensions.

Datasets DS[1-7] are used as input to test the performance of each algorithm with respect to variable number of clusters and dimensions. While dataset DS[8-12] are used as input to test the efficiency and scalability of each algorithm implemented using Bounds Files with respect to variable number of data points, and mappers.

Figure 5.1 shows a graphical representation of the distribution of 2D data points in four datasets with variations of SD values, and fixed $n = 5000$, $d = 2$, and $k = 8$. Note how clusters separation start from well-separated in Figure 5.1a,

Name	No. of points (n)	No. of dimensions (d)	Size (MB)
covertype	581,012	55	72
mnist	60,000	784	104

Table 5.2: Characteristics of real-world datasets. (MB=Megabyte)

where $SD = 0.01$, to heavily overlapped clusters in Figure 5.1d, where $SD = 0.5$.

Real-world datasets: To observe the practicality of the proposed algorithms on real-world settings, two naturally-clustered datasets have been used in the experimental work. A brief description of the characteristics of real dataset is provided in Table 5.2.

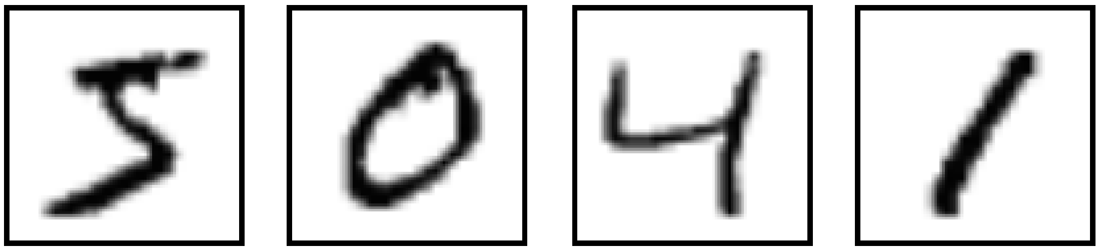
The first dataset *covertype*, contains collected observations of trees from four areas of the Roosevelt National Forest in Colorado. The dataset contains 581,012 observations, where each observation has 55 integer attributes. The collected data represent information about the types of soil, the wilderness areas, elevation, slope, forest cover type, and several other characteristics. This dataset is publicly available at the UCI Machine Learning Repository [111].

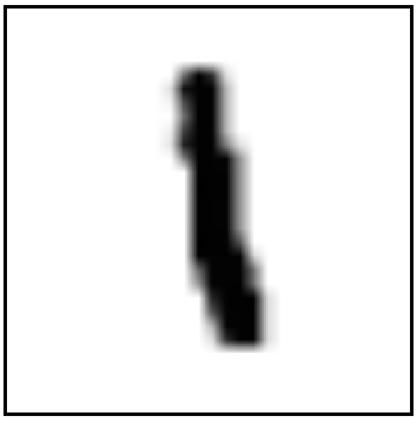
The second dataset *mnist* contains a total of 60,000 images of handwritten digits (0-9) collected from approximately 250 writers. Each pixel in each image holds an integer in the range [0,255], where 0 means the pixel is completely white, and 255 means the pixel is completely black. This produces a 28×28 matrix of integers. This matrix is flattened to form a 784 ($28 \times 28 = 784$) dimensional vector.

The aim of using this dataset is to cluster similar hand written digits together by using the integer values that represent the grey-level of each pixel.

Figure 5.2a shows an example of four images of the hand written digits 5, 0, 4 and 1. Figure 5.2b illustrates how a hand-written image of digit number 1 would be represented as a 14×14 matrix, where dimensions were reduced, and the value of each pixel was normalised to the range [0,1].

This dataset has been used in many works to test the scalability and efficiency of


(a) Four hand written digits 5, 0, 4, and 1



$$\simeq \begin{bmatrix}
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&.6&.8&0&0&0&0&0&0\\
0&0&0&0&0&0&.7&1&0&0&0&0&0&0\\
0&0&0&0&0&0&.7&1&0&0&0&0&0&0\\
0&0&0&0&0&0&.5&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.4&0&0&0&0&0\\
0&0&0&0&0&0&0&1&.7&0&0&0&0&0\\
0&0&0&0&0&0&0&1&1&0&0&0&0&0\\
0&0&0&0&0&0&0&.9&1&.1&0&0&0&0\\
0&0&0&0&0&0&0&.3&1&.1&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0\\
0&0&0&0&0&0&0&0&0&0&0&0&0&0
\end{bmatrix}$$

(b) Sample matrix of size 14 × 14 pixels representing digit number 1. The pixels' values are normalized to the range [0,1]

Figure 5.2: Figure 5.2a shows an example of four images of hand written digits (5, 0, 4, 1) from the mnist dataset. Figure 5.2b shows an image of the hand written digit (1) when represented in a 14 × 14 matrix of pixels [2].
.

K-Means. One of the reasons behind choosing this dataset is its high dimensionality ($d = 784$), where the performance of K-Means is heavily affected by computing point-centre distances. In section 5.6, mnist have been used as input to compare the performance of implementations of K-Means variants using BFs with NKM-H, TIKM-H, NKM-S, and TIKM-S. The dataset is available online at [112].

5.2 Hardware and Software Setup

Hardware: Apache Hadoop and Apache Spark are deployed on the same cluster which consists of 1 master node and 16 worker nodes. The master node has 2 AMD CPUs running at 3.1GHz with 8 cores each, and 8x8GB DDR3 RAM, and 6x3TB Near Line SAS disks running at 7200 rpm. Each worker node has 1 Intel

CPU running at 3.1 GHz with 4 cores, 4x4GB DDR3 RAM, and a 1x1TB SATA disk running at 7200 rpm. All the nodes run CentOS-6 (x86_64) operating system.

Software: The cluster uses Hadoop version 2.2.0 to run MapReduce on YARN. HDFS is configured with 128 MB default block size, and a replication factor of 3 replicas for each file. The default JVM heap size is 1 GB per task.

Apache Spark 2.1.1 is deployed on the same cluster as Hadoop, where YARN is used as the cluster manager, and HDFS as the distributed file system.

All algorithms were implemented in Java and compiled using JDK 1.7.0_79.

5.3 Experimental Design

The performance of parallel K-Means is affected by many factors. In order to give a clear picture of the performance of each algorithm, the experimental work was designed to address the major factors that could influence the performance of the optimised algorithms in several domains including: number of cluster, number of dimensions, number of data points, and number of mappers.

This section explains the configuration setup for the experimental work in terms of, the initialisation method of cluster centroids, the definitions and the choice of values for parameters related to Hadoop and K-Means, and an explanation of the evaluation metrics that are used to evaluate the performance of each algorithm.

Initial centroids: as mentioned earlier in section 2.3, K-Means is sensitive to the choice of initial centroids. This means running K-Means multiple times on a given dataset, each time with a different set of initial centroids leads to different final outputs for each run. To guarantee that this is not the case, all algorithms in a given experiment operate on the same initial set of centroids, that is, all algorithms use the same initialisation seed that is used to generate the initial set of centroids from the input dataset.

Total number of iterations: is the number of iterations that an algorithm executes until convergence or until reaching the maximum number of iterations.

Maximum number of iterations ($\boldsymbol{e}$): is a parameter provided by the user where an algorithm runs until it reaches this number or converges before that. The number of maximum iterations is set to 20.

Most of the clustering work in K-Means usually occurs in the few first iterations where the distance moved by cluster centroids is considrably large and data points change assignments more rapidly. As the number of iterations gets larger and the algorithm is closer to convergence, cluster centroids start to stabilise and do not move much. Consequently, most data points do not change their cluster assignment and stay at the same clusters that they were assigned to at the first few iterations [20] [21]. This means that the most challenging part for the optimised algorithms is in the early iterations and if these algorithms do well in the first iterations, they will achieve even better efficiency in later iterations. Furthermore, triangle inequality optimisations guarantee that the quality of the final clustering output is intact and equivalent to the original Lloyd's algorithm (clusterin quality is explained in detail in sections 2.3.1.2 and 5.3.2). This is why we decided to stop the algorithms at a fixed number of iterations in all the experiments in sections 5.4, 5.5 and 5.6.

Number of reducers ($\boldsymbol{r}$): The number of reducers is set to 1 on experiments that test each algorithm with a variable number of clusters and dimensions in sections 5.4 and 5.5.1. This is because in these tests the reduce time is not the performance bottleneck and increasing r would not have a significant effect on performance. The number of reducers is 20 on tests with a variable number of points in section 5.5.2 because the size of the datasets is relativity large, and after testing against several number of reducers, it was observed that 20 reducers is the optimal number for this particular test.

5.3.1 Evaluation Metrics

Each iteration of K-Means on Hadoop consists of three major phases: map, shuffle, and reduce. The major operations that consumes the majority of K-Means running time occur in the map phase. To fully understand the time consumed by each operation in the map phase, the map time is broken down into three major operations: 1) the average time to compute centre-centre distances, 2) the average time to compute point-centre distances, and 3) the average time to write extra information to HDFS. The shuffle time and reduce time are also reported.

The following is a detailed description of the evaluation metrics that are used to evaluate the performance of each algorithm.

- **Average iteration time** is the average running time per iteration over the total number of iterations that an algorithm has executed. This time includes: the CPU time, the I/O time, and the communication time. To compute the average time per iteration, the time for each iteration is obtained from Hadoop's job history log files at the end of each iteration. Then, after all iterations complete running, the average time spent by each iteration is computed by dividing the sum of all iterations' times over the total number of iterations. The iteration time does not include the time to initialise cluster centroids because it is a one time cost that occurs only once in each test. Later in this section, the average iteration time is inspected in terms of the variation in the iteration times in each algorithm and whether it can be a representative measurement index.

- **Speedup**: In general, speedup measures the improvement in speed for an enhanced algorithm over a baseline algorithm [113]. In this work, the performance of an optimised algorithm is reported as the speedup relative to NKM-H algorithm, where speedup is defined as the ratio of the average iteration time in NKM-H to the average iteration time of an optimised algorithms. One exception is when algorithms are tested against variable number of mappers where the baseline is the average iteration time on single mapper. Hence, speedup in such experiments is defined as the ratio of the average iteration time of an algorithm on a single mapper to the average iteration time for the same algorithm on a multiple number of mappers. For each algorithm, the average speedup over 10 trials is reported.

- **Average number of distance calculations** is the average number of point-centre distance calculations per iteration over the total number of iterations. To obtain the average, each mapper performs a certain number of distance computations, this number is aggregated with the number of computations obtained from all mappers. The summation of the number of distance computations that are obtained from all the mappers is maintained at the end of each iteration. Finally, the total number of distance computations over all iterations is computed and averaged over the total number of iterations.

- **Average time to compute point-centre distances**: To obtain the time to compute point-centre distances, in a given mapper, the total time consumed by point-centre distance computations for points assigned to this mapper is computed. After the completion of all mappers, the average time per mapper over the number of mapper is computed. After that, The total of these averages is divided by the total number of iterations to obtain the average time per iteration.

- **Average time to compute centre-centre distances**: Computing centre-centre distances is part of the map stage where each mapper in the optimised algorithms performs this process once in the *setup* function. First, the average time to compute centre-centre distances per mapper over the total number of mappers is computed in each iteration. Then, the average time to compute these distances per iteration over the total number of iterations is reported.

- **Average time to write extra information to HDFS** is the time each mapper takes to write the required information using BFs or EVs. Since this operation is executed by each mapper, the average time to write extra information per mapper over the number of mappers is computed, then the average time per iteration over the total number of iterations is reported.

- **Average shuffle time**: The shuffle time is the time to transfer intermediate data from mappers to reducers over the network. The shuffling of intermediate data starts as a predefined percentage (default is 5%) of the total number of mappers complete successfully, and ends when the last record on the last to finish mapper is transferred to its assigned reducer. The shuffle is part of the reduce phase. The average shuffle time is obtained from Hadoop's job history log files. The average shuffle time per Reducer over the total number of reducers is computed. This time is then averaged over the total number of iterations.

- **Average reduce time**: The reduce time is the time taken by Reducers to sort, merge and reduce the intermediate data fetched from mappers. The reduce time is obtained from Hadoop's history log files. The average time per reducer over the total number of Reducers is then computed, and the average reduce time per iteration over the total number of iterations is reported.

Is the Average Iteration Time a Representative Measurement Metric?

The section aims to examine the impact of the first iteration on the average iteration time and determine whether the average iteration time is qualified to be used as a measurement metric. To achieve this aim, each algorithm is tested on dataset DS4 and as Figure 5.3 shows, the first iteration time, the average iteration time *including* the first iteration, and the average iteration time *excluding* the first iteration are reported for each algorithm. Furthermore, the standard deviation (SD) is computed for each averaged iteration time (including and excluding the first iteration time) to show how much the iteration times differ from the average.

Figure 5.3 shows that there is no significant variation in the iteration times for all algorithms except for ELK-H-EV and CMP-H-EV. In these two algorithms the SD is relatively large. The average iteration time including the first iteration is 207($\pm$22.2) in ELK-H-EV, and 225.1($\pm$9.2) in CMP-H-EV. This variation in time is a result of the effect of the time consumed by first iteration. The first iteration takes longer time to complete compared to subsequent iterations as can noticed in the figure.

The time of the first iteration differ in general between algorithms that use EVs and all other algorithms because of the large time spent on writing EVs to HDFS. In addition, the first iteration does not avoid as many distance computations as

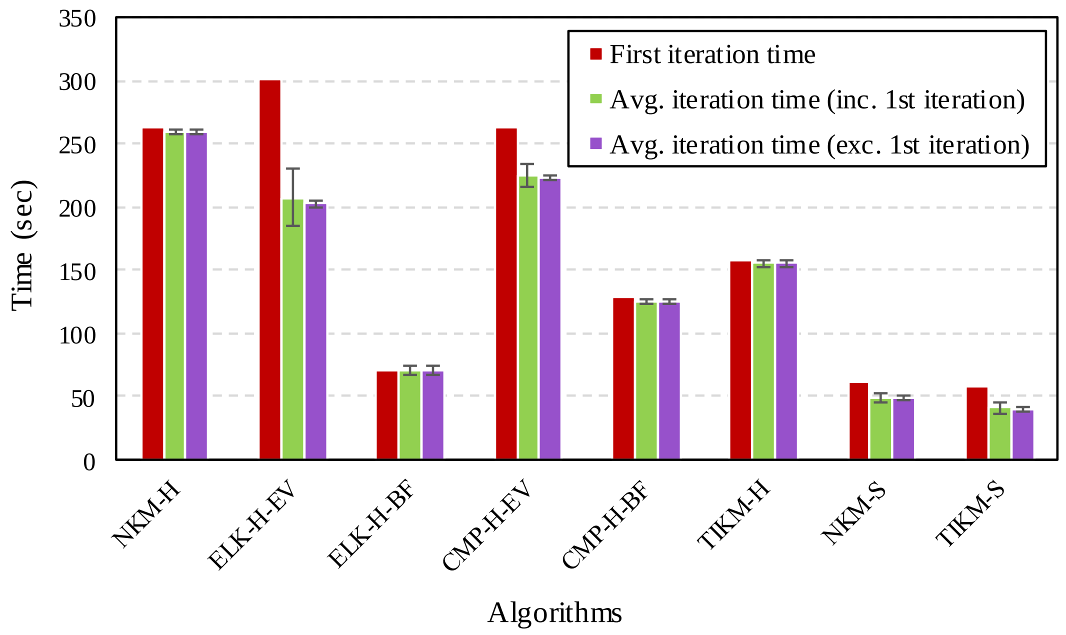


Figure 5.3: An illustration of three different times for each algorithm: 1) The running time of the first iteration, 2) the average running time per iteration and the standard deviation *including* the first iteration time, and 3) the average running time per iteration and the standard deviation *excluding* the first iteration time. (Dataset: DS4, $n = 100000$, $d = 512$, $k = 128$, $e = 20$).

subsequent iterations because the distance bounds and cluster assignments are not effective yet. This leads to a slow performance in the first iteration which can be even worse than the first iteration in NKM-H as Figure 5.3 illustrates. However, the difference between the average iteration times (including and excluding the first iteration time) in ELK-H-EV and CMP-H-EV is not significant. Therefore, the average iteration time is used a measurement index and the reported average in the experiments to follow includes the time of the first iteration.

5.3.2 Clustering Quality

In theory, the quality of the final output in the triangle inequality implementations is guaranteed to be equivalent to Lloyd's K-Means algorithm. In practice, the quality of the clustering results of the optimised algorithms were validated, in the early stages of this research, by comparing them with the clustering results of NKM-H. Given an input dataset, all algorithms were executed with the same initial centroids and were run until convergence. The produced cluster centroids after each iteration in an optimised algorithm were compared with the cluster centroids produced by NKM-H for the same iteration. If the sets of centroids in the tested algorithms match, and the algorithms converge in the same number of iterations, this indicates that the algorithms are deterministically equivalent, which was the case in all our implementations.

5.4 Comparative Analysis of All Implementations on Hadoop

The aim of this section is to investigate the scalability and efficiency of K-Means implementations using EVs and BFs with a wide range of number of clusters (k) and dimensions (d). Another aim is to determine the best and worse range of k and d for each algorithm. To accomplish these aims, algorithms: ELK-H-EV, CMP-H-EV, ELK-H-BF, and CMP-H-BF are tested against variable number of clusters k and dimensions d.

First, the obtained results from testing the algorithms against a variable number of clusters ($8 \leq k \leq 2048$) and fixed values for d and n ($d = 512$, $n = 100000$) are analysed. Then, the results from running each algorithm on variable number of dimensions ($8 \leq d \leq 2048$) and fixed values for k and n ($k = 128$, $n = 100000$) are discussed.

5.4.1 Variable Number of Clusters

In order to give an insight on the effectiveness of the optimised algorithms on avoiding distance computations, the number of distance computations for each algorithm is discussed first.

Distance Calculations

Figure 5.4 shows the average number of distance computations per iteration over the total number of iterations for NKM-H, ELK-H-BF, CMP-H-BF and TIKM-H.

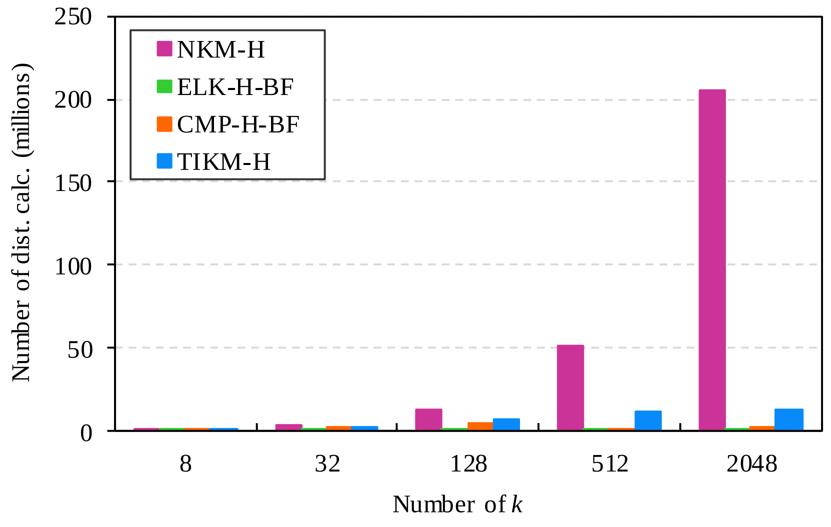


Figure 5.4: Average number of distance computations performed per iteration for four K-Means algorithms with variable number of clusters. (Dataset: DS4, $n = 100000$, $d = 512$, $e = 20$)

ELK-H-EV and ELK-H-BF perform an equal number of distance computations at each iteration. This is also true for CMP-H-EV and CMP-H-BF. Thus, a discussion about the number of distance computations for one algorithm can be applied to the other.

ELK-H-BF efficiently eliminates a large number of distance computations with all variations of k. Figure 5.4 shows that ELK-H-BF can eliminate around 76% when $k = 8$, and around 98% when $k = 512$ and 2048.

CMP-H-BF works best with large number of clusters on well-separated clusters where it eliminates 98% and 99% when $k = 512$ and 2048, but skips only 13% and 11% distance computations with $k = 8$ and 32 respectively.

Since TIKM-H implements the simplest approach to avoid distance computations, it does not prune many computations with small k. For instance, only 0.3% and 5% of the distance computations are skipped when $k = 8$ and 32, respectively. However, the skipped distance computations rises to up to 78% when $k = 512$, and 94% when $k = 2048$.

Speedup

As explained in section 5.3.1, the performance of each optimised algorithm is reported as speedup relative to the performance of NKM-H. Having higher speedups means better performance.

Figure 5.5 shows the speedup per iteration over the total number of iterations for five parallel implementations of K-Means on Hadoop relative to the NKM-H algorithm. The algorithms are tested against variable number of clusters to examine the influence of k on the performance of each algorithm. Dataset DS4 ($d = 512$, $n = 100,000$) is the input in this experiment (see Table 5.1).

It can be noticed that, in general, the speedup for algorithms implemented with BFs is higher than the ones implemented using EVs. When $8 \leq k \leq 128$, ELK-H-EV and CMP-H-EV perform the same or worse than NKM-H. This is due to

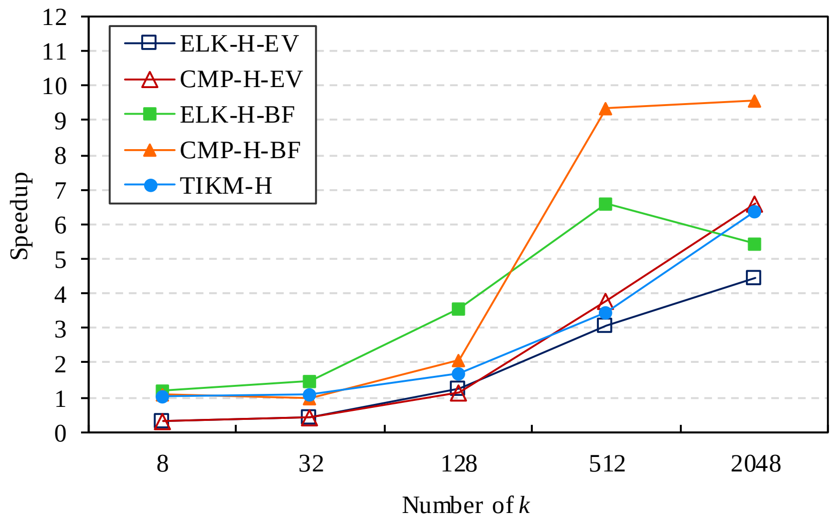


Figure 5.5: Speedup per iteration for five parallel implementations of K-Means on Hadoop relative to NKM-H, tested against variable number of clusters. ($n = 100000$, $d = 512$, $e = 20$)

the time to write EVs to HDFS in each iteration outweighs the time gained from skipping distance computations.

Although speedups are achieved by all algorithms when $512 \leq k \leq 2048$, it can be clearly noticed that using BFs to write and read distance bounds and cluster assignments is more effective than using EVs. For example, while the speedup of ELK-H-BF is 6.6x and 5.4x where $k = 512$ and 2048 respectively, ELK-H-EV achieves speedups of 3.4x and 4.4x for the same number of k. The difference is even more clear for CMP-H-BF and CMP-H-EV. For instance, CMP-H-BF is 9.3x and 9.6x faster than NKM-H when $k = 512$ and 2048, while CMP-H-EV is 3.8x and 6.6x faster with the same numbers of k. The speedup in ELK-H-BF drops from 6.6x when $k = 512$ to 5.4x when $k = 2048$ due to the increase in the time to write BFs which is dependent on k. The difference in the performance between EV and BF implementations is attributed to the amount of I/O overhead created by each approach from writing extra information to HDFS. The impact of the overhead is discussed in detail in the next section.

As the number of clusters gets larger than 32, TIKM-H starts to benefit from the pruned distance computations combined with the light computational overhead from centre-centre computations. The algorithm gains more speedups as the number of clusters increases.

Overhead Analysis

This section presents a detailed analysis of the time consumed by each task per iteration in each algorithm tested against variable number of clusters. To achieve this, the average iteration time is broken down into three main operations: time to compute point-centre distances, time to compute centre-centre distances, time to write extra information to HDFS. In addition, the time to perform the rest of operations is also reported. Section 5.3.1 explains all these measurement metrics except for the time of rest of operations. The rest of operations time is the time consumed by any operation other than the three operations just mentioned. Some of the operations included in the time consumed by the rest of operations are: the time to shuffle intermediate data, reduce time, time to merge the output of multiple reducers, time to replicate the reducer output, job setup time, required time by job tracker to contact application masters and assign map-reduce tasks, time spent by worker nodes to send heartbeat signals to JobTracker, time taken by NameNode to assign storage block and create input splits, etc.

In Figure 5.6, each bar represents the average time consumed by three major operations, and the rest of all other operations. The x-axis represents variable number of clusters k, and each k contains the set of algorithms that are tested against. The main objective is to study the impact of the overhead and distance computations on the performance of each algorithm. In addition, the figure shows the bottleneck in each approach. The tested algorithms are: CMP-H-BF, CMP-H-EV, ELK-H-BF, ELK-H-EV, TIKM-H, and NKM-H.

It might be noticed that the average iteration time in Figure 5.6 does not reflect the exact speedup gained by some algorithms. That is because the rest of operations time includes the time for all other operations, and the time for some of these operations overlaps with other operations. One example is the shuffle time where it overlaps with the map time. That is, Hadoop starts shuffling intermediate data from mappers to reducers while mappers are still processing data.

The following remarks can be drawn from Figure 5.6:

- The bottleneck in NKM-H is the time consumed by distance computations. The optimised approaches overcome this bottleneck but with a trade-off. The trade-off is the time spent on writing and reading distance bounds and cluster assignments, and computing centre-centre distances. As can be seen in Figure 5.6, for number of clusters where $8 \leq k \leq 128$, the impact of writing the extra information to HDFS can be clearly seen on CMP-H-EV and ELK-H-EV, where the performance of both is either worse or almost equal to NKM-H. The difference in performance between EV and BF implementations can be clearly noticed when $k = 512$ where the time to write extra information to HDFS in ELK-H-EV and CMP-H-EV is much larger than the time in ELK-H-BF and CMP-H-BF, respectively.

- The effect of centre-centre distance computations does not start to emerge until $k \geq 512$. The time consumed by these computations is not significant when $k = 512$. However, when $k = 2048$, this time increases dramatically, and in some cases, it becomes the dominant cost. For example, when $k = 2048$, CMP-H-BF and TIKM-H algorithms spend about 70% and 40% of their time computing these distances, respectively.

- Although TIKM-H eliminates the least number of distance computations compared to Elkan and Compare-means algorithms, the small overhead that is created by computing k^2 centre-centre distances makes it an excellent competitor with all other optimised algorithms, especially when $k \geq 512$.

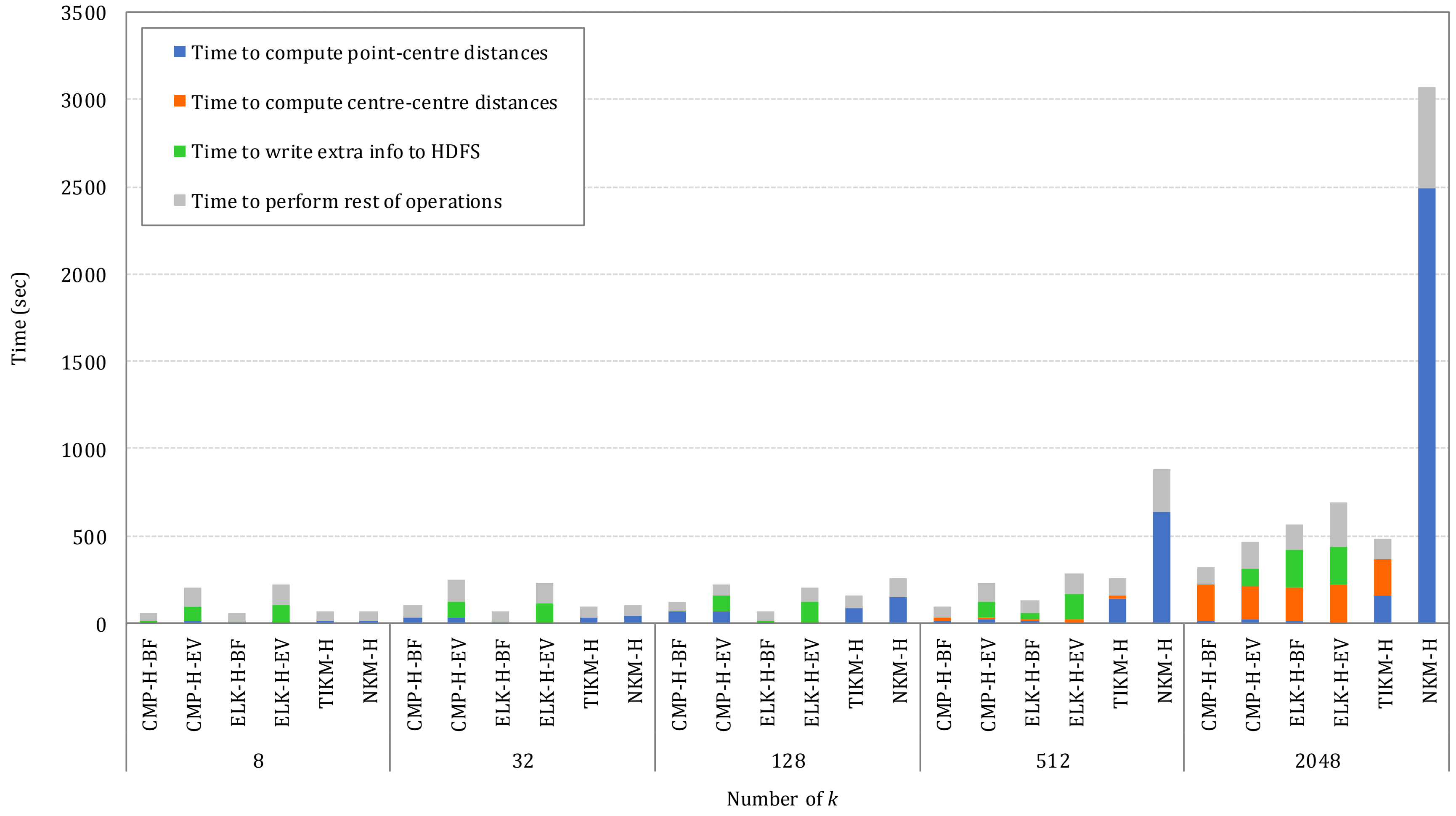


Figure 5.6: Average running time per iteration for six K-Means implementations tested against variable number of clusters (k). Each bar represents the average iteration time over 20 iterations for one algorithm divided into four operations times, 1) Time to compute point-centre distances, 2) Time to compute centre-centre distances, 3) Time to write extra information to HDFS, and 4) Time for the rest of all other operations.

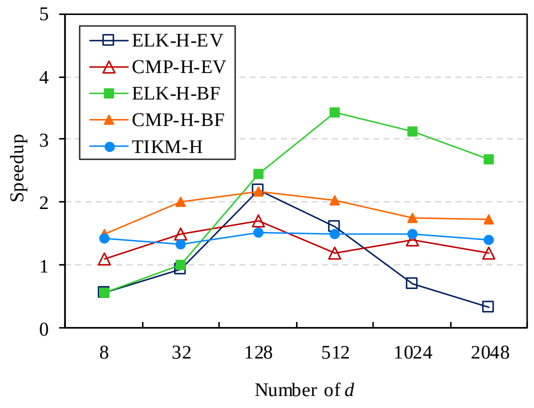
5.4.2 Variable Number of Dimensions

In the previous experiment, it was shown in Figure 5.5 that the speedup of ELK-H-EV, CMP-H-EV, CMP-H-BF, and TIKM-H relative to NKM-H keeps increasing as the number of clusters increases. It can be also observed from Figure 5.5 that the speedup for these algorithms started to increase when $k = 128$. In order to measure the ability for the mentioned algorithms to accelerate with even higher dimensions, each algorithm is tested with a variable number of dimensions ($8 \leq k \leq 2048$), while the number of clusters is fixed at $k = 128$. Datasets DS[1-6] are used as input in this experiment.

