2019 IEEE 21st International Conference on High Performance Computing and Communications; IEEE 17th International Conference on Smart City; IEEE 5th International Conference on Data Science and Systems

K-means parallel acceleration for sparse data dimensions on Flink

1st Zihao Zeng
dept. College of Computer Science and Electronic Engineering
Hunan University
Changsha, China

2nd Kenli Li
dept. College of Computer Science and Electronic Engineering
Hunan University
Changsha, China

3rd Mingxing Duan
dept. College of Computer Science and Electronic Engineering
Hunan University
Changsha, China

4th Chubo Liu
dept. College of Computer Science and Electronic Engineering
Hunan University
Changsha, China

5th Xiangke Liao
dept. College of Computer
National University of Defense Technology
Changsha, China

***Abstract*—The K-means algorithm is a clustering algorithm which widely used in various applications, and it's running time is dramatically increased as the data size expanded. When the volume of data exceeds the range that can be carried by a single machine, the parallel operation of the algorithm must be implemented by using a distributed computing framework. Generally, during the parallel operation of the task, there are differences among the running time of each task due to the data skew, and the running progress of the entire job is determined by the task with the longest running time. In this paper, we propose an optimal data partitioning method for the application of the k-means algorithm on the sparsely dimensioned dataset to eliminate the data skew problem and further accelerate the parallel execution of the algorithm. Experimental evaluation on large-scale text datasets demonstrate the effectiveness of our partitioning approach on Flink.**

***Keywords*—K-means; Flink; Sparse vector; Data skew**

I. INTRODUCTION

With the rapid development of the Internet, massive data is produced and collected in various industries; such as social media, economics, medical and transportation. These data are mostly unstructured, such as audio data, text data, images, etc., which can be explored to obtain more valuable information.

Cluster analysis, a task of divides a set of data into different clusters, so that data with similar features are partitioned into one cluster, and only the smallest similarity remains among the clusters [1][2]. In [3], lots of clustering algorithms have been developed. Because of simple implementation and ideal clustering effect, the popularity of k-means algorithm increasing in the cluster analysis. It is at top 10 data mining algorithms [4] as well. The idea of k-means can be divided into two steps: first, k center points are randomly assigned, and each sample is assigned to the center nearest to it; Second, recalculate a new center point for each cluster. Repeating these two steps until the center points converges or computing reaches the maximum number of iterations. As we all know, with the continuous increasing of size of datasets and the extension of feature dimensions, the running time of the k-means algorithm based on CPU or GPU alone is unbearable. Therefore, it is a promising method to accelerate the K-means algorithm in parallel. The parallel k-means algorithm based on big distributed data processing platform such as Hadoop or Spark has also been studied by many researchers and has achieved better performance.

Hadoop is an open source MapReduce framework for distributed big data storage and processing [5][6][7]. Due to its high fault tolerance and scalability, it has been widely used in early big data processing issues. Xun *et al.* [8] proposed a parallel mining algorithm for frequent item-sets using MapReduce. Mahout [9] has implemented K-means parallelization algorithm on Hadoop MapReduce. Compared to Hadoop, Spark and Flink [10][11][24], a memory-based big data processing framework, has better performance with iterative machine learning algorithms. Since it caches both input data and intermediate data in memory without repeatedly reading and writing data on disk, which saves a huge amounts of disk I/O operation time. The reason why Flink is better than Spark is its automatic optimization function for iteration and efficient memory management.

Many researchers have focused on the fact that the data input by the k-means algorithm is a dense vector, and their methods will become relatively low efficient when most samples are represented as high-dimensional sparse vectors. A simple example, when the word's one-hot code is used as a word vector, the text is represented by the word frequency of the word in each text, there are often hundreds of thousands or even millions of words in a large text dataset where a large number of zero elements exist and they do not need to participate in calculations. Due to the different distribution of non-zero elements of each high-dimensional feature vector, Spark or Flink causes severe load imbalance when loading data and allocating tasks, resulting in poor performance.

Motivated by these, this paper focuses on the parallel implementation of k-means algorithm on Flink for samples with high-dimensional sparse vectors, and achieves load balancing by designing reasonable data partitions as a pre-processing step for k-means algorithm.

The remainder of this paper is organized as follows: Section II and III review related work and provide more background information. Section IV mainly describes parallel implementation details based on the Flink for the k-means algorithm. Our experimental design and results analysis are given in Section V. Section VI concludes this paper.

