CMPT 815 Research Project

Performance Comparison of Big Data Benchmarks on Big Data Analytics Frameworks - Hadoop And Spark

Name: Jenia Afrin Jeba

Student ID:11259150

Submitted to: Dr. Derek Eager

Table of Contents

List of tables

List of figures

Chapter 1
Introduction

At this moment, the word “Big Data” has become a buzzword not only in industry but also in fields of academia. Now-a-days, data is produced from almost everywhere: starting from regular huge amount of users’ activities in social media to smart phones, from the regular commercial and business transactions to industrial productions, from wireless sensor controlling systems to engineering and scientific computing etc. [18][19].

In a survey of DOMO named as “the Data Never Sleeps 3.0” which was published in 2015 [18]-[20], the following statistics was mentioned in that post, that in every minute:

- 347,222 tweets were sent by Twitter users;
- 1,736,111 photos were liked by Instagram users;
- over 4,166,667 likes were posted by Facebook users;
- 51,000 apps were downloaded by Apple users;
- 110,040 calls were made by Skype users;

All these large statistics have introduced people to the latest hot topic of the world - the field of Big Data.

In today’s technology and business world of 21st century, data is the most valuable asset for the growth of the staircases of success [18]. Analysis of this huge volume of data is important for gaining insights and understanding patterns that can aid in producing strategic decisions required for any area. Therefore, the necessity for Big Data technologies and initiatives are expanding exponentially and every discipline is now concentrating in making use of Big Data technologies and analytics [18]- [20]. Vast amount of data sets is referred as Big Data which may be in structured or unstructured form. The method of analyzing bulky data sets to highlight the important patterns and insights is referred as Big Data analytics [21]-[23].

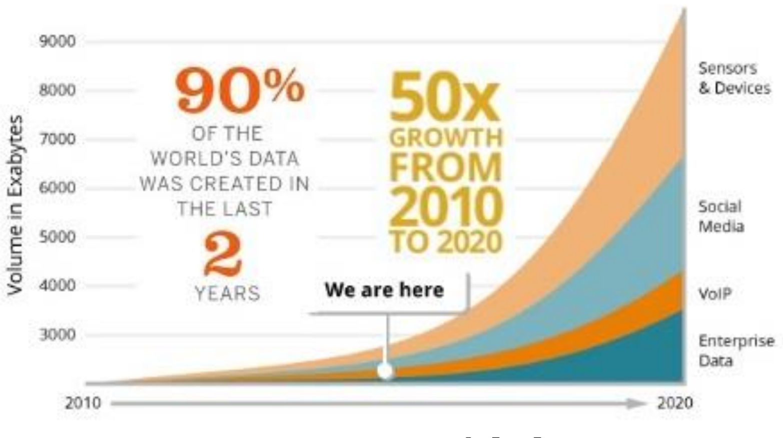


Fig 1: Data growth [18]

1.1 Background

The concept of resolving big data problem was originated from Google for the very first time. Google, the most extensively used search engine which gathers enormous quantity of data everyday, reckoned out with technologies – Hadoop and MapReduce for the management, storage and analysis of data [5][6] [12] [14].

In the field of big data analytics, Hadoop is now at the focus of a spreading network of big data technologies whose users include a wide variety of renowned companies and organizations such as - Google, Facebook, Twitter Amazon, EBay, Yahoo! IBM and LinkedIn [14]. Hadoop is an open source distributed processing framework that performs processing of large amount of data and storage for big data applications which execute in clustered systems. Hadoop can process several forms of unstructured, semi-structured and structured data, giving users more extensibility for accumulating data, analyzing and processing data in a better way than relational databases like – SQL. It provides support for data mining, machine learning, predictive analytics and advanced analytics initiatives etc. [13]-[16]. Hadoop performs all these operations with the help of HDFS (Hadoop Distributed File System) and MapReduce where HDFS offers a distributed storage and MapReduce offers the distributed processing for Hadoop [15] [16].