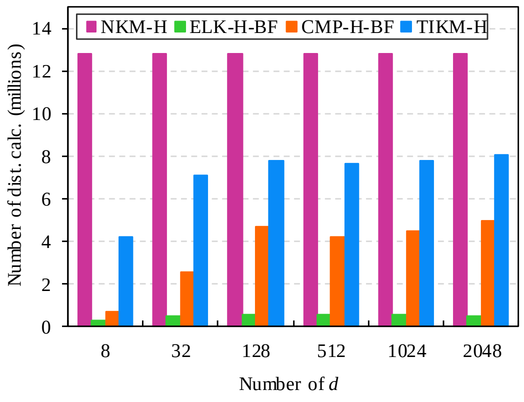
Figure 5.7a illustrates the speedup, relative to NKM-H, of the five optimised algorithms: ELK-H-EV, CMP-H-EV, ELK-H-BF, CMP-H-BF, and TIKM-H tested against variable number of dimensions.

In Figure 5.7a, the speedup of ELK-H-EV reaches the peak when $d = 128$ (2.2x) and starts to decline as d gets larger than 128. Although ELK-H-EV eliminates most distance computations (see Figure 5.7b) with all variations of d, the speedup of ELK-H-EV drops to 0.3x when $d = 2048$. This drop in speed is caused by the dramatic increase in the overhead from writing $\frac{n}{p}$ EVs to HDFS in each iteration, where the size of each EV is $d + k + 2$ (illustrated in Figure 5.7c).

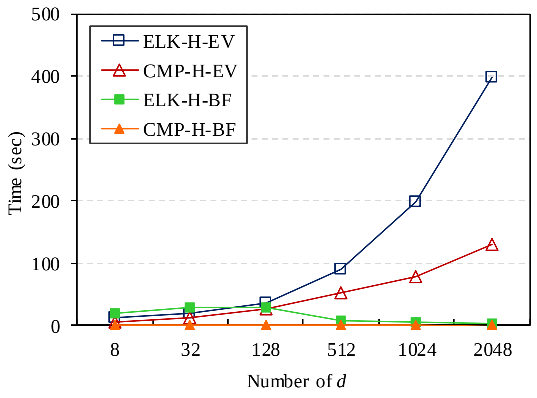
Figure 5.7c shows the average time per iteration to write EVs and BFs to HDFS for each optimised algorithm. The time to write BFs is insignificant in ELK-H-BF and CMP-H-BF as the number of dimensions increases. This is because the size of each record stored to a BF is $k + 2$, which is independent from d. On the other hand, ELK-H-EV and CMP-H-EV suffer from the increase of d, especially when $k \geq 1024$ where the time to write EVs becomes the dominant cost in ELK-H-EV.



(a) Speedup relative to NKM-H



(b) Avg. no. of distance calculations



(c) Avg. time per iteration to write EVs and BFs

Figure 5.7: Results of testing each algorithm against variable number of dimensions (d). Figure 5.7a shows the speedup of each algorithm relative to NKM-H, Figure 5.7b depicts the average number of distance computations per iteration over the total number of iterations, and Figure 5.7c illustrates the average time per iteration to write EVs and BFs to HDFS. ($k = 128$, $n = 100000$, $e = 20$)

5.5 Detailed Analysis of Implementations using a BF

It can be observed from the previous experiments that using BFs to implement K-Means variants has more potential to scale with increasing numbers of k and d than variants implemented with EVs. This is because algorithms that use BFs generate smaller overhead than algorithms that use EVs. For this reason, the focus on this section will be on optimised algorithms that are implemented using BFs.

5.5.1 Variable Number of Clusters and Dimensions

This section aims to show the impact of the number of clusters (k) and the number of dimensions (d) on the performance of ELK-H-BF, CMP-H-BF, and TIKM-H. The performance of these algorithms is compared against the performance of NKM-H to examine if the proposed optimisations can achieve any speedups with increased values of k and d. Another aim is to determine the range of k and d where each algorithm achieves its best and worse performance.

The values of k and d varies from small, medium and large number of clusters and dimensions, where $8 \leq k \leq 2048$ and $8 \leq d \leq 512$. The number of data points is fixed at $n = 100,000$, and the number of reducers $r = 1$. The datasets DS[1-4] are used as input to test the performance of each algorithm with clustered data. Dataset DS7 is used as an input to test each algorithm with uniform random data, which is the worst case for the K-Means variants that were implemented in this work. The real dataset covertype is used to test each algorithm with a real-world dataset.

To examine the impact of the major operations that influences the behaviour of each algorithm, Figures 5.8, 5.9, 5.10, and 5.11, show the average iteration time, the average number of distance computations, the average shuffle time and the average time to write extra information to HDFS, per iteration over the total number of iterations, receptively. Each Figure contains five sub-figures where each sub-figure represents an experiment on a given dataset with fixed number dimensions and data points, and variable number of clusters. The sub-figures are ordered based on the dataset that is used as an input. The sub-figures have the same order in all Figures. For example, if we want to examine the speedup, number of distance computations, shuffle time, and time to write extra information to HDFS for dataset DS1 with $d = 8$, we will look at the Figures 5.8a, 5.9a, 5.10a. and 5.11a.

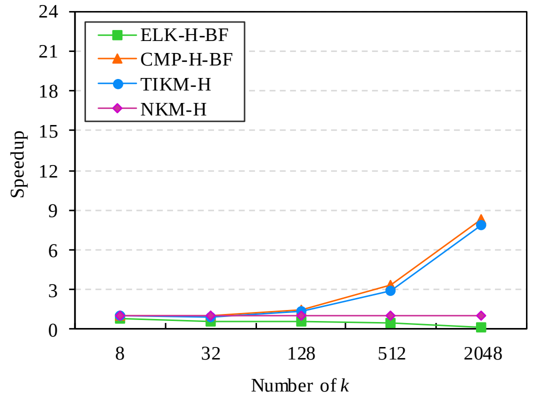
Note that the sub-figures for the results of clustered datasets (DS[1-4]) and the uniform random dataset (DS7) are shown together. The reason is to make it easier to compare the impact of the underlying structure of the input data on the performance of the optimised algorithms.

First, the analysis of the experimental work on clustered datasets (i.e. datasets DS[1-4]) is discussed, followed by the analysis of the experimental work on the uniform random dataset DS7. Finally, the results of the experimental work on the real-dataset *covertype* are discussed.

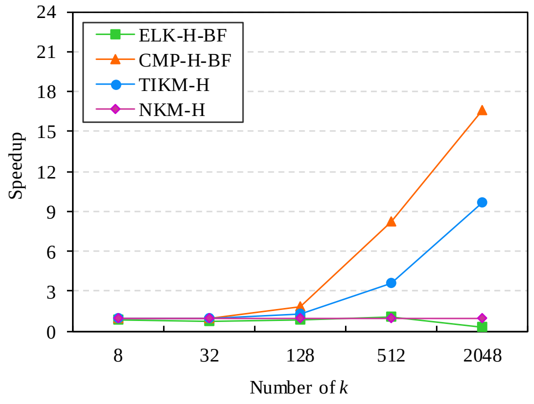
Clustered Datasets (DS[1-4])

From Figure 5.8a to Figure 5.8d, it can be observed that, in general, CMP-H-BF outperforms NKM-H, ELK-H-BF, and TIKM-H when $512 \leq k \leq 2048$ for all the tests on variations of *d*. The highest speedup that CMP-H-BF achieves relative to NKM-H is 21.2x where $d = 128$ and $k = 2048$ (Figure 5.8c). This can be attributed to two reasons: 1) CMP-H-BF eliminates larger number of distance computations that is close to ELK-H-BF and larger than TIKM-H, which can be observed in Figures 5.9[a-d], and 2) the small overhead CMP-H-BF generates compared to ELK-H-BF, as can be seen in Figures 5.11[a-d].

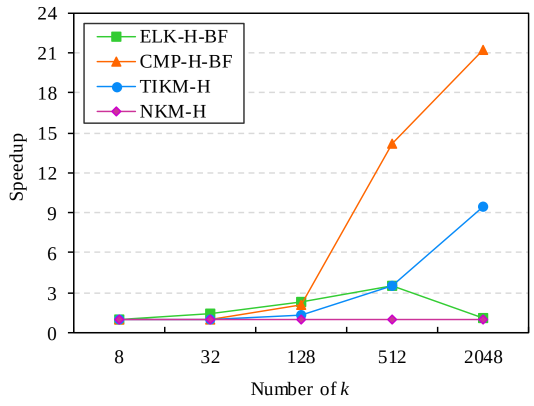
The best performance for ELK-H-BF is when $128 \leq d \leq 512$ and $128 \leq k \leq 2048$, as shown in Figures 5.8c and 5.8d. This is because distance computations



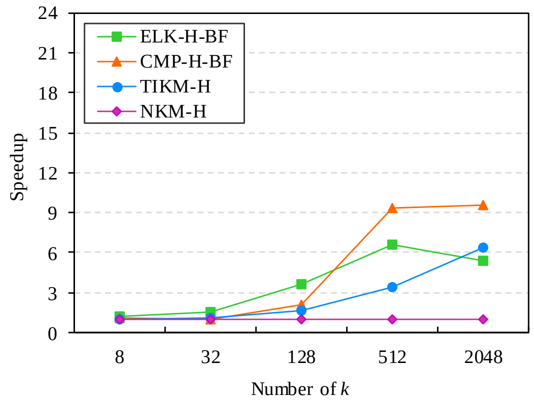
(a) $d = 8$, DS1



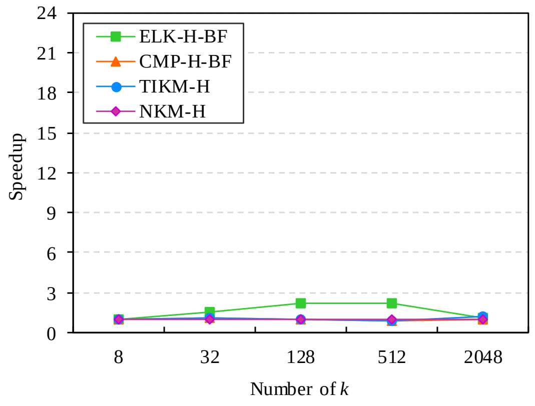
(b) $d = 32$, DS2



(c) $d = 128$, DS3

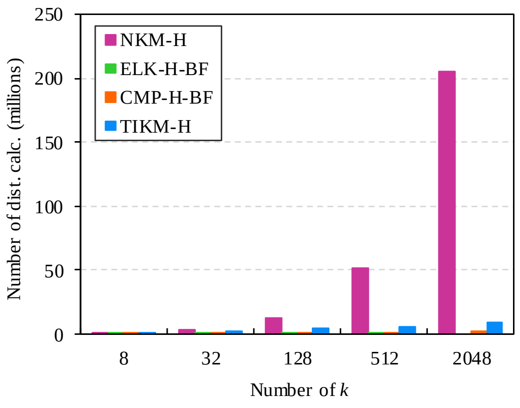


(d) $d = 512$, DS4

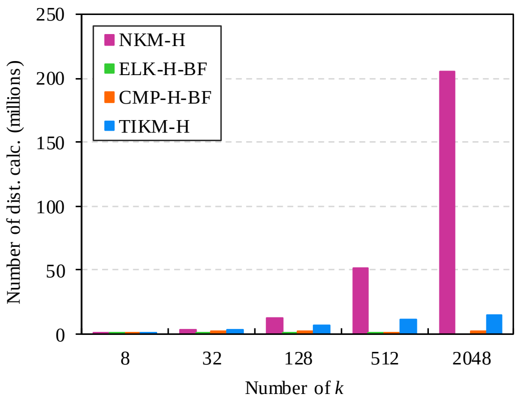


(e) $d = 512$, DS7

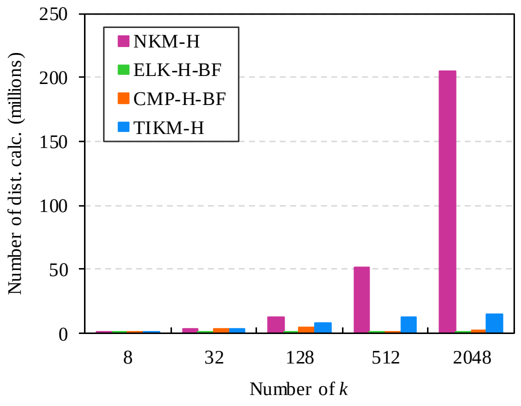
Figure 5.8: Speedup of each optimised algorithm relative to NKM-H. The Figures from 5.9a to 5.9d show the results for experiments on clustered datasets DS[1-4]. Figure 5.9e shows results for experiments on uniform random dataset DS7. Speedup is defined here as: avg iteration time(NKM-H)/avg iteration time(optimised). ($n = 100,000$, $e = 20$)



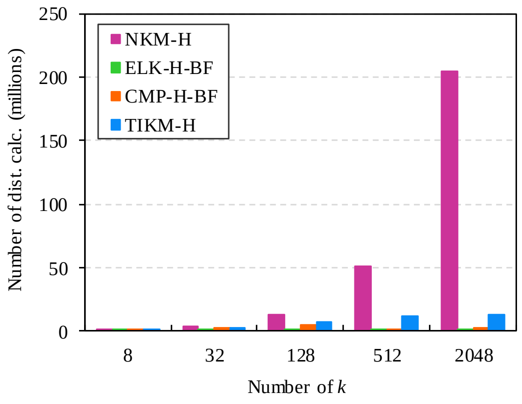
(a) $d = 8$, DS1



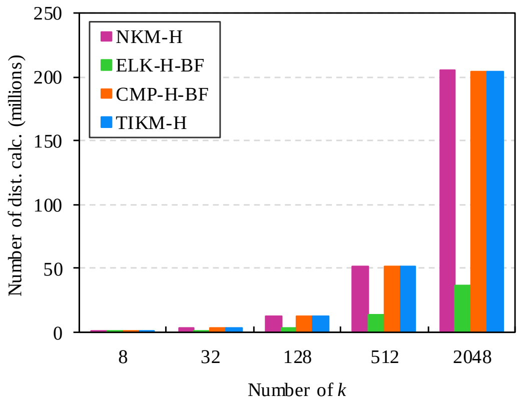
(b) $d = 32$, DS2



(c) $d = 128$, DS3

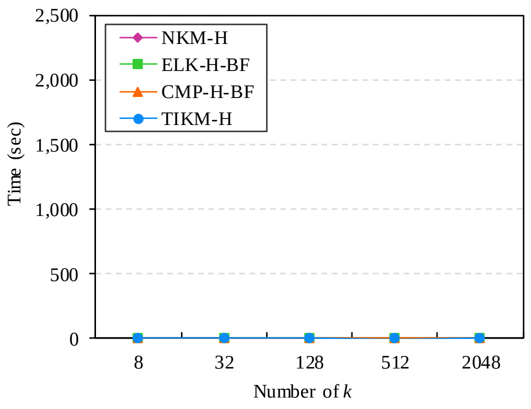


(d) $d = 512$, DS4

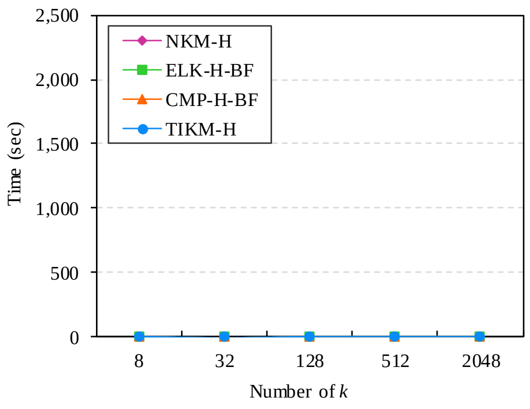


(e) $d = 512$, DS7

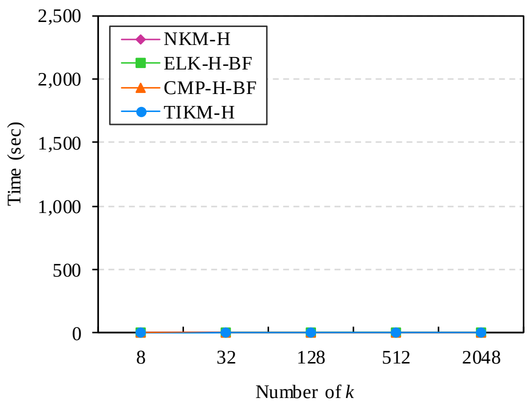
Figure 5.9: Average number of distance calculations per iteration over the number of iterations for each algorithms. The Figures from 5.9a to 5.9d show the results for experiments on clustered datasets DS[1-4]. Figure 5.9e shows results for experiments on uniform random dataset DS7. ($n = 100,000$, $e = 20$)



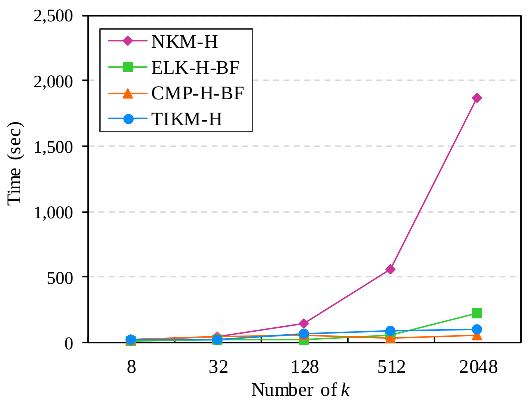
(a) $d = 8$, DS1



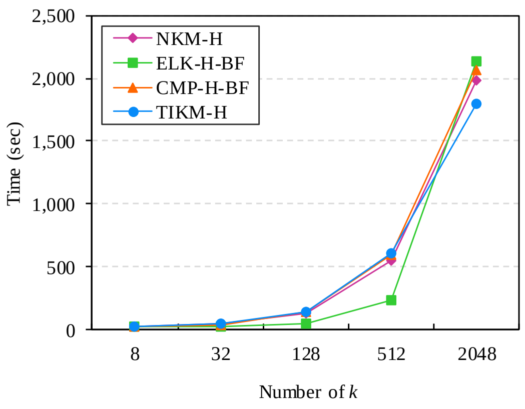
(b) $d = 32$, DS2



(c) $d = 128$, DS3

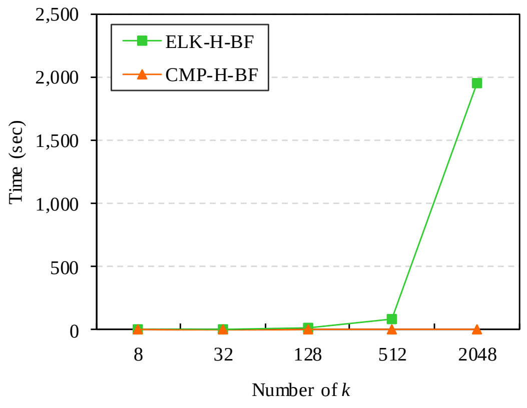


(d) $d = 512$, DS4

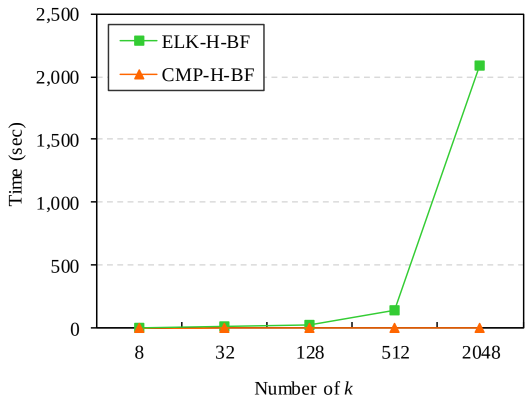


(e) $d = 512$, DS7

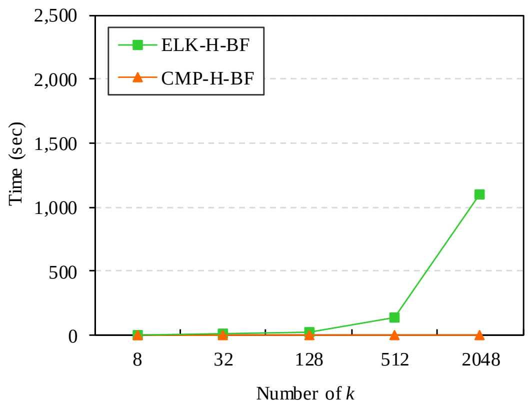
Figure 5.10: Average shuffle time per iteration over the number of iterations for each algorithms. The Figures from 5.10a to 5.10d show the results for experiments on clustered datasets DS[1-4]. Figure 5.10e shows results for experiments on uniform random dataset DS7. ($n = 100,000$, $e = 20$)



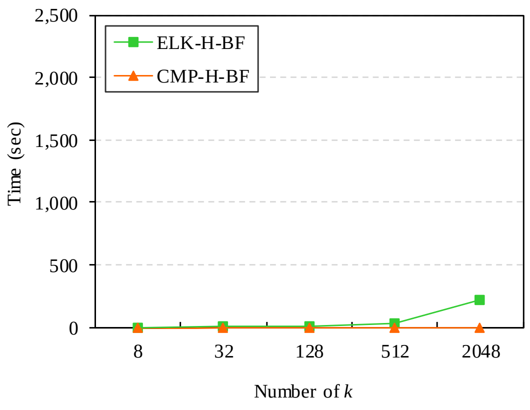
(a) $d = 8$, DS1



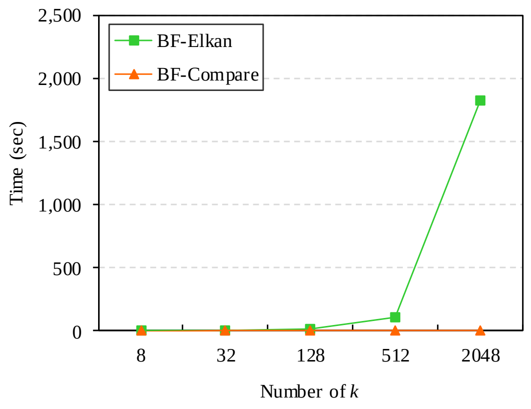
(b) $d = 32$, DS2



(c) $d = 128$, DS3



(d) $d = 512$, DS4



(e) $d = 512$, DS7

Figure 5.11: Average time to write extra information to HDFS per iteration over the total number of iterations for the optimised algorithms ELK-H-BF, and CMP-H-BF. The Figures from 5.11a to 5.11d show the results for experiments on clustered datasets DS[1-4]. Figure 5.11e shows results for experiments on uniform random dataset DS7. ($n = 100,000$, $e = 20$)

become the dominant cost in NKM-H and ELK-H-BF eliminates more than 95% of these computations. Furthermore, the time gained from pruning distance computations outweighs the time wasted on reading and writing distance bounds and cluster assignments. Although ELK-H-BF eliminates the largest number of distance computations compared to the other two algorithms, the overhead it generates affects the performance greatly.

For small numbers of clusters and dimensions where $8 \leq k, d \leq 32$, no significant improvements in speed are reported for all the optimised algorithms. In fact, the performance of ELK-H-BF is even worse than NKM-H in some cases (e.g. 0.1x of speedup when $d = 8$ and $k = 2048$). This is because even though the optimised algorithms eliminate some distance computations, the time that NKM-H spends on distance computations is already small, and the time gained from eliminating distance computations does not compensate the time spent on reading and writing the extra information.

The impact of the performed distance computations on the shuffle time can be seen in Figures 5.10[a-d]. Although the same amount of intermediate data in each test for all algorithms (optimised and NKM-H) is transferred from mappers to reducers over the network, the shuffle time can be affected by the amount of performed distance computations because of the overlap between the map and shuffle phases. The shuffle time in a given iteration starts as 5% of the total number of mappers complete their work, and ends when the last pair of cluster index and its associated data point in the last to finish mapper is transferred to its assigned reducer. This gives the optimised algorithms the advantage to speedup the shuffle time as these algorithms take less time to compute distances and, as a consequence, start and finish shuffling intermediate data sooner than NKM-H. This can be clearly seen in Figures 5.10d and 5.11e where both figures illustrate the average shuffle time per iteration for tests on clustered and uniform datasets where $d = 512$.

Uniform Random Dataset (DS7)

The optimised algorithms perform well when they operate on well-clustered datasets [63] [19]. This is particularly true for CMP-H-BF because it does not use any distance bounds and it relies only on the simple triangle inequality in Lemma 2.3.1 (see section 2.3.1.1 in Chapter 2) to skip redundant distance computations. ELK-H-BF, however, uses a large set of distance bounds combined with triangle inequality, which makes it more powerful on eliminating unnecessary distance computations but with larger I/O overhead.

As Figure 5.8e shows, there is no gain in speedup for CMP-H-BF and TIKM-H relative to NKM-H. This is caused by the small number of eliminated distance calculations, which is bellow 1% of the total number of distance computations in both algorithms (see Figure 5.9e). This also affects the shuffling time where both algorithms spend the same time as NKM-H in shuffling the data from mappers to reducers.

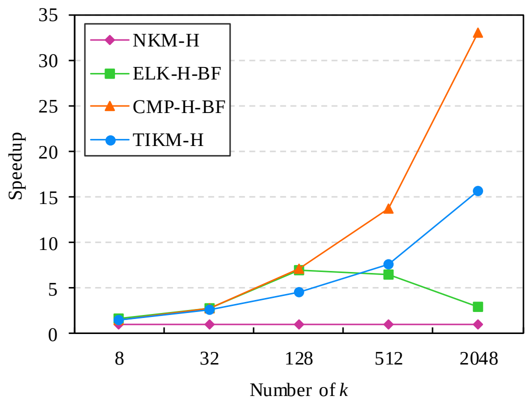
ELK-H-BF, on the other hand, eliminates up to 82% (when $k = 2048$) distance computations from the total number of distance computations. This is reflected on the speedup where ELK-H-BF was 2.8x times faster than NKM-H when the number of clusters are in the range of $128 \leq k \leq 512$.

This experiment clearly shows how the sparsity of clusters could effects the performance of K-Means variants that rely on triangle inequality to skip distance computations.

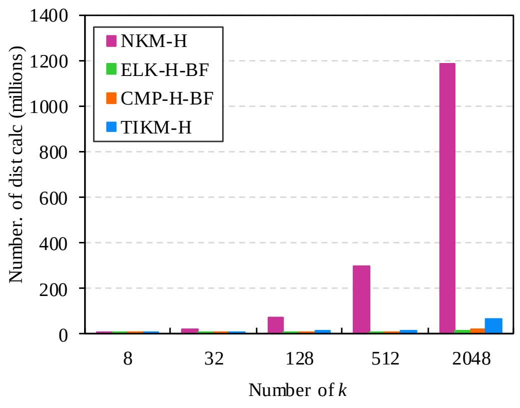
Real-world Dataset (covertype)

To study the performance of each algorithm with real-world settings, the real dataset *covertype* is used as an input for each algorithm and tested against variable number of clusters ($8 \leq k \leq 2048$). Please see section 5.1 and Table 5.2 for further details regarding real datasets.

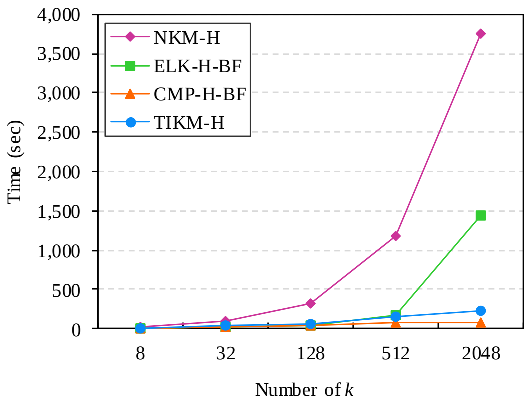
In general, CMP-H-BF and TIKM-H achieve high speedups relative to NKM-



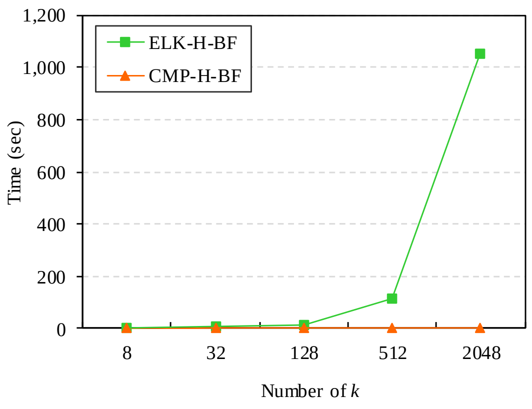
(a) Speedup relative to NKM-H



(b) Avg. no. of distance computations



(c) Avg. Shuffle time



(d) Avg time to write BFs

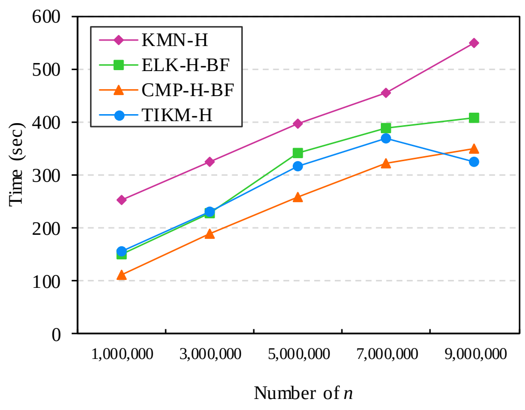
Figure 5.12: Results of testing each algorithm on the real-world dataset *covertype*. Figure 5.12a depicts the speedup of each algorithm relative to NKM-H, Figure 5.12b shows the average number of distance computations per iteration, Figure 5.12c shows average shuffle time per iteration, and Figure 5.12d illustrates the average time to write BFs to HDFS. Each algorithm is tested with respect to variable number of clusters. (Dataset: mnist, $n = 581,012$, $d = 55$, $e = 20$)

H as the number of clusters increases, as it can be observed from Figure 5.12a. The speedups for CMP-H-BF and TIKM-H, relative to NKM-H, are 33x and 15x, respectively, where $k = 2048$. ELK-H-BF, on the other hand, achieves a speedup of 7.2x when $k = 128$ then the speedup starts to drop as the number of clusters gets larger until it reaches 3x when $k = 2048$. This drop in speed in ELK-H-BF is due to the increase of the overhead that is generated from writing distance bounds and cluster assignments to HDFS as Figure 5.12d shows.

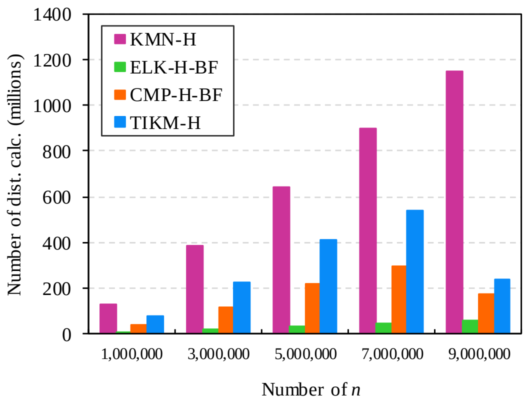
5.5.2 Variable Number of Data Points

This section aims to test the performance of each algorithm against an increased number of data points (n). Each algorithm is tested against five clustered datasets, DS[8-12] (see Table 5.1), each with a variable number of data points and constant number of clusters $k = 128$, and dimensions $d = 128$. The number of data points starts at 1,000,000 and increases by 2,000,000 data points in the following datasets until it reaches 9,000,000 data points. As the number of data points increases, the number of mappers increases accordingly. The number of mappers is obtained by dividing the total size of the dataset over the size of the HDFS block size (the block size in this experiment is 128 MB). For example, dataset DS8 ($n = 1,000,000$) is processed on 13 mappers, while dataset DS12 is processed on 83 mappers.

