Efficient clustering techniques on Hadoop and Spark

Article

Accepted Version

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 ← ∞
5      for j ← 1 to k do
6        d ← d(x_i, c_j)
7        if d < minDistance then
8          minDistance ← d
9          assign x_i to c_j
10       end
11     end
12   end
13   for j ← 1 to k do
14     c_j ← (1/|c_j|) Σ_{x∈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,*

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

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 ← 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 ← value minDistance ← ∞
5  |  a ← −1
6  |  for j ← 1 to k do
7  |  |  d ← d(x, c_j)
8  |  |  if d < minDistance then
9  |  |  |  minDistance ← d
10 |  |  |  a ← 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 ← 0
5  |  sum ← (0,0,...,0)
6  |  foreach x ∈ values do
7  |  |  sum ← sum + x //vector sum
8  |  |  pointsCounter ← pointsCount + 1
9  |  end
10 |  c'_j ← 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.

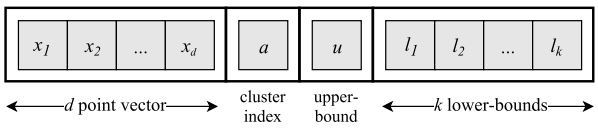


Figure 1: Structure of an Extended Vector in ELK-H-EV.

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} ← d(c_i, c_j) //for all 1 ≤ i, j ≤ 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 ← point
8      for j ← 1 to k do
9          t_j ← false
10     end
11     minDistance ← ∞
12     for j ← 1 to k do
13         if t_j then continue
14         d ← d(w.x, c_j)
15         w.l_j ← d
16         if d < minDistance then
17             minDistance ← d
18             w.u ← minDistance
19             w.a ← j
20             for z ← j + 1 to k do
21                 if s_{j,z} ≥ 2 * d then
22                     t_z ← 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} ← d(c_i, c_j), for all 1 ≤ i, j ≤ k
4      compute h_j ← min_{j≠j'} d(c_j, c_{j'}) * 0.5, for all
         1 ≤ j, j' ≤ k
5  Function map(offset, value):
6      let w be an Extended Vector
7      w ← value
8      //update k lower-bounds
9      for j ← 1 to k do
10         w.l_j ← max[w.l_j − m_j, 0]
11     end
12     w.u ← w.u + m_{w.a} //update upper-bound
13     g ← true //flag to check if u is updated
14     d1, d2 ← 0
15     if w.u ≤ h(w.a) then continue
16     for j ← 1 to k do
17         if (j ≠ w.a)
18         &(w.u > w.l_j)
19         &(w.u > s_{w.a,j} * 0.5) then
20             if g then
21                 d1 ← d(w.x, c_{w.a})
22                 w.u ← d1
23                 w.l_{w.a} ← d1
24                 g ← false
25             else
26                 d1 ← w.u
27             end
28             if d1 > w.l_j or d1 > s_{w.a,j} * 0.5 then
29                 d2 ← d(w.x, c_j)
30                 w.l_j ← d2
31                 if d2 < d1 then
32                     w.a ← j
33                     w.u ← d2
34                     g ← false
35                 end
36             end
37         end
38     end
39     write w to HDFS
40     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 7: CMP-H-EV-Mapper(k)