Nevertheless, Hadoop and MapReduce are not appropriate for executing all types of applications; they are not suitable for handling small files, not efficient for iterative processing, cannot manage the live data firmly, has comparatively slow processing speed, inefficient for caching etc. [16] [25]. To get the better of the drawbacks of Hadoop, an alternative framework has come into the scene which is new in-memory runtime systems called Spark [17][21]. Spark executes works almost in the same way as Hadoop does, but the basic difference is that the calculations are performed in memory (RAM), so it performs faster than Hadoop. But again, the performance of Spark is to some extent limited when it is used with the distributed cluster manager YARN because it causes RAM overhead memory leaks [24] [27].

So, whether to choose Hadoop or Spark? Both of them are the most leading distributed systems for big data processing and analytics. Primarily, for disk-heavy processes, Hadoop is used with the MapReduce and Spark is used for in-memory processing architecture. Both applications are frequently used together, both have their own strengths and weaknesses, both have similarities and dissimilarities, therefore it’s imperative to know the structures and characteristics when deciding to execute applications on them.

In computing, a benchmark is defined as the action of executing a computer program / a set of programs for evaluating the comparative performance of an object, usually by executing some specific standard experiments and trials against it [10] [22]. There are several functional-level and application-level big data benchmarks such as - WordCount, PageRank, Sort, TeraSort, K-means etc. [21] [22]. In this project, I want to run these benchmarks on Spark and Hadoop distributed systems for analyzing various characteristics of these two major big data analytic platforms and accordingly measure and compare the difference in their performances such as - the maximum and average CPU performance and usage of memory, the execution time, the throughput rate, the throughput/node rate etc. with different sizes of input datasets.

My objectives will include studying associated research works to explore the inspirations, studying and understanding the existing benchmarks, the conduct of execution of the applications on disk-based data analytic framework Hadoop and in-memory data analytics framework Spark, running the benchmarks on them and then generate comparison graphs of their performances.

Hadoop and Spark both have their own strengths and weaknesses, so they will certainly provide different performance responses when different benchmarks with different input data sets of varying sizes are tested on them. The WordCount, Sort, and TeraSort are yielded as instances in Hadoop library files and the Spark versions are provided by HiBench [4].

The WordCount function's task is to read an input text file to count the occurrences of each word in that file [4] [22]. The Terasort function comprises of three applications: Teragen for generating input data, TeraSort for sorting the input data, and TeraValidata for checking the output [4]. PageRank is used for ranking the website pages according to their importance. Every page is assigned a numerical value named 'PageRank' that characterizes the significance of that particular webpage. It states that a page gets more significance if it is inter-linked with more pages with greater PageRank values [11].

Machine learning lets the computer to form models on the basis of data. There is an algorithm termed as Naive Bayes Classifier which is used for categorizing text and it has two phases: training and testing. In the former phase, the classifier is taught by any sample text file and then develop a model. In the latter phase, based on the model, the classifier processes the input data set that is given to it for testing [4] [22].

Exploring the comparison of performance of Hadoop and Spark has always been an interesting research area both in academia and industry [10] [17] [21] [22] [24] [25]. For each benchmark, each engine will perform differently when parameters are changed. So, from this project I'll be able to figure out which platform is providing better performance for each benchmark.

Hopefully, the resources that I'm and I'll be going through, will guide me to conduct my research and fulfill the aforesaid objectives eventually.

1.2 Overview of the work

The main work of my work includes:

• to set up Hadoop and Spark clusters and then run the benchmark applications on them, which are: web search benchmark PageRank, micro benchmark WordCount, benchmark TeraSort, and machine learning benchmark K-means.

• to measure the speedup, throughput, running time, CPU usage, maximum and average memory requirements for all these benchmarks for different sizes of input data sets on the two platforms.

• to compare the performance variances on the aforementioned aspects of these two platforms based on the runtime features of these benchmarks.

• to present comparison graphs of the experimental results in the following chapters and evaluate them for various benchmarks.

This report is organized as follows: Chapter 2 explains more detailed working mechanisms of Hadoop HDFS and Hadoop MapReduce. Chapter 3 explains more details on Spark's working mechanisms together with RDD. Chapter 4 demonstrates experimental designs and corresponding results. Chapter 5 includes the conclusion.