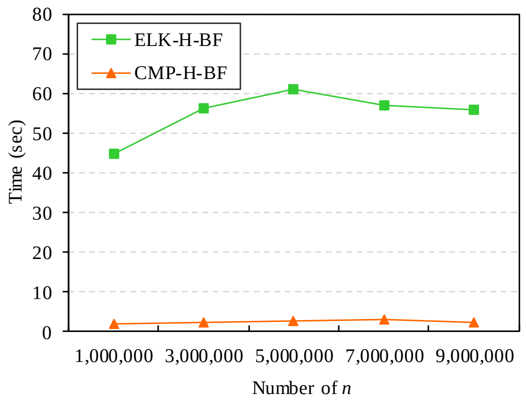
Figure 5.13b illustrates the average number of distance computations per iteration and Figure 5.13a plots the average running time per iteration over the total number of iterations for each algorithms. The impact of the reduction in distance computations can be clearly observed in these two figures. When the number of data points is in the range of $1,000,000 \leq n \leq 7,000,000$, CMP-H-BF and TIKM-H skip around 40% and 70% distance computations, respectively. The number of skipped distance computations increases for both algorithms when $n = 9,000,000$ to about 85% for CMP-H-BF and 80% for TIKM-H, which in return reduces the iteration time for both algorithms (see Figure 5.13a). Although ELK-H-BF eliminates most of the distance computations (about 95%), the time to write BFs to HDFS, illustrated in Figure 5.13c, makes the algorithm runs at almost the same speed as TIKM-H, except when $n = 9,000,000$, where TIKM-H is faster. This is because TIKM-H takes advantage of the light overhead and the large amount of skipped distance computations compared to the number of distance computations that was skipped where $n < 9,000,000$.



(a) Avg iteration time.



(b) Avg. no. of distance computations.



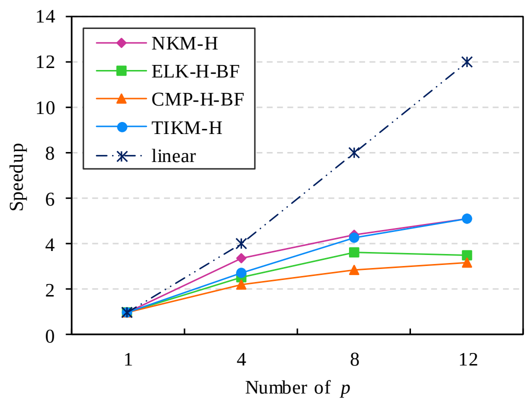
(c) Avg. time to write BFs to HDFS.

Figure 5.13: Results of testing each algorithm against variable number of data points (n), ($d = 128$, $k = 128$, $e = 20$)

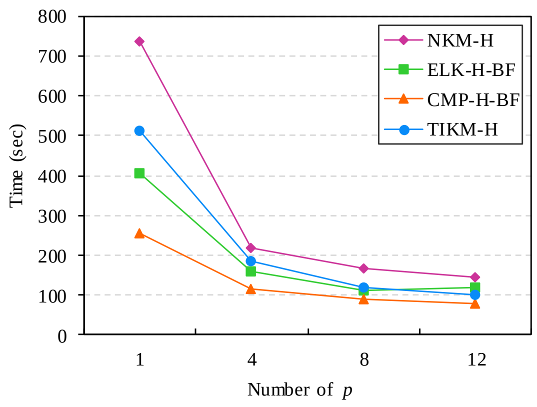
5.5.3 Variable Number of Mappers

The aim of this section is to investigate the scalability of each algorithm as the number of processing elements, which in this case is the number of mappers (p), is increased. To achieve this purpose, all algorithms run on a fixed problem size where $n = 1,000,000$, and $d = 128$, and the number of mappers is varied. The number of mappers starts from $p = 1$ and increased by 4 mapper each time and up to $p = 12$.

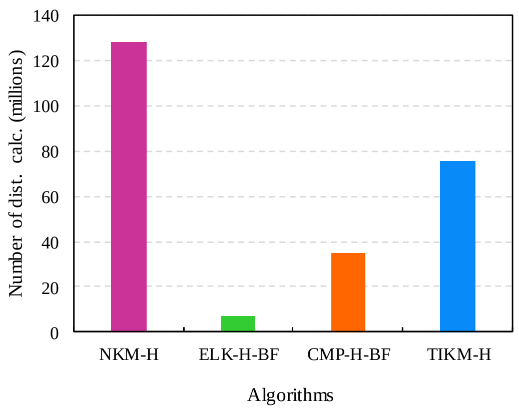
Figure 5.14a shows the speedup of each algorithm tested against variable number of mappers. The speedup in this experiment measures the relative gain in performance of executing each algorithm in parallel against executing the same al-



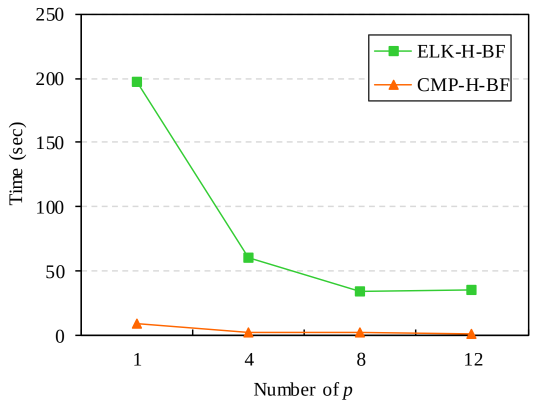
(a) Speedup relative to 1 mapper.



(b) Avg. iterations time.



(c) Avg. no. of distance computations.



(d) Avg. time to write BFs.

Figure 5.14: Results of testing each algorithm against variable number of mappers (p), (Dataset: DS6, $n = 1,000,000$, $d = 128$, $k = 128$, $e = 20$)

gorithm on a single mapper. To compute the speedup, the average iteration time for an algorithm is measured on a single mapper and multiple mappers, and the former is divided by the latter. An ideal parallel algorithm achieves speedups linear to p, which is hard to achieve considering the communication and I/O overheads in the parallel system.

Note that the test results for the speedup of each algorithm in Figure 5.14a is independent from the results of other algorithms. This is because the baseline configurations for each test is different from the other. For example, the speedup of NKM-H on multiple number of mappers is relative to the speed of the same algorithm on a single mapper. For this reason Figure 5.14b illustrates the average iteration time for each algorithm over the total number of iterations to be able to compare the running time of the algorithms on a different number of mappers.

Figure 5.14a shows that NKM-H and TIKM-H gain more parallel speedups from adding more mappers compared to ELK-H-BF and CMP-H-BF. This is because each mapper in NKM-H and TIKM-H computes larger numbers of distances compared to ELK-H-BF and CMP-H-BF as Figure 5.14c shows. This means that each mapper in NKM-H and TIKM-H spends a large amount of time computing distances and distributing this workload over a multiple number of mappers would lead to more gain in parallel speedups.

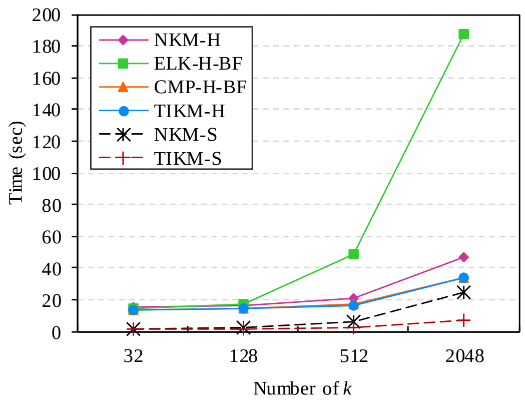
In Figure 5.14b modest improvement is achieved in terms of decreasing the average iteration time for CMP-H-BF by adding more mappers. This is because the algorithm has a small overhead (see Figure 5.14d) and prunes around 75% of distance computations (see Figure 5.14c) which makes it already fast on a single mapper (≈ 3 times faster than NKM-H).

ELK-H-BF avoids about 95% of distance computations which compensates the time wasted on writing BFs to HDFS. Furthermore, because of this overhead, the algorithm benefits from adding additional mappers, especially when $p = 4$ and 8, as Figure 5.14d illustrates.

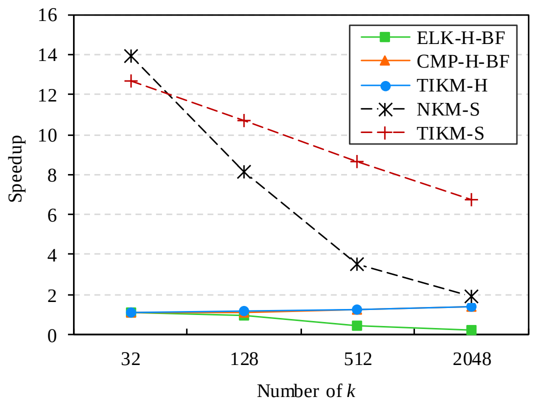
5.6 Comparative Analysis of K-Means Implementations on Hadoop and Spark

This section presents the results obtained from the experimental work on Apache Spark and compares these results against experimental work on Apache Hadoop. The goal of this experiment is to provide a comparative analysis between the performances of NKM-H, ELK-H-BF, CMP-H-BF, TIKM-H, NKM-S, and TIKM-S.

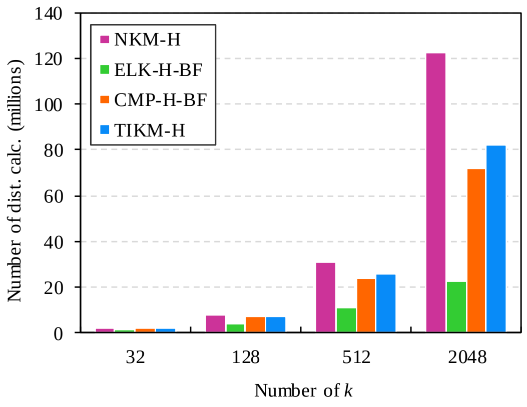
The experiments are executed on the real dataset *mnist*, and tested against variable number of clusters where $32 \leq k \leq 2048$, with fixed $d = 748$, and $n =$



(a) Avg. iterations time.



(b) Speedup relative to NKM-H.



(c) Avg. no. of distance computations.

Figure 5.15: Results of testing algorithms on Hadoop using BFs and algorithms on Spark on real dataset mnist with respect to variable number of clusters. Figure 5.15a shows the average iteration time for each algorithm, Figure 5.15b shows the speedup of each algorithm relative to NKM-H, and Figure 5.15c shows the average number of distance computations for each algorithm. Note that algorithms NKM-S and TIKM-S are not included in Figure 5.15c because the number of distance calculations in NKM-H and NKM-S is identical, and this can be also applied to TIKM-H and TIKM-S. (Dataset: mnist, $n = 60,000$, $d = 784$, $e = 20$)

60000 (see section 5.1). Note that the size of the dataset is 104 MB, which is smaller than the default HDFS block size of 128 MB. This means if 128 MB is used as the size of the block size, the dataset will by processed by only one mapper. Therefore, the HDFS block size is set to 20 MB instead of 128 MB in order to make the dataset processed by more than one mapper/executor.

Figure 5.15a shows the average running time per iteration for all algorithms where each algorithm is tested against variable number of clusters. It can be observed that NKM-S is faster than all K-Means implementations on Hadoop for $32 \leq k \leq 128$. This is attributed to the caching mechanism in Spark where input data is distributed over the cluster executor nodes and cached in-memory in the first iteration and reused in subsequent iterations in the form of Resilient Distributed Datasets (RDDs). This feature, unlike Hadoop, reduces the I/O and communication overheads. However, as k increases, distance computations become the bottleneck and the running time starts to increase to the point where it comes very close to the running time of CMP-H-BF and TIKM-H when $k = 2048$.

TIKM-S, on the other hand, outperforms all algorithms including NKM-S when $128 \leq k \leq 2048$. This is because TIKM-S skips around 17%, 33%, and 45% of distance computations when $k = 128, 512$, and 2048, respectively, as can be seen in Figure 5.15c, with a small overhead from computing k^2 centre-centre distances performed by each executor.

Figure 5.15b plots the speedup per iteration relative to NKM-H for each algorithm. CMP-H-BF and TIKM-H were able to reduce the large gap in speedup between them and NKM-S when $k = 2048$. That is, when $k = 2048$, CMP-H-BF and TIKM-H achieve 1.4x speedups, while NKM-S achieves and 1.9x. This makes CMP-H-BF and TIKM-H compete with NKM-S when the number of clusters is large.

5.7 Summary

This chapter presented the analysis of the experimental work that was carried out to evaluate the effectiveness of the proposed K-Means optimisations based on triangle inequality. NKM-H was used as a baseline for the new optimised implementations of K-Means using triangle inequality. The algorithms were tested against variable number of clusters (k), dimensions (d), data points (n) and mappers (p).

The impact of the triangle inequality optimisations on the number of distance computations was analysed first. The experiments showed that ELK algorithm outperforms all other algorithms in terms of the number of skipped distance computations. This can be attributed to the distance bounds that are used in ELK which allows the algorithm to do more comparisons and avoid more computations. However, the generated overhead from writing these bounds to HDFS had a considerable affected on the performance of ELK.

The performance of implementations of K-Means optimisations using EVs and BFs was investigated. The results showed that the performance of K-Means implementations with EVs can be greatly affected by the generated overhead from writing extra information to HDFS. For example, when the algorithms were tested against variable number of clusters, the speedup achieved by ELK-H-BF is much greater than the one achieved by ELK-H-EV when $k \geq 128$. Since CMP algorithm writes only one cluster assignment to HDFS in each iteration, CMP-H-BF algorithm was able to achieve notable speedups as the number of clusters and dimensions was increased. An interesting finding is that the TIKM-H algorithm achieved notable speedups, despite the fact that it adopts the simplest optimisation, due to its light-weight overhead.

The main operations that generated additional overheads in EVs and BFs implementations were investigated. It was noticed that writing the extra information to HDFS in each iteration affects the performance in implementations with both methods, EVs and BFs. However, this overhead is more evident on EVs implementations because of the large size of each EV compared to each record stored on BFs. Furthermore, the overhead from computing centre-centre distances is insignificant when $k \leq 512$. As k gets larger than 512, the impact of centre-centre computations can degrade the performance. Testing the algorithms against variable number of d showed the superiority of implementations with BFs. This is because the records in BFs are independent from d where only the extra information are written. Implementations with EVs, on the other hand, store data points in d dimensions with the extra information. Therefore, as d increases, the overhead from writing EVs also increases and becomes the performance bottleneck.

The performance of algorithms implemented with BFs was compared against variable number of k, d, n and p. Three types of datasets were used as input: clustered, uniform random dataset, and real-world datasets. With clustered datasets, CMP-H-BF achieved the highest speedups (21x where $d = 128$ and $k = 2048$). TIKM-H achieved moderate speedups but its speedup improved as k and d increased. On the other hand, tests on uniform random datasets showed how ELK-H had the advantage of using distance bounds to eliminate a large number of distance computations compared to CMP-H and TIKM-H. In this setting, ELK-H-BF was the only algorithm to be able to improve the speedups, while the other algorithms had approximately the same performance as NKM-H.

A comparison of BF implementations on Hadoop and two implementations of K-Means on Spark was discussed. The algorithms were tested against variable number of k. The results showed that CMP-H-BF is a good candidate that can accelerate the NKM-H to the point where its performance is close to NKM-S. TIKM-S outperformed all other algorithms when $128 \leq k \leq 2048$, taking advantage of the cached in-memory datasets, and the gain in speed from eliminating distance computations using the basic triangle inequality optimisation.

Chapter 6

Conclusions

This chapter summarises the main contributions of this thesis and presents the conclusions that can be drawn from the investigations that were carried out through the journey of this work. Future work that could add valuable contributions to this research is discussed in the last section.

The main aim of this thesis is to improve the efficiency and scalability of the Naive K-Means algorithm on Hadoop (NKH-H). To achieve this aim, the rich body of research was thoroughly investigated to identify efficient approaches that can speedup the clustering process in the Naive K-Mean. Among the large number of approaches that attempt to enhance the running time, using triangle inequality to skip redundant distance computations can accelerate the performance of K-Mean while maintaining the exact clustering results produced by the Naive K-Means. Such approaches meet the first and second objectives of this research to design and implement parallel solutions for K-Means on Hadoop that are more efficient and deterministically produce the same clustering results of Lloyd's algorithm and to adopt optimisation techniques based on triangle inequality. .

The implementation of variants that use triangle inequality on Hadoop is a challenging task. This is because these variants need to carry extra information from one iteration to the next and Hadoop does not provide a mechanism to exchange intermediate data between two consecutive MapReduce jobs. Hence, two new techniques: K-Means on Hadoop using an Extended Vector (EV), and K-Means on Hadoop using a Bounds File (BF), were introduced in Chapter 4 to give Hadoop the ability to transfer intermediate data from one iteration to the next without modifying the internal components of Hadoop. The new proposed techniques achieve the third objective of this project which is to provide a mechanism to carry extra information from one iteration to the next on Hadoop. To achieve this objective, two optimisations (Elkan's algorithm and Compare-means) were implemented on Hadoop using each technique to test the effectiveness of each one. Furthermore, an implementation of K-Means on Hadoop using the basic triangle inequality, which does not require any information from the previous iteration, was introduced. This implementation attempted to compensate the small number of skipped distance computations (compared to Elkan's and Compare-means) with the small overhead.

As for the final objective, which was to Measure the ability of the new solutions to improve the efficiency of Lloyd's K-Means on Hadoop and their ability to scale with an increased number of clusters, dimensions and data points, the experimental work in Chapter 5 compared the performance of implementations that used EVs and BFs with respect to various numbers of clusters and dimensions. From these tests it was found that algorithms that use BFs could scale better than those using EVs. Therefore, further tests were performed on implementations with BFs to examine the behaviour of these algorithms with variable number of clusters, dimensions, data points and mappers. Finally, implementations of K-Means on Hadoop using BFs were compared with two implementations of K-Means on Spark. The results showed that our implementations could compete with the Naive K-Means on Spark. However, implementing K-Means on Spark with the basic triangle inequality optimisation made it outperform all other implementations on large number of clusters.

6.1 Contributions

Two approaches were proposed to pass extra information from one iteration to the next on Hadoop. The first approach used a data structure called Extended Vector (EV), which stores the input data point vector and any extra information related to this data point into one EV. This EV would then be the input for the next iteration. The second approach stores the extra information into files called Bounds Files (BFs), where each entry in a given BF is a collection of extra information that is associated with an input data point. In both approaches, the extra information are written to HDFS, which generated I/O and communication overheads.

To evaluate each approach, two K-Means variants that adopt triangle inequality to reduce the number of distances computations were implemented using each approach. The following K-Means variants were implemented on Hadoop to evaluate each proposed techniques:

- Elkan's algorithm on Hadoop using an Extended Vector (ELK-H-EV),
- Elkan's algorithm on Hadoop using a Bounds File (ELK-H-BF),
- Compare-means on Hadoop using an Extended Vector (CMP-H-EV),
- Compare-means on Hadoop using a Bounds File (CMP-H-BF).

Furthermore, an optimised version of K-Means called Triangle Inequality K-Means on Hadoop (TIKM-H) was introduced. This version used the most basic form of triangle inequality to skip redundant distance computations, which do not require any information from previous iterations.

In addition, two implementations of K-Means on Spark were provided to investigate and compare the performance of K-Means on Hadoop and Spark. In particular, two implementations of K-Means on Spark were introduced: Naive K-Means on Spark (NKM-S) and Triangle Inequality K-Means on Spark (TIKM-S).

The experimental work provided a comparison between implementations of K-Means using EVs and BFs, implementations of K-Means using BFs tested against variable numbers of clusters, dimensions, points and mappers, and implementations of K-Means on Hadoop using BFs and two implementations of K-Means on Apache Spark.

6.2 Key Findings

To test the efficiency and scalability of each of the proposed algorithms relative to NKM-H, each algorithm was tested against various number of parameters that have a major impact on the performance of K-Means in general, and parallel K-Means on Hadoop in particular. That is, the proposed implementations were tested against variable number of clusters, dimensions, data points, and mappers.

The comparative analysis of EV and BF approaches showed that significant speedups could be achieved by implementations using both approaches. However, implementations that use BFs are more efficient and scalable than those that use EVs to pass information to subsequent iterations. As the number of clusters and dimensions increases, the overhead that is generated from writing EVs to HDFS increases dramatically.

An extensive experimental investigation was conducted on algorithms that use BFs to show the ability of this approach to scale with an increased number of clusters and dimensions with artificial and real datasets. The following conclusions can be drawn from the results of these experiments:

- It was found through the use of clustered and uniform random datasets that the best performance of the optimised algorithms that use triangle inequality is with datasets that have well-separated clusters. This is because more distance computations can be avoided with well-clustered datasets.

- The optimised algorithms did not achieve any significant speedups relative to NKM-H with low number of dimensions and clusters. The number of distance computations must be large enough to compensate the time spent on writing/reading extra information by the gained time from skipping distance computations in the optimised algorithms.

- It was noticed that because of the overlap between the map phase and the shuffle phase (i.e. mappers start transferring intermediate data as the work of 5% of the total number of mappers is complete), the optimised algorithms take advantage of this overlap by reducing the overhead created by distance computations in the NKM-H, which resulted in a reduction in the shuffle time.

The comparison between the performances of algorithms that were implemented on Hadoop using BFs and the two implementations of K-Means on Spark showed the superiority of TIKM-S over all the implementations on Hadoop and Spark as the number of clusters was increased. Combining the in-memory caching mechanism that Spark employs with the simple triangle inequality optimisation gave TIKM-S the ability to outperform all the other implementations.

6.3 Future Work

This section discusses the future work that could lead to valuable contributions to this research.

Through out this work, it was shown that a parallel implementation of efficiently optimised K-Means solutions can lead to a fast and highly scalable version of K-Means. The key issue this project focused on was to use triangle inequality to reduce the number of distance computations in parallel K-Means on Hadoop. This work could be enhanced even more by considering other factors.

As it has been discussed earlier in this thesis, K-Means is sensitive to the choice of the initial set of cluster centroids. In this work the initial cluster centroids were chosen randomly from the input dataset. Therefore, there is still a scope of further improvement on the efficiency by applying an effective centroids initialisation technique from the wide-range of techniques in the literature. A good set of initial cluster centroids could lead to a faster convergence, and could also make optimisations based on triangle inequality prune larger numbers of distance computations.

The K-Means implementations that were presented in this work are implemented on Hadoop and Spark. It would be interesting to compare these implementations with implementations of K-Means on other distributed computing frameworks such as Twister [23] and Piccolo [88].

This work chose two K-Means variants (Elkan's algorithm and Compare-means) to test the effectivness of the proposed approaches to pass information from one iteration to the next on Hadoop. Future work could include implementations of other variants that triangle inequality, or require information from previous iterations, such as Hamerly's algorithm [67], and adaptive K-Means [95].

The new implementations can be added to the clustering algorithms that are provided by Apache Mahout [101]. Apache Mahout is a library implemented on top of Apache Hadoop that offers various scalable Machine Learning algorithms including clustering algorithms such as K-Means (Lloyd's version).

Appendices

Appendix A

Publications

This appendix includes two published conference papers and one accepted journal paper. My contributions to all the papers are as follows:

- Carried out literature review of related work.
- Designed and implemented the parallel algorithms.
- Run experiments on Hadoop and Spark, collected and analysed the results.
- Wrote the initial draft of the papers, integrated contributions from co-authors, and led the submission.

[1] S. Al Ghamdi, G. Di Fatta, F. T. Stahl, "Optimisation Techniques for Parallel K-Means on MapReduce," in *Proceedings of the 8th International Conference on Internet and Distributed Computing Systems, IDCS 2015*, Windsor, UK, pp. 193-200, September 2015.

[2] S. Al Ghamdi, G. Di Fatta, "Efficient Parallel K-Means on MapReduce Using Triangle Inequality," in *Proceedings of the 2017 IEEE 3rd International Conference on Big Data Intelligence and Computing*, Orlando, Florida, USA, pp. 985-992,

November 2017.

[3] S. Al Ghamdi, G. Di Fatta, "Efficient Clustering Techniques on Hadoop and Spark," *International Journal of Big Data Intelligence*, [in press].

Optimisation Techniques for Parallel K-Means on MapReduce

Sami Al Ghamdi, Giuseppe Di Fatta, Frederic Stahl

School of Systems Engineering, University of Reading,
Whiteknights, Reading, RG6 6AY, United Kingdom
s.a.m.alghamdi@pgr.reading.ac.uk,
g.difatta@reading.ac.uk, f.t.stahl@reading.ac.uk

Abstract. The K-Means algorithm is one the most efficient and widely used algorithms for clustering data. However, K-Means performance tends to get slower as data grows larger in size. Moreover, the rapid increase in the size of data has motivated the scientific and industrial communities to develop novel technologies that meet the needs of storing, managing, and analysing large-scale datasets known as *Big Data*. This paper describes the implementation of parallel K-Means on the MapReduce framework, which is a distributed framework best known for its reliability in processing large-scale datasets. Moreover, a detailed analysis of the effect of distance computations on the performance of K-Means on MapReduce is introduced. Finally, two optimisation techniques are suggested to accelerate K-Means on MapReduce by reducing distance computations per iteration to achieve the same deterministic results.

Keywords: K-Means; Parallel K-Means; Clustering; MapReduce.

1 Introduction

Clustering is the process of partitioning data points in a given dataset into groups (clusters), where data points in one group are more similar than data points in other groups. Cluster analysis plays an important role in the Big Data problem. For example, it has been used to analyse gene expression data, and in image segmentation to locate objects' borders in an image.

K-Means [1] is one of the most popular and widely used clustering algorithms. K-means has been extensively studied and improved to cope with the rapid and exponential increase in the size of datasets. One obvious solution is to parallelise K-Means. K-Means have been parallelised based on different environments such as Message Passing Interface (MPI) [2] and MapReduce [3].

For a given number of iterations, the computational complexity of K-Means is dominated by the distance computations required to determine the nearest centre for each data point. These operations consume most of the algorithm's run-time because, in each iteration, the distance from each data point to each centre has to be calculated. Various optimisation approaches have been introduced to tackle this issue. Elkan [4] applied the triangle inequality property to eliminate unnecessary distance computations on high dimensional datasets. An optimisation technique based on multidimensional trees (KD-Trees) [5] was proposed by Pelleg and Moore [6] to accelerate K-Means. Judd et al. [7] presented a parallel K-Means formulation for MPI and used two approaches to prune unnecessary distance calculations. Pettinger and Di Fatta [8] [9] proposed a parallel KD-Tree K-Means algorithm for MPI, which overcomes the load imbalance problem generated by KD-Trees in distributed computing systems. Different approaches have been proposed to improve K-Means efficiency on MapReduce by reducing the number of iterations. However, we intend to accelerate K-Means on MapReduce by reducing distance computations per iteration.

This paper describes the implementation of K-Means on MapReduce with a mapper-combiner-reducer approach and how the iterative procedure is accomplished on MapReduce. In Addition, it presents some preliminary results relative to the effect of distance calculations on the performance of K-Means on MapReduce. Finally, two approaches are suggested to improve the efficiency of K-Means on MapReduce.

The rest of the paper is organised as follows: Section 2 briefly introduces K-Means and MapReduce, and presents a detailed description of Parallel K-Means on MapReduce. Section 3 reports the experimental results. Section 4 presents the work in progress. Finally, section 5 concludes the paper.

2 Parallel K-Means on MapReduce

2.1 K-Means

Given a set X of n data points in a d-dimensional space $\mathbb{R}^d$, and an integer k that represents the number of clusters, K-Means partitions X into k clusters by assigning each $x_i \epsilon X$ to its nearest cluster centre, or centroid, $c_j \epsilon C$, where C is the set of k centroids. Given a set of initial centroids, data points are assigned to clusters and cluster centroids are recalculated: this process is repeated until the algorithm converges or meets an early termination criterion. The goal of K-Means is to minimise the objective function known as the Sum of Squared Error $(SSE) = \sum_{j=1}^{k} \sum_{i=1}^{n_j} ||x_i - c_j||^2$, where x is the i^{th} data point in the j^{th}cluster and n_j is the number of data points in the j^{th}cluster. The time complexity for K-Means is $O(nkd)$ per iteration.

2.2 MapReduce

MapReduce [3] is a programming paradigm that is designed to, efficiently and reliably, store and process large-scale datasets on large clusters of commodity machines.

In this paradigm, the input data is partitioned and stored as blocks (or input-splits) on a distributed file system such as Google File System (GFS) [10], or Hadoop Distributed File System (HDFS) [11]. The main phases in the MapReduce model are *Map*, *Shuffle*, and *Reduce*. In addition, there is an optional optimisation phase called *Combine*. The MapReduce phases are explained as follows:

In the *Map* phase, the user implements a *map* function that takes as an input the records inside each input-split in the form of key1-value1 pairs. Each map function processes one pair at a time. Once processed, a new set of intermediate key2-value2 pairs is outputted by the mapper. Next, the output is spilled to the disk of the local file system of the computing machine. In the *Shuffle* phase the mappers' output is sorted, grouped by key (key2) and shuffled to reducers. Once the mappers' outputs are transferred across the network, the *Reduce* phase proceeds where reducers receive the input as key2-list(value2) pairs. Each reducer processes the list of values associated to each unique key2. Then, each reducer produces results as key3-value3 pairs, which are written to the distributed file system. The *Combine* phase is an optional optimisation on MapReduce. Combiners minimise the amount of intermediate data transferred from mappers to reducers across the network by performing a local aggregation over the intermediate data.

2.3 Parallel K-Means on MapReduce Implementation

Parallel K-Means on MapReduce (PKMMR) has been discussed in several papers (e.g., [12][13]). However, in this paper we explain, in details, how *counters* are used to control the iterative procedure. Moreover, we show the percentage of the average time consumed by distance computations. PKMMR with a combiner consists of: *Mapper*, *Combiner*, *Reducer* and a user program called *Driver* that controls the iterative process. In the following sections, a data point is denoted as dp, a cluster identifier as c_id, the combiner's partial sum and partial count as p_sum and p_count.

Driver Algorithm

The Driver is a process that controls the execution of each K-Means iterations in MapReduce and determines its convergence or other early termination criteria. The pseudocode is described in Algorithm-1. The Driver controls the iterative process through a user defined counter called $global\_counter$ (line 2). The global_counter is used as a termination condition in the while loop. The counter is incremented in the Reducer if the algorithm does not converge or an early termination condition is not met, otherwise, the counter is set to zero and the while loop terminates. Besides configuring, setting, and submitting the MapReduce job, the Driver also merges multiple reducers' outputs into one file that contains all updated centroids.

Algorithm-1: *Driver*

```
1:  Select k initial cluster centroids randomly;
2:  global_counter := 1 //initialised and modified in Reducer (Algorithm-4)
3:  while global_counter > 0 or a termination condition is not met do
4:      Configure and setup a MapReduce job;
5:      Send initial set of centroids to computing nodes,
6:      Run the MapReduce job;
7:      if number of reducers > 1 then
8:          Merge reducers output into one file
9:      end if
10:     global_counter := Counter(global_counter).getValue();
11: end while
```

Mapper Algorithm

Each Mapper processes an individual input-split received from HDFS. Each Mapper contains three methods, *setup*, *map* and *cleanup*. While the map method is invoked for each key-value pair in the input-split, setup and cleanup methods are executed only once in each run of the Mapper. As shown in Algorithm-2, setup loads the centroids to c_list. The map method takes as input the offset of the dp and the dp as key-value pairs, respectively. In lines 4-10, where the most expensive operation in the algorithm occurs, the loop iterates over the c_list and assigns the dp to its closest centroid. Finally, the mapper outputs the c_id and an object consists of the dp and integer 1. Because it is not guaranteed that Hadoop is going to run the Combiner, Mapper and Reducer must be implemented such that they produce the same results with and without a Combiner. For this reason, an integer 1 is sent with the dp (line 11) to represent p_count in case the combiner is not executed.

Algorithm-2: *Mapper*

```
Method   setup ( )
 1:  Load centroids to c_list;
Method   map (key, value)
 1:  Extract dp vector from value;
 2:  c_id := -1;
 3:  min_distance := ∞;
 4:  for i := 0 to c_list.size -1 do
 5:      distance := EuclideanDistance(c_list[i], dp)
 6:      if distance < min_distance then
 7:          min_distance := distance;
 8:          c_id := i;
 9:      end if
10:  end for
11:  output (c_id, (dp, 1));
```

Algorithm-3: *Combiner*

```
Method   setup ( )
 1:  Load centroids to c_list;
Method   reduce(c_id, list<values>)
 1:  p_count := 0,  p_sum := 0;
 2:  for value in values do
 3:      Extract dp vector from value;
 4:      p_sum := p_sum + the vector sum of dps in d-dimensions;
 5:      p_count := p_count + 1;
 6:  end for
 7:  output(c_id, (p_sum, p_count))
```

Combiner Algorithm
As shown in Algorithm-3, the Combiner receives from the Mapper (key, list(values)) pairs, where key is the c_id, and list(values) is the list of dps assigned to this c_id along with the integer 1. In lines 2-6, the Combiner performs local aggregation where it calculates the p_sum, and p_count of dps in the list(values) for each c_id. Next, in line 7, it outputs key-value pairs where key is the c_id, and value is an object composed of the p_sum and p_count.