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

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} ← d(c_i, c_j) //for all 1 ≤ i, j ≤ k
4  Function map(offset, value)
5      x ← value
6      let b be a collection that stores the extra
        information for x
7      for j ← 1 to k do
8          t_j ← false
9      end
10     minDistance ← ∞
11     for j ← 1 to k do
12         if t_j then continue
13         d ← d(x, c_j)
14         b.l_j ← d
15         if d < minDistance then
16             minDistance ← d
17             b.u ← minDistance
18             b.a ← j
19             for z ← j + 1 to k do
20                 if s_{j,z} ≥ 2 * d then
21                     t_z ← 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} ← d(c_i, c_j), for all 1 ≤ i, j ≤ k
4      compute h_j ← min_{j≠j'} d(c_j, c_{j'}) * 0.5, for all
        j ∈ 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 ← value
9      let b be a collection that stores the cluster
        index assigned to x
10     b ← f(pointsCounter)
11     //update k lower-bounds
12     for j ← 1 to k do
13         b.l_j ← max[b.l_j − m_j, 0]
14     end
15     b.u ← b.u + m_{b.a} //update upper-bound
16     g ← true //flag to check if u is updated
17     d1, d2 ← 0
18     if b.u ⩽ h_{b.a} then continue
19     for j ← 1 to k do
20         if (j ≠ b.a) &(b.u > b.l_j)
            &(b.u > s_{b.a,j} * 0.5) then
21             if g then
22                 d1 ← d(x, c_{b.a})
23                 b.u ← d1
24                 b.l_{b.a} ← d1
25                 g ← false
26             else
27                 d1 ← b.u
28             end
29             if d1 > b.l_j or d1 > s_{b.a,j} * 0.5 then
30                 d2 ← d(x, c_j)
31                 b.l_j ← d2
32                 if d2 < d1 then
33                     b.a ← j
34                     b.u ← d2
35                     g ← false
36                 end
37             end
38         end
39     end
40     pointsCounter ← 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.

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.

Algorithm 10: CMP-H-BF-Mapper(k)

```
1  Function setup():
2      load centroids to C
3      compute s_{i,j} ← d(c_i, c_j) //for all 1 ≤ i, j ≤ 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 ← 1
8      end
9  Function map(offset, value)
10     x ← value
11     let b be a collection that stores the cluster
         index assigned to x
12     if iteration == 1 then
13         b.a ← 1 //initialise cluster assignment
14     else
15         b ← f(pointsCounter)
16     end
17     minDistance ← d(x, c_{b.a})
18     d ← 0
19     for j ← 1 to k do
20         if s_{j,b.a} ≥ 2 * minDistance or j == b.a
             then
21             continue
22         end
23         d ← d(x, c_j)
24         if d < minDistance then
25             minDistance ← d
26             b.a ← j
27         end
28     end
29     if iteration > 1 then
30         pointsCounter ← pointsCounter + 1
31     end
32     write b to a BF on HDFS
33     output(b.a, x)
```

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} ← d(c_i, c_j) //for all 1 ≤ i, j ≤ k
Function map(offset, value)
    x ← value
    //initialise all values in t
    for j ← 1 to k do
        t_j ← false
    end
    minDistance ← ∞
    for j ← 1 to k do
        if t_j then
            continue
        end
        d ← d(x, c_j)
        if d < minDistance then
            minDistance ← d
            a ← j
            for z ← j + 1 to k do
                if s_{j,z} ≥ 2 * d then
                    t_z ← 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 ($28 \times 28 = 784$) 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		8	15
DS2		32	28
DS3		128	235
DS4	100,000	512	941
DS5		1024	1884
DS6		2048	3788
DS7		512	947
DS8	1,000,000		1638
DS9	3,000,000		3584
DS10	5,000,000	128	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

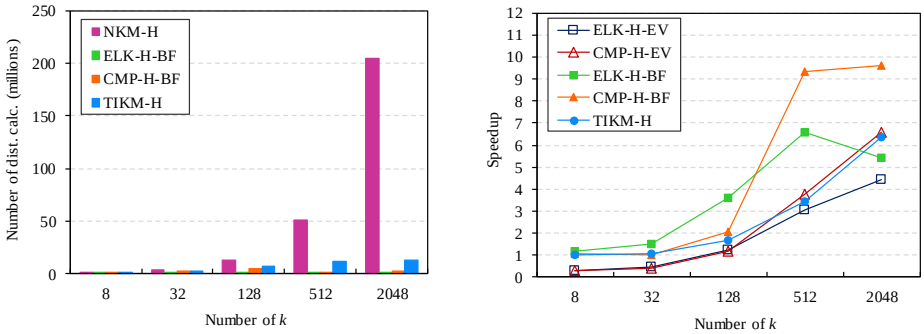
This section aims to investigate the impact of the number of clusters (k) and the number of dimensions (d) on the performance of ELK-H-BF, CMP-H-BF, and TIKM-H compared with NKM-H. The values of k and d varies from small, medium and large where $8 \leq k \leq 2048$ and $8 \leq d \leq 512$. The number 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.