II. RELATED WORK

Cluster analysis has been an important research topic for data mining researchers for many years. In the past, under the condition of small data size, researchers mainly focused on the

978-1-7281-2058-4/19/$31.00 ©2019 IEEE
DOI 10.1109/HPCC/SmartCity/DSS.2019.00284

optimization problem of k-means algorithm itself. For example, the initial centroid is not randomly selected, but generated by an algorithm [13][14][15][16]. With the continuous extension of data scale and the development of big data technology, researchers have shifted their focus to the parallel implementation of k-means algorithm and large-scale data storage on distributed big data processing platforms such as Hadoop or Spark.

In the literature [25], Zhao *et al.* proposed a method of parallelization based on MapReduce for K-means algorithm. Bahman *et al.* [26] realized the parallelization of the initialization centroid steps in the K-means++. Kusuma *et al.* [17] proposed the intelligent K-means algorithm on Spark for big data clustering, and using batch of data instead using original Resilient Distributed Dataset which is the abstraction of datasets on Spark. Li *et al.* [18] discussed the parallel version of GPU-based K-means algorithm and adopted shared memory and registers to avoid multiple accesses of global memory for high-dimensional data. V.Santhi and Rini Jose [19] analyzed the performance of the Bat and Firefly optimization algorithms which were used to obtain the initial centroids and executed the modified K-means algorithm in Apache Spark environment.

Different from the work above, some researchers focus on the k-means clustering algorithm which is designed for the hardware layer and trying to break through existing memory constraints. Li *et al.* [20] proposed a novel method of three-level data partition strategy and run it on Sunway TaihuLight supercomputer, one of the world's fastest supercomputers. Their approach partitions the dataset size n, the number of clusters centroids k and the feature vector dimension d on different hardware levels hierarchically, so that the total value of k∗d is no longer limited by the size of memory. Their work achieved performance of less than 18 seconds per iteration for a large-scale clustering case in parallel.

In contrast to the mentioned work, we focus on proposing an efficient approach that can implement the k-means algorithm in parallel in an in-memory computing architecture on Flink for a large-scale dataset with high-dimensional sparse vectors.

III. Background

A. Flink

Apache Flink is an open source computing platform for distributed stream and batch data processing, mainly implemented in Java. Similar to Spark, it is also based on memory computing, while retaining the fault tolerance and scalability of MapReduce. Flink has been designed to run in all common cluster environments and provides two abstract models of DataSet and DataStream for distributed data, which can be represented by object in Java or Scala. In general, Flink runtime mainly contains two types of processes: JobManagers and TaskManagers. Fig.1 shows the architecture of Flink. The JobManagers's role is a master-side, which is responsible for applying resources, and managing and distributing task from user by dividing it into several subtasks for each nodes. It will manage subtasks and distributed it into TaskMangers. TaskManagers (also called workers) is at slave-side whose main function is to receive and execute the tasks (or more specifically, the subtasks) sent by the JobManagers, and communicate with the JobManagers to feedback the tasks status information. To control how many tasks a TaskManager accepts, a TaskManager has so called task slots (at least one). Each task slot allocates the memory managed by the TaskManager evenly, and the number of task slots also indicates the highest parallelism of the current task.