Chapter 2
Hadoop

The Apache Hadoop is an open-source software framework for big data solution which offers the distributed processing of huge input data sets across computer clusters with the help of specific programming model in an efficient, scalable and reliable way. In this chapter, the details of HDFS, and MapReduce which are the two core technologies of Hadoop are presented based on its authorized documentation [5] [6].

2.1 Hadoop Distributed File System (HDFS)

The Hadoop Distributed File System (HDFS), one of the core elements of Hadoop, is an expansible file system which offers distributed storage for large data-concentrated applications. It is built using inexpensive commodity hardware PCs which are designed to be fault-tolerant. There is a master server named as “NameNode” and numerous slave servers named as “DataNodes” in a HDFS cluster. The architecture of HDFS was taken from the official Hadoop website and is shown in Figure 2.1 [5] [7].

The main job of the NameNode is to manage the metadata which comprises of - names of the files and their corresponding locations in the distributed clusters, the number of replicas of each file and the respective permissions granted for the clients to access those files etc. The main job of a DataNode is to provide the storage for data [7] [10]. The idea used in HDFS is to split the data into small pieces named as “blocks” which are automatically replicated for providing fault tolerance and then store those blocks in DataNodes all over the distributed cluster. Though the default value of replication factor is set to 3, for optimization users can set the block size by their own too. The way of placing replicas is the crucial factor for ensuring reliability and performance of HDFS. According to the default architectural design, the first two replicas are always kept on the similar rack but on distinct DataNodes and the third replica is kept on another different rack [5].

If a client requests read/write from/to HDFS, it must first talk to the NameNode for getting the access permission and the locations of the files and then contact with the DataNodes [5]. The NameNode never stores the actual data, rather it acts as a manager of all the DataNodes; each DataNode periodically sends message (the message is called “heartbeat”) to the NameNode via which the NameNode detects that particular node is alive. The NameNode will stop sending I/O request if it doesn’t receive heartbeat from DataNode and will mark that node as a dead node. Then the NameNode will re-replicate the stored data of the dead DataNode on another node.

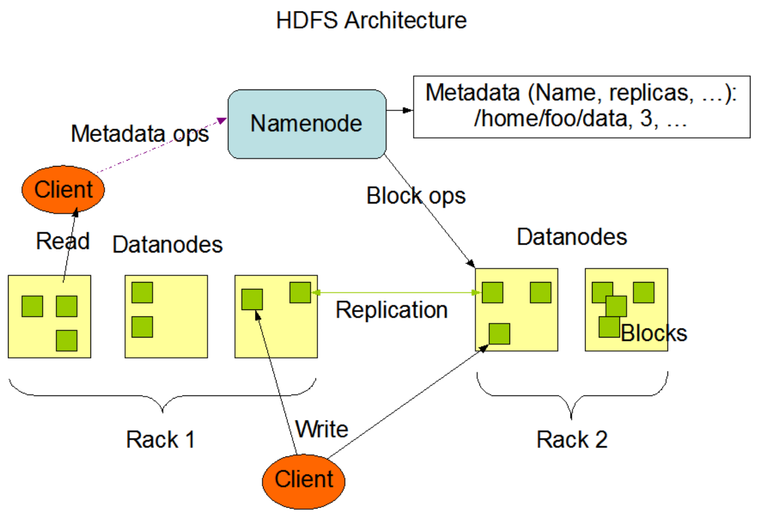


Fig 2.1: HDFS architecture [5]

2.2 MapReduce

MapReduce is the processing part of Hadoop which provides programming model for parallel and distributed processing of large amount of input datasets [14] – [16]. According to the working mechanism of MapReduce and Hadoop, the processing of data is performed at the location of the data itself rather than moving it to anywhere else which cuts down the data transfer cost.