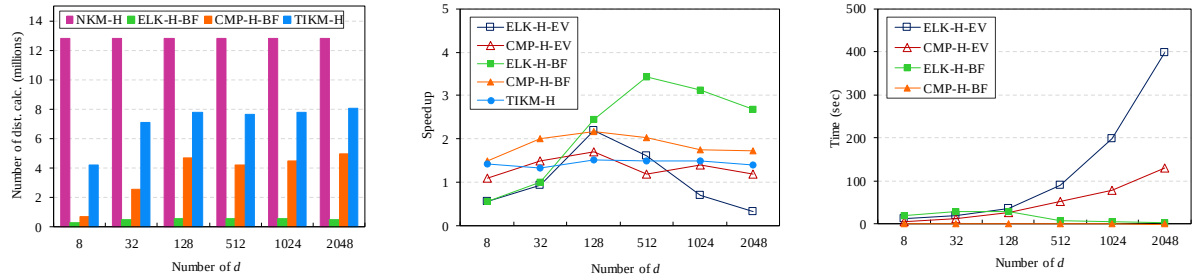
For small numbers of clusters and dimensions where $8 \leq k, d \leq 32$, no significant improvements in speed are reported for all the optimised algorithms. This is because even though the optimised algorithms eliminate some distance computations, the time that NKM-H spends on distance computations 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



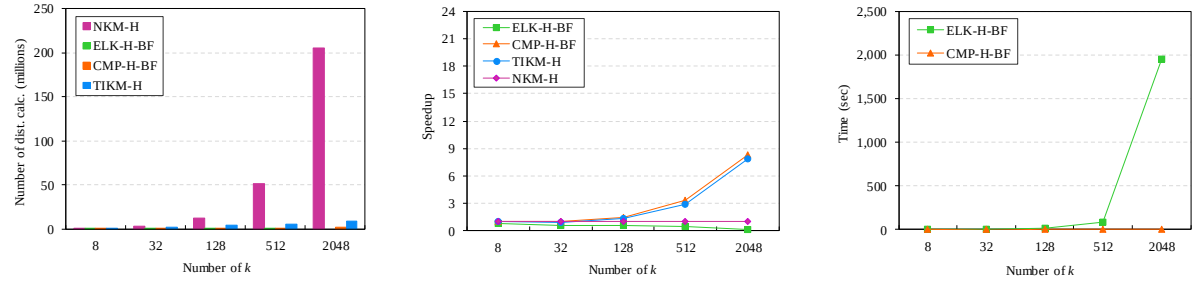
(a) Average number of distance calculations (b) Speedup relative to NKM-H

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).

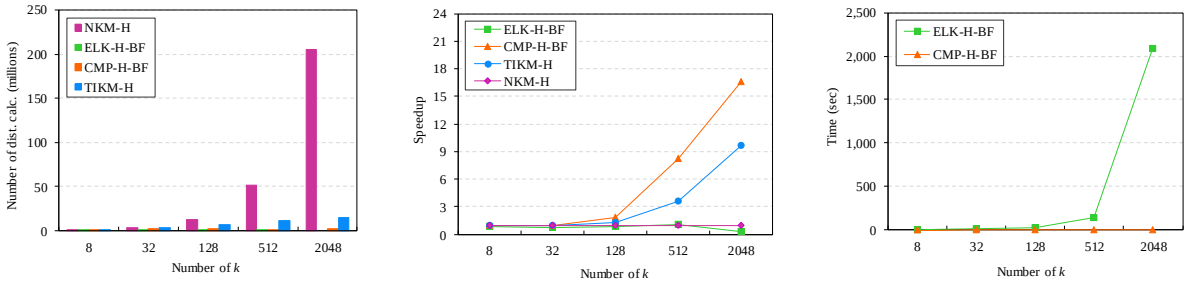


(a) Avg. no. of distance calculations (b) Speedup relative to NKM-H (c) Avg. time to write EVs and BFs

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, $d = 8$)