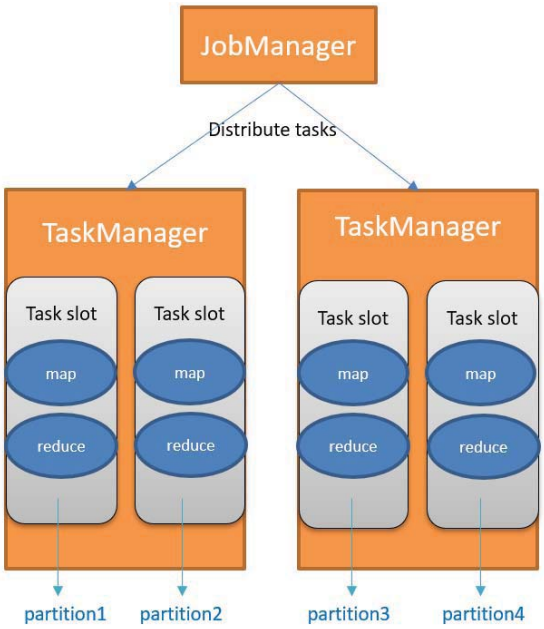

Fig. 1. Flink architecture

There are several parallel transformation and action operations which can be performed on DataSet, and different parallelism can be set for each operation. They are encapsulated as a series of high-level interfaces (e.g., Map, FlatMap, Filter, Reduce, Join), enabling programmers to work easily. Each data item in the DataSet is represented as a key-value pair, and the kernel calculation process in Flink is still based on the MapReduce calculation model, mainly including the Map and Reduce stages. In the Map stage, the input key-value pairs generate a series of key-value pairs based on user-defined tasks through a Mapper. Reduce is the data aggregation stage, which takes the key-value pairs output from the Map stage as input. According to the user-defined tasks, data with the same conditions is entered into the same Reducer and the final result of the job is produced.

B. K-means

K-means is an unsupervised learning clustering algorithm. Its main idea is rather simple: For a sample dataset in a given European space, the sample dataset is divided into k different clusters, according to the distance between the samples. Let the points in the cluster to be as close as possible, while the distance between the clusters as large as possible. To be formalized, given n samples $X = \{x_1, x_2, \dots, x_n\}$, where x_i is a d-dimensional vector, $i \in \{1, 2, \dots, n\}$, assuming a set of centroids $C = \{c_1, c_2, \dots, c_k\}$, the goal of the algorithm is to minimize the squared error *Cost(C)*:

$$\mathrm{Cost(C)} = \sum_{i=1}^{n} \|x_i - clu(x_i)\|_2^2$$

where $clu(x_i)$ is the closest centroid to x_i:

$$clu(x_i) = argmin_{c_j \in \{c_1, \dots, c_k\}} \left(\|x_i - c_j\|_2^2 \right)$$

It is not easy to find the least square error of the above formula. This is an NP-hard problem [21], so only heuristic iterative methods can be used [22]. K-means is mainly an iterative process of the following two stages:

$$(1)\quad clu(x_i) = argmin_{c_j \in \{c_1,\dots,c_k\}} \left(\|x_i - c_j\|_2^2\right)$$

$$(2)\quad c_j = \frac{\sum_{arg\ clu(x_i)=c_j} x_i}{|arg\ clu(x_i) = c_j|}$$

After random initialization of the centroid set, the algorithm can be compressed into these two steps. First, for each sample, its distance from each centroid is calculated, then, assign it to the centroid closest to it. Second, for each centroid, the centroid is updated according to the mean vector of the assigned samples. The algorithm keeps iterating these two steps until the changes of centroids are less than a given threshold or the maximum number of iterations is reached.

IV. The Presented Method

In this section, a data partitioning method for the sparse sample vector dimension problem will be introduced. This method would achieve better load balancing, so that the calculation amount of each compute node in cluster will be roughly the same. Then, an explanation of the parallel implementation process of the algorithm based on Flink. The overall process is shown in Figure 2.

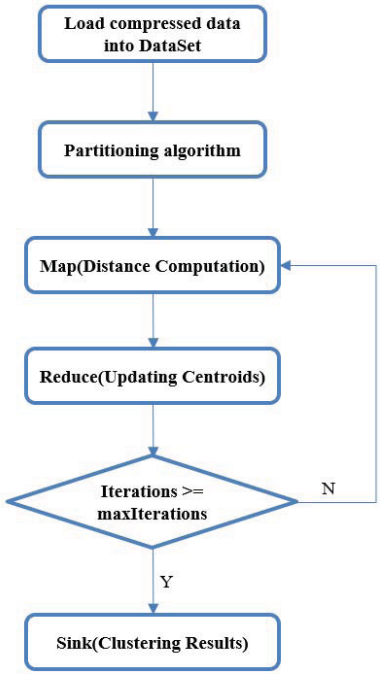

Fig. 2. Algorithm execution flow

A. Data partition

1) Problem Definition

Apparently, the calculation of distance is the most time-consuming part of the k-means algorithm in the first step, which requires distance of that each point to every centroid. For a given number of samples *n*, the number of centroids *k*, and the sample vector dimension *d*, the time complexity of this process is $O(nkd)$. However, the measurement of similarity between samples is not limited to the only Euclidean distance method, and the cosine similarity between the sample vectors can be used instead:

$$\text{cosDistance}(x_i, c_j) = 1 - \frac{\vec{x_i} \cdot \vec{c_j}}{\|x_i\|_2 \cdot \|c_j\|_2}$$

After the normalization of features, the modulus of the vector can be directly ignored when calculation of the cosine distance of the sample. At this time, the cosine distance is equal to the Euclidean distance; also, the two samples with a small cosine distance also have a small Euclidean distance.