Reducer Algorithm
After the execution of the Combiner, the Reducer receives (key, list(values)) pairs, where key is the c_id and each value is composed of p_sum and p_count. In lines 2-6 of Algorithm-4**,** instead of iterating over all the dps that belong to a certain c_id, p_sum and p_count are accumulated and stored in total_sum and total_count, respectively. Next, the new centroid is calculated and added to new_c_list. In lines 9-11, a convergence criterion is tested. If the test holds, then the global_counter is incremented by one, otherwise, the global_counter's value does not change (stays zero) and the algorithm is terminated by the Driver.

Algorithm-4: *Reducer*

```
Method    setup ( )
 1:  Load centroids to c_list; //holds current centroids
 2:  global_counter = 0;
 3:  Initialise new_c_list; //holds updated centroids
Method    reduce(c_id, list<values>)
 1:  total_sum, total_count,  new_centroid, old_centroid  = 0;
 2:  for value in values do
 3:      Extract dp vector from value;
 4:      total_sum := total_sum + value.get_p_sum();
 5:      total_count := total_count + value.get_p_count();
 6:  end for
 7:  new_centroid :=  total_sum /  total_count;
 8:  add new_centroid to new_c_list
 9:  if new_centroid has changed  or a threshold is not reached then
 10:     Increment global_counter by 1
 11: end if
 12: output(c_id, dp)
Method    cleanup( )
 1:  Write new centroids in new_c_list to HDFS;
```

3 Experimental Results

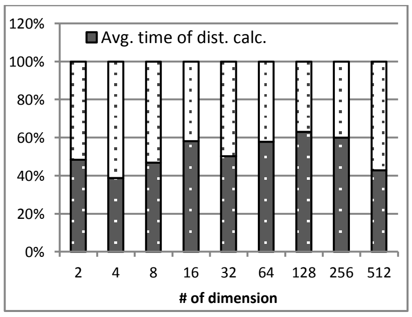
To evaluate PKMMR, we run the algorithm on a Hadoop [14] 2.2.0 cluster of 1 master node and 16 worker nodes. The master node has 2 AMD CPUs running at 3.1GHz with 8 cores each, and 8x8GB DDR3 RAM, and 6x3TB Near Line SAS disks running at 7200 rpm. Each worker node has 1 Intel CPU running at 3.1 GHz with 4 cores, and 4x4GB DDR3 RAM, and a 1x1TB SATA disk running at 7200 rpm.

The datasets used in the experiments are artificially generated where data points are randomly distributed. Additionally, initial cluster centroids are randomly picked from the dataset [1]. The number of iterations is fixed in all experiments at 10.

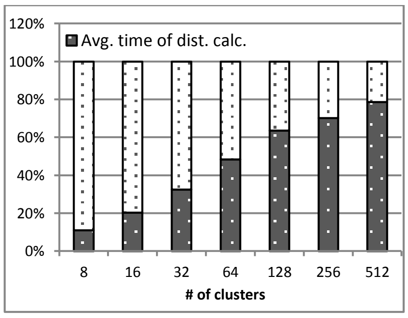
To show the effect of distance calculations on the performance of PKMMR, we run the algorithm with different number of data points *n*, dimensions *d* and clusters *k*. The percentage of the average time consumed by distance calculations in each iteration is represented by the grey area in each bar in the Fig. 1-(a), 1-(b), and 1-(c). The white dotted area represents the percentage of the average time consumed by other MapReduce operations per iteration including job configuration and distribution, map tasks (excluding distance calculations) and reduce tasks.

In each run, we compute the average run-time for one iteration by dividing the total run-time over the number of iterations. Then, the average run-time consumed by distance calculations per iteration is computed.

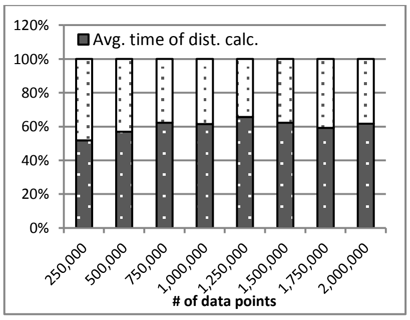
We run PKMMR with a varied number of *d*, while *n* is fixed at 1,000,000, and *k* is fixed at 128. Fig.1-(a) shows that 39% (*d*=4) to 63% (*d*=128) of the average iteration time is consumed by distance calculations.



(a) Avg. time consumption with variable number of *d*. *n*=1000000, *k*=128.



(b) Avg. time consumption with variable number of *k*. *n*=1000000, *d*=128.



(c) Avg. time consumption with variable number of *n*. *d*=128, *k*=128.

Fig. 1. Percentage of the average consumed time by distance calculations per iteration with variable number of *d*, *k* and *n*.

PKMMR is also run with a variable number of *k*, while *n* is set to 1,000,000 and *d* is set to 128. In Fig.1-(b), it can be clearly seen the tremendous increase in the percentage of consumed time by distance calculations per iteration from 11% (*k*=8) to 79% (*k*=512). In this experiment, distance calculations become a performance bottleneck as the number of clusters increases, which is more likely to occur while processing large-scale datasets.

Fig. 1-(c) illustrates the percentage of the average time of distance calculations when running PKMMR with variable number of *n*, while *d*=128 and *k*=128. As it can be observed, distance calculations consume most of the iteration time. About 65% of the iteration time is spent on distance calculations when *n*=1,250,000. Therefore, reducing the number of required distance calculations will most likely accelerates the iteration run-time and, consequently, improves the overall run-time of PKMMR.

4 Work in Progress

We intend to accelerate the performance of K-Means on MapReduce by applying two methods to reduce the distance computations in each iteration. Firstly, triangle inequality optimisation techniques are going to be implemented and tested with high dimensional datasets. However, such techniques usually require extra information to be stored and transferred from one iteration to the next. As a consequence, large I/O and communication overheads may hinder the effectiveness of this approach if not taken into careful consideration. Secondly, efficient data structures, such as KD-trees or other space-partitioning data structures [15], will be adapted to MapReduce and used with K-Means. Two issues will be investigated in this approach. First, inefficient performance with high dimensional datasets that has been reported in [6]. Second, load imbalance that was addressed in [8][9].

5 Conclusions

In this paper we have described the implementation of parallel K-Means on the MapReduce framework. Additionally, a detailed explanation of the steps to control the iterative procedure in MapReduce has been presented. Moreover, a detailed analysis of the average time consumed by distance calculations per iteration has been discussed. From the preliminary results, it can be clearly seen that most of the iteration time is consumed by distance calculations. Hence, reducing this time might contribute in accelerating K-Means on the MapReduce framework. Two approaches are under investigations, which are, respectively, based on the triangle inequality property and space-partitioning data structures.

References

1. S. Lloyd, “Least Squares Quantization in PCM,” *IEEE Trans Inf Theor*, vol. 28, no. 2, pp. 129–137, 1982.
2. I. S. Dhillon and D. S. Modha, “A Data-Clustering Algorithm on Distributed Memory Multiprocessors,” in *KDD’99 Workshop on High Performance Knowledge Discovery*, London, UK, 1999, pp. 245–260.
3. J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” in *Proceedings of the 6th Conference on Symposium on Operating Systems Design & Implementation - Volume 6*, Berkeley, CA, USA, 2004, pp. 10–10.
4. C. Elkan, “Using the Triangle Inequality to Accelerate k-Means,” presented at the International Conference on Machine Learning - ICML, 2003, pp. 147–153.
5. J. Bentley, “Multidimensional binary search trees used for associative searching,” *Commun ACM*, vol. 18, no. 9, pp. 509–517, 1975.
6. D. Pelleg and A. Moore, “Accelerating Exact K-means Algorithms with Geometric Reasoning,” in *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, New York, NY, USA, 1999, pp. 277–281.
7. D. Judd, P. K. Mckinley, and A. K. Jain, “Large-scale parallel data clustering,” *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 20, pp. 871–876, 1998.
8. D. Pettinger and G. Di Fatta, “Scalability of efficient parallel K-Means,” in *2009 5th IEEE International Conference on E-Science Workshops*, 2009, pp. 96–101.
9. G. Di Fatta and D. Pettinger, “Dynamic Load Balancing in Parallel KD-Tree K-Means”, IEEE Int.l Conference on Scalable Computing and Communications, 2010, pp. 2478-2485.
10. S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google File System,” in *Proceedings of the 19th ACM Symposium on Operating Systems Principles*, New York, NY, USA, 2003, pp. 29–43.
11. K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop Distributed File System,” in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, Washington, DC, USA, 2010, pp. 1–10.
12. W. Zhao, H. Ma, and Q. He, “Parallel K-Means Clustering Based on MapReduce,” in *Cloud Computing*, 2009, pp. 674–679.
13. B. White, T. Yeh, J. Lin, and L. Davis, “Web-scale Computer Vision Using MapReduce for Multimedia Data Mining,” in *Proceedings of the Tenth International Workshop on Multimedia Data Mining*, New York, NY, USA, 2010, pp. 9:1–9:10.
14. Apache Hadoop [Online]. Available: http://hadoop.apache.org/. [Accessed: 03-Jan-2015].
15. D. Pettinger, G. Di Fatta, “Space Partitioning for Scalable K-Means”, IEEE The Ninth International Conference on Machine Learning and Applications (ICMLA 2010), 12-14 Dec. 2010, Washington DC, USA, pp. 319-324.

2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress

Efficient Parallel K-Means on MapReduce Using Triangle Inequality

Sami Al Ghamdi, Giuseppe Di Fatta
Department of Computer Science
University of Reading
Whiteknights, Reading, RG6 6AY, United Kingdom
Email: s.a.m.alghamdi@pgr.reading.ac.uk, g.difatta@reading.ac.uk

***Abstract*—K-Means is one of the most efficient and popular clustering algorithms that has been around for more than 50 years. The naive implementation of K-Means spends the vast majority of its time computing redundant distance calculations from each point to all cluster centres. This issue has been extensively studied and methods based on the triangle inequality principle have been used to eliminate unnecessary distance calculations. Most triangle inequality optimisations cache extra information (distance bounds and cluster assignments) from one iteration to eliminate the need of computing exact distances in the next. This work takes these optimisations one step further and integrates them into an accelerated version of K-Means on a well-known distributed computing framework known as MapReduce to produce an efficient and highly scalable K-Means for big data. Although MapReduce is considered as one of the most reliable and fault tolerant distributed computing frameworks, one of its major drawback is that it does not support iterative algorithms such as K-Means, and does not cache any data between two consecutive iterations, which is required in most triangle inequality optimisations. Therefore, this work introduces two new approaches to pass information from one iteration to the next to accelerate K-Means. The first approach is called K-Means on MapReduce using Extended Vector (KMMR-EV). The second approach is called K-Means on MapReduce using Bounds Files (KMMR-BF). These approaches achieve speedups up to 4.5x for KMMR-EV and 6.8x for KMMR-BF, with respect to the naive implementation of K-Means on MapReduce (KMMR-N). An extensive experimental work, with real and synthetic datasets, has been conducted on Apache Hadoop (an open-source implementation of MapReduce), along with an overhead analysis to show the effectiveness of both approaches.**

I. Introduction

Due to the vast amounts of data that has been generated during the last decade, new technologies and algorithms had to be developed to cope with this exponential explosion of generated data. Cluster analysis is one of the important techniques that aims to explore the hidden structure of the data. Clustering is the process of dividing the data into groups (clusters), where data points in the same group has more similarities than data points in other groups [1].

Selected as one the top ten data mining algorithms [2], the K-Means algorithm [3] [4] [5] is considered as one the most widely used clustering algorithms [6]. K-Means gained its popularity from its simplicity and efficiency. Given a set $\mathcal{X} = \{x_1, x_2, ..., x_n\}$, where n is the number of data points in a d-dimensional space $\mathbb{R}^d$, partitioned into k clusters $\mathcal{C} = \{C_1, C_2, ..., C_k\}$, K-Means aims to minimise the Sum of Squared Error $SSE = \sum_{j=1}^{k} \sum_{x \in c_j} \parallel x - c_j \parallel^2$, where j is the index of the j-th $C \in \mathcal{C}$, and c_j is the centroid (mean of points) of C_j. The most used and straightforward implementation of K-Means is known as Lloyd's algorithm [1], which in this work is referred to as *Naive K-Mean*. The Naive K-Means algorithm starts by randomly picking k initial cluster centroids. Then, the distance from each $x \in \mathcal{X}$ to each $c_j \in \mathcal{C}$ is computed and x gets assigned to its closest c_j. Next, C_j is moved to the mean of all points assigned to it. The algorithm iterates until it converges (where cluster centroids do not move any more), or an early termination condition is met. K-Means finds a local minimum solution in $O(ndk)$ running time per iteration.

This work aims to improve the scalability and efficiency of the Naive K-Means, taking into consideration two conditions: 1) obtaining the exact same final centres as the Naive K-Mean; 2) working on an original and unmodified scalable distributed framework. Therfore, this work introduces a parallel implementation of Elkan's K-Means [6], which is deterministically equivalent to Naive K-Mean, on a well-known distributed framework known as MapReduce [7]. Apache Hadoop [8] is the most famous open source implementation of MapReduce, and the algorithms' implementations in this work is based on Hadoop's programming API. Elkan's algorithm, efficiently, eliminates redundant distance computations related to each data point and produces the same exact results as the Naive K-Means. The number of distance computations is reduced to be closer to $O(n)$ instead of $O(nk)$ per iteration as in the Naive K-Means. Section III-A explains triangle inequality in detail.

To apply triangle inequality to K-Means, extra information (bounds and cluster assignments) is required to be passed from one iteration to the next to avoid unnecessary distance computations. This is not a straightforward process in MapReduce because the framework does not directly support iterative algorithms, and does not cache data between two consecutive MapReduce jobs [9]. Therefore, this work presents two new approaches that pass information from one iteration to the next in MapReduce to accelerate K-Means. The first approach called, K-Means on MapReduce using Extended Vector (*KMMR-EV*), appends the extra information in the current iteration to the input data vector and forms an extended vector that is used as an input in the next iteration. The second approach is K-Means on MapReduce using Bounds Files (*KMMR-BF*), where extra information is written into separate files on Hadoop Distributed File System (HDFS), and the required information is retrieved on next iterations from these files. Both approaches have been extensively experimented on Hadoop, with real and synthetic datasets with variable values for *d* and *k* that range from 4 to 2048. The results show improvement in efficiency for both approaches, compared to Naive K-Means on MapReduce (KMMR-N), with speedups of 6.8x for KMMR-BF, and 4.5x for KMMR-EV when $k = 1024$ and $d = 512$.

Many alternative ditributed computing frameworks, such as Spark [10] and Twister [9], that support iterative algorithms could be used instead of Hadoop. However, this work shows that significant speedups can be achieved when using the MapReduce paradigm on Hadoop by optimising the algorithms and using these optimised algorithms as drop-in solutions.

The rest of the paper is organised as follows: Section 2 reviews the related work. Section 3 explains the triangle inequality and how it can be applied to K-Means and explains the implementation of the two new approaches. Section 4 discusses the experimental results. Finally, section 5 concludes the paper and discusses the future work.

II. Related Work

Triangle inequality has been used in many works to eliminate unnecessary distance computations. Elkan [6] introduced an efficient K-Means algorithm, which this work builds on, using triangle inequality. For each point, Elkan's algorithm keeps one upper-bound on the distance from that point to its closest centroid, and *k* lower-bounds on the distance from the same point to each centroid. Although the algorithm works efficiently with high dimensional data, it needs to store and retrieve: *n* upper-bounds, *n* cluster assignments (*cid*), and *nk* lower-bounds. It also requires $O(k^2)$ time to compute centre-centre distances. In [11] and [12], the authors, independently and differently, extend Elkan's algorithm to make it work with fewer number of lower bounds in an attempt to reduce the resulted overhead from maintaining *nk* lower-bounds as in Elkan's. In [13], Elkan's algorithm has been parallelised over a shared-memory, multicore machine, and compared against other variants of K-Means.

A parallel implementation of K-Means on distributed memory multiprocessors based on Message Passing Interface MPI was introduced by Dhillon and Modha in [14]. The algorithm partitions the original dataset into a number of subsets. Then, each processor works on an independent subset where distance calculations are performed and each point is assigned to its closest centroid. Then, partial sums and *SSE*s are collected and new centroids are calculated. This process is repeated until the algorithm converges. The authors in [15] introduced parallel K-Means based on MPI and used a method called Spheres of Guaranteed Assignment which follows the concept of pruning unnecessary distance calculations per iteration based on triangle inequality but without maintaining upper or lower bounds.

In [16], the authors presented a parallel K-Means on Twister [9], which is an optimised implementation of the MapReduce framework that supports iterative algorithms based on publish/subscribe messaging infrastructure and caches static data in memory, to cluster high dimensional social image data. The authors applied triangle inequality to reduce distance computations based on Elkan's [6] work, except that instead of keeping *nk* lower-bounds, they keep fewer number of lower-bounds. The work presented in this paper is different in terms of using the standard unoptimised MapReduce programming paradigm which is implemented by Haddop. The work in [17] breaks down the running time of each iteration on K-Means on MapReduce and shows that the time to compute distances is the bottleneck in K-Means.

K-Means++ [18] is a variant of K-Means which carefully selects the initial set of centroids that has a constant factor away from the optimum solution. K-means|| or Scalable K-mean++ [19] and Competitive K-Means [20], address a downside of the k-means++ initialisation which is its inherently sequential nature and provide solutions to make it work efficiently on a parallel environment, specifically, MapReduce.

In [21], K-Means is implemented on MapReduce and its efficiency is improved by using locality sensitive hashing *LSH* to divide points into buckets where the original points are transformed into weighted representative points. This method is used to prune unnecessary distance computations by computing the distance of a given point with only a small number of centres that exist in the same bucket as the point. The algorithm was tested with real datasets and shows improvement in speed by 67% and 76% when *k* is 1500 and 3000 respectively, compared to scalable K-Means++. However, the dimensionality of both datasets is low (26 and 41 dimensions) which does not give a full understanding of the algorithm's behaviour with high dimensional datasets.

III. K-Means on MapReduce

A. K-Means and Triangle Inequality

This section briefly reviews Elkan's [6] method to reduce the number of distance computations by using trinagle inequality. The triangle inequality property states that, for any three points a, b, and c, $\| a-c \| \leq \| a-b \| + \| b-c \|$. Elkan's algorithm applies the triangle inequality to K-Means as follows: let *c* and c' be two centroids, where *c* is the closest centroid to point *x*, and c' is any other centroid:

$$if \;\; \tfrac{1}{2} \| c - c' \| \geq \| x - c \| \;\; then \;\; \| x - c' \| \geq \| x - c \| \quad (1)$$

which means that *x* is closer to *c* and there is no need to calculate $\| x - c' \|$.

Elkan's algorithm sets an upper-bound (u) on the distance between *x* and *c*, where *c* is the closest centroid to *x*, such that $u \geq \| x - c \|$. From (1), if $u \leq \frac{1}{2} min_{c' \neq c} \| c - c' \|$, then all distance calculations related to point *x* can be avoided because no other centroid can be closer to *x* than the current centroid *c*. Furthermore, a lower-bound (l) is set on the distance between x and each centroid c', such that $l \leq \| x - c' \|$.

TABLE I
NOTATIONS DESCRIPTION

Notation	Description
X	The input dataset of size n
k	Number of clusters
C[]	An array of size k holds the list of centroids
newC[]	An array of size k holds the list of new centroids
S[][]	A 2D-array holds $k\times$ k centre-centre distances
P[]	A boolean array of size k holds the status of centroids
H[]	An array of size k holds half minimum distance to the second closest centroid from each centroid
u	upper-bound from each $x \in X$ to closest $c \in C$
l	k lower-bounds from each $x \in X$ to all $c \in C$
cid	Cluster index (ID) of the closest $c \in C$ from $x \in X$

Before the end of each iteration, upper and lower bounds are updated by, first, computing the distance moved by each centroid, then adding this distance to the upper-bound, and subtracting it from the each lower-bound. The algorithm caches centre-centre distances along with half the distance between each centre and its closest other centre. Further explanation of the process of eliminating distance computations can be found in section III-D.

Table I describes the notations that appear in following sections.

B. MapReduce and Hadoop

MapReduce [7], is a distributed computing framework that was introduced by Google in 2004, and Apache Hadoop [8] is the most famous open-source platform that implements MapReduce. In MapReduce, the input dataset is partitioned into subsets, known as *input-splits*. These input-splits are distributed over Hadoop Distributed File System (HDFS). The input-splits are then transferred to the mappers as key-value pairs. Each mapper represents a node in the cluster and processes an individual input-split. The output of each mapper is sorted by key and sent to the allocated reducer as key-value pairs. Each reducer receives the output of the mappers as key-list(values) pairs, which represent the values that was associated to each key by the mapper. Finally, each reducer outputs the results to HDFS. An optional optimisation known as a *combiner* can be used to minimise the amount of intermediate data transferred from mappers to reducers across the network by performing a local aggregation over the intermediate data [22].

C. KMMR-N: Naive K-Means on MapReduce

Due to the lack of support to iterative algorithms in the MapReduce framework, a controller program, called here a *Driver*, has been implemented to control the iterative process, merge the centroids' files that are emitted by Reducers, and checks the algorithm's convergence. In the map phase, each mapper processes an individual input-split that is received from HDFS. Inside the Mapper, centroids are loaded to memory in the *setup* method. After that, the map function receives the byte offset and the data point as key-value pairs. The map function assigns each data point to its closest centroid and outputs the centroid's index (*cid*) as the key, and data point as the value. Then, the Mapper's output is sorted by *cid* and sent or shuffled to the assigned Reducer. Each Reducer receives the *cid* as key and the assigned data points to this cid as a list of iterable values. After loading the centroids to memory in the *setup* method, the reduce method re-computes the new centroid and compares the new and old centroids. If the centroids are not equal, the convergence status is set to not converged, consequently, the Driver runs a new iteration. Finally, each Reducer outputs the new centroids to HDFS and the centroids from all Reducers are merged in the *Driver*.

D. K-Means on MapReduce using Triangle Inequality

To apply the triangle inequality to K-Means, the algorithm requires extra information to be passed from one iteration to the next. In particular, it requires the following for each input data point: an upper-bound from the data point to its closest centroid, the *cid* of the assigned cluster, and *k* lower-bounds from each data point to each centroid. This work introduces two approaches that implement KMMR with triangle inequality:

1) KMMR-EV: K-Means on MapReduce using Extended Vector: This approach applies the triangle inequality to KMMR by extending the input data vector. That is, in the current iteration, required extra information is appended to the original data vector which forms the extended vector (EV) and this vector is then written to HDFS and used as an input in the next iteration. Fig. 1 shows the structure of the extended vector which starts with *k* lower-bounds, followed by an upper-bound, the assigned *cid*, and the original vector of size *d* dimensions. So, the EV's size becomes: $d + k + 2$.

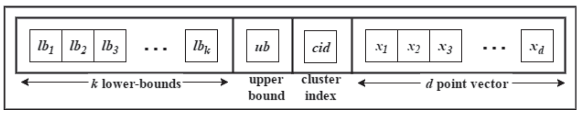


Fig. 1. Structure of Extended Vector (*EV*)

The implementation of KMMR-EV consists of three major phases, a *driver* that controls the iterative process, a *map* phase that assigns each point to its closest centroid, and a *reduce* phase that computes the means of the points assigned to each *cid* and produce new centroids. The detailed explanation of the phases is as follows:

Driver: The Driver (Algorithm 1) randomly picks the initial set of centroids, initialises the MapReduce job, and controls the iterative process.

Map phase: Since distances are not initialised yet, the mapper in the first iteration (Algorithm 2) works as an initialisation step to initialise upper and lower bounds. Each time the distance from $ev.point$ to any centroid c_j is calculated, the lower-bound that corresponds c_j is set to that distance in line 11. Line 14 sets the upper-bound to the distance from the point to its closest centroid. The boolean array *skip[]* is initialised to *false* and holds the status of each centroid. In line 9, if *true*, the distance calculation to centroid c_j is skipped, otherwise,

Algorithm 1: Driver(k, X)

```
select k initial cluster centroids randomly
iteration ← 1
while not converged or an early termination condition is not met do
    send the centroids' file to all computing nodes
    if $iteration == 1$ then
        set mapper to EVInitMapper //Algorithm 2
    else
        set mapper to EVElkanMapper //Algorithm 3
    end
    run the MapReduce job
    if $numberofreducers > 1$ then
        merge reducers output into one file
    end
    reducer/s (Algorithm 4) check for convergence
    $iteration \leftarrow iteration + 1$
end
```

Algorithm 2: EVInitMapper(k, C)

```
Function setup():
    compute S[][]
Function map(key, value)
    declare new ExtendedVector ev
    ev.point ← $value$
    initialise all values in skip[k] to $false$
    min_distance ← $\infty$
    for $j \leftarrow 0$ to $k - 1$ do
        if skip[j] then continue
        distance ← getDistance(ev.point, C[j])
        ev.l(j, distance) //set l(x, j)
        if $distance < min\_distance$ then
            min_distance ← distance
            ev.u ← min_distance //set u(x)
            ev.cid ← c //set cid
            for $z \leftarrow j + 1$ to $k - 1$ do
                if $S[j][z] \geq 2 * distance$ then
                    skip[z] ← true
                end
            end
        end
    end
    writeToHDFS(ev)
    output(ev.cid, ev.point)
```

the distance is calculated. Lines 16-20 uses inequality (1) to set the skip status of the next centroid in the centroids array $C[]$.

The second mapper (Algorithm 3), which is executed on iterations > 1, takes as input a key-value pair, where $value$ is the Extended Vector that was stored by the mapper in the previous iteration. In lines 5-8, the upper and lower-bounds are updated by adding the centroid's movement to the upper-bound, and subtracting it from each lower-bound. The centroid's movement is part of the data structure that holds the centroid's vector and is computed and stored at the end of the reduce stage. The movement can be retrieved by calling getMovement(). In line 11, $H[ev.cid]$ is half the distance from the centroid associated with the current point to its closest other centroid. If the test holds, all distance calculations associated to the currently processed point are skipped. In line 15, $S[ev.cid][j]$ is the distance from the centroid that was assigned to the current point to centroid c_j. If the tests in lines 13-15 does not hold, the distance computation to currently processed centroid is skipped. The distance from the current point to any centroid other than one assigned to the point does not get calculated until line 24, where the tests at line 23 repeats the tests at line 13 but with an updated *u*. All the bounds are stored to the ExtendedVector object (ev) and, finally, ev is written to HDFS, and the mapper outputs the point (ev.point) with the index of its closest centroid (ev.cid).

Reduce phase: the implementation of the reducer (Algorithm 4) is the same in all approaches including KMMR-N. Each reducer receives the *cid* as the key and the list of points assigned to *cid* as list of values. Each reducer processes each *cid* with its associated points independently. The reducer iterates over the points and computes the average of all points assigned to input *cid* to produce the new centroid. The old and new centroids are compared and if they are not equal or does not satisfy a certain threshold, the convergence status is set to *not converged*, which, consequently, makes the *Driver* runs one more iteration.

2) KMMR-BF: K-Means on MapReduce using Bounds Files: This approach stores the extra information into files called Bounds Files (*BF*). For each point, each mapper stores one upper-bound, *k* lower-bounds, and the *cid* of the assigned cluster centroid to the point. Each BF corresponds to an input-split processed by a mapper. To find out which BF corresponds to which input-split, the BF's name is set to be the *starting byte offset* of the currently processed input-split. So, in the current iteration the mapper looks in HDFS for the BF's name, which was stored in last iteration, that matches the currently processed input-split's starting byte offset and loads all the extra information to memory. The size of each BF would be $\frac{n}{p}(k + 2)$, where *p* is the number of mappers.

Since most of this algorithm's implementation matches KMMR-EV implementation in the last section, the main implementation's differences will be pointed out in this section. As in KMMR-EV, KMMR-BF has two mapper implementations, the first mapper runs on the first iteration and the second runs on iterations > 1. The triangle inequality is also applied in the same way as KMMR-EV. The main difference is in the way of storing and retrieving extra information. KMMR-BF reads the bounds from BF and loads them to memory in the *setup* method. Then, it proceeds the centres elimination process as in KMMR-EV. After that it assigns the point to its closest centre and outputs the *cid* with its associated point to reducers. Next, when each mapper finishes processing all

Algorithm 3: EVElkanMapper(*k*, *C*)

```
Function setup():
    compute S[][]
Function map(key, value):
    create a new ExtendedVector ev from $value$
    for $j \leftarrow 0$ to $k-1$ do
        ev.l(j) ← max[ev.l(j) − getMovement(j), 0]
    end
    ev.u ← ev.u + getMovement(ev.cid)
    update_u ← true //flag to check if u is updated
    d1, d2 ← 0
    if $ev.u \leqslant H[ev.cid]$ then continue
    for $j \leftarrow 0$ to $k-1$ do
        if $(j \neq ev.cid)$
        & $(ev.u > ev.l(j))$
        & $(ev.u > S[ev.cid][j] * 0.5)$ then
            if $update\_u$ then
                d1 ← getDistance(ev.point, c(ev.cid))
                ev.u ← d1
                ev.l(ev.cid) ← d1
                update_u ← false
            end
            d1 ← ev.u
            if $d1 > ev.l(j)$ or $d1 > S[ev.cid][j] * 0.5$
            then
                d2 ← getDistance(ev.point, c(j))
                ev.l(j) ← d2
                if $d2 < d1$ then
                    ev.cid ← j
                    ev.u ← d2
                    update_u ← false
                end
            end
        end
        writeToHDFS(ev)
        output (ev.cid, ev.point)
    end
```

n/p records, where p is the number of mappers, the updated bounds are written to a BF on HDFS in the *cleanup* method.

E. Overhead Analysis

Both approaches generate I/O and memory overheads that could become the performance bottleneck for large number of *k* clusters and *n* points in *d* dimensions. Because Hadoop's files are immutable (i.e. records cannot be modified). Therefore, any information that needs to be passed from one iteration to the next must be written to a file and then read in the next iteration. In case of KMMR-EV, for *p* number of mappers, each mapper writes *n/p* EVs, each EV contains *d* input data vector, *k* lower-bounds from each point to all centroids, an *u* from each point to its closest centroid, and the *cid* of the closest centroid. So, KMMR-EV writes $\frac{n}{p}(d+k+2)$ EVs to HDFS in each iteration.

Algorithm 4: EVReducer(*k*, *C*)

```
Function setup():
    initialise newC[k]
Function reduce(cid, points):
    sum ← (0,0,0,...,0)
    foreach $p \in points$ do
        sum ← sum + $p$
    end
    $newC[cid] \leftarrow$ sum$/|points|$
    if $newC[cid] \neq C[cid]$ then not converged yet

Function cleanup():
    writeToHDFS(newC)
```

KMMR-BF, on the other hand, requires $\frac{n}{p}(k+2)$ read/write operations for *k* lower-bounds, an upper-bound, and a *cid*, which results in lower overhead compared to KMMR-EV.

In terms of memory overhead, both approaches require $O(k^2)$ memory space and time for centre-centre distances in each iteration. Furthermore, KMMR-BF requires $\frac{n}{p}(k+2)$ memory for lower/upper bounds and cluster assignments. So, the whole memory space required by KMMR-BF is $O(\frac{n}{p}(k+2)+k^2)$.

IV. Experimental Results

A. Experiment Setup

The experiments are executed on a Hadoop 2.6.0 cluster of 1 master node and 16 worker nodes. The master node has 2 AMD CPUs running at 3.1GHz with 8 cores each, and 8x8GB DDR3 RAM, and 6x3TB Near Line SAS disks running at 7200 rpm. Each worker node has 1 Intel CPU running at 3.1 GHz with 4 cores, and 4x4GB DDR3 RAM, and a 1x1TB SATA disk running at 7200 rpm.

The number of iterations is fixed to 20 iterations for all tests. We chose 20 because, after few iterations the centres do not move much [6], hence, 20 iterations will give a good intuition of the performance in general and the number of skipped distance calculations per iteration. Each experiment is run for 10 times and the average is reported. The number of reducers in all experiments = 1.