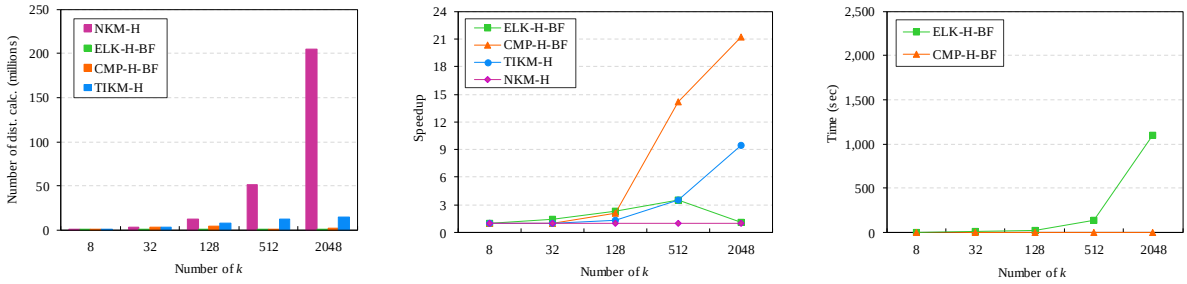
computations. This is reflected on the speedup where ELK-H-BF was 2.8x times faster than NKM-H when the number of clusters are in the range of $128 \leq k \leq 512$.

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.



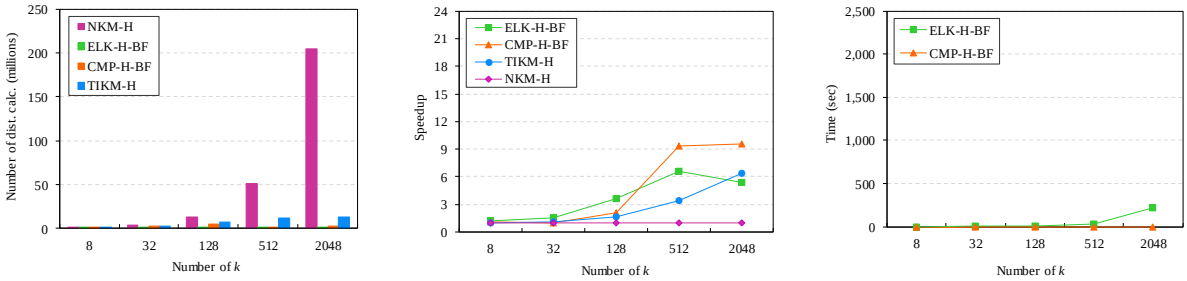
(a) Avg. no. of distance calculations (b) Speedup relative to NKM-H (c) Avg. time to write EVs and BFs

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, $d = 128$)



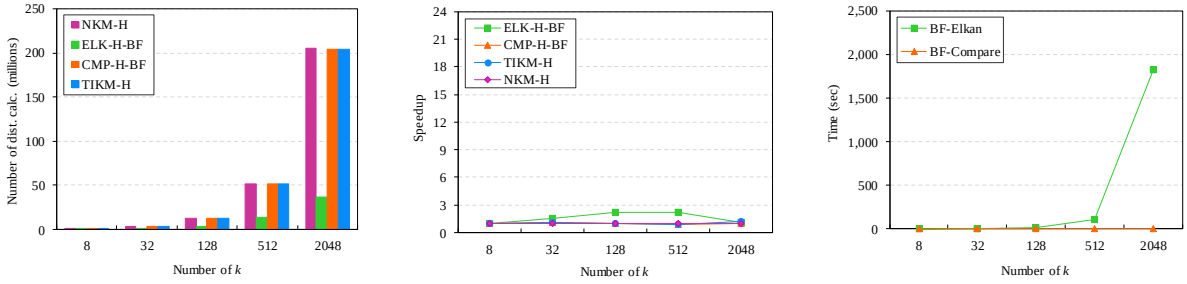
(a) Avg. no. of distance calculations (b) Speedup relative to NKM-H (c) Avg. time to write EVs and BFs