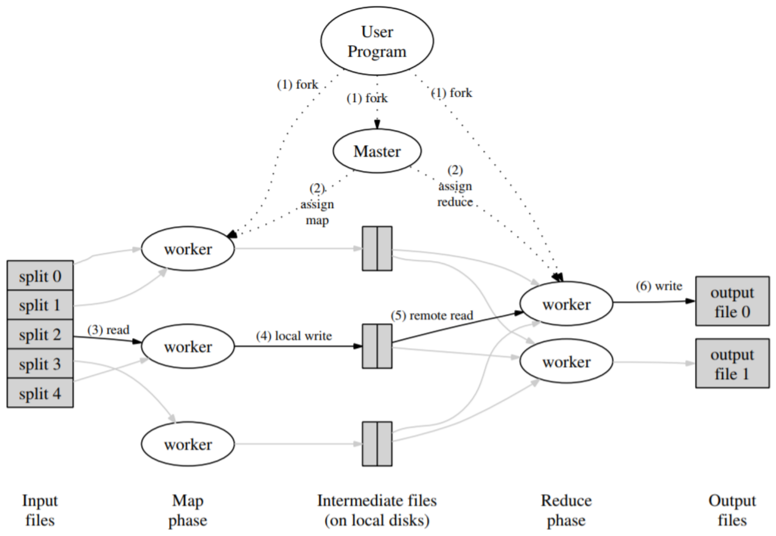


Fig 2.2: MapReduce architecture [13] [15]

As shown in fig 2.2, the cluster in MapReduce has also a master server namely “JobTracker” and several worker servers namely “TaskTracker” where the JobTracker splits a large incoming job into smaller tasks and assigns those computational tasks to the TaskTrackers. The whole process is divided into three phases: map the key-value pairs, shuffle them and then reduce [16] [28]. The Map function uses particular key-value pairs to map the input data and then group together the values which have the same key; the shuffle function simply sorts the results produced in the mapping stage and hands them over to the TaskTracker who will be reducing them into final results; the Reduce function then takes the intermediate values with the same keys and merges them together to produce a single result. Like HDFS, TaskTrackers also send heartbeats to the JobTracker for similar purpose [15] [16].

Chapter 3
Spark

Spark is another cluster computing based distributed open source framework for dealing with big data issues which has been designed to support iterative applications in a scalable and fault tolerant way. Spark uses in-memory computing which lets it execute programs up to almost 100x faster than Hadoop MapReduce [1] [2]. If spark is used with composition of both memory and disks, then it can run almost 10x faster. In Spark, the distributed shared memory system is known as Resilient Distributed Dataset (RDD), which supports the reprocess of data and intermediate results back in memory. Spark also provides APIs in Scala, Java, R shells and Python [2]. It can also run on YARN (Yet Another Resource Negotiator, which is a cluster resource manager) and can use data from HDFS. In this chapter, the details of Spark is presented based on its authorized documentation [1] [2].

3.1 Spark Components

The Figure 3.1 shows a brief overview about how Spark runs on distributed clusters. The main program is the driver program which synchronizes among all the applications and contains the SparkContext. For getting processing and memory resources for executing the programs, the SparkContext requires to link to a cluster manager. There are three cluster managers on which Spark can run: Standalone mode, Mesos mode and YARN mode. Resources are allocated to executors by cluster managers and applications run on the executors. SparkContext is responsible for sending tasks to executors [1] [2] [8].

3.1.1 Resilient Distributed Dataset (RDD)

The Resilient Distributed Dataset (RDD) is a collection of data items which are fragmented into partitions, can be stored in the memory of the slave nodes and segregated across the machines of the distributed cluster which can be run on parallelly. RDDs can be formed by any file in the existing in HDFS or parallelizing any Scala file in the driver program and then converting it. Spark uses lineage information to recover from node failures [8] [9]. When any partition is lost Spark rebuilds that from lineage without returning back to the checkpoint. RDDs automatically recover from node failures. When Spark starts a new job, the worker nodes read those data from HDFS and then cache it to main memory. RDDs reduce reading/writing from/to disks which actually speeds up the execution process. RDD is read-only files [17] [21].