The wall-clock time is measured for all experiments and after measuring the average time spent by each mapper/reducer, the average time per iteration is calculated. For example, let each mapper runs in T_m time, then the average map time per iteration T_M for *M* number of mappers where $m \in M$, that iterates for *E* number of iterations where $e \in E$ is $T_M = \frac{1}{E}\sum_{e=1}^{E}\sum_{m=1}^{M}\frac{T_m}{M}$.

The speedup of the new approaches is compared against the KMMR-N where speedup is calculated as the average iteration time of KMMR-N divided by the average iteration time of the optimised approach.

B. Datasets

Real and synthetic datasets have been used in the experiments. The real dataset is:

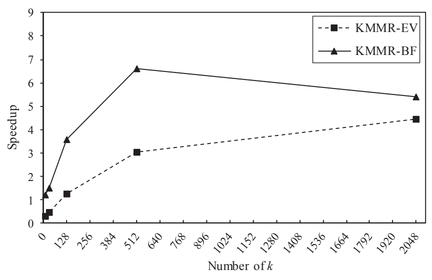


Fig. 2. Speedup relative to KMMR-N with variable k

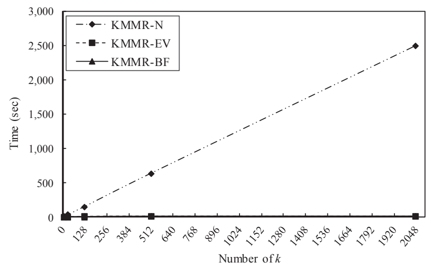


Fig. 4. Average time to compute distances per iteration with variable k

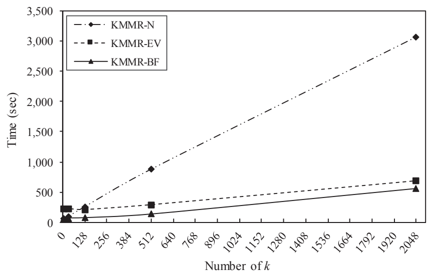


Fig. 3. Average run time per iteration with variable k

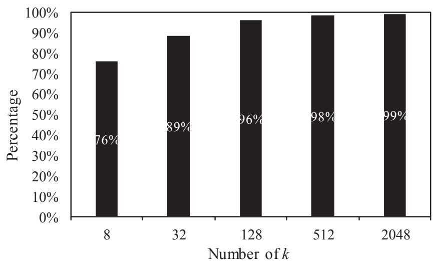


Fig. 5. Percentage of skipped dist. calc. per iteration with variable k

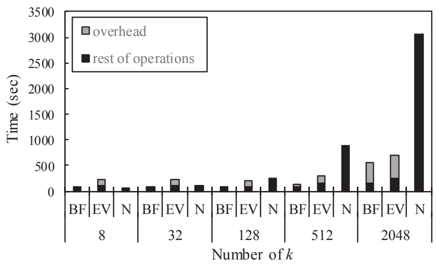


Fig. 6. Average overhead per iteration vs. rest of operations with variable k

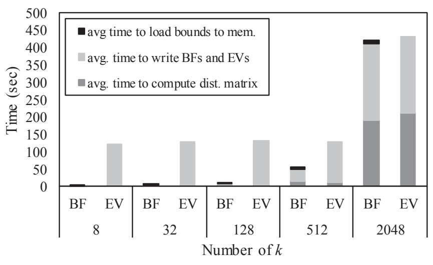


Fig. 7. Overhead average time per iteration broken down into three main operations with variable k

1) **KDD04**: A public dataset for protein homology prediction task of KDDCUP 2004 [23]. It contains 145751 points in 74 dimensions which describe the sequence alignment match between the native protein sequence and the sequence that is tested for homology with size of 63 MB (MB=Megabyte, GB=Gigabyte).
The next synthetic datasets were generated with well separated spherical Gaussian clusters:
2) **GaussianD-*d***: 10 datasets have been generated to test the performance with variable dimensions d, where d = 4, 8, 16, 32, 64, 128, 256, 512, 1024 and 2048, n = 100,000 and k = 128, and the size ranges from 7 MB for GaussianD-*4* to 3.7 GB for GaussianD-*2048*. GaussianD-*512* was used on testing the performance with variable k.
3) **GaussianN-*n***: Four datasets have been artificially generated to test the performance with variable number of data points n, where $n = 5 \times 10^5, 10^6, 5 \times 10^6$ and 10^7, in 64 dimensions and with the size ranges from 590 MB for GaussianN-5×10^5 to 11.5 GB for GaussianN-10^7.

C. Results

The dataset GaussianD-512 is used in testing the effect of variable number of clusters k. Fig. 2 and 3 show the speedup and the runtime per iteration for KMMR-BF and KMMR-EV relative to KMMR-N with variable number of clusters k. For low to medium k (4-32), KMMR-BF and KMMR-EV run almost with the same speed as KMMR-N. However, the gain in speed for both starts when k=64 and reaches the peek when k=1024 (6.8x for KMMR-BF and 4.5x for KMMR-EV) where the benefit from eliminating distance calculations outweighs the overhead from computing distance matrix, and reading/writing bounds. Both algorithms start to get slower when k=2048 because the amount of overhead that increases with respect to k. However, even with the large amount of overhead, both algorithms are still faster than KMMR-N because as the overhead time in KMMR-BF and KMMR-EV

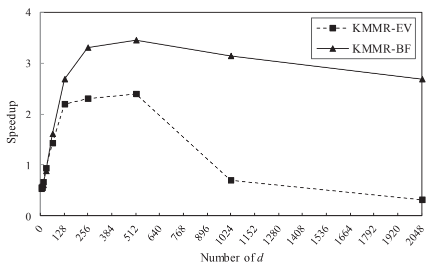


Fig. 8. Speedup relative to KMMR-N with variable d, ($k = 128$)

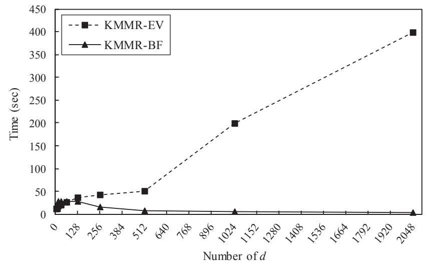


Fig. 9. Average time per iteration to write EVs and BFs with variable d

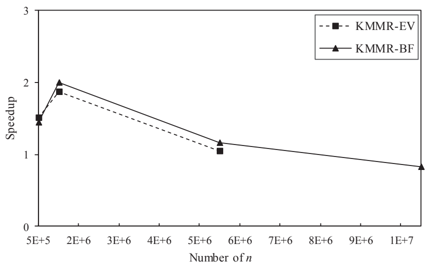


Fig. 10. Speedup relative to KMMR-N with variable n

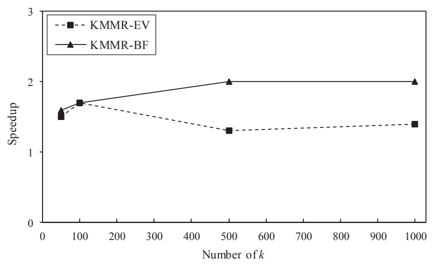


Fig. 11. Speedup relative to KMMR-N for dataset KDD04 with variable k

increases with respect to k, the time of computing distances also increases in KMMR-N and the time consumed by the overhead in KMMR-BF and KMMR-EV is compensated by the saved time from eliminating distance computations.

Fig. 4 shows the average time of distance calculations per iteration which is linear to k and dominates the runtime in KMMR-N, while its insignificant and almost constant in KMMR-BF and KMMR-EV. Fig. 5 shows the percentage of the average skipped distance computations per iteration where it reaches 99% for large k.

For lack of space in Fig. 6 and 7, KMMR-N, KMMR-EV, and KMMR-BF are shortened to N, EV, and BF, respectively. In Fig. 6, each bar represents the average iteration time divided into: overhead time, and rest of operations time, where the former does not apply to KMMR-N. The overhead time consists of three major operation: 1) computing k^2 centre-centre distances, 2) reading bounds from HDFS and loading them into memory (occurs in KMMR-BF only), and 3) writing EVs and BFs to HDFS. The rest operations time includes: the time to stup and distribute the job, point-centre distance calculations, shuffle, reduce, data files replication, etc. Three remarks can be noticed: 1) For large number of k ($k > 512$) the overhead becomes the performance bottleneck instead of distance calculations. 2) Despite the generated overhead, both approaches run faster than KMMR-N because of the gain in speed from skipping distance calculations. 3) For a small number of k, the overhead in KMMR-EV dominates the iteration run time which makes it slower than KMMR-N and KMMR-BF.

Fig. 7 shows the average time of overhead per iteration broken into three three major operations that were mentioned previously. It can be seen that KMMR-EV struggles more than KMMR-BF with the large overhead created from writing EVs to HDFS even with small k. The time to compute the matrix of centre-centre distances is obviously the same in both approaches and can significantly affects the running time with large values of k as can be notices in $k = 2048$. The time to load BFs to memory is not significant in KMMR-BF even with large values of k.

Fig. 8 shows the speedup relative to KMMR-N, and Fig. 9 shows the average time to write EVs and BFs to HDFS in KMMR-BF and KMMR-EV per iteration. The datasets GaussianD-d have been used in this test, where d varies from 4 to 2048, and k is fixed at 128 clusters. As the number of d increases, KMMR-BF gains speedups (up to 3.7x when $d = 512$), while KMMR-EV gains speedup up to 2.1x when $d = 256$ and its performance drops as d increases (0.3x when d=2048) because of the large overhead generated from writing $\frac{n}{p}(d+k+2)$ EVs per mapper in each iteration, where p is the number of mappers. As can be seen in Fig. 9 that the time to write EVs to HDFS increases dramatically as d increases in KMMR-EV.

Fig. 10 shows the speedup of KMMR-EV and KMMR-BF relative to KMMR-N using the datasets GaussianN-n for variable number of n and fixed values for d=64 and k=100. As the number of n increases, speedup for both approaches reaches $\approx$ 2x when n=10^6 and declines when $n = 10^7$ because the I/O overhead and the shuffle time become the dominant costs. This is because as n increases to very large values, the number of mappers also increases which, consequently, increases the time to write EVs and BFs (which are replicated over data nodes), and increases the communication between mappers and reducers. Adding to these reasons, the limitation in KMMR-EV which makes the algorithm increases the initial number of mappers assigned to the MapReduce job (see section III-D1), testing KMMR-EV with $n = 10^7$ required a large number of mappers (90 mappers), which made the test hard to execute.

Fig. 11 shows the speedup for KMMR-BF and KMMR-EV relative to KMMR-N for the real dataset KDD04. The dataset has been tested with variable values of k = 50, 100, 500 and 1000. When k is 50 to 100, the speedup is ≈ 1.6x for both algorithms. When $500 \leq k \leq 1000$, KMMR-BF's speedup increases to 2x, while KMMR-EV's decreases to $\approx$ 1.3x for both k values. This is because the time to write EVs in KMMR-EV is larger than the time to write BFs, and KMMR-EV reaches the point where it cannot compensate the time spent on reading/writing EVs with the time saved from skipping distance computations.

V. Conclusion

This paper has presented the implementation of two new techniques to introduce triangle inequality optimisations to K-Means on MapReduce. The first approach, KMMR-EV, appends the required bounds to the original input data vector and uses this extended vector (EV) as an input in the next iteration. While in the second approach, KMMR-BF, the bounds are stored on files called bounds files (BF) and the required information are retrieved from these files. An extensive experimental analysis has been carried out using real world and synthetic datasets, to study the performance and the overhead in a wide range of the input parameters, such as variable n points, d dimensions, and k clusters. The experimental results show that both approaches can decrease the running time compared to KMMR-N with speedups of 4.5x for KMMR-EV and 6.8x for KMMR-BF. The trade-off from using these two approaches has been examined in the experimental work and explained in the overhead analysis.

KMMR-BF performs faster than KMMR-EV and KMMR-N when applied to datasets with high dimensional space d, where $128 \leq d \leq 2048$, and large number of clusters k, where $256 \leq k \leq 2048$. For large values of n, d and k (e.g. Fig 8 and 10), the overhead in KMMR-EV could outweighs the gained time from eliminating distance computations and it becomes slower than KMMR-N.

The future work will observe the effeciency of other K-Means optimisations on MapReduce, such as Hamerly's algorithm [11] with only one lower-bound, and Compare-means [24] using only triangle inequality with centre-centre distances. Furthermore, the performance of these algorithms will be examined and compared on different distributed frameworks such as Spark and Twister.

References

[1] A. K. Jain, "Data clustering: 50 years beyond K-means," *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651–666, Jun. 2010.

[2] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg, "Top 10 algorithms in data mining," *Knowledge and Information Systems*, vol. 14, no. 1, pp. 1–37, Jan. 2008.

[3] E. Forgy, "Cluster analysis of multivariate data: efficiency versus interpretability of classifications," *Biometrics*, vol. 21, pp. 768–769, 1965.

[4] J. MacQueen, "Some methods for classification and analysis of multivariate observations." The Regents of the University of California, 1967.

[5] S. Lloyd, "Least Squares Quantization in PCM," *IEEE Trans. Inf. Theor.*, vol. 28, no. 2, pp. 129–137, 1982.

[6] C. Elkan, "Using the Triangle Inequality to Accelerate k-Means," 2003, pp. 147–153.

[7] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI'04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[8] "Welcome to Apache Hadoop!" [Online]. Available: http://hadoop.apache.org/

[9] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, "Twister: A Runtime for Iterative MapReduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC '10. New York, NY, USA: ACM, 2010, pp. 810–818.

[10] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[11] G. Hamerly, "Making k-means Even Faster," in *SDM*, 2010, pp. 130–140.

[12] J. Drake and G. Hamerly, "accelerated k-means with adaptive distance bounds," in *OPT2012: the 5th NIPS Workshop on Optimization for Machine Learning*, 2012.

[13] G. Hamerly and J. Drake, *Accelerating Lloyds Algorithm for k-Means Clustering*. Springer International Publishing, 2015, pp. 41–78, dOI: 10.1007/978-3-319-09259-1_2.

[14] I. S. Dhillon and D. S. Modha, "A Data-Clustering Algorithm on Distributed Memory Multiprocessors," in *KDD99 Workshop on High Performance Knowledge Discovery*. London, UK: Springer-Verlag, 1999, pp. 245–260.

[15] D. Judd, P. K. Mckinley, and A. K. Jain, "Large-scale parallel data clustering," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, pp. 871–876, 1998.

[16] J. Qiu and B. Zhang, "Mammoth Data in the Cloud: Clustering Social Images," in *Cloud computing and big data*. Amsterdam: IOS Press, 2013, pp. 231–246.

[17] S. Ghamdi, G. Fatta, and F. Stahl, "Optimisation Techniques for Parallel K-Means on MapReduce," in *Proceedings of the 8th International Conference on Internet and Distributed Computing Systems - Volume 9258*, ser. IDCS 2015. Windsor, UK: Springer-Verlag New York, Inc., 2015, pp. 193–200.

[18] D. Arthur and S. Vassilvitskii, "K-means++: The Advantages of Careful Seeding," in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '07. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[19] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, "Scalable K-means++," *Proc. VLDB Endow.*, vol. 5, no. 7, pp. 622–633, Mar. 2012.

[20] T. H. Rui Mximo Esteves, "A new approach for accurate distributed cluster analysis for Big Data: competitive K-Means," *Int. J. of Big Data Intelligence*, vol. 1, no. 1/2, pp. 50 – 64, 2014.

[21] Q. Li, P. Wang, W. Wang, H. Hu, Z. Li, and J. Li, "An Efficient K-means Clustering Algorithm on MapReduce," in *Database Systems for Advanced Applications*, ser. Lecture Notes in Computer Science, S. S. Bhowmick, C. E. Dyreson, C. S. Jensen, M. L. Lee, A. Muliantara, and B. Thalheim, Eds. Bali, Indonesia: Springer International Publishing, Apr. 2014, pp. 357–371, dOI: 10.1007/978-3-319-05810-8_24.

[22] T. White, *Hadoop: the definitive guide*. Farnham: O'Reilly, 2012.

[23] "KDD Cup 2004 - Datasets." [Online]. Available: http://osmot.cs.cornell.edu/kddcup/datasets.html

[24] S. J. Phillips, "Acceleration of K-Means and Related Clustering Algorithms," in *Revised Papers from the 4th International Workshop on Algorithm Engineering and Experiments*, ser. ALENEX '02. London, UK, UK: Springer-Verlag, 2002, pp. 166–177.

Efficient Clustering Techniques on Hadoop and Spark

Sami Al Ghamdi

Department of Computer Science,
University of Reading,
Reading, UK
E-mail: s.a.m.alghamdi@pgr.reading.ac.uk

Giuseppe Di Fatta

Department of Computer Science,
University of Reading,
Reading, UK
E-mail: g.difatta@reading.ac.uk

Abstract: Clustering is an essential data mining technique that divides observations into groups where each group contains similar observations. K-Means is one of the most popular clustering algorithms that has been used for over fifty years. Due to the current exponential growth of the data, it became a necessity to improve the efficiency and scalability of K-Means even further to cope with large-scale datasets known as Big Data. This paper presents K-Means optimisations using triangle inequality on two well-known distributed computing platforms: Hadoop and Spark. K-Means variants that use triangle inequality usually require caching extra information from the previous iteration, which is a challenging task to achieve on Hadoop. Hence, this work introduces two methods to pass information from one iteration to the next on Hadoop to accelerate K-Means. The experimental work shows that the efficiency of K-Means on Hadoop and Spark can be significantly improved by using triangle inequality optimisations.

Keywords: K-Means; Hadoop; Spark; MapReduce; Efficient Clustering; Triangle Inequality K-Means

Biographical notes: Sami Al Ghamdi is a PhD candidate in the Department of Computer Science at University of Reading. He earned his Master' degree in Computer Science from Saint Joseph's University, Philadelphia, USA, in 2012. In 2003 he earned his Bachelor's degree in Computer Science from King Abdulaziz University, Jeddah, Saudi Arabia. From 2004 to 2008, he worked as a Computer Engineer and joined the Department of Computer Science in Al-Baha University, Saudi Arabia, as a Lecturer in 2009. His research interests include distributed computing, parallel algorithms and big data solutions.

Dr. Giuseppe Di Fatta is an Associate Professor of Computer Science and the Head of the Department of Computer Science at the University of Reading, UK. In 1999, he was a research fellow at the International Computer Science Institute (ICSI), Berkeley, CA, USA. From 2000 to 2004, he was with the High-Performance Computing and Networking Institute of the National Research Council, Italy. From 2004 to 2006, he was with the University of Konstanz, Germany. His research interests include data mining algorithms, distributed and parallel computing, big data in sciences and data-driven multidisciplinary applications. He has published over 100 articles in peer-reviewed conferences and journals. He serves in the editorial board of the Elsevier Journal of Network and Computer Applications. He is the co-founder of the IEEE ICDM Workshop on Data Mining in Networks and has chaired several international events, such as the 2015 International Conference on Internet and Distributed Computing Systems.

1 Introduction

The last two decades witnessed an exponential growth of the data generated by many sources such as, scientific experiments, social media Web sites, government statistics, sensor networks, and many other. For example, the Large Hadron Collider project (LHC), which provides more knowledge about the universe by accelerating particles and examining the results from their collisions, is expected to produce around 50 petabytes of data in 2017, and the collected data could reach 10 gigabytes per second (WLCG, 2017). YouTube users eceeded 1 billion users, where 100 hours of videos are uploaded every minute, and 135,000 hours are watched (YouTube, 2017). eBay stores and preocess about 150 billion new records daily (Lin and Dyer, 2010). In order to cope with this rapid increase in the data, novel solutions are developed to manage and process large-scale datasets known as big data.

Collecting, storing and managing the data is a crucial process. However, the data itself is worthless unless meaningful knowledge can be extracted from it. For this reason, various innovative techniques were developed over the years dedicated to knowledge discovery. One of the essential approaches to unveil the hidden patterns in a given set of observations is to divide these observations into a number of groups (clusters), such that observations in one group have more similarities than observations in other groups. This process is known as *clustering* or *cluster analysis*. Clustering algorithms are developed and used in many fields such as engineering, computer science, life and medical sciences, astronomy and earth sciences, social sciences and economics (Xu and Wunsch, 2009). Most clustering algorithms, however, are computationally expensive or iterative in nature. This made the clustering task very challenging, especially when dealing with large and high-dimensional datasets. Therefore, the focus has been shifted lately to parallel clustering solutions on distributed processing models to overcome these challenges. One of the most popular and attractive distributed processing models is known as MapReduce (Dean and Ghemawat, 2008). Apache Hadoop (Apache, 2017) provides an open-source implementation of the MapReduce programming model. The popularity of MapReduce comes from its ability to offer a reliable and fault-tolerant parallel programming paradigm without the need to deal with the underlying details of the distributed system, such as data distribution and tasks scheduling.

Ranked as one of the top ten data mining algorithms (Wu et al., 2008), K-Means takes the number of clusters as an input and iterates over the input data points until it converges. In each iteration, the standard implementation of K-Means, known as Lloyd's algorithm (Lloyd, 1982), computes the distance from each data point to all cluster centroids. This process is a performance bottleneck in K-Means. Most of these distance calculations, however, are redundant and can be avoided using geometric approaches based on triangle inequality. Although these approaches could produce efficient versions of K-Means, most of them require using extra information from the previous iteration. This is not a straightforward task to achieve under the MapReduce programming paradigm that Hadoop implements. MapReduce does not have the ability to cache any information between two consecutive iterations. Therefore, this paper introduces two approaches that allow Hadoop to pass intermediate data from one iteration to the next in order to be able to implement highly scalable and efficiently optimised K-Means algorithms based on triangle inequality.

The aim of this paper is to improve the efficiency and scalability of Lloyd's K-Means on Hadoop while maintaining the same deterministic clustering results that Lloyd's algorithm produces. Some K-Means variants that are based on triangle inequality can be more efficient and deterministically equivalent to Lloyd's K-Means. However, implementing such variants on Hadoop is a challenging task. This is because most of these variants require the use of some extra information (e.g. distance bounds and cluster assignments) from the previous iteration to be able to eliminate unnecessary distance computations and Hadoop does not cache intermediate data between two consecutive iterations. Therefore, this work presents two techniques to store required intermediate data in one iteration for it to be used in the next. The first technique appends the extra information to the original input data vector and forms an *Extended Vector* (EV). The second technique stores the extra information that corresponds to each data point into a file called a *Bounds File* (BF).

To evaluate the effectiveness of the proposed techniques, two optimised algorithms, Elkan's algorithm and Compare-means algorithm, are implemented using each technique and tested with real and artificially generated datasets. The performance of each optimised algorithm is compared against the performance of Lloyd's K-Means on Hadoop, which is referred to in the remaining of this work as Naive K-Means on Hadoop (NKM-H). Furthermore, an implementation of K-Means on Hadoop and Spark using the most basic form of triangle inequality to skip distance computations is introduced and also compared with the algorithms mentioned earlier. The experimental work investigates the impact of several important factors that influence the performance of K-Means. These factors include variable number of clusters (k), dimensions (d) and data points (n). The results show that variants of K-Means based on triangle inequality implemented on Hadoop with the proposed techniques can achieve significant speedups relative to NKM-H. For example, Elkan's K-Means and Compare-means on Hadoop using Bounds Files outperform NKM-H by upto 7x and 33x speedups, respectively.

The remainder of the paper is organised as follows: Section 2 reviews the related work. Section 3 presents a background about K-Means and how it can be optimised using triangle inequality. A brief introduction to MapReduce, Hadoop, and Spark is also presented in Section 3. Section 4 explains the implementation of the Naive K-Means on Hadoop. Section 5 Introduces the new implementations of efficient K-Means on Hadoop. Section 6 explains the implementation of two K-Means implementations on Spark. Section 7 discusses the experimental results. Finally, section 8 concludes the paper and discusses the future work.

1.1 Contributions

The contributions of this paper are:

- The design and the development of two techniques: K-Means on Hadoop using an Extended Vector (EV) and K-Means on Hadoop using a Bounds File (BF). These techniques give Hadoop the ability to pass information from one iteration to the next on iterative algorithms;

- Parallel implementations of K-Means variants on Hadoop using EVs and BFs to evaluate the effectiveness of the proposed approaches;

- An extensive experimental analysis that tests the scalability and efficiency of implementations of K-Means on Hadoop using BFs and EVs with respect to the number of clusters, dimensions, data points, and mappers;

2 Realted Work

A parallel implementation of K-Means on distributed memory multiprocessors based on Message Passing Interface MPI was introduced by (Dhillon and Modha, 2002). The algorithm partitions the original dataset into a number of subsets. Then, each processor works on an independent subset where distance calculations are performed and each point is assigned to its closest centroid. Then, partial sums and *SSE*s are collected and new centroids are calculated. This process is repeated until the algorithm converges. (Judd et al., 1998) introduced parallel K-Means based on MPI and used a method called Spheres of Guaranteed Assignment which follows the concept of pruning unnecessary distance calculations per iteration based on triangle inequality but without maintaining upper or lower bounds.

(Zhang and Qiu, 2013) presented a parallel K-Means on Twister (Ekanayake et al., 2010), which is an optimised implementation of the MapReduce framework that supports iterative algorithms based on publish/subscribe messaging infrastructure and caches static data in memory to cluster high dimensional social image data. Triangle inequality was used to reduce distance computations based on (Elkan, 2003) work, except that instead of keeping *nk* lower-bounds, a fewer number of lower-bounds is maintained. The work presented in this paper is different in terms of adopting the standard unoptimised MapReduce programming paradigm which is implemented by Hadoop.

K-Means++ (Arthur and Vassilvitskii, 2007) is a variant of K-Means which carefully selects the initial set of centroids that has a constant factor away from the optimum solution. K-means|| or Scalable K-mean++ (Bahmani et al., 2012) and Competitive K-Means (Esteves et al., 2014), address a downside of the k-means++ initialisation which is its inherently sequential nature and provide solutions to make it work efficiently on a parallel environment, specifically, MapReduce.

In (Li et al., 2014), K-Means was implemented on MapReduce and its efficiency was improved by using locality sensitive hashing *LSH* to divide points into buckets where the original points are transformed into weighted representative points. This method is used to prune unnecessary distance computations by computing the distance of a given point with only a small number of centres that exist in the same bucket as the point. The algorithm was tested with real datasets and shows improvement in speed by 67% and 76% when k is 1500 and 3000 respectively, compared to scalable K-Means++. However, the dimensionality of both datasets is low (26 and 41 dimensions) which does not give a full understanding of the algorithm's behaviour with high dimensional datasets.

The work in (Shi et al., 2015), compares MapReduce and Spark in terms of three major architectural components: shuffle, execution model, and caching. On both frameworks, five algorithms were tested: Word Count, Sort, K-Means, linear regression, and PageRank. In K-Means, three artificially generated datasets were used as input where each point has 20 dimensions and the number of data points for each dataset are: 1 million, 200 million, and 1 billion. The results showed that K-Means on Spark was 1.5x faster than K-Means on MapReduce in the first iteration, and 5x faster in subsequent iterations.

3 Background

3.1 K-Means

Ranked as one of the top ten most influential data mining algorithms (Wu et al., 2008), K-Means is a well-known clustering algorithm that partitions data into clusters of similar features. Simplicity, efficiency, and straight-forward implementation made K-Means one the most used algorithms in cluster analysis (Jain, 2010). K-Means was proposed independently in different works (Steinhaus, 1956); (Lloyd, 1982); (Ball and Hall, 1965); (MacQueen, 1967) targeting different problems.

K-Means has been used in many fields to cluster variant types of data. Some of the applications that K-Means was applied to are:

- Colour quantisation where the pixels of an image grouped into clusters (Celebi, 2011); (Kanungo et al., 2002).

- Market segmentation (Kuo et al., 2002), where markets are broken down into meaningful segments, such as segmenting buyers habits based on age groups.

- Analysis of gene expression data (Tavazoie et al., 1999); (Yeung et al., 2003).

- Documents clustering (Effat et al., 2016); (Steinbach et al., 2000), where similar documents are grouped into one cluster while other documents are assigned to other clusters.

Algorithm 1: Sequential Naive K-Means(X, k)

```
1  select $k$ initial cluster centroids randomly from $X$
2  while not converged and an early termination
     condition is not met do
3      for $i = 1$ to $n$ do
4          $minDistance \leftarrow \infty$
5          for $j \leftarrow 1$ to $k$ do
6              $d \leftarrow \mathrm{d}(x_i, c_j)$
7              if $d < minDistance$ then
8                  $minDistance \leftarrow d$
9                  assign $x_i$ to $c_j$
10             end
11         end
12     end
13     for $j \leftarrow 1$ to $k$ do
14         $c_j \leftarrow \frac{1}{|c_j|} \sum_{x \in c_j} x$ //Compute the mean
15     end
16 end
```

Lloyd's K-Means

The basic K-Means algorithm was independently proposed by (Steinhaus, 1956); (Lloyd, 1982); (Ball and Hall, 1965); and (MacQueen, 1967). The focus of this paper is on Lloyd's algorithm which is the most commonly used version (Celebi et al., 2013); (Hamerly and Drake, 2015). Lloyd's algorithm is referred to in the remainder of this paper as *Naive K-Means*.

Given a set $\mathcal{X} = \{x_1, x_2, ..., x_n\}$, where n is the number of data points in a d-dimensional space $\mathbb{R}^d$, partitioned into k clusters $\mathcal{C} = \{C_1, C_2, ..., C_k\}$, K-Means aims to minimise the Sum of Squared Error $SSE = \sum_{j=1}^{k} \sum_{x \in c_j} d(x - c_j)^2$, where j is the index of the j-th $C \in \mathcal{C}$, c_j is the centroid (mean of points) of C_j, and d(.,.) is the Euclidean distance between two points. Algorithm 1 describes the pseudo-code of the Naive K-Means where it starts by randomly picking k initial cluster centroids. Then, the distance from each $x \in \mathcal{X}$ to each $c_j \in \mathcal{C}$ is computed and x gets assigned to its closest c_j. In line 14, the location of each c_j is updated by computing the mean of all points assigned to each cluster, where $|c_j|$ is the number of points assigned to cluster C_j. The algorithm iterates until it converges where cluster centroids do not move any more or an early termination condition is met. K-Means finds a local minimum solution in $O(ndk)$ running time per iteration.

The next section explains how triangle inequality can be used on eliminating redundant distance computations from data points to centroids in the Naive K-Means.

3.2 Using Triangle Inequality to Accelerate K-Means

The most expensive operation in K-Means is computing the distance from each data point to all centres to find the centre with the minimum distance. One of the most important remarks in K-Means is that after a few number of iterations, most data points do not change their cluster assignment, especially with well-clustered datasets. The reason behind this is that after a few number of iterations the movement of cluster centroids is insignificant (Elkan, 2003); (Hamerly and Drake, 2015). Thus, most of the distance calculations from points to centroids are redundant, and this is where triangle inequality excels.

In general, the main goal of using triangle inequality with K-Means is to prove that a given point in the input dataset is closer to a certain centroid without the need to calculate the distance to other centroids. Triangle inequality was used in different ways to prune distance calculations. For a point x and two cluster centroids a and b, the following are some of the cases that triangle inequality can be applied to K-Means (Elkan, 2003); (Hamerly and Drake, 2015):

1. Show that x is closer to a than b, with calculating only $d(x, a)$ and $d(a, b)$.
2. Form an upper-bound from x to its closest centroid.
3. Form a lower-bound from x to one or more centroid.

The following Lemma is used in finding the closest centroid from a given point by using pre-calculated centre-centre distances and the distance from the point to its previously assigned centroid.