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

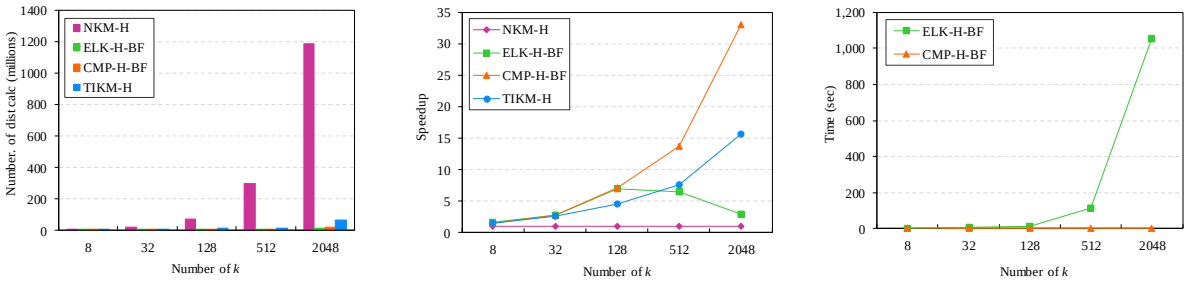
This section aims to test the performance of each algorithm against a variable number of data points (n), where $1,000,000 \leq n \leq 9,000,000$. Each algorithm is tested against five clustered datasets, DS[8-12], each with a variable number of data points and constant number of clusters $k = 128$, 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.



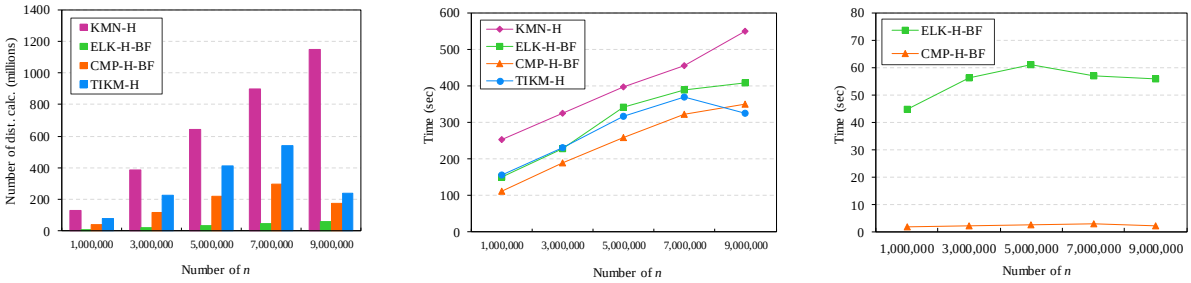
(a) Avg. no. of distance calculations (b) Speedup relative to NKM-H (c) Avg. time to write EVs and BFs

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

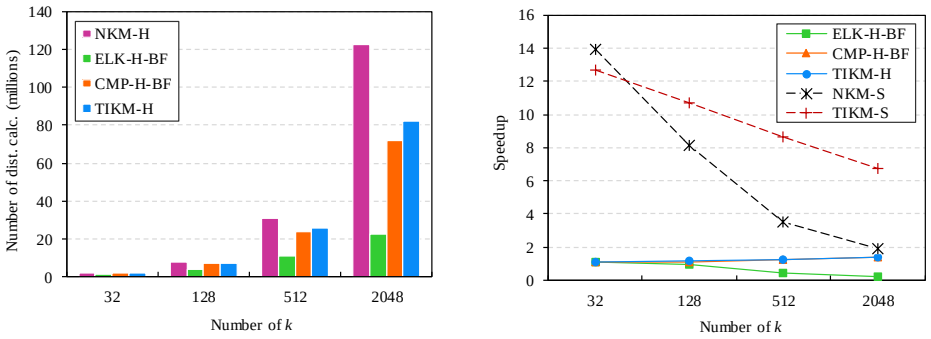
Figure 10: Results of tests on variable number of points (n). Average distance computations per iteration (Figure 10(a)), average iteration time (Figure 10(b)), and average time to write EVs and BFs (Figure 10(c)). (Dataset: DS[8-12], $d = 128$, $k = 128$)

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.