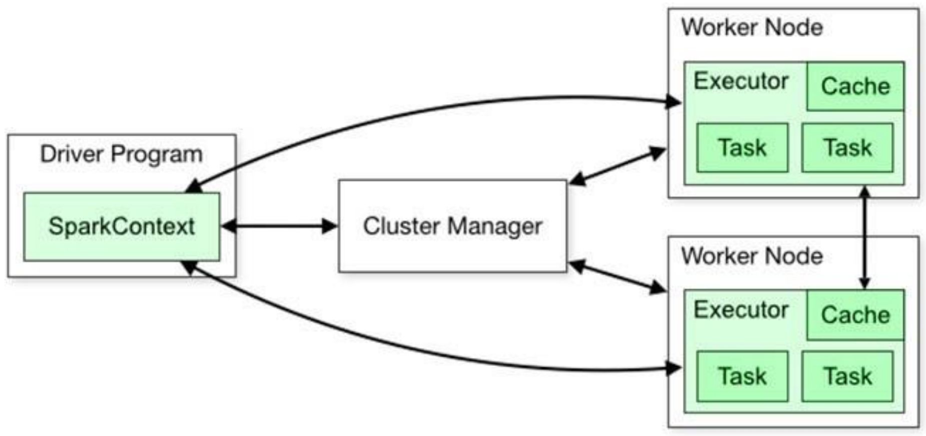


Fig 3.1: Spark architecture [1] [2]

When a spark application is submitted by a user, the driver program contacts with the cluster manager for negotiating the resources. On behalf of the driver, executors on the worker nodes are then launched by the cluster manager. Based on data placement, tasks are sent to the cluster manager by the driver. The tasks register themselves with the driver program before executors start execution, to facilitate the driver with a complete view regarding all the executors [1] [26]. Then the executors begin accomplishing the tasks which are allocated by the driver. The driver program will keep watching the set of executors during running the spark application. By tracing the position of cached data, data placement is decided and it lets the driver program in the spark architecture to plan for the forthcoming tasks. The driver program will abort all the executors and set free the resources from the cluster manager when its main () method exits or when the stop () method of the Spark Context is being called [2] [27].

Chapter 4
Experiment results and analysis

4.1 Setup of Cluster

The cluster that I set up for this experiment, was made up of seven computers. One of them served both as a master and a slave node and other machines served as slave nodes. The hardware and software configuration being used for the cluster is as follows:

- 7 nodes were interconnected by each 1Gbps-Ethernet
- Ubuntu 16.04 operating system and JAVA 1.7.0 version were used for all the nodes
- Each CPU consisted of 8 cores
- assigned 8GB RAM to each node
- used Hadoop 2.7.1, Spark 1.3.1 for running all the benchmarks
- used the Spark and Hadoop benchmarks and data generators presented in HiBench Benchmark Suite 7.0 version [4], (links are given in table 4.1)
  - ✓ The HiBench Benchmark Suite was downloaded from this web page: https://github.com/intel-hadoop/HiBench
  - ✓ Mapper number in Hadoop, partition number in Spark was 49 each
  - ✓ kept the block replication of HDFS to be 3 and the size of block to be 128MB; ran Spark on YARN with yarn-client mode, arranged the Spark configuration with the parameters given below:
    - 8 executors
    - 8 executor cores
    - 20G executor memory
    - 10G driver memory

Benchmark	**Platform**	**Source**
PageRank	Hadoop	HIBENCH HOME/src/pegasus/target/pegasus-2.0-SNAPSHOT.jar
PageRank	Spark	org.apache.spark.examples.SparkPageRank
WordCount	Hadoop	HADOOP HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.1.jar
WordCount	Spark	com.intel.sparkbench.wordcount.ScalaWordCount
TeraSort	Hadoop	HADOOP HOME/share/hadoop/mapreduce/hadoop-mapreduce-examples-2.7.1.jar
TeraSort	Spark	com.intel.sparkbench.terasort.ScalaTeraSort
K-means	Hadoop	MAHOUT HOME/bin/mahout kmeans
K-means	Spark	org.apache.spark.examples.mllib.DenseKMeans

Table 4.1: Benchmarks

4.2 Benchmarks

In the following segment, the four benchmarks which are used to compare the Hadoop and Spark performances are described. These benchmarks – PageRank, WordCount, TeraSort and K-means, are divided into three groups as presented in table 4.2. Intel provided these Hadoop and Spark workloads which are contained in HiBench suite and the data generator present in HiBench suite created all the input data files.