When the sample dimensions are sparse, each sample data can be compressed and stored into two arrays $Ndata = \{val_1, val_2, \dots, val_h\}$ $Ncol = \{col_1, col_2, \dots, col_h\}$, where val_i represents the value of the i-th non-zero element of the current sample, col_i represents the index of the i-th non-zero element in the original data for all $i \in \{1, 2, \dots, h\}$, h is the number of non-zero elements of the current sample, and h is far less than the true dimension of the sample, so that the calculation is focused on the non-zero element. Flink's default data partitioning rule is to divide the data into different partitions (each data partition corresponds to a task slot) by taking the hash code of the key of each data to the total number of partitions. It has the possibility of generating a hash collision, which means that different input data items may produce the same hash code through the hash function. Although the Flink programming interface provides the *rebalance* operator to sequentially assign each piece of data to different partitions in a round-robin manner, the data skew caused by the difference in the length of each piece of data cannot be solved. Since the k-means algorithm requires all samples to be assigned a nearest centroid before performing centroid update, this will result in a waiting delay, with a total of non-zero elements having a large number of data blocks taking a long time, while the data blocks with less zero elements is less time consuming. That is, in a Flink cluster environment, tasks that take a short time need to wait for a long-term task before the Reduce phase. In the case where all blocks are computed in parallel, the time overhead of the first step of the k-means algorithm will be the computation time of the data block containing the most non-zero elements. As the number of iterations increases, this extra time overhead will multiply.

For the sample dataset **A**, each sample is not separable. Therefore, we consider set A as an ordered set of sparse vectors, where $A = \{r_1, r_2, \dots, r_n\}$, r_i represents the compressed form of sparse vector for $i \in \{1, 2, \dots, n\}$. The partition defining the sample dataset **A** is set $\beta = \{A_1, A_2, \dots, A_m\}$, where $A_i \in A$ and $A_i \cap A_j = \emptyset$ for all $i \neq j$ and $i, j \in \{1, 2, \dots, m\}$, meanwhile, $A_1 \cup A_2 \cup \dots \cup A_m = A$. The time overhead of each block A_i participating in the distance calculation is proportional to the sum of the non-zero elements of all sparse vectors in its block. Our goal is to find an optimal partition $\beta = \{A_1, A_2, \dots, A_m\}$ of A, which minimizes the total time overhead for parallel computing of individual data blocks. Parallel time overhead $\text{Time}(\beta)$ is as follows:

$$\text{Time}(\beta) = max_{i \in \{1,2,\dots,m\}}(Num(A_i))$$

where $Num(A_i)$ is the total number of non-zero elements of block A_i. The average number of non-zero elements is defined as:

$$Avg(A) = \frac{Num(A)}{m}$$

2) An Optimal Partitioning Method

Since the individual samples are existing independently, the order in which we exchange the sample vectors in dataset **A** does not affect the calculation results. First, we sort the set **A** in the ascending order by the number of non-zero elements in the sample vector. At this time, $A = \{r'_1, r'_2, \dots, r'_n\}$, and $Num(r'_i) \le Num(r'_{i+1})$ for all $i \in \{1, 2, \dots, n-1\}$.

Theoretically, the optimal partitioning strategy should remain that the number of non-zero elements in each block in the same as $Avg(A)$. Otherwise, if there are data blocks with the number of non-zero elements below the average, there must be some data blocks with non-zero elements above the average, such partitions are not optimal.

Noticeably, the larger the number of partitions, the smaller the amount of computation carried by each block after the optimal partition. However, the number of partitions represents the number of compute nodes in the cluster; as a finite resource that cannot be increasing without limitation. Therefore, we only consider the optimal partition of dataset **A** in the case where the number of partitions is fixed. Unfortunately, the sample vectors are the smallest processing unit, so there is no guarantee that each data block will contain exactly the $Avg(A)$ non-zero elements.