Lemma 1 *Let x be a point, and p and q be two centroids,*

$$\textit{if } d(p, q) \geq 2d(x, p) \textit{ then } d(x, q) \geq d(x, p)$$

Proof. From the triangle inequality property, it is known that:

$$d(p, q) \leq d(x, p) + d(x, q)$$
$$d(p, q) - d(x, q) \leq d(x, p).$$

The left hand side can be written as:

$$d(p, q) - d(x, q) \geq 2d(x, p) - d(x, p) = d(x, p).$$

Hence:

$$d(x, p) \leq d(x, q).$$

□

The usage of Lemma 1 was proposed by (Hodgson, 1988). Hodgson's approach compared a given centroid c with only its closest centroid c', that is, if $d(x, c) < d(c, c')$ then the distance calculation to only c' is avoided. In (Orchard, 1991), triangle inequality was used to improve the search of the nearest-neighbor. For a given point x and a candidate nearest-neighbor y, the author showed that another point z cannot be closer to x if Lemma 1 holds. The same approach was applied to K-Means by (Phillips, 2002) on an algorithm called Compare-means. (Elkan, 2003) algorithm uses Lemma 1 with a set of upper and lower bounds on the distance from each data point to cluster centroids to avoid a large number of distance computations. The following sections show how the scalability of Compare-means and Elkan's algorithms can be improved by implementing them on distribution fashion on Hadoop.

3.3 MapReduce and Apache Hadoop

MapReduce is a programming paradigm that is designed to store and process large-scale datasets efficiently and reliably on large clusters of commodity machines. MapReduce is designed to provide a high performance parallel execution of programs without dealing with underlying details of the distributed system such as scheduling, distribution and fault-tolerance.

In the MapReduce paradigm, the input data is stored on a distributed file system (e.g. Hadoop Distributed File System (HDFS)). The input and output data are in the form of *key-value* pairs. The computation process is expressed by implementing two functions: *map* and *reduce* from the MapReduce library, which are typically implemented by the user.

Hadoop is a popular open-source implementation of MapReduce that is widely used by many organisations such as Yahoo!, Facebook, Twitter and IBM to manage and analyse massive amounts of daily generated data (White, 2012). The dataflow in Hadoop consists of three phases: 1) map phase; 2) shuffle phase; and 3) reduce phase.

In the *map phase*, as the input dataset loaded to HDFS, it is split into what is known as input-splits. The number of mappers equals the number of input-splits and the size of each input-split can be modified (default 128 MB). Each mapper processes one input-split independently. The *map* function takes as an input the records in each input-split in the form of *key* and *value* (<K,V>) pairs and outputs a new <K2,V2> pair. In the *shuffle phase*, each reducer uses HTTP protocol to fetch its own partition from the mappers' output files that reside on the mappers' nodes. The shuffle starts as a predefined percentage (default is 5%) of mappers complete their work. Finally, the *reduce phase* starts after each reducer fetches its own partition from the mapper's output files. Before invoking the *reduce* function, the reducer merges and sorts the the mappers' output files fetched from different mappers and then the *reduce* method is invoked and each reducer outputs the resulted <K3,V3> pairs to HDFS.

Limitations: despite the advantages that Hadoop offers to store, manage, and process large-scale datasets, several limitations are addressed in many works (e.g. (Mohebi et al., 2016); and (Grolinger et al., 2014)). Some of the limitations that are specific to the support of iterative Machine Learning algorithms such as K-Means are:

- Absence of loop-aware task scheduling where each iteration is a new MapReduce Job.
- Reload and reshuffle of static data which creates an unnecessary I/O and communication overheads.
- Lack of support to cache and retrieve information from previous iterations. This limitation imposes extra complexities on iterative algorithms that require information from previous iterations in order to proceed their work efficiently. This paper investigates this limitation in particular.

3.4 Apache Spark

Apache Spark (Zaharia et al., 2010) is a distributed framework that is designed to process large-scale working sets that are reused over multiple parallel operations in-memory. The goal of Spark is to process iterative machine learning algorithms and interactive analytics problems faster than Hadoop MapReduce while maintaining the fault tolerance and scalability of MapReduce. Spark can operate on several clusters managers (e.g. Hadoop YARN) or as a standalone system.

Two main abstractions are provided by Spark to process parallel applications, Resilient Distributed Datasets (RDDs) and parallel operations. An RDD is a collection of immutable (read-only) objects partitioned among cluster nodes that can be rebuilt in case a partition is lost. RDDs can be cached in-memory once across worker nodes (*executors*) and reused by applications that run on multiple parallel operations. Parallel operations can be either *transformations*, where an RDD can be transformed from a file on stable storage, or from another existing RDD; or *actions*, where a value is returned to the application driver, or stored on a data storage.

4 Naive K-Means on Hadoop

This section explains the implementation of the Naive K-Means on Hadoop (NKM-H). The implementation of NKM-H consists of three main classes: Driver, Mapper and Reducer.

Driver: The Driver starts by randomly selecting the initial set of centroids from the input dataset and sends the centroids file to the mappers. The Driver controls the iterative process where a new MapReduce job is set and initiated for each iteration. In case of having more than one reducer, the Driver merges the centroid files produced by each reducer into one file which becomes the input centroids file in the following iteration. Algorithm 2 describes the pseudo-code of the Driver.

Mapper: Each mapper consists of three functions, *setup*, *map*, and *cleanup*. While the *map* function is invoked for each record in the input-split, *setup* and *cleanup* are executed only once on each run of the mapper class. As shown in Algorithm 3, *setup* reads the set of centres and loads them to C. Then, the *map* function takes as an input, key-value pairs where the

Algorithm 2: Driver(X, k)

```
1  $C \leftarrow$ select $k$ initial cluster centroids from $X$ randomly
2  while not converged or an early termination condition is not met do
3      send the set of centroids $C$ to mappers
4      set mapper to NKM–H-Mapper
5      set reducer to NKM-H-Reducer
6      run a new MapReduce job
7      if $numberOfReducers > 1$ then
8          merge reducers output into one file
9      end
10 end
```

Algorithm 3: NKM-H-Mapper(k)

```
1  Function setup():
2      load centroids from HDFS to $C$
3  Function map(offset, value)
4      $x \leftarrow value$ $minDistance \leftarrow \infty$
5      $a \leftarrow -1$
6      for $j \leftarrow 1$ to $k$ do
7          $d \leftarrow \mathrm{d}(x, c_j)$
8          if $d < minDistance$ then
9              $minDistance \leftarrow d$
10             $a \leftarrow j$
11         end
12     end
13     output$(a, x)$
```

key is the offset of the data point in the input file, and the value is the data point itself. Subsequently, the *map* function iterates over C to find the centroid with the minimum distance from the input data point. Finally, the index of the closest centroid (a) is emitted to the reducers with its assigned data point as a key-value pair.

Reducer: After each mapper outputs a key-value pair, these pairs are grouped by key and sent to the reducer in the form of (*key*, *list(values)*) pairs, where *key* is the cluster index j and *values* are the data points that were assigned to centroid c_j by the mappers. In Algorithm 4, the *setup* function initialises C' which holds the set of updated centroids. In the *reduce* function, the vector sum of all the points in the list is calculated and stored in **sum**. The updated centroid, which is represented by the mean of the data points in each cluster, is calculated by dividing the **sum** over the count of the points in each cluster. Finally, each reducer writes the new centroids in C' to HDFS. Note that since the Reducer's implementation is identical in all the following implementations of K-Means it implementation will not be discussed in further section.

As discussed in section 3.3, Hadoop does not support iterative algorithms, particularly, Hadoop does not have the ability to cache intermediate data between two consecutive MapReduce jobs. The following

Algorithm 4: NKM-H-Reducer

```
1  Function setup():
2      let $C$‘ be a list holds the new centroids
3  Function reduce(j, values):
4      $pointsCounter \leftarrow 0$
5      sum $\leftarrow$ (0,0,...,0)
6      foreach $x \in values$ do
7          sum $\leftarrow$ sum $+ x$ //vector sum
8          $pointsCounter \leftarrow pointsCount + 1$
9      end
10     $c'_j \leftarrow$ sum$/pointsCounter$
11     load $c'$ to $C'$
12 Function cleanup():
13     write recodes in $C'$ to HDFS
```

sections discuss the challenge of implementing K-Means variants that use triangle inequality and require extra information from the previous iteration and presents two techniques to overcome such challenge.

5 Efficient K-Means based on Triangle Inequality on Hadoop

As it was explained in section 3.3, one of Hadoop's limitations is its lack to cache intermediate data between two consecutive MapReduce jobs. Several K-Means variants, such as Elkan's algorithm (Elkan, 2003), Hamerly's algorithm (Hamerly, 2010), Adaptive algorithm (Drake and Hamerly, 2012), and Compare-means algorithm (Phillips, 2002), require information from the previous iteration to use them in the current iteration to avoid computing the exact distances from data points to centroids. Therefore, this section introduces two approaches: K-Means on Hadoop using an Extended Vector (EV); and K-Means on Hadoop using a Bounds File (BF). These approaches aim to allow Hadoop to pass information from one iteration to the next to efficiently accelerate the K-Means algorithm. Furthermore, the implementation of two optimised algorithms, Elkan's algorithm and Compare-Means algorithm, on Hadoop using each approach is explained.

In general, the assignment of data points to their closest centres in K-Means on Hadoop is the responsibility of the mappers, while the reducers are responsible for aggregating points belonging to each centroid and producing the new set of centroids. Therefore, the optimisation steps occur in the map phase and, as a consequence, several mapper algorithms will be discussed in the next sections. On the other hand, the implementation of the reducer in all of the proposed solutions is identical to the reducer in Algorithm 4.

5.1 K-Means on Hadoop using an Extended Vector (EV)

This section explains the use of a data structure called Extended Vector (EV) to pass extra information from one iteration to the next. The idea of the Extended Vector is to append any required information in the current iteration to the original input data vector to form an EV. This EV will be the input in the next iteration where the input data along with any extra information associated to it can be read together. Therefore, the Extended Vector can be defined as: a data structure that stores the input data vector and any extra information related to this data vector in a given iteration, in order to be used in subsequent iterations. This can be considered as the straight-forward solution to the problem of passing information between iterations in Hadoop. Two K-Means variants are implemented using this approach, Elkan's algorithm and Compare-means. The following sections will explain the implementation of each algorithm on Hadoop using an EV.

5.1.1 Elkan's Algorithm on Hadoop using EVs (ELK-H-EV)

The implementation of Elkan's algorithm on Hadoop using an Extended Vector is referred to as *ELK-H-EV*. Elkan's algorithm efficiently eliminates a large number of unnecessary distance computations while maintaining the same output as the Naive K-Means. In addition to the need of computing the k^2 centre-centre distances at the beginning of each iteration, the algorithm needs to cache the following information in one iteration and use them in the next:

- n upper-bounds on the distance between each data point and its assigned centroid.
- nk lower-bounds on the distance between each data point and each centroid.
- n cluster assignments for each data point from the previous iteration.

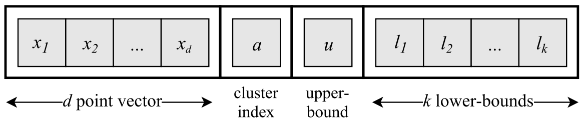


Figure 1: Structure of an Extended Vector in ELK-H-EV.

EV Size: Since extra information is associated with each data point, the required information will be appended to the data point, which forms the Extended Vector (EV). Figure 1 illustrates the structure of an EV in ELK-H-EV. Each EV in ELK-H-EV consists of a data point vector in d dimensions, one upper-bound for the distance from the point to its closest centroid, one cluster assignment index from the previous iteration and k lower-bounds for the distances from the point to each centre. Therefore, the size of each EV in ELK-H-EV is $d + k + 2$. This means that each mapper writes $\frac{n}{p}(d + k + 2)$ EVs to HDFS per iteration.

Notation	Description
X	Input dataset of size n
C	The set of cluster centroids of size k
k	Number of clusters
c_j	Cluster centroid, where $c_j \in C$, with $1 \leq j \leq k$
c'_j	New location for centroid c_j
c_a	Closest centroid to data point x, where $1 \leq a \leq k$
$s_{i,j}$	Distance between centroids c_i and c_i, where $1 \leq i, j \leq k$ and $i \neq j$
h_j	Half minimum distance from c_j to its closest centroid
m_j	Distance that centroid c_j has moved in the last iteration, i.e. d(c_j,c'_j)
u	An upper-bound from data point $x \in X$ to its closest centroid c_a
l_j	A lower-bound from data point $x \in X$ to centroid c_j
w	An ExtendedVector class object which stores the data vector $w.x$ ($x \in X$) and required extra information

Table 1 Notations description

Table 1 describes the notations that are used in the pseudo-codes.

Algorithm

The implementation of each of the following algorithms can be divided into three major phases:

1. A *driver* that initiates the MapReduce jobs and controls the iterative process. Because the implementation of the driver in all algorithms is similar to the driver in section 4, Algorithm 2, only significant changes will be highlighted to avoid redundancy.

2. A *map* phase that assigns each point to its closest centroid (distance computation elimination steps occur in this phase).

3. A *reduce* phase that computes the means of points assigned to each cluster centroid and produces new set of centroids. The implementation of the reducer is identical to the reducer in Algorithm 4, section 4.

Driver: The driver in ELK-H-EV and ELK-H-BF, which will be explained later in section 5.2.1, is similar to the driver's implementation in Algorithm 2. One exception is that because ELK-H's implementation has two mappers' implementations, it runs the first mapper

Algorithm 5: ELK-H-EV-Mapper-1(k)

```
1  Function setup():
2      load centroids $C$
3      compute $s_{i,j} \leftarrow d(c_i, c_j)$ //for all $1 \leq i, j \leq k$
4  Function map(offset, point)
5      let $w$ be an Extended Vector
6      let $t$ be a boolean list of size $k$
7      $w.x \leftarrow point$
8      for $j \leftarrow 1$ to $k$ do
9          $t_j \leftarrow false$
10     end
11     $minDistance \leftarrow \infty$
12     for $j \leftarrow 1$ to $k$ do
13         if $t_j$ then continue
14         $d \leftarrow$ d($w.x$, $c_j$)
15         $w.l_j \leftarrow d$
16         if $d < minDistance$ then
17             $minDistance \leftarrow d$
18             $w.u \leftarrow minDistance$
19             $w.a \leftarrow j$
20             for $z \leftarrow j + 1$ to $k$ do
21                 if $s_{j,z} \geq 2 * d$ then
22                     $t_z \leftarrow true$
23                 end
24             end
25         end
26     end
27     write $w$ to HDFS
28     output($w.a$, $w.x$)
```

(Algorithm 5) in the first iteration, and the second mapper (Algorithm 6) in subsequent iterations.

Map phase: ELK-H requires two mappers' implementations, the first mapper is executed in the first iteration, and the second mapper is executed in subsequent iterations. This is because in the first iteration distance bounds and cluster assignments are not initialised yet. Therefore, the first mapper runs in the first iteration and initialises the distance bounds and cluster assignments, and the second mapper runs in subsequent iterations and performs the techniques for eliminating unnecessary distance computations.

First Mapper: The first mapper initialises the distance bounds and the cluster assignments. Furthermore, the algorithm uses Lemma 1 to skip some distance computations where information from the previous iteration is not required. The pseudo-code in Algorithm 5 shows how upper and lower bounds associated to each input data point x are initialised in ELK-H-EV. w represents an ExtendedVector (EV) class object with the index of the assigned cluster centroid (a), the upper-bound (u), the lower-bound (l), and the data point (x), as members of w. First, a new Extended Vector (w) is declared in line 5, then, the input data point is assigned to $w.x$. The distance from the input data point $w.x$ to the closest centroid is assigned to the upper-bound $w.u$. The lower-bound $w.l_j$ is set to the distance from point $w.x$ to any centroid c_j. Lemma 1 states that: given two centres p and a, and a point x, if $d(p, a) \geq 2d(x, p)$ then $d(x, a) \geq d(x, p)$. This Lemma can be used to skip the distance computation from $w.x$ to the next centroid in the centroids list. To achieve this, t holds the skip status of each centroid, that is, if the distance computation from $w.x$ to centroid c_j can be skipped, c_j's status in t_j will be *true*, otherwise, it is *false*. Line 13 tests the status of the currently processed centroid. The distance computation to this centroid is avoided if its status is true. Lines 14-19 find the closest centroid from $w.x$. Then in line 20 the distance from the current centroid to the next centroid is extracted from structure s, and line 21 tests Lemma 1 to check if the distance to the next centroid can be eliminated. If the test holds, the skip status of the next centroid is set to *true* and the distance computation to it is skipped. In line 27 w is written to HDFS. EVs that are written by each mapper will be the input for the mappers in the next iteration. Finally, the mapper outputs data point ($w.x$) and its assigned cluster index ($w.a$) to reducers as a key-value pair.

Second Mapper: Algorithm 6 illustrates the pseudo-code of the second mapper in ELK-H-EV, which is executed on iterations > 1. The second mapper takes as input key-value pairs, where each value represents an EV that was stored by a mapper in the previous iteration. In lines 9-12, the lower and upper bounds are updated. The distance (m_j) that centroid c_j has moved in the previous iteration is added to the upper-bound and subtracted from each lower-bound. The centroid's movement is part of the data structure that holds the centroid's vector and is computed and stored at the end of the reduce stage. If the test in line 15 holds, all distance calculations associated to the currently processed point are skipped. Furthermore, if any of the three tests in lines 14-16 does not hold, the distance computation to currently processed centroid is avoided. The distance from the point $w.x$ to any centroid other than the one assigned to the it does not get calculated until line 29, where the tests at line 28 repeats the tests at lines 18 and 19 but with an updated upper-bound $w.u$. At this point w acquires updated values for the assigned cluster index a, the upper-bound u, and the lower-bounds l_j ($1 \leq j \leq k$) and can be written to HDFS at line 39. Finally, the mapper outputs the point $w.x$ with the index of its closest centroid $w.a$ to the reducers.

5.1.2 Compare-means on Hadoop using EVs (CMP-H-EV)

Compare-means (Phillips, 2002) is a variant of K-Means that also uses triangle inequality to skip redundant distance computations. While Elkan's algorithm uses a combination of distance bounds and triangle inequality to eliminate unnecessary distance computations, Compare-means uses only triangle inequality without any distance bounds. The only required information from the previous iteration is the cluster assignment for

Algorithm 6: ELK-H-EV-Mapper-2(k)

```
1  Function setup():
2      load centroids to $C$
3      compute $s_{i,j} \leftarrow d(c_i, c_j)$, for all $1 \leq i, j \leq k$
4      compute $h_j \leftarrow min_{j \neq j'} d(c_j, c_{j'}) * 0.5$, for all
         $1 \leq j, j' \leq k$
5  Function map(offset, value):
6      let $w$ be an Extended Vector
7      $w \leftarrow value$
8      //update $k$ lower-bounds
9      for $j \leftarrow 1$ to $k$ do
10         $w.l_j \leftarrow \max[w.l_j - m_j, 0]$
11     end
12     $w.u \leftarrow w.u + m_{w.a}$ //update upper-bound
13     $g \leftarrow true$ //flag to check if $u$ is updated
14     $d1$, $d2 \leftarrow 0$
15     if $w.u \leq h(w.a)$ then continue
16     for $j \leftarrow 1$ to $k$ do
17         if $(j \neq w.a)$
18         $\&(w.u > w.l_j)$
19         $\&(w.u > s_{w.a,j} * 0.5)$ then
20             if $g$ then
21                 $d1 \leftarrow$ d($w.x$, $c_{w.a}$)
22                 $w.u \leftarrow d1$
23                 $w.l_{w.a} \leftarrow d1$
24                 $g \leftarrow false$
25             else
26                 $d1 \leftarrow w.u$
27             end
28             if $d1 > w.l_j$ or $d1 > s_{w.a,j} * 0.5$ then
29                 $d2 \leftarrow d(w.x, c_j)$
30                 $w.l_j \leftarrow d2$
31                 if $d2 < d1$ then
32                     $w.a \leftarrow j$
33                     $w.u \leftarrow d2$
34                     $g \leftarrow false$
35                 end
36             end
37         end
38     end
39     write $w$ to HDFS
40     output($w.a$, $w.x$)
```

Algorithm 7: CMP-H-EV-Mapper(k)

```
1  Function setup():
2      load centroids to $C$
3      compute $s_{i,j} \leftarrow d(c_i, c_j)$ //for all $1 \leq i, j \leq k$
4  Function map(offset, value)
5      let $w$ be an Extended Vector
6      if $iteration == 1$ then
7          $w.x \leftarrow value$
8          $w.a \leftarrow 1$
9      end
10     $minDistance \leftarrow$ d($w.x$, $c_{w.a}$)
11     $d \leftarrow 0$
12     for $j \leftarrow 1$ to $k$ do
13         if $s_{j,w.a} \geq 2 * minDistance$ or $j == w.a$
            then
14             continue
15         end
16         $d \leftarrow$ d($w.x$, $c_j$)
17         if $d < minDistance$ then
18             $minDistance \leftarrow d$
19             $w.a \leftarrow j$
20         end
21     end
22     write $w$ to HDFS
23     output($w.a$, $w.x$)
```

each data point. The implementation of Compare-means on Hadoop using an Extended Vector is referred to as CMP-H-EV.

As in ELK-H-EV, CMP-H-EV needs to compute k^2 centre-centre distances at the beginning of each mapper. In addition, the algorithm needs to cache one cluster assignment for each data point from last iteration. Each EV in CMP-H-EV consists of a data point vector of size d dimensions, and one cluster assignment index from the previous iteration. Therefore, the size of each EV in CMP-H-EV is $d + 1$. This means that each mapper writes $\frac{n}{p}(d + 1)$ EVs to HDFS per iteration.

Algorithm

Map phase: Unlike ELK-H-EV, CMP-H-EV has only one mapper because it does not need to initialise any distance bounds. As mentioned previously in this section, the only extra information CMP-H-EV needs from the previous iteration is the index (a) of the assigned cluster to each data point, which needs to be initialised in the first iteration. In this situation, a is initialised to 1 in the first iteration for all data points, which is the index of the first centroid in the centroids list C.

The pseudo-code in Algorithm 7 describes the steps of the mapper in CMP-H-EV. First, it can be observed that CMP-H-EV's algorithm is simpler than ELK-H-EV with regards to the method each algorithm eliminates distance computations. This simplicity makes the algorithm lighter than ELK-H-EV in terms of I/O overhead, but this come on the cost of the amount of skipped distance computation.

In the first iteration, the *map* function receives the byte-offset of the input record and the data point vector as a key-value pair. A new Extended Vector (w) is declared in line 5 and the received value (data point) is assigned to $w.x$. The index for the cluster centroid that was assigned to $w.x$ in the previous iteration is initialised to one for all data points, which is the index of the first centroids in the centroids list. Consequently, *minDistance* in line 10 will be the distance from $w.x$ to the first centroid in the centroids list. Distance computations are avoided if the test in line 13 holds. The test in line 13 uses Lemma 1, which states that: for two centres c_1 and c_2, and a data point x, if we know that $d(c_1, c_2) \geq 2d(x, c_1)$ then $d(x, c_2) \geq d(x, c_1)$, and $d(x, c_2)$ can be avoided. CMP-H-EV performs this test at line 13 using the last centroid that point $w.x$ was assigned to in the previous iteration ($w.a$). If the test does not hold, the distance to the centroid is computed as in NKM-H. Finally, w is written to HDFS, and the pair ($w.a$,$w.x$) is emitted to the reducers.

In iterations > 1, the *map* function receives the value as an EV that contains the data point $w.x$ and cluster index for the centroids that point $w.x$ was assigned to in the previous iteration. The algorithm then attempts to skip distance computations at line 13.

5.2 K-Means on Hadoop using a Bounds File (BF)

This section introduces the second approach called K-Means on Hadoop using a Bounds File (BF). The idea behind this approach is motivated by the large overhead EVs create when processing large number of clusters and dimensions. Thus, BFs attempt to reduce the overhead from writing EVs to HDFS in each iteration.

A *Bounds File (BF)* can be defined as a flat file that is written to HDFS in the each mapper, where each record in this file represents an extra information that is associated to a data point in the input dataset. In other words, in a given iteration, each mapper stores the desired extra information related to each input data point on a file on HDFS, this file is called a Bounds File. Unlike implementations that use EVs, each record in a BF stores only the extra information without the data point. These files can then be read by the mappers in subsequent iterations and each point is joined with its corresponding extra information.

The following sections explain the implementations of two K-Means variants: Elkan's algorithm, and Compare-means on Hadoop using BFs. Sections 5.1.1 and 5.1.2 explained the implementation steps of both algorithms on Hadoop using EVs (ELK-H-EV and CMP-H-EV) with an explanation of the method each algorithm follows to eliminate distance computations. Therefore, the following sections will focus on how to store extra information in one iteration and retrieve it in the next using BFs.

5.2.1 Elkan's Algorithm on Hadoop using BFs (ELK-H-BF)

In a given iteration, each mapper in Elkan's algorithm on Hadoop using a Bounds File (ELK-H-BF) writes one upper-bound, k lower-bounds, and one cluster assignment, which are associated to each data point to a BF on HDFS. In the following iteration, each mapper finds the BF that corresponds the input-split that was assigned to that mapper and loads all the extra information in the BF to memory. At this point, each mapper acquires the extra information that each data point needs to proceed with the elimination process.

How to identify which BF corresponds to which input-split? Hadoop splits the original input dataset into a number of input-splits where each mapper processes an individual input-split. The splitting mechanism does not change from one iteration to another, that is, each input-split contains the same data points in the same order from one iteration to the next. However, the input-split processed by a given mapper in one iteration could be processed by a different mapper on different node in the next iteration. This issue causes a difficulty in associating each BF to its corresponding input-split. To solve this issue, the BF's name is set to be the *starting byte offset* of the currently processed input-split. Hence, in given iteration, the mapper searches HDFS for the BF with the name that matches the *starting byte offset* of the input-split assigned to this mapper in the current iteration. The contents of the BF are then loaded the memory of the mapper's node. Since the order of the records in the input-split does not change from one iteration to another, the order of the records on the input-split will match the order of records in the corresponding BF.

BF Size: In a given iteration, each mapper in ELK-H-BF writes the following extra information for each data point to a BF: one upper-bound for the distance from the point to its closest centroid, one cluster assignment index from the previous iteration, and k lower-bounds for the distances from the point to each centre. Therefore, each record in a BF in ELK-H-BF is of size: $K + 2$, which makes the size of each BF $\frac{n}{p}(k + 2)$ per iteration, where n is the total number of data points, and p is the number of mappers.

Algorithm

Map phase: Similar to ELK-H-EV (section 5.1.1), ELK-H-BF requires two mappers' implementations, the first mapper runs in the first iteration and initialises the distance bounds and cluster assignments, while the second mapper runs in subsequent iterations and performs the techniques for eliminating unnecessary distance computations.

First mapper: Algorithm 8 shows the pseudo-code of the first mapper in ELK-H-BF, where most of the steps are similar to the steps in Algorithm 5, except that ELK-H-BF stores and reads extra information to/from BFs.

Each time the distance from the input data point to a given centroid c_j is calculated (line 13), the the lower-bound $b.l_j$ is set to that distance in line 14. Additionally, when the distance to the closest centroid is determined, the upper-bound ($b.u$) is set to that distance in line 17, and the index of this closets centroid is assigned to $b.a$ in line 18. At this point all the extra information for point x are acquired and can be written to a BF in line 26.

Second mapper: The pseudo-code of ELK-H-BF's second mapper is shown in Algorithm 9. ELK-H-BF follows the same method that ELK-H-EV uses on eliminating distance computations, which was illustrated

Algorithm 8: ELK-H-BF-Mapper-1(k)

```
1  Function setup():
2      load centroids to $C$
3      compute $s_{i,j} \leftarrow d(c_i, c_j)$ //for all $1 \leq i, j \leq k$
4  Function map(offset, value)
5      $x \leftarrow value$
6      let $b$ be a collection that stores the extra
         information for $x$
7      for $j \leftarrow 1$ to $k$ do
8          $t_j \leftarrow false$
9      end
10     $minDistance \leftarrow \infty$
11     for $j \leftarrow 1$ to $k$ do
12         if $t_j$ then continue
13         $d \leftarrow \mathrm{d}(x, c_j)$
14         $b.l_j \leftarrow d$
15         if $d < minDistance$ then
16             $minDistance \leftarrow d$
17             $b.u \leftarrow minDistance$
18             $b.a \leftarrow j$
19             for $z \leftarrow j + 1$ to $k$ do
20                 if $s_{j,z} \geq 2 * d$ then
21                     $t_z \leftarrow true$
22                 end
23             end
24         end
25     end
26     write $b$ to a BF on HDFS
27     output($b.a$, $x$)
```

in Algorithm 6. The two algorithms differ in the method of reading and writing cluster assignments and distance bounds from/to HDFS. The second mapper assumes that the extra information was stored to a BF by a mapper in the previous iteration. Therefore, each mapper searches HDFS for the BF that corresponds to the input-split that is assigned to this mapper (line 6). When the BF is located, each record in the BF is parsed to a collection structure called b in the algorithm, where the size of b is $k + 2$ (k lower-bounds, one upper-bound, and one cluster assignment). All b's are then loaded to the list f. The *map* function reads each b from f that corresponds to each data point and uses the information in b to eliminate distance computations. Before sending the output to the reducers, each b is written to a BF on HDFS in line 41. This BF is then read by a mapper in the following iteration.

5.2.2 Compare-means on Hadoop using BFs (CMP-H-BF)

The detailed explanation of the method Compare-means follows to eliminate accelerate K-Means is discussed in section 5.1.2. Therefore, the focus of this section is on how Compare-means on Hadoop writes and reads the cluster assignment for each data point from the previous iteration using Bounds Files.

Algorithm 9: ELK-H-BF-Mapper-2(k)

```
1  Function setup():
2      load centroids from DistributedCache to $C$
3      compute $s_{i,j} \leftarrow d(c_i, c_j)$, for all $1 \leq i, j \leq k$
4      compute $h_j \leftarrow min_{j \neq j'} d(c_j, c_{j'}) * 0.5$, for all
         $j \in k$
5      let $f$ be a list that stores the cluster
         assignments for all data point
6      find the BF that corresponds to the
         input-split assigned to this mapper and load
         its records to $f$
7  Function map(offset, value):
8      $x \leftarrow value$
9      let $b$ be a collection that stores the cluster
         index assigned to $x$
10     $b \leftarrow f(pointsCounter)$
11     //update $k$ lower-bounds
12     for $j \leftarrow 1$ to $k$ do
13         $b.l_j \leftarrow \max[b.l_j - m_j, 0]$
14     end
15     $b.u \leftarrow b.u + m_{b.a}$ //update upper-bound
16     $g \leftarrow true$ //flag to check if $u$ is updated
17     $d1, d2 \leftarrow 0$
18     if $b.u \leqslant h_{b.a}$ then continue
19     for $j \leftarrow 1$ to $k$ do
20         if ($j \neq b.a$) &($b.u > b.l_j$)
             &($b.u > s_{b.a,j} * 0.5$) then
21             if $g$ then
22                 $d1 \leftarrow \mathrm{d}(x, c_{b.a})$
23                 $b.u \leftarrow d1$
24                 $b.l_{b.a} \leftarrow d1$
25                 $g \leftarrow false$
26             else
27                 $d1 \leftarrow b.u$
28             end
29             if $d1 > b.l_j$ or $d1 > s_{b.a,j} * 0.5$ then
30                 $d2 \leftarrow d(x, c_j)$
31                 $b.l_j \leftarrow d2$
32                 if $d2 < d1$ then
33                     $b.a \leftarrow j$
34                     $b.u \leftarrow d2$
35                     $g \leftarrow false$
36                 end
37             end
38         end
39     end
40     $pointsCounter \leftarrow pointsCounter + 1$
41     write $b$ to a BF on HDFS
42     output($b.a$, $x$)
```

BF Size: In a given iteration, each mapper in CMP-H-BF writes the index for the cluster assigned to each data point in the previous iteration to a BF. Therefore, each mapper writes a BF of size: $\frac{n}{p}$ per iteration, where n is the total number of data points, and p is the number of mappers.

Algorithm

Map phase: The pseudo-code in Algorithm 10 illustrates the implementation steps of the mapper in CMP-H-BF. In the first iteration, the index of the assigned cluster to point x from previous iteration is initialised to one, which is the first centroid in the centroids list C. If the test at line 20 holds, the distance computation to centroid c_j is skipped. After assigning x to its closest centroids c_j, index j is assigned to $b.a$ which is then written to a BF on HDFS. This process is repeated on subsequent iterations where previous cluster assignments can be read from BFs. Therefore, in the *setup* function, the records of the BF that corresponds the input-split that is assigned to the mapper is loaded to f. The *map* function can read updated cluster assignments (line 15) from the previous iteration for each data point.