Category	Benchmark
Micro Benchmark	WordCount TeraSort
Machine Learning	K-means
Web Search	PageRank

Table 4.2: Categories of benchmarks used in the experiment

4.2.1 Web Search Benchmark

It was Google which used PageRank for ranking the website pages according to their significance factor for the very first time. For representing the importance of each page, a numerical value is assigned to each page which is called PageRank. While a page links to another page, it casts votes to that page. A webpage which has higher PageRank value, has more impact to vote other pages. Therefore, a page will get more votes if it is linked to more and more pages which have got larger PageRank scores [11].

4.2.2 Micro Benchmark

The WordCount and TeraSort are presented as samples in Hadoop library and HiBench provided the Spark versions. In the WordCount program the idea is to read any input text file and then calculate the total of how many times a particular word has occurred in that file. TeraSort, a recognized benchmark package on Hadoop which consists of three applications: Teragen, a MapReduce program which is used to create or generate input data (in Terabytes); TeraSort, to sort that input data and TeraValidata, to validate or check the output [4] [10].

4.2.3 Machine Learning Benchmark

Machine learning is a core branch of artificial intelligence which lets the computer / machine shape analytical models grounded on data, recognize significant patterns and create decisions with minimal human involvement [10].
There is an algorithm in machine learning named as K-means algorithm which is used for assembling similar objects into “k” clusters; user can decide the value of “k”. In this algorithm, initially, k data points for each cluster are randomly selected which are called “centroids” [17]. Afterwards, based on the distance to the nearest centroid’s location, the input data points form clusters. By calculating the average of all the points and passing on those points to clusters to find new centroids, the algorithm keeps computing new centroids for all clusters repeatedly until it combines all of the data [16].

4.3 Experimental Results and Analysis

In the following segment, the results and analysis of the experiment for executing the benchmarks described above, are presented. All these benchmarks have been run on Hadoop and Spark distributed systems. For comparing their performance, I've recorded the running time (in seconds), the CPU usage (in percentage %), the average and maximum memory usage (in percentage %), the throughput (bytes / seconds) and the throughput per node (bytes / seconds) on these two platforms. Five data sets for each (tiny, small, large, huge, gigantic) were chosen and ran.

4.3.1 Comparison between Spark and Hadoop

4.3.1.1 PageRank

The running time (in seconds) consumption (shown in table 4.3) and performance comparison for executing PageRank on Hadoop and Spark are represented in the following graph (fig 4.1).

The range of the input data sets being used here were from 4 million to 64 million pages and the sizes of the input were ranged from 1 GB to almost 19.9 GB. It is noticeable from the graph that when the input increased from 4 million to 16 million pages, Spark gave a better performance compared to Hadoop. But, as the input size continued to increase more than that (i.e. 32M and more), then Spark started to fall off behind Hadoop and took a running time of almost more than 2 times than Hadoop.

Input	**4M**	**8M**	**16M**	**32M**	**64M**
Hadoop	405.168 s	575.289 s	926.766 s	1691.601 s	4078.037 s
Spark	154.887 s	404.639 s	834.513 s	1965.088 s	9953.462 s

Table 4.3: The running time (in seconds) consumption comparison for executing PageRank on Hadoop and Spark