The partitioning method we proposed is divided into two stages. In the first stage, we divide the dataset into 2m partitions with similar numbers of elements; in the second stage, we optimize the partition we obtained from the first stage through aggregation.

The first step we took is based on a greedy strategy to obtain the optimal partition, traversing the entire set **A** in order and making a selection of cutting point positions to find the optimal partition. First, we select the first cut point and divide the set into two parts, and use the first part as a partition, and the other part treats it as a subproblem that needs to be divided into $(m-1)$ partitions. Then we solve the subproblem and use the same strategy to decompose down. Obviously, the choice of cutting point position is very critical. Therefore, each choice we make is a heuristic-based local optimal choice. This is based on the assumption that in the case where the subproblem can reach the optimal solution, and the subproblem can be combined with the greedy choice to obtain the optimal solution of the original problem.

Specifically, we traverse the vector set while counting the number of non-zero elements. When the number of statistics reaches $Avg(A)$, we consider whether to use the current position as the cut point. After making the selection, we update this average which based on the number of non-zero elements in the remaining unpartitioned sample vectors in the set and continue traversing backwards. Each sample vector is marked with the initial partition number at the end of the traversal. *Algorithm 1* shows the initial partition details.

The sample data has been grouped into 2m partitions $\beta = \{A_1, A_2, \dots, A_{2m}\}$ by *Algorithm 1*, but there are still small value errors for the number of non-zero elements in different partitions. We continue to sort set β according to the number of non-zero elements in each partition, where $\beta = \{A'_1, A'_2, \dots, A'_{2m}\}$ and $Num(A'_i) \le Num(A'_{i+1})$ for all $i \in \{1, 2, \dots, 2m-1\}$, finally by aggregating the first partition and the last partition into a new partition, and aggregating the second partition and the second last partition into another partition, keeping the way we aggregated and so on and so forth, thus then we could have produce the final partition.

Algorithm 1: Data partition based on greedy strategy

```
// remainNum: the sum of the unpartitioned vector lengths;
// tempSum: Temporary variable of statistical vector length;
// avgNum: the average length of unpartitioned vectors;
// partitionId: the partition number;
Input: Array of sample vector lengths vLen[n].
Output: Array of partition numbers vPart[n].
1.  Initialize the number of partitions m;
2.  sort(vLen);
3.  remainNum ← sum(vLen);
4.  tempSum ← 0;
5.  avgNum ← remainNum / m;
6.  parttionId ← 1;
7.  for i ← 1 to n do
8.      vPart[i] ← partitionId;
9.      if (tempSum + vLen[i]) < avgNum then
10.         tempSum ← tempSum + vLen[i];
11.     else
12.         remainNum ← remainNum – tempSum;
13.         if remainNum / (m-1) ≥ (tempSum + vLen[i]) then
14.             tempSum ← 0;
15.             remainNum ← remainNum – vLen[i];
16.         else
17.             vPart[i] ← vPart[i] + 1;
18.             tempSum ← vLen[i];
19.         end if
20.         partitionId ← partitionId + 1;
21.         m ← m – 1;
22.         avgNum ← remainNum / m;
23.     end if
24. end for
```

B. Parallel implementation of K-means algorithm for sparse data on Flink

First, we read the compressed sample vector from HDFS and load it into the DataSet object. At this point, the data has been partitioned by Flink by default. However, each operator of Flink is calculated in units of partitions, which does not have the operability of a single piece of data. We need to export the DataSet as a *List*, where each element represents the length of each sample vector, and each element has a unique mapping to each piece of data in the DataSet. Second, we run the algorithm above on the exported *List* to get the partition number corresponding to each element and collect each element to generate a new DataSet object. Finally, we combine the two DataSet objects with a *join* operator and use the *partitionByCustom* function to repartition the data according to the partition number.

The process of running the k-means algorithm on Flink after data partitioning is roughly the same as that of other distributed frameworks. By randomly selecting the initial centroid and broadcasting it to each node, and defining a *map* function to calculate the distance from the sample to the

centroid, and defining the *reduce* function to update the centroid. Here, the processing time of each *map* subtask is the same, so that the algorithm is accelerated when running in parallel at this stage.