Algorithm 10: CMP-H-BF-Mapper(k)

```
1  Function setup():
2      load centroids to $C$
3      compute $s_{i,j} \leftarrow d(c_i, c_j)$ //for all $1 \leq i, j \leq k$
4      if $iteration > 1$ then
5          let $f$ be a list that stores the cluster
             assignments for all data point
6          locate the BF that corresponds to the
             input-split assigned to this mapper and
             load its records to $f$
7          $pointsCounter \leftarrow 1$
8      end
9  Function map(offset, value)
10     $x \leftarrow value$
11     let $b$ be a collection that stores the cluster
         index assigned to $x$
12     if $iteration == 1$ then
13         $b.a \leftarrow 1$ //initialise cluster assignment
14     else
15         $b \leftarrow f(pointsCounter)$
16     end
17     $minDistance \leftarrow$ d($x$, $c_{b.a}$)
18     $d \leftarrow 0$
19     for $j \leftarrow 1$ to $k$ do
20         if $s_{j,b.a} \geq 2 * minDistance$ or $j == b.a$
             then
21             continue
22         end
23         $d \leftarrow$ d($x$, $c_j$)
24         if $d < minDistance$ then
25             $minDistance \leftarrow d$
26             $b.a \leftarrow j$
27         end
28     end
29     if $iteration > 1$ then
30         $pointsCounter \leftarrow pointsCounter + 1$
31     end
32     write $b$ to a BF on HDFS
33     output($b.a$, $x$)
```

5.3 *Triangle Inequality K-Means on Hadoop (TIKM-H)*

This section explains the implementation of Triangle Inequality K-Means on Hadoop (TIKM-H). As illustrated in Algorithm 11, TIKM-H uses the most basic form of triangle inequality to skip redundant distance computations from points to cluster centroids. That is why it was named after triangle inequality. By the *most basic form of triangle inequality* we mean that this approach does not require any information from the previous iteration to skip distance computations. This approach needs to compute only the intra centre distances at the start of each mapper. In fact, the method TIKM-H follows to skip distance computations is the same as the one used in the first mapper of ELK-H-EV (Algorithm 5), and ELK-H-BF (Algorithm 8), where the centre-centre distances are computed at the *setup* function of each mapper and the map function tests the inequality in Lemma 1 to check if the distance to the next centroids in the list can be skipped (see Algorithms 5 or 8 for a detailed explanation).

This approach does not have the potential to prune lots of distance computations compared to ELK-H and CMP-H. However, its very small overhead could make it a good competitor to ELK-H and CMP-H on situations where the overhead becomes the dominant cost.

6 K-Means on Spark

Two versions of K-Means are implemented on Spark. The first version implements the Naive K-Means on Spark (NKM-S), and the second implements the Triangle Inequality K-Means on Spark (TIKM-S).

6.1 *Naive K-Means on Spark (NKM-S)*

The driver caches the input dataset in-memory for both algorithms in the first iteration as an RDD (RDD-1). RDD-1 is partitioned and distributed over a number of worker nodes (executors) where each executor finds the closest centroid from each data point and returns the index of the cluster centroid associated with the data point as a pair to driver. The driver creates a new RDD (RDD-2) of size n composed of pairs of data points and the index of their assigned centroids. To update the location of each centroid, the vector sum of points assigned to each centroid and the count of these points are required to compute the mean of points in each cluster. Therefore, RDD-2 is reduced by key (key is the centroid's index) to compute the vector sum and the number of points in each cluster is counted. Finally, the mean of points in each cluster (represents the new centroid) is computed in the driver. The old and new centroids are compared in the driver and new a iteration starts in case of failed convergence.

Algorithm 11: TIKM-H-Mapper(k)

```
Function setup():
    load centroids to $C$
    compute $s_{i,j} \leftarrow d(c_i, c_j)$ //for all $1 \leq i, j \leq k$
Function map(offset, value)
    $x \leftarrow value$
    //initialise all values in $t$
    for $j \leftarrow 1$ to $k$ do
        $t_j \leftarrow false$
    end
    $minDistance \leftarrow \infty$
    for $j \leftarrow 1$ to $k$ do
        if $t_j$ then
            continue
        end
        $d \leftarrow \mathrm{d}(x, c_j)$
        if $d < minDistance$ then
            $minDistance \leftarrow d$
            $a \leftarrow j$
            for $z \leftarrow j + 1$ to $k$ do
                if $s_{j,z} \geq 2 * d$ then
                    $t_z \leftarrow true$
                end
            end
        end
    end
    output($a$, $x$)
```

6.2 Triangle Inequality K-Means on Spark (TIKM-S)

TIKM-S uses the same approach used in TIKM-H (section 5.3). A basic triangle inequality optimisation based on Lemma 1 is used to eliminate unnecessary distance computations where the only required information is the centre-centre distances before computing the distance from each point to each centroid.

TIKM-S was implemented because it has a light overhead compared to Elkan and Compare-means algorithms. This gives TIKM-S the potential to gain speedup with a relatively small overhead.

7 Experimental Work

7.1 Software and Hardware

Hardware: Apache Hadoop and Apache Spark are deployed on the same cluster which consists of 1 master node and 16 worker nodes. The master node has 2 AMD CPUs running at 3.1GHz with 8 cores each, and 8x8GB DDR3 RAM, and 6x3TB Near Line SAS disks running at 7200 rpm. Each worker node has 1 Intel CPU running at 3.1 GHz with 4 cores, 4x4GB DDR3 RAM, and a 1x1TB SATA disk running at 7200 rpm. All the nodes run CentOS-6 (x86_64) operating system.

Software: The cluster uses Hadoop version 2.2.0 to run MapReduce on YARN. HDFS is configured with 128 MB default block size, and a replication factor of 3 replicas for each file. The default JVM heap size is 1 GB per task.

Apache Spark 2.1.1 is deployed on the same cluster as Hadoop, where YARN is used as the cluster manager, and HDFS as the distributed file system.

All algorithms were implemented in Java and compiled using JDK 1.7.0_79.

7.2 Datasets

The datasets used in the experimental work are either artificially generated datasets or real-world datasets.

Artificial datasets: Table 2 describes the characteristics of each artificial and real-world dataset in terms of its number of data points (n), number of dimensions (d), and the size in megabytes (MB). The data points in datasets DS[1-6] and DS[8-12] are normally distributed around 128 centres to form 128 well-separated clusters. This was done by, first, generating 128 centre vectors with a uniform distribution in $\mathbb{R}^d$. Then, an equal number of data points was generated and assigned to each centre with an independent univariate Gaussian distribution for each dimension. Except for dataset DS7, datasets DS[1-12] have a constant standard deviation $SD = 0.02$. This low SD generates clusters with high density around the centre vectors which creates well-separated clusters. The data points in dataset DS7 are generated with a uniformly random distribution where there is no underlying structure in the data. Dataset DS7 was generated to test the worst performance for the optimised algorithms where there are no meaningful clusters to be found.

Real-world datasets: To observe the practicality of the proposed algorithms on real-world settings, two naturally-clustered datasets haven been used in the experimental work.

The first dataset *covertype*, contains collected observations of trees from four areas of the Roosevelt National Forest in Colorado. The dataset contains 581,012 observations, where each observation has 55 integer attributes. The collected data represent information about the types of soil, the wilderness areas, elevation, slop, forest cover type, and several other characteristics. The dataset is publicly available at the UCI Machine Learning Repository (Blackard et al., 1998).

The second dataset *mnist* contains images of handwritten digits. Each image is represented by a 28 × 28 array. This array is flattened to form a 784 (28 × 28 = 784) dimensional vector, where each number in each dimension describes the darkness level of a specific pixel. The total number of images in this dataset is 60,000 images. The dataset is available online at (LeCun et al., 1998).

Name	Points (n)	Dimensions (d)	Size (MB)
DS1	100,000	8	15
DS2		32	28
DS3		128	235
DS4		512	941
DS5		1024	1884
DS6		2048	3788
DS7		512	947
DS8	1,000,000	128	1638
DS9	3,000,000		3584
DS10	5,000,000		5836
DS11	7,000,000		8192
DS12	9,000,000		10588
covertype	581,012	55	72
mnist	60,000	784	104

Table 2 Characteristics of artificial and real-world datasets.

7.3 Evaluation Metrics

Each iteration of K-Means on Hadoop consists of three major phases: map, shuffle, and reduce. The major operations that consumes the majority of K-Means running time occur in the map phase. Therefore, to fully understand the time consumed by each operation in the map phase, the map time is broken down into three major operations: 1) the average time to compute centre-centre distances, 2) the average time to compute point-centre distances, and 3) the average time to write extra information to HDFS. The shuffle time and reduce time are also reported.

The following is a detailed description of the evaluation metrics that are used to evaluate the performance of each algorithm.

- **Average iteration time** is the average running time per iteration over the total number of iterations that an algorithm has executed. This time includes: the CPU time, the I/O time, and the communication time. To compute the average time per iteration, the time for each iteration is obtained from Hadoop's job history log files at the end of each iteration. After all iterations complete running, the average time spent by each iteration is computed by dividing the sum of all iterations' times over the total number of iterations. The iteration time dose not include the time to initialise cluster centroids because it is a one time cost that occurs only once in each test.

- **Speedup**: In general, speedup measures the improvement in speed for an enhanced algorithm over a baseline algorithm (Grama et al., 2003). In this work, the performance of an optimised algorithm is reported as the speedup relative to NKM-H algorithm, where speedup is defined as the ratio of the average iteration time in NKM-H to the average iteration time of an optimised algorithms. For each algorithm, the average speedup over 10 trials is reported.

- **Average number of distance calculations** is the average number of point-centre distance calculations per iteration over the total number of iterations.

- **Average time to compute point-centre distances**: To obtain the time to compute point-centre distances, in a given mapper, the total consumed time by point-centre distance computations for points assigned to this mapper is computed. After the completion of all mappers, the average time per mapper over the number of mapper is computed. After that, The total of these averages is divided by the total number of iterations to obtain the average time per iteration.

- **Average time to compute centre-centre distances**: The average time to compute centre-centre distances per mapper over the total number of mappers is computed in each iteration. Then, the average time to compute these distances per iteration over the total number of iterations is reported.

- **Average shuffle time**: The average shuffle time per reducer over the total number of reducers is computed. This time is then averaged over the total number of iterations.

7.4 Comparative Analysis of All Implementations on Hadoop

The aim of this section is to investigate the scalability and efficiency of K-Means implementations using EVs and BFs with a wide range of number of clusters (k) and dimensions (d). Another aim is to determine the best and worse range of k and d for each algorithm. To accomplish these aims, algorithms: ELK-H-EV, CMP-H-EV, ELK-H-BF, and CMP-H-BF are tested against variable number of clusters k and dimensions d.

7.4.1 Variable Number of Clusters

This experiment uses dataset DS4 as input to test the performance of each algorithm with a variable number of clusters (k). Note that the number of distance computations in ELK-H-EV and CMP-H-EV is equivalent to ELK-H-BF and CMP-H-BF respectively. As shown in Figure 2(a) ELK-H-BF efficiently eliminates a large number of distance computations with all variations of k. ELK-H-BF eliminates around 76% when $k = 8$, and around 98% when $k = 512$ and 2048. CMP-H-BF works best with large number of clusters on well-separated clusters where it eliminates 98% and 99% when $k = 512$ and 2048, but skips only 13% and 11% distance computations with $k = 8$ and 32 respectively.

Since TIKM-H implements the simplest approach to avoid distance computations, it does not prune many computations with small k. For instance, only 0.3% and 5% of the distance computations are skipped when $k = 8$ and 32, respectively. However, the skipped distance computations rises to up to 78% when $k = 512$, and 94% when $k = 2048$.

It can be noticed from Figure 2(b) that, in general, the speedup for algorithms implemented with BFs is higher than the ones implemented using EVs. When $8 \leq k \leq 128$, ELK-H-EV and CMP-H-EV perform the same or worse than NKM-H. This is because the time to write EVs to HDFS in each iteration outweighs the time gained by skipping distance computations. When $k = 512$ and 2048, the speedups of ELK-H-BF are 6.6x and 5.4x, while ELK-H-EV achieves speedups of 3.4x and 4.4x. CMP-H-BF, on the other hand, is 9.3x and 9.6x faster than NKM-H when $k = 512$ and 2048, while CMP-H-EV is 3.8x and 6.6x faster with the same numbers of k. The speedup in ELK-H-BF drops from 6.6x when $k = 512$ to 5.4x as k in creases to 2048 due to the increase in the time to write BFs which is dependent on k. As the number of clusters gets larger than 32, TIKM-H starts to benefit from the pruned distance computations combined with the light computational overhead from centre-centre computations. The algorithm gains more speedups as the number of clusters increases.

7.4.2 Variable Number of Dimensions

It can be observed from Figure 2(b) in the previous experiment that the speedup of ELK-H-EV, CMP-H-EV, CMP-H-BF, and TIKM-H started to increase when $k = 128$. In order to measure the ability to accelerate with higher dimensions, each algorithm is tested with variable number of dimensions ($8 \leq k \leq 2048$), while the number of clusters is fixed at $k = 128$. Datasets DS[1-6] are used as input in this experiment.

In Figure 3(b), the speedup of ELK-H-EV reaches the peek when $d = 128$ (2.2x) and starts to decline as d gets larger than 128. Although ELK-H-EV eliminates most distance computations (see Figure 3(a)) with all variations of d, the speedup of ELK-H-EV drops to 0.3x when $d = 2048$. This drop in speed is caused by the dramatic increase in the overhead from writing EVs to HDFS (see Figure 3(c)).

7.5 Detailed Analysis of Implementations using BFs

It can be observed from the previous experiments that using BFs to implement K-Means variants has more potential to scale with increasing numbers of k and d than variants implemented with EVs. Therefore, further tests were carried on with BF implementations on various datasets with various number of clusters (k), dimensions (d) and data points (n).

7.5.1 Variable Number of Clusters and Dimensions

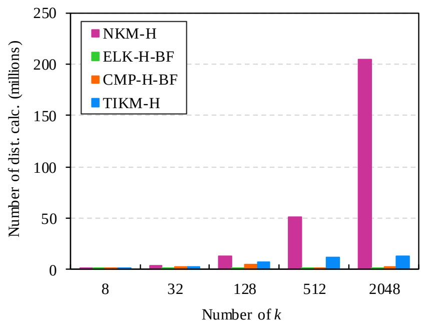
This section aims to investigate the impact of the number of clusters (k) and the number of dimensions (d) on the performance of ELK-H-BF, CMP-H-BF, and TIKM-H compared with NKM-H. The values of k and d varies from small, medium and large where $8 \leq k \leq 2048$ and $8 \leq d \leq 512$. The number of data points is fixed at $n = 100,000$, and the number of reducers $r = 1$. While datasets DS[1-4] are used as input to test the performance of each algorithm with clustered data, dataset DS7 is used as an input to test each algorithm with uniform random data, which is the worst case for the optimised K-Means implementations presented here. The real dataset covertype is used to test each algorithm with a real-world dataset.

From Figure 4 to Figure 7, it can be observed that, in general, CMP-H-BF outperforms NKM-H, ELK-H-BF, and TIKM-H when $512 \leq k \leq 2048$ for all the tests on variations of d. The highest speedup that CMP-H-BF achieves relative to NKM-H is 21.2x where $d = 128$ and $k = 2048$ (Figure 6(b)). This can be attributed to two reasons: 1) CMP-H-BF eliminates larger number of distance computations that is close to ELK-H-BF and larger than TIKM-H, which can be observed in Figures 4(a), 5(a), 6(a) and 7(a); and 2) the small overhead CMP-H-BF generates compared to ELK-H-BF, as can be seen in Figures 4(c), 5(c), 6(c) and 7(c).

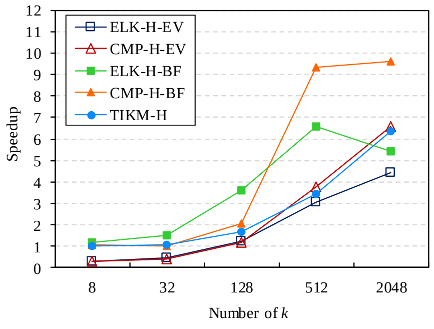
The best performance for ELK-H-BF with artificial datasets is when $128 \leq d \leq 512$ and $128 \leq k \leq 2048$, as shown in Figures 6(b) and 7(b). This is because distance computations become the dominant cost in NKM-H and ELK-H-BF eliminates more than 95% of these computations. Furthermore, the time gained from pruning distance computations outweighs the time wasted on reading and writing distance bounds and cluster assignments. Although ELK-H-BF eliminates the largest number of distance computations compared to the other two algorithms, the overhead it generates affects the performance greatly.

For small numbers of clusters and dimensions where $8 \leq k, d \leq 32$, no significant improvements in speed are reported for all the optimised algorithms. This is because even though the optimised algorithms eliminate some distance computations, the time that NKM-H spends on distance computations is already small, and the time gained from eliminating distance computations does not compensate the time spent on reading and writing the extra information.

The results of tests on uniformly random dataset DS7 is illustrated in Figure 8(b). Figure 8(b) shows that there is no gain in speedup for CMP-H-BF and TIKM-H relative to NKM-H. This is caused by the small number of eliminated distance calculations, which is bellow 1% of the total number of distance computations in both algorithms (see Figure 8(a)). ELK-H-BF, on the other hand, eliminates up to 82% (when $k = 2048$) distance computations from the total number of distance

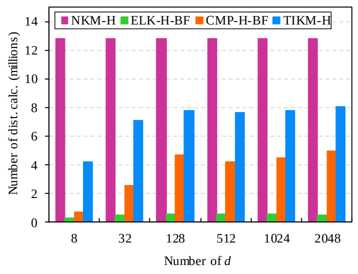


(a) Average number of distance calculations

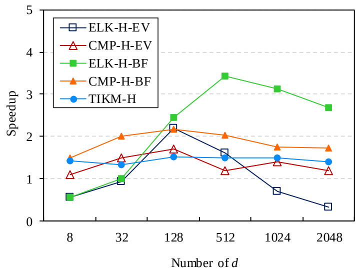


(b) Speedup relative to NKM-H

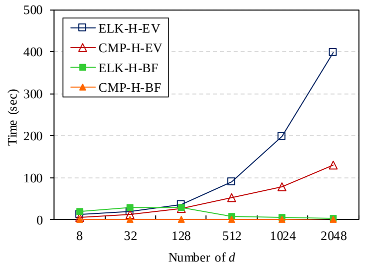
Figure 2: Average distance computations per iteration over the total number of iterations shown in Figure 2(a), and speedup relative to NKM-H shown in Figure 2(b). Each algorithm is tested against variable number of clusters (k).



(a) Average number of distance calculations

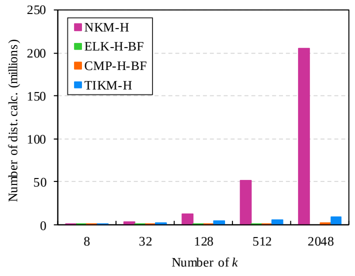


(b) Speedup relative to NKM-H

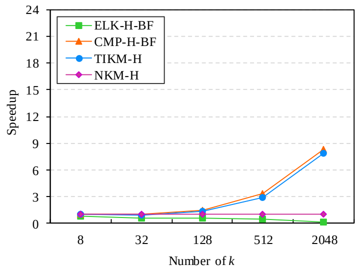


(c) Avg. time per iteration to write EVs and BFs

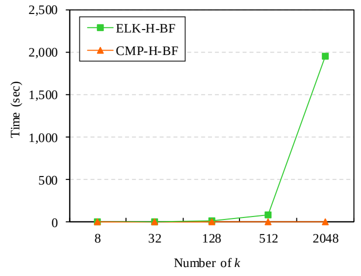
Figure 3: Average distance computations per iteration over the total number of iterations shown in Figure 3(a), speedup relative to NKM-H shown in Figure 3(b), and the average time to write EV's and BFs to HDFS shown in Figure 3(c). Each algorithm is tested against variable number of dimensions (d).



(a) Avg. no. of distance calculations



(b) Speedup relative to NKM-H

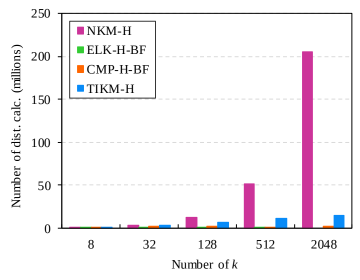


(c) Avg. time to write EVs and BFs

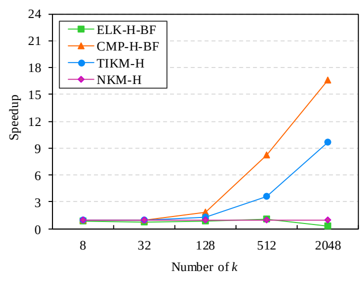
Figure 4: Average distance computations per iteration (Figure 4(a)), speedup relative to NKM-H (Figure 4(b)), and average time to write EVs and BFs (Figure 4(c)). (Dataset: DS1, $d = 8$)

computations. This is reflected on the speedup where ELK-H-BF was 2.8x times faster than NKM-H when the number of clusters are in the range of $128 \leq k \leq 512$.

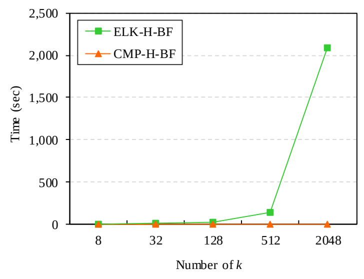
To study the performance of each algorithm with real-world settings, the real dataset covertype is used as an input for each algorithm and tested against variable number of clusters ($8 \leq k \leq 2048$). In general, CMP-H-BF and TIKM-H achieve high speedups relative to NKM-H as the number of clusters increases, as it can be observed from Figure 9(b). The speedups for CMP-H-BF and TIKM-H, relative to NKM-H, are 33x and 15x, respectively, where $k = 2048$. ELK-H-BF, on the other hand, achieves a speedup of 7.2x when $k = 128$ then the speedup starts to drop as the number of clusters gets larger until it reaches 3x when $k = 2048$. This drop in speed in ELK-H-BF is due to the increase of the overhead that is generated from writing distance bounds and cluster assignments to HDFS as Figure 9(c) shows.



(a) Avg. no. of distance calculations

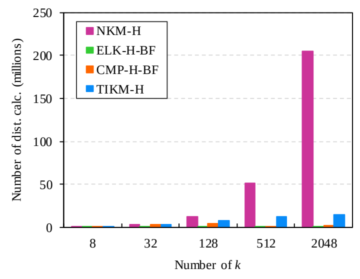


(b) Speedup relative to NKM-H

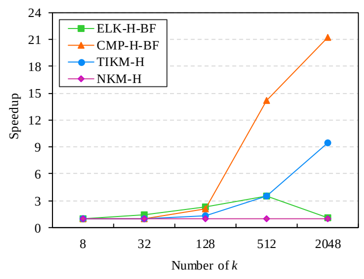


(c) Avg. time to write EVs and BFs

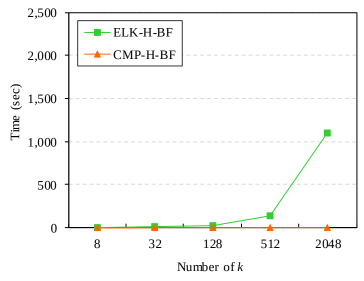
Figure 5: Average distance computations per iteration (Figure 5(a)), speedup relative to NKM-H (Figure 5(b)), and average time to write EVs and BFs (Figure 5(c)). (Dataset: DS2, $d = 32$)



(a) Avg. no. of distance calculations

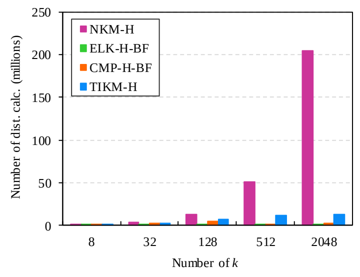


(b) Speedup relative to NKM-H

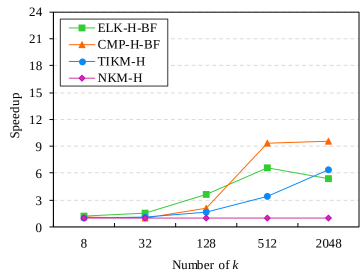


(c) Avg. time to write EVs and BFs

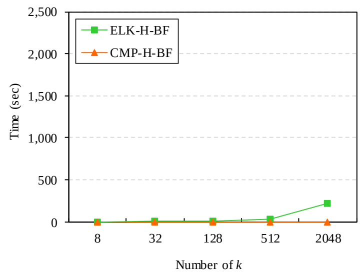
Figure 6: Average distance computations per iteration (Figure 6(a)), speedup relative to NKM-H (Figure 6(b)), and average time to write EVs and BFs (Figure 6(c)). (Dataset: DS3, $d = 128$)



(a) Avg. no. of distance calculations



(b) Speedup relative to NKM-H



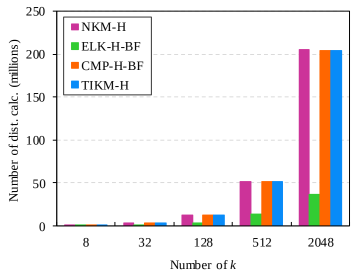
(c) Avg. time to write EVs and BFs

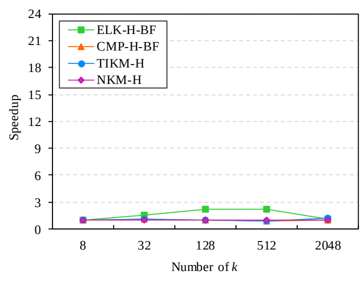
Figure 7: Average distance computations per iteration (Figure 7(a)), speedup relative to NKM-H (Figure 7(b)), and average time to write EVs and BFs (Figure 7(c)). (Dataset: DS4, $d = 512$)

7.5.2 Variable Number of Data Points

This section aims to test the performance of each algorithm against a variable number of data points (n), where $1,000,000 \leq n \leq 9,000,000$. Each algorithm is tested against five clustered datasets, DS[8-12], each with a variable number of data points and constant number of clusters $k = 128$, and dimensions $d = 128$.

Figure 10(a) illustrates the average number of distance computations per iteration and Figure 10(b) plots the average running time per iteration over the total number of iterations for each algorithms. The impact of the reduction in distance computations can be clearly observed in these two figures. When the number of data points is in the range of $1,000,000 \leq n \leq 7,000,000$, CMP-H-BF and TIKM-H skip around 40% and 70% distance computations, respectively. The number of skipped distance computations increases for both algorithms when $n = 9,000,000$ to about 85% for CMP-H-BF and 80% for TIKM-H, which in return reduces the iteration time for both algorithms (see Figure 10(b)). Although ELK-H-BF eliminates most of the distance computations (about 95%), the time to write BFs to HDFS, illustrated in Figure 10(c), makes the algorithm runs at almost the same speed as TIKM-H, except when $n = 9,000,000$, where TIKM-H is faster.


(a) Avg. no. of distance calculations


(b) Speedup relative to NKM-H

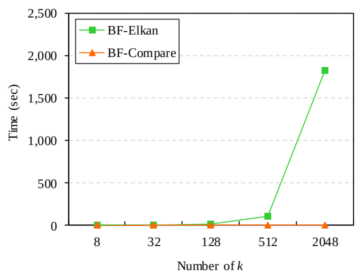
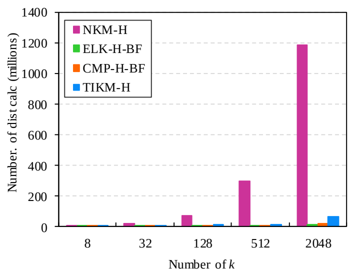
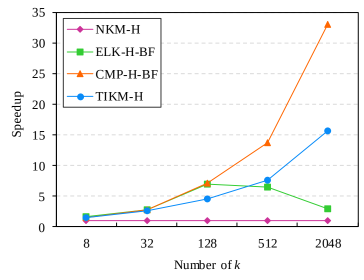

(c) Avg. time to write EVs and BFs

Figure 8: Average distance computations per iteration (Figure 8(a)), speedup relative to NKM-H (Figure 8(b)), and average time to write EVs and BFs (Figure 8(c)). (Dataset: DS7 (uniform), $d = 512$)


(a) Avg. no. of distance calculations


(b) Speedup relative to NKM-H

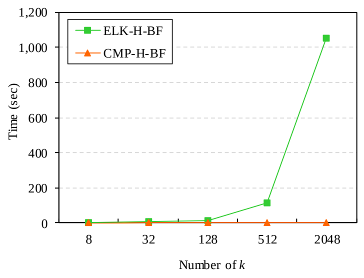
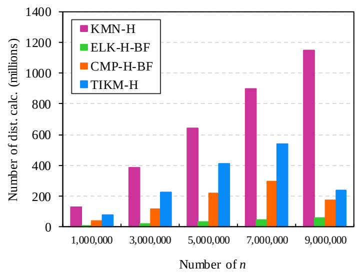
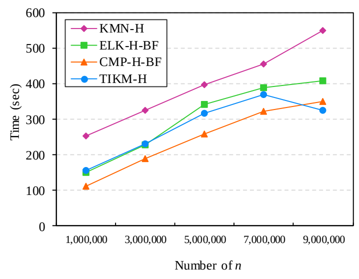

(c) Avg. time to write EVs and BFs

Figure 9: Average distance computations per iteration (Figure 9(a)), speedup relative to NKM-H (Figure 9(b)), and average time to write EVs and BFs (Figure 9(c)). (Dataset: covertype, $d = 55$)


(a) Avg. no. of distance calculations


(b) Speedup relative to NKM-H

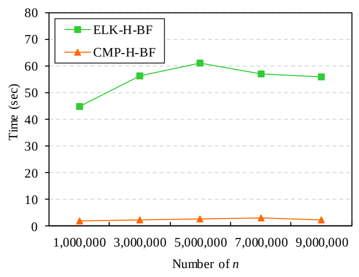

(c) Avg. time to write EVs and BFs

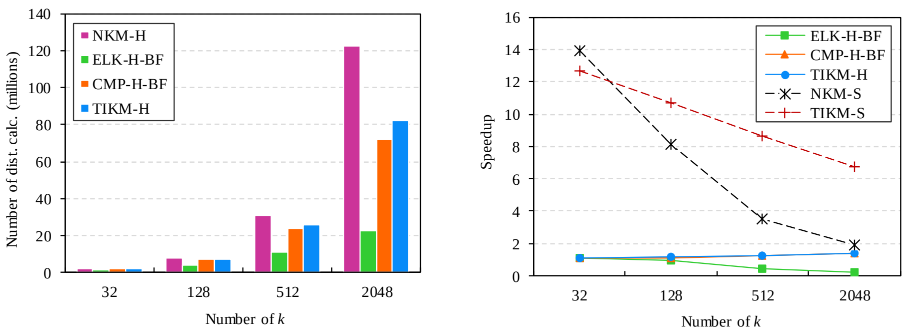
Figure 10: Results of tests on variable number of points (n). Average distance computations per iteration (Figure 10(a)), average iteration time (Figure 10(b)), and average time to write EVs and BFs (Figure 10(c)). (Dataset: DS[8-12], $d = 128$, $k = 128$)

This is because TIKM-H takes advantage of the light overhead and the large amount of skipped distance computations compared to the number of distance computations that was skipped where $n < 9,000,000$.

7.5.3 Comparative Analysis of K-Means Implementations on Hadoop and Spark

This section presents the results obtained from the experimental work on Apache Spark and compares these results against experimental work on Apache Hadoop. The goal of this experiment is to provide a comparative analysis between the performances of NKM-H, ELK-H-BF, CMP-H-BF, TIKM-H, NKM-S, and TIKM-S.

The experiments are executed on the real dataset *mnist*, and tested against variable number of clusters where $32 \leq k \leq 2048$, with fixed $d = 748$ and $n = 60000$. It can be observed from Figure 11(b) that NKM-S is faster than all K-Means implementations on Hadoop for $32 \leq k \leq 128$. This is attributed to the caching mechanism in Spark where input data is distributed over the cluster executer nodes and cached in-memory in the first iteration and reused in subsequent iterations in the form of Resilient Distributed Datasets (RDDs).