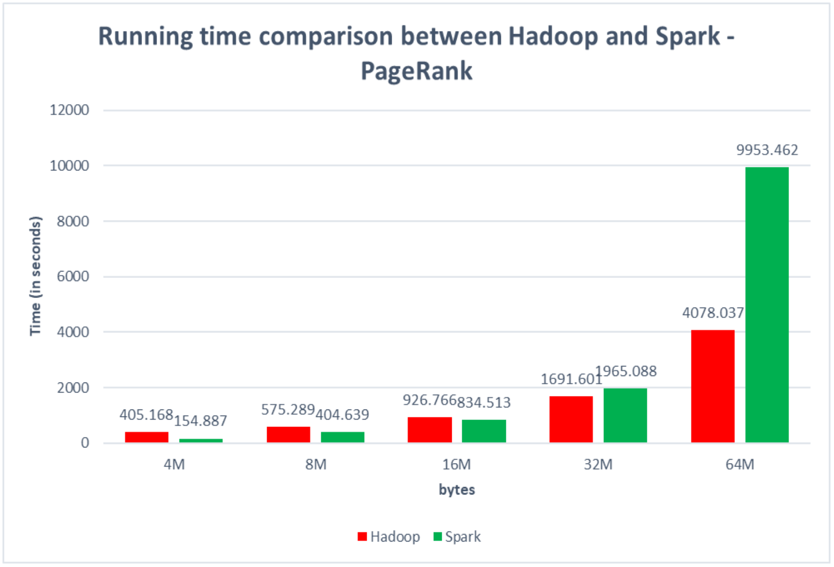


Fig 4.1: The running time (in seconds) performance comparison for executing PageRank on Hadoop and Spark

The reason behind it is that PageRank requires more memory resources because it runs with iterations. For iterative operations running on Spark, it is tough to pre-determine the exact amount of memory requirement. During each iteration, Spark stores the intermediate results in memory as newly created RDDs which are just read-only in nature. It wouldn't have caused a problem if the size of intermediate results remained at constant level. Because, then the amount of memory consumption between two serial iterations wouldn't been have that much significant. But the escalation of memory consumption between two sequential iterations becomes significant when the extent of the intermediate outcomes is proportionate to the extent of the input dataset (such as-PageRank).

So, since Spark uses in-memory calculation, it has a memory resource consumption of up to 2 times higher than that of Hadoop. Accordingly, after the input data size becomes too large, Spark isn't able to provide enough memory (RAM) for newly formed RDDs. Then Spark has to start replacement policy (replacing previous data to disk for providing space to new data) which not only starts effecting Spark's runtime performance but also increases the memory usage and CPU consumption as shown in fig 4.2 (a), (b) respectively. On the other hand, for Hadoop, the memory and CPU usage vary somewhat with varying input size. Comparatively, Spark saves CPU resources with small inputs than Hadoop.

Again, when the input size increases from 4MB to 16 MB, the throughput for Hadoop also increases but it decreases after 32 MB as shown in fig 4.3. The throughput for Spark decreases with the increasing input size.

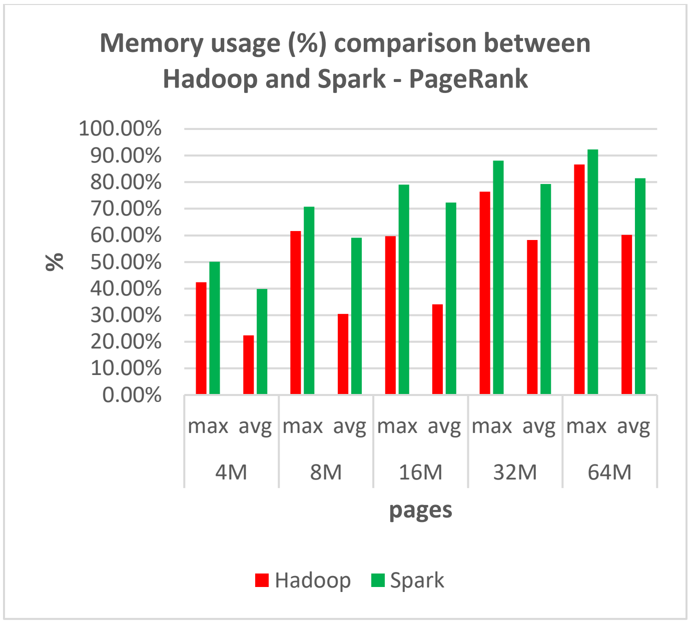
Input	**4M**		**8M**		**16M**		**32M**		**64M**	
	max	avg	max	avg	max	avg	max	avg	max	avg
Hadoop	42.34%	22.47%	61.67%	30.45%	59.67%	34.11%	76.46%	58.26%	86.68%	60.17%
Spark	50.12%	39.91%	70.78%	59.08%	79.03%	72.38%	88.15%	79.34%	92.25%	81.44%

(a)