V. Experimental Results

A. Datasets and baseline

In the experiment we used three datasets, and the number of samples and feature dimensions in each dataset are shown in Table I. Among them, *new3* and *la12* are large text datasets from *CLUTO* [12], and the third is the large movie review dataset used in the literature [23]. We normalize the word frequency features to produce new text files for our experiments. Our experimental environment is a virtual machine environment created for Flink with one master node and two worker nodes. Each node has eight-core processor and 64G memory available.

Datasets	Samples	Features
La12	6279	31472
New3	9558	83487
Movie review	25000	89527

Table I: Experimental Datasets

We set the k-means algorithm for different iterations to compare the execution time differences of our partitioning strategy with Flink's default data partitioning method.

B. Results

From Figure 3, Figure 4 and Figure 5, we can see that after our data partitioning, the K-means algorithm execution time is always faster than the execution time of the algorithm under Flink default data partition. When the number of iterations increases, the gap of execution time becomes larger and larger, almost linearly increasing. At the same time, we changed the original data partition without affecting the accuracy of the k-means algorithm. Since our proposed partitioning method only accelerates the parallel execution of the first-stage distance calculation of the K-means algorithm, it does not affect the selection of the nearest centroid for each sample and the subsequent update centroid.

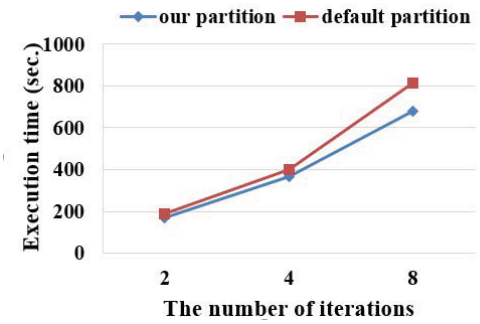

Fig. 3. Movie review dataset

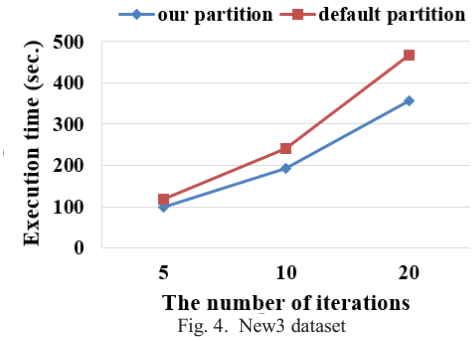

Fig. 4. New3 dataset

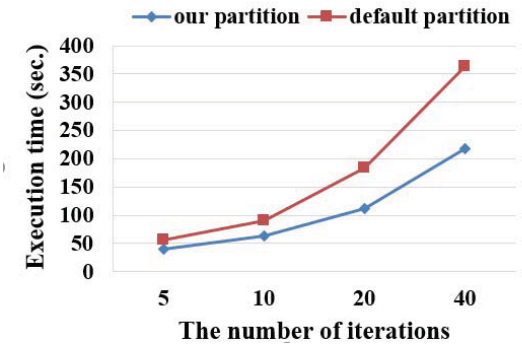

Fig. 5. La12 dataset

VI. Conclusion

In this paper, for the case where the sample data is a high-dimensional sparse vector, and the task waiting delay problem that may exist when the k-means algorithm is directly loaded, we design a data partitioning method based on greedy strategy as the preprocessing of k-means algorithm. The process enables each task to achieve load balancing, and finally minimizing the total execution time. The effectiveness of our method is demonstrated by experimental comparison which used three large-scale text datasets.

References

[1] A. K. Jain and R. C. Dubes, Algorithms for Clustering Data, Prentice-Hall, 1988.

[2] P.-N. Tan, M. Steinbach, and V. Kumar, Introduction to Data Mining, Addison-Wesley Companion Book Site 2006.

[3] P.-N. Tan, M. Steinbach, and V. Kumar, Introduction to Data Mining. Addison-Wesley, 2005.

[4] 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 Information Systems, vol. 14, pp. 1-37, 2008.

[5] Verma, Chitresh, and Rajiv Pandey. "Big Data representation for grade analysis through Hadoop framework." 2016 6th *International Conference-Cloud System and Big Data Engineering (Confluence)*. IEEE, 2016.

[6] Sivaraman, E., and R. Manickachezian. "High performance and fault tolerant distributed file system for big data storage and processing using hadoop." *Intelligent Computing Applications (ICICA)*, 2014 International Conference on. IEEE, 2014.