(a) Avg. no. of distance calculations (b) Speedup relative to NKM-H

Figure 11: Results of testing algorithms on Hadoop using BFs and algorithms on Spark on real dataset mnist with variable number of clusters (k). Average distance computations per iteration illustrated in Figure 11(a) and speedup relative to NKM-H illustrated in Figure 11(b). (Dataset: mnist, $d = 748$, $n = 60000$)

This feature, unlike Hadoop, reduces the I/O and communication overheads. However, as k increases, distance computations become the bottleneck and the speedup of NKM-S starts to decline to the point where it becomes very close to the running time of CMP-H-BF and TIKM-H when $k = 2048$.

TIKM-S, on the other hand, outperforms all algorithms including NKM-S when $128 \leq k \leq 2048$. This is because TIKM-S skips around 17%, 33%, and 45% of distance computations when $k = 128, 512$ and 2048, respectively, as can be seen in Figure 11(a), with a small overhead from computing k^2 centre-centre distances performed by each executer. CMP-H-BF and TIKM-H were able to reduce the large gap in speedup between them and NKM-S when $k = 2048$. That is, when $k = 2048$, CMP-H-BF and TIKM-H achieve 1.4x speedups, while NKM-S achieves and 1.9x. This makes CMP-H-BF and TIKM-H compete with NKM-S when the number of clusters is large.

8 Conclusion

The aim of this paper was to improve the efficiency and scalability of K-Means. To achieve this aim efficient variants of K-Means were implemented on Hadoop and Spark. The variants used triangle inequality to reduce the number of distance computations in each iteration. Some of these variants required extra information from the previous iteration, which Hadoop does not support. Therefore, two techniques , Extended Vectors (EVs) and Bounds Files (BFs), were proposed to allow Hadoop to pass required extra information from one iteration to the next. Furthermore, the performance of several optimisations of K-Means was investigated on Hadoop and Spark.

The comparative analysis of EV and BF approaches showed that significant speedups could be achieved by implementations using both approaches. However, implementations that use BFs are more efficient and scalable than those that use EVs to pass information to subsequent iterations. As the number of clusters and dimensions increases, the overhead that is generated from writing EVs to HDFS increases dramatically.

It was found through the use of clustered and uniform random datasets that the best performance of the optimised algorithms that use triangle inequality is with datasets that have well-separated clusters. This is because more distance computations can be avoided with well-clustered datasets.

The optimised algorithms did not achieve any significant speedups relative to NKM-H with low number of dimensions and clusters. The number of distance computations must be large enough to compensate the time spent on writing/reading extra information by the gained time from skipping distance computations in the optimised algorithms.

The comparison between the performances of algorithms that were implemented on Hadoop using BFs and the two implementations of K-Means on Spark showed the superiority of TIKM-S over all the implementations on Hadoop and Spark as the number of clusters was increased. Combining the in-memory caching mechanism that Spark employs with the simple triangle inequality optimisation gave TIKM-S the ability to outperform all the other implementations.

In the future work it would be interesting to compare these implementations with implementations of K-Means on other distributed computing frameworks such as Twister (Ekanayake et al., 2010) and Piccolo (Power and Li, 2010). In addition, implementations of other variants based on triangle inequality such as Hamerly's algorithm (Hamerly, 2010), and Adaptive K-Means (Drake and Hamerly, 2012) could be tested and compared on Hadoop and Spark.

References

Apache (2017). *Welcome to Apache Hadoop.* `http://hadoop.apache.org/` [Accessed: 15/10/2017].

Arthur, D. and Vassilvitskii, S. (2007). k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics.

Bahmani, B., Moseley, B., Vattani, A., Kumar, R., and Vassilvitskii, S. (2012). Scalable k-means++. *Proceedings of the VLDB Endowment*, 5(7):622–633.

Ball, G. and Hall, D. (1965). Isodata: A novel method of data analysis and pattern classification. Technical report, Stanford Research Institute, Menlo Park.

Blackard, J., Dean, D., and Anderson, C. (1998). *Covertype Data Set.* `https://archive.ics.uci.edu/ml/datasets/covertype` [Accessed: 02/04/2017].

Celebi, M. (2011). Improving the performance of k-means for color quantization. *Image and Vision Computing*, 29(4):260–271.

Celebi, M., Kingravi, H., and Vela, P. (2013). A comparative study of efficient initialization methods for the k-means clustering algorithm. *Expert Systems with Applications*, 40(1):200–210.

Dean, J. and Ghemawat, S. (2008). Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113.

Dhillon, I. and Modha, D. (2002). A data-clustering algorithm on distributed memory multiprocessors. In *Large-Scale Parallel Data Mining*, pages 245–260. Springer.

Drake, J. and Hamerly, G. (2012). Accelerated k-means with adaptive distance bounds. In *5th NIPS workshop on optimization for machine learning*, pages 42–53.

Effat, N., Divya, S., Sirisha, D., and Venkatesan, M. (2016). Enhanced k-means clustering approach for health care analysis using clinical documents. *International Journal of Pharmaceutical and Clinical Research*, 8(1):60–64.

Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S., Qiu, J., and Fox, G. (2010). Twister: a runtime for iterative mapreduce. In *Proceedings of the 19th ACM international symposium on high performance distributed computing*, pages 810–818. ACM.

Elkan, C. (2003). Using the triangle inequality to accelerate k-means. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 147–153.

Esteves, R. M., Hacker, T., and Rong, C. (2014). A new approach for accurate distributed cluster analysis for big data: competitive k-means. *International Journal of Big Data Intelligence 5*, 1(1-2):50–64.

Grama, A., Gupta, A., Karypis, G., and Kumar, V. (2003). *Introduction to Parallel Computing.* Pearson Education. Addison-Wesley, 2 edition.

Grolinger, K., Hayes, M., Higashino, W. A., L'Heureux, A., Allison, D. S., and Capretz, M. A. (2014). Challenges for mapreduce in big data. In *Services (SERVICES), 2014 IEEE World Congress on*, pages 182–189. IEEE.

Hamerly, G. (2010). Making k-means even faster. In *Proceedings of the 2010 SIAM international conference on data mining*, pages 130–140. SIAM.

Hamerly, G. and Drake, J. (2015). Accelerating lloyds algorithm for k-means clustering. In *Partitional clustering algorithms*, pages 41–78. Springer.

Hodgson, M. (1988). Reducing the computational requirements of the minimum-distance classifier. *Remote Sensing of Environment*, 25(1):117–128.

Jain, A. (2010). Data clustering: 50 years beyond K-means. *Pattern Recognition Letters*, 31(8):651–666.

Judd, D., Mckinley, P., and Jain, A. (1998). Large-scale parallel data clustering. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20:871–876.

Kanungo, T., Mount, D., Netanyahu, N., Piatko, C., Silverman, R., and Wu, A. (2002). An efficient k-means clustering algorithm: Analysis and implementation. *IEEE transactions on pattern analysis and machine intelligence*, 24(7):881–892.

Kuo, R., Ho, L., and Hu, C. (2002). Integration of self-organizing feature map and k-means algorithm for market segmentation. *Computers & Operations Research*, 29(11):1475–1493.

LeCun, Y., Cortes, C., and Burges, C. (1998). *THE MNIST DATABASE.* `http://yann.lecun.com/exdb/mnist/` [Accessed: 02/04/2017].

Li, Q., Wang, P., Wang, W., Hu, H., Li, Z., and Li, J. (2014). An efficient k-means clustering algorithm on mapreduce. In *International Conference on Database Systems for Advanced Applications*, pages 357–371. Springer.

Lin, J. and Dyer, C. (2010). *Data-Intensive Text Processing with Mapreduce.* Morgan & Claypool Publishers, San Rafael, Calif.

Lloyd, S. (1982). Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137.

MacQueen, J. (1967). Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA.

Mohebi, A., Aghabozorgi, S., Ying Wah, T., Herawan, T., and Yahyapour, R. (2016). Iterative big data clustering algorithms: a review. *Software: Practice and Experience*, 46:107–129.

Orchard, M. (1991). A fast nearest-neighbor search algorithm. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*, pages 2297–2300. IEEE.

Phillips, S. (2002). Acceleration of k-means and related clustering algorithms. In *Workshop on Algorithm Engineering and Experimentation*, pages 166–177. Springer.

Power, R. and Li, J. (2010). Piccolo: Building fast, distributed programs with partitioned tables. In *OSDI*, volume 10, pages 1–14.

Shi, J., Qiu, Y., Minhas, U. F., Jiao, L., Wang, C., Reinwald, B., and Özcan, F. (2015). Clash of the titans: Mapreduce vs. spark for large scale data analytics. *Proceedings of the VLDB Endowment*, 8(13):2110–2121.

Steinbach, M., Karypis, G., and Kumar, V. (2000). A comparison of document clustering techniques. In *KDD workshop on text mining*, volume 400, pages 525–526. Boston.

Steinhaus, H. (1956). Sur la division des corp materiels en parties. *Bull. Acad. Polon. Sci*, 1:801–804.

Tavazoie, S., Hughes, J., Campbell, M., Cho, R., and Church, G. (1999). Systematic determination of genetic network architecture. *Nature genetics*, 22(3):281.

White, T. (2012). *Hadoop: the definitive guide*. O'Reilly, Farnham.

WLCG (2017). *WLCG - Worldwide LHC Computing Grid*. Website. `http://wlcg-public.web.cern.ch/about` [Accessed: 03/10/2017].

Wu, X., Kumar, V., Quinlan, R., Ghosh, J., Yang, Q., Motoda, H., McLachlan, G., Ng, A., Liu, B., Yu, P., Zhou, Z., Steinbach, M., Hand, D., and Steinberg, D. (2008). Top 10 algorithms in data mining. *Knowledge and Information Systems*, 14(1):1–37.

Xu, R. and Wunsch, D. (2009). *Clustering*, volume 10. Wiley-IEEE Press.

Yeung, K., Medvedovic, M., and Bumgarner, R. (2003). Clustering gene-expression data with repeated measurements. *Genome biology*, 4(5):R34.

YouTube (2017). *YouTube Statistics*. `http://www.youtube.com/yt/press/statistics.html` [Accessed: 03/10/2017].

Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I. (2010). Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95.

Zhang, J. and Qiu, B. (2013). Mammoth data in the cloud: Clustering social images. volume 23, page 231. IOS Press.

Bibliography

[1] A. C. Murthy, V. K. Vavilapalli, D. Eadline, J. Niemiec, and J. Markham, *Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2*, 1st ed. Upper Saddle River, NJ: Addison Wesley, mar 2014.

[2] "Visualizing MNIST: An Exploration of Dimensionality Reduction," accessed 2018-07-28. [Online]. Available: http://colah.github.io/posts/2014-10-Visualizing-MNIST/

[3] "The Worldwide LHC Computing Grid (WLCG)," accessed 2017-01-20. [Online]. Available: http://wlcg-public.web.cern.ch/about

[4] "Youtube statistics," accessed 2017-09-16. [Online]. Available: http://www.youtube.com/yt/press/statistics.html

[5] J. Lin and C. Dyer, *Data-Intensive Text Processing with Mapreduce*, G. Hirst, Ed. San Rafael, Calif.: Morgan & Claypool Publishers, April 2010.

[6] D. Laney, "3d data management: Controlling data volume, velocity, and variety," Department of Computer Science, Michigan State University, East Lansing, Michigan, Tech. Rep. 949, February 2001.

[7] M. A. Beyer and D. Laney, "The importance of big data: A definition," *Gartner Publications*, pp. 1–9, 2012.

[8] I. A. T. Hashem, I. Yaqoob, N. B. Anuar, S. Mokhtar, A. Gani, and S. U. Khan, “The rise of “big data” on cloud computing: Review and open research issues,” *Information Systems*, vol. 47, pp. 98–115, 2015.

[9] A. Mohebi, S. Aghabozorgi, T. Ying Wah, T. Herawan, and R. Yahyapour, “Iterative big data clustering algorithms: a review,” *Software: Practice and Experience*, vol. 46, no. 1, pp. 107–129, Jan. 2016.

[10] M. D. Assunção, R. N. Calheiros, S. Bianchi, M. A. S. Netto, and R. Buyya, “Big Data computing and clouds: Trends and future directions,” *Journal of Parallel and Distributed Computing*, vol. 79–80, pp. 3–15, May 2015.

[11] J. S. Ward and A. Barker, “Undefined by data: a survey of big data definitions,” *arXiv preprint arXiv:1309.5821*, 2013.

[12] R. Xu and D. Wunsch, *Clustering.* John Wiley & Sons, 2009, vol. 10.

[13] S. Shahrivari and S. Jalili, “Single-pass and linear-time k-means clustering based on mapreduce,” *Information Systems*, vol. 60, pp. 1–12, 2016.

[14] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” *Commun. ACM*, vol. 51, no. 1, pp. 107–113, Jan. 2008.

[15] “Welcome to apache hadoop!” accessed 2017-04-16. [Online]. Available: http://hadoop.apache.org/

[16] T. White, *Hadoop: the definitive guide.* Farnham: O’Reilly, 2012.

[17] X. Wu, V. Kumar, J. R. Quinlan, J. Ghosh, Q. Yang, H. Motoda, G. J. McLachlan, A. Ng, B. Liu, P. S. Yu, Z.-H. Zhou, M. Steinbach, D. J. Hand, and D. Steinberg, “Top 10 algorithms in data mining,” *Knowledge and Information Systems*, vol. 14, no. 1, pp. 1–37, Jan. 2008.

[18] S. Lloyd, “Least Squares Quantization in PCM,” *IEEE Trans. Inf. Theor.*, vol. 28, no. 2, pp. 129–137, 1982.

[19] C. Elkan, “Using the triangle inequality to accelerate k-means,” in *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 2003, pp. 147–153.

[20] D. Judd, P. K. Mckinley, and A. K. Jain, “Large-scale parallel data clustering,” *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 20, pp. 871–876, 1998.

[21] G. Hamerly and J. Drake, “Accelerating Lloyd’s Algorithm for k-Means Clustering,” in *Partitional Clustering Algorithms*, M. E. Celebi, Ed. Springer International Publishing, 2015, pp. 41–78, dOI: 10.1007/978-3-319-09259-1_2.

[22] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, “Spark: Cluster Computing with Working Sets,” in *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, ser. HotCloud’10. Berkeley, CA, USA: USENIX Association, 2010, pp. 10–10.

[23] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, “Twister: A Runtime for Iterative MapReduce,” in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, ser. HPDC ’10. New York, NY, USA: ACM, 2010, pp. 810–818.

[24] H. Lee, M. Kang, S.-B. Youn, J.-G. Lee, and Y. Kwon, “An experimental comparison of iterative mapreduce frameworks,” in *Proceedings of the 25th ACM International on Conference on Information and Knowledge Management*, ser. CIKM ’16. New York, NY, USA: ACM, 2016, pp. 2089–2094.

[25] A. K. Jain, M. N. Murty, and P. J. Flynn, “Data Clustering: A Review,” *ACM Comput. Surv.*, vol. 31, no. 3, pp. 264–323, Sep. 1999.

[26] J. Han, *Data mining: concepts and techniques*, 3rd ed. Burlington, MA: Elsevier, 2011.

[27] P. Berkhin, *A Survey of Clustering Data Mining Techniques.* Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, pp. 25–71.

[28] D. R. Cutting, D. R. Karger, J. O. Pedersen, and J. W. Tukey, "Scatter/-gather: A cluster-based approach to browsing large document collections," in *Proceedings of the 15th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '92. New York, NY, USA: ACM, 1992, pp. 318–329.

[29] M. Steinbach, G. Karypis, and V. Kumar, "A comparison of document clustering techniques," in *In 6th ACM SIGKDD, World Text Mining Conference*, 2000.

[30] de Souto Marcilio CP, C. I. G., de Araujo Daniel SA, L. T. B., and S. Alexander, "Clustering cancer gene expression data: a comparative study," *BMC Bioinformatics*, vol. 9, no. 1, p. 497, Nov 2008.

[31] J. Hou, W. Liu, E. Xu, and H. Cui, "Towards parameter-independent data clustering and image segmentation," *Pattern Recognition*, vol. 60, pp. 25–36, 2016.

[32] R. J. Kuo, L. Ho, and C. Hu, "Cluster analysis in industrial market segmentation through artificial neural network," *Computers & Industrial Engineering*, vol. 42, no. 2, pp. 391–399, 2002.

[33] T. Zhang, R. Ramakrishnan, and M. Livny, "Birch: an efficient data clustering method for very large databases," in *ACM Sigmod Record*, vol. 25, no. 2. ACM, 1996, pp. 103–114.

[34] S. Guha, R. Rastogi, and K. Shim, "Cure: an efficient clustering algorithm for large databases," in *ACM Sigmod Record*, vol. 27, no. 2. ACM, 1998, pp. 73–84.

[35] L. I. Kuncheva and S. T. Hadjitodorov, "Using diversity in cluster ensembles," in *Systems, man and cybernetics, 2004 IEEE international conference on*, vol. 2. IEEE, 2004, pp. 1214–1219.

[36] Y. Cheng, "Mean shift, mode seeking, and clustering," *IEEE transactions on pattern analysis and machine intelligence*, vol. 17, no. 8, pp. 790–799, 1995.

[37] M. Ester, H.-P. Kriegel, J. Sander, X. Xu *et al.*, "A density-based algorithm for discovering clusters in large spatial databases with noise." in *Kdd*, vol. 96, no. 34, 1996, pp. 226–231.

[38] A. K. Jain, "Data clustering: 50 years beyond K-means," *Pattern Recognition Letters*, vol. 31, no. 8, pp. 651–666, Jun. 2010.

[39] H. Steinhaus, "Sur la division des corp materiels en parties," *Bull. Acad. Polon. Sci*, vol. 1, pp. 801–804, 1956.

[40] G. Ball and D. Hall, "Isodata: A novel method of data analysis and pattern classification," Stanford Research Institute, Menlo Park, Tech. Rep., 1965.

[41] J. MacQueen *et al.*, "Some methods for classification and analysis of multivariate observations," in *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, vol. 1, no. 14. Oakland, CA, USA., 1967, pp. 281–297.

[42] Celebi and M. Emre, "Improving the performance of k-means for color quantization," *Image and Vision Computing*, vol. 29, no. 4, pp. 260–271, 2011.

[43] T. Kanungo, D. M. Mount, N. S. Netanyahu, C. D. Piatko, R. Silverman, and A. Y. Wu, "An Efficient k-Means Clustering Algorithm: Analysis and

Implementation," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 24, no. 7, pp. 881–892, Jul. 2002.

[44] Juang, Li-Hong, Wu, and Ming-Ni, "Mri brain lesion image detection based on color-converted k-means clustering segmentation," *Measurement*, vol. 43, no. 7, pp. 941–949, 2010.

[45] R. Kuo, L. Ho, and C. Hu, "Integration of self-organizing feature map and k-means algorithm for market segmentation," *Computers and Operations Research*, vol. 29, no. 11, pp. 1475 – 1493, 2002.

[46] S. Tavazoie, J. D. Hughes, M. J. Campbell, R. J. Cho, and G. M. Church, "Systematic determination of genetic network architecture," *Nature Genetics*, vol. 22, no. 3, pp. 281–285, Jul. 1999.

[47] K. Y. Yeung, M. Medvedovic, and R. E. Bumgarner, "Clustering gene-expression data with repeated measurements," *Genome Biology*, vol. 4, no. 5, p. R34, Apr. 2003.

[48] E. Naaz, D. Sharma, D. Sirisha, and V. M, "Enhanced k-means clustering approach for health care analysis using clinical documents," *International Journal of Pharmaceutical and Clinical Research*, vol. 8, no. 1, pp. 60–64, 2016.

[49] M. Steinbach, G. Karypis, V. Kumar *et al.*, "A comparison of document clustering techniques," in *KDD workshop on text mining*, vol. 400, no. 1. Boston, 2000, pp. 525–526.

[50] M. E. Celebi, H. A. Kingravi, and P. A. Vela, "A comparative study of efficient initialization methods for the k-means clustering algorithm," *Expert Systems with Applications*, vol. 40, no. 1, pp. 200–210, Jan. 2013.

[51] P. Berkhin, “A Survey of Clustering Data Mining Techniques,” in *Grouping Multidimensional Data*, J. Kogan, C. Nicholas, and M. Teboulle, Eds. Springer Berlin Heidelberg, Jan. 2006, pp. 25–71.

[52] J. M. Peña, J. A. Lozano, and P. Larrañaga, “An empirical comparison of four initialization methods for the K-Means algorithm,” *Pattern Recognition Letters*, vol. 20, no. 10, pp. 1027–1040, Oct. 1999.

[53] D. S. Hochbaum and D. B. Shmoys, “A best possible heuristic for the k-center problem,” *Math. Oper. Res.*, vol. 10, no. 2, pp. 180–184, May 1985.

[54] D. Arthur and S. Vassilvitskii, “K-means++: The Advantages of Careful Seeding,” in *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA ’07. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2007, pp. 1027–1035.

[55] P. S. Bradley and U. M. Fayyad, “Refining Initial Points for K-Means Clustering,” in *Proceedings of the Fifteenth International Conference on Machine Learning*, ser. ICML ’98. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1998, pp. 91–99.

[56] G. Hamerly and C. Elkan, “Learning the k in k-means,” in *Advances in neural information processing systems*, 2004, pp. 281–288.

[57] D. Pelleg and A. Moore, “X-means: Extending k-means with efficient estimation of the number of clusters,” in *In Proceedings of the 17th International Conf. on Machine Learning.* Morgan Kaufmann, 2000, pp. 727–734.

[58] R. E. Kass and L. Wasserman, “A reference bayesian test for nested hypotheses and its relationship to the schwarz criterion,” *Journal of the american statistical association*, vol. 90, no. 431, pp. 928–934, 1995.

[59] D. Pelleg and A. Moore, "Accelerating exact k-means algorithms with geometric reasoning," in *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '99. New York, NY, USA: ACM, 1999, pp. 277–281.

[60] D. Pettinger and G. Di Fatta, "Scalability of efficient parallel K-Means," in *2009 5th IEEE International Conference on E-Science Workshops*, Dec. 2009, pp. 96–101.

[61] J. Bentley, "Multidimensional binary search trees used for associative searching," *Commun. ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[62] K. Alsabti, S. Ranka, and V. Singh, "An Efficient K-Means Clustering Algorithm," in *In Proceedings of IPPS/SPDP Workshop on High Performance Data Mining*, 1998.

[63] A. W. Moore, "The anchors hierarchy: Using the triangle inequality to survive high dimensional data," in *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence.* Morgan Kaufmann Publishers Inc., 2000, pp. 397–405.

[64] M. E. Hodgson, "Reducing the computational requirements of the minimum-distance classifier," *Remote Sensing of Environment*, vol. 25, no. 1, pp. 117 – 128, 1988.

[65] M. T. Orchard, "A fast nearest-neighbor search algorithm," in *[Proceedings] ICASSP 91: 1991 International Conference on Acoustics, Speech, and Signal Processing*, Apr. 1991, pp. 2297–2300 vol.4.

[66] S. J. Phillips, "Acceleration of K-Means and Related Clustering Algorithms," in *Revised Papers from the 4th International Workshop on Algorithm Engineering and Experiments*, ser. ALENEX '02. London, UK, UK: Springer-Verlag, 2002, pp. 166–177.

[67] G. Hamerly, “Making k-means even faster,” in *Proceedings of the 2010 SIAM international conference on data mining.* SIAM, 2010, pp. 130–140.

[68] R. E. Bellman, *Adaptive control processes: a guided tour.* Princeton university press, 2015, vol. 2045.

[69] M. Steinbach, L. Ertöz, and V. Kumar, “The challenges of clustering high dimensional data,” in *New directions in statistical physics.* Springer, 2004, pp. 273–309.

[70] J. Dean and S. Ghemawat, “MapReduce: Simplified Data Processing on Large Clusters,” in *Proceedings of the 6th Conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, ser. OSDI’04. Berkeley, CA, USA: USENIX Association, 2004, pp. 10–10.

[71] “Amazon emr,” accessed 2017-09-16. [Online]. Available: https://aws.amazon.com/emr/

[72] S. Ghemawat, H. Gobioff, and S.-T. Leung, “The Google File System,” in *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’03. New York, NY, USA: ACM, 2003, pp. 29–43.

[73] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, “The Hadoop Distributed File System,” in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST ’10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[74] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, “HaLoop: Efficient Iterative Data Processing on Large Clusters,” *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 285–296, Sep. 2010.

[75] P. C. Zikopoulos, D. DeRoos, K. Parasuraman, T. Deutsch, D. Corrigan, and J. Giles, *Harness the power of Big Data: the IBM Big Data platform.* New York; Singapore: McGraw-Hill, 2013, oCLC: 829742565.

[76] K. Grolinger, M. Hayes, W. Higashino, A. L'Heureux, D. Allison, and M. Capretz, "Challenges for MapReduce in Big Data," in *2014 IEEE World Congress on Services (SERVICES)*, Jun. 2014, pp. 182–189.

[77] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, "Parallel Data Processing with MapReduce: A Survey," *SIGMOD Rec.*, vol. 40, no. 4, pp. 11–20, Jan. 2012.

[78] D. Singh and C. K. Reddy, "A survey on platforms for big data analytics," *Journal of Big Data*, vol. 2, no. 1, p. 8, Oct. 2014.

[79] F. Li, B. C. Ooi, M. T. Özsu, and S. Wu, "Distributed Data Management Using MapReduce," *ACM Comput. Surv.*, vol. 46, no. 3, pp. 31:1–31:42, Jan. 2014.

[80] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "iMapReduce: A Distributed Computing Framework for Iterative Computation," in *Proceedings of the 2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, ser. IPDPSW '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 1112–1121.

[81] A. Pavlo, E. Paulson, A. Rasin, D. J. Abadi, D. J. DeWitt, S. Madden, and M. Stonebraker, "A comparison of approaches to large-scale data analysis," in *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '09. New York, NY, USA: ACM, 2009, pp. 165–178.

[82] C. Doulkeridis and K. Nørvåg, "A survey of large-scale analytical query processing in mapreduce," *The VLDB Journal*, vol. 23, no. 3, pp. 355–380, Jun. 2014.

[83] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking: Bringing order to the web," in *Proceedings of the 7th International World Wide Web Conference*, Brisbane, Australia, 1998, pp. 161–172.

[84] "Logistic regression," accessed 2017-04-16. [Online]. Available: https://en.wikipedia.org/wiki/Logistic_regression

[85] "Apache mesos," accessed 2015-01-03. [Online]. Available: http://mesos.apache.org/

[86] "AWS |Amazon Elastic Compute Cloud (EC2) - Scalable Cloud Hosting," accessed 2015-01-03. [Online]. Available: http://aws.amazon.com/ec2/

[87] "MLlib is Apache Spark's scalable machine learning library," accessed 2017-09-16. [Online]. Available: https://spark.apache.org/mllib/

[88] R. Power and J. Li, "Piccolo: Building fast, distributed programs with partitioned tables." in *OSDI*, vol. 10, 2010, pp. 1–14.

[89] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: a system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 2010, pp. 135–146.

[90] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "Mapreduce online." in *Nsdi*, vol. 10, no. 4, 2010, p. 20.

[91] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: a distributed framework for prioritized iterative computations," in *Proceedings of the 2nd ACM Symposium on Cloud Computing*. ACM, 2011, p. 13.

[92] I. S. Dhillon and D. S. Modha, “A Data-Clustering Algorithm on Distributed Memory Multiprocessors,” in *KDD’99 Workshop on High Performance Knowledge Discovery.* London, UK: Springer-Verlag, 1999, pp. 245–260.

[93] K. Kerdprasop and N. Kerdprasop, “Parallelization of K-means Clustering on Multi-core Processors,” in *Proceedings of the 10th WSEAS International Conference on Applied Computer Science*, ser. ACS’10. Stevens Point, Wisconsin, USA: World Scientific and Engineering Academy and Society (WSEAS), 2010, pp. 472–477.

[94] E. Lusk, N. Doss, and A. Skjellum, “A high-performance, portable implementation of the MPI message passing interface standard,” *Parallel Computing*, vol. 22, pp. 789–828, 1996.

[95] J. Drake and G. Hamerly, “Accelerated k-means with adaptive distance bounds,” in *5th NIPS workshop on optimization for machine learning*, 2012, pp. 42–53.

[96] W. Zhao, H. Ma, and Q. He, “Parallel K-Means Clustering Based on MapReduce,” in *Cloud Computing*, ser. Lecture Notes in Computer Science, M. G. Jaatun, G. Zhao, and C. Rong, Eds. Springer Berlin Heidelberg, Jan. 2009, pp. 674–679.

[97] A. A. Golghate and S. W. Shende, “Parallel k-means clustering based on hadoop and hama,” *International Journal of Computing and Technology*, vol. 1, no. 2014, pp. 33–37, 2014.

[98] P. Anchalia, A. Koundinya, and N. Srinath, “MapReduce Design of K-Means Clustering Algorithm,” in *2013 International Conference on Information Science and Applications (ICISA)*, Jun. 2013, pp. 1–5.

[99] I. K. Savvas and M. T. Kechadi, “Mining on the cloud-k-means with mapreduce.” in *CLOSER*, 2012, pp. 413–418.

[100] R. Esteves, R. Pais, and C. Rong, “K-means Clustering in the Cloud – A Mahout Test,” in *2011 IEEE Workshops of International Conference on Advanced Information Networking and Applications (WAINA)*, Mar. 2011, pp. 514–519.

[101] “Apache mahout: Scalable machine learning and data mining,” accessed 2017-04-16. [Online]. Available: https://mahout.apache.org/

[102] J. Zhang and B. Qiu, “Mammoth data in the cloud: Clustering social images,” in *Cloud Computing and Big Data.* IOS Press, 2013, vol. 23, p. 231.

[103] B. Bahmani, B. Moseley, A. Vattani, R. Kumar, and S. Vassilvitskii, “Scalable K-means++,” *Proc. VLDB Endow.*, vol. 5, no. 7, pp. 622–633, Mar. 2012.

[104] T. H. Rui Máximo Esteves, “A new approach for accurate distributed cluster analysis for Big Data: competitive K-Means,” *Int. J. of Big Data Intelligence*, vol. 1, no. 1/2, pp. 50 – 64, 2014.

[105] X. Cui, P. Zhu, X. Yang, K. Li, and C. Ji, “Optimized big data k-means clustering using mapreduce,” *The Journal of Supercomputing*, vol. 70, no. 3, pp. 1249–1259, 2014.

[106] Q. Li, P. Wang, W. Wang, H. Hu, Z. Li, and J. Li, “An Efficient K-means Clustering Algorithm on MapReduce,” in *Database Systems for Advanced Applications*, ser. Lecture Notes in Computer Science, S. S. Bhowmick, C. E. Dyreson, C. S. Jensen, M. L. Lee, A. Muliantara, and B. Thalheim, Eds. Bali, Indonesia: Springer International Publishing, Apr. 2014, pp. 357–371, dOI: 10.1007/978-3-319-05810-8_24.

[107] J. Shi, Y. Qiu, U. F. Minhas, L. Jiao, C. Wang, B. Reinwald, and F. Özcan, "Clash of the titans: Mapreduce vs. spark for large scale data analytics," *Proc. VLDB Endow.*, vol. 8, no. 13, pp. 2110–2121, Sep. 2015.

[108] B. Wang, J. Yin, Q. Hua, Z. Wu, and J. Cao, "Parallelizing k-means-based clustering on spark," in *International Conference on Advanced Cloud and Big Data, CBD 2016, Chengdu, China, August 13-16, 2016*, 2016, pp. 31–36.

[109] "Cluto–software for clustering high-dimensional datasets, version 2.1.1," accessed 2017-07-16. [Online]. Available: http://glaros.dtc.umn.edu/gkhome/views/cluto

[110] S. Gopalani and R. Arora, "Comparing apache spark and map reduce with performance analysis using k-means," *International Journal of Computer Applications*, vol. 113, no. 1, pp. 8–11, March 2015, full text available.

[111] J. Blackard, D. Dean, and C. Anderson, "Covertype data set," accessed 2017-09-16. [Online]. Available: https://archive.ics.uci.edu/ml/datasets/covertype

[112] Y. LeCun, C. Cortes, and C. Burges, "The mnist database," accessed 2017-09-16. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[113] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, 2nd ed., ser. Pearson Education. Addison-Wesley, 2003.