[7] Pal, Amrit, et al. "A Performance Analysis of MapReduce Task with Large Number of Files Dataset in Big Data Using Hadoop." *Communication Systems and Network Technologies (CSNT)*, 2014 Fourth International Conference on. IEEE, 2014.

[8] Y. Xun, J. Zhang, and X. Qin, "FiDoop: Parallel mining of frequent itemsets using mapreduce," *IEEE Trans*. Syst., Man, Cybern., Syst., vol. 46, no. 3, pp. 313–325, Mar. 2016.

[9] S. Owen, R. Anil, T. Dunning, and E. Friedman, *Mahout in action*. Manning Shelter Island, 2011.

[10] M. Zaharia, M. Chowdhury, and T. Das, “Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing,” in *Proceedings of USENIX conference on Networked Systems Design and Implementation*, 2012, pp. 2–2.

[11] (2018). *Flink Programming Guide*. Accessed on Dec. 21, 2018. [Online]. Available: http://flink.apache.org/

[12] G. Karypis, “Cluto–software for clustering high-dimensional datasets, version 2.1.1,” Oct. 2007. [Online]. Available: http://glaros.dtc.umn.edu/gkhome/views/cluto.

[13] D. Arthur and S. Vassilvitskii. k-means++: The advantages of careful seeding. In SODA, pages 1027–1035, 2007.

[14] MIRKIN, B. .“Clustering for Data Mining: A Data Recovery Approach”. 2005. Boca Raton FL: Chapman and Hall/CRC

[15] Komarasamy, G., Wahi, A.: An optimized k-means clustering technique using bat algorithm. Eur. J. Sci. Res. 84, 263–273 (2012)

[16] Mathew, J., Vijayakumar, R.: Scalable parallel clustering approach for large data using parallel k means and firefly algorithms. In: *IEEE International Conference on High Performance Computing and Applications*, Bhubaneswar, India (2014). https://doi.org/10.1109/ICHPCA.2014.7045322.

[17] Kusuma, I., Ma’sum, M.A., Habibie, N., Jatmiko, W., Suhartanto, H.: In design of intelligent k-means based on spark for big data clustering. In: *IEEE International Workshop on Big Data and Information Security*, Jakarta, Indonesia (2016). https://doi.org/10.1109/IWBIS.2016.7872895.

[18] Li Y , Zhao K , Chu X , et al. Speeding up K-Means Algorithm by GPUs[C]// 2010 10th *IEEE International Conference on Computer and Information Technology*. IEEE, 2010.

[19] Santhi V., Jose R. (2018) Performance Analysis of Parallel K-Means with Optimization Algorithms for Clustering on Spark. In: Negi A., Bhatnagar R., Parida L. (eds) Distributed Computing and Internet Technology. ICDCIT 2018. Lecture Notes in Computer Science, vol 10722. Springer, Cham.

[20] L. Li, *et al*., "Large-Scale Hierarchical k-means for Heterogeneous Many-Core Supercomputers," in *2018 SC18: The International Conference for High Performance Computing, Networking, Storage, and Analysis (rtioSC)*, Dallas, TX, US, 2018 pp. 160-170.

[21] James Newling and Franc¸ois Fleuret. Nested mini-batch k-means. In Advances in *Neural Information Processing Systems*, pages 1352–1360, 2016.

[22] Stuart Lloyd. Least squares quantization in pcm. *IEEE transactions on information theory*, 28(2):129–137, 1982.

[23] Maas, Andrew L. , et al. "Learning Word Vectors for Sentiment Analysis." *Meeting of the Association for Computational Linguistics*: Human Language Technologies Association for Computational Linguistics, 2011.

[24] Chen C , Li K , Ouyang A , et al. GFlink: An In-Memory Computing Architecture on Heterogeneous CPU-GPU Clusters for Big Data[C]// 2016 45th *International Conference on Parallel Processing (ICPP)*. IEEE, 2016.

[25] Zhao W , Ma H , He Q . Parallel K-Means Clustering Based on MapReduce[C]// *Proceedings of the 1st International Conference on Cloud Computing*. Springer, Berlin, Heidelberg, 1970.

[26] Bahmani B , Moseley B , Vattani A , et al. Scalable K-Means++[J]. *Proceedings of the VLDB Endowment*, 2012, 5(7).