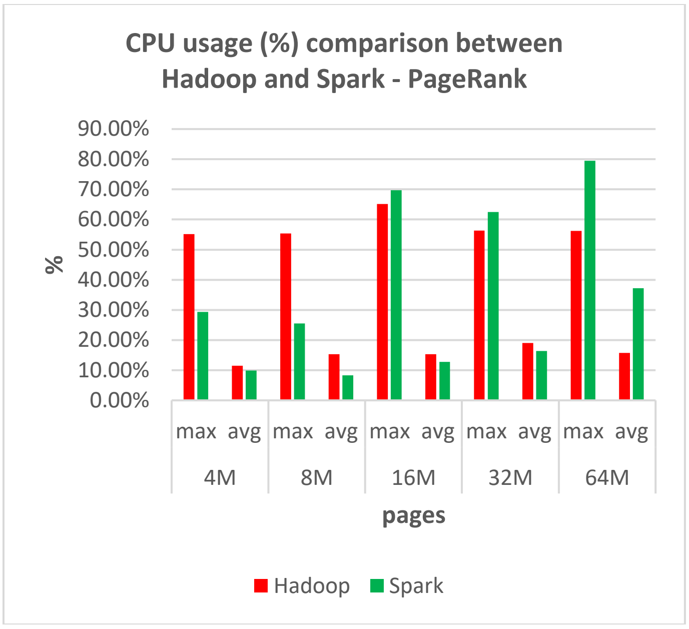
Input	**4M**		**8M**		**16M**		**32M**		**64M**	
	max	avg	max	avg	max	avg	max	avg	max	avg
Hadoop	55.18%	11.46%	55.38%	15.31%	65.09%	15.32%	56.37%	19.06%	56.27%	15.77%
Spark	29.37%	9.91%	25.47%	8.27%	69.72%	12.74%	62.44%	16.34%	79.48%	37.19%

(b)

Table 4.4: a) Maximum and average memory usage % and b) Maximum and average CPU usage % comparison for executing PageRank on Hadoop and Spark



(a)



(b)

Fig 4.2: a) Maximum and average memory usage % and b) Maximum and average CPU usage % comparison for executing PageRank on Hadoop and Spark

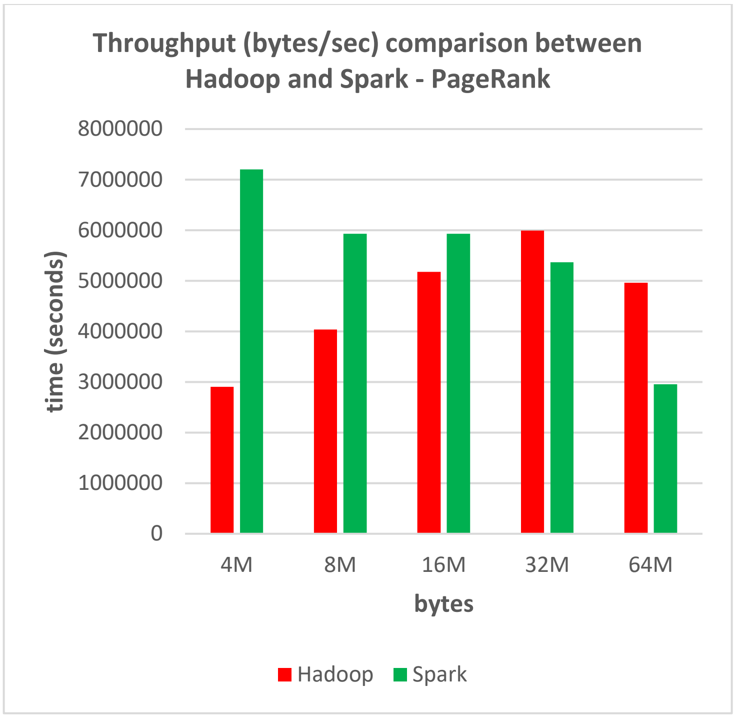
Input	**4M**	**8M**	**16M**	**32M**	**64M**
Hadoop	2902844 b/s	4035562 b/s	5174344 b/s	5994546 b/s	4962112 b/s
Spark	7201756 b/s	5932724 b/s	5932124 b/s	5369058 b/s	2957188 b/s

(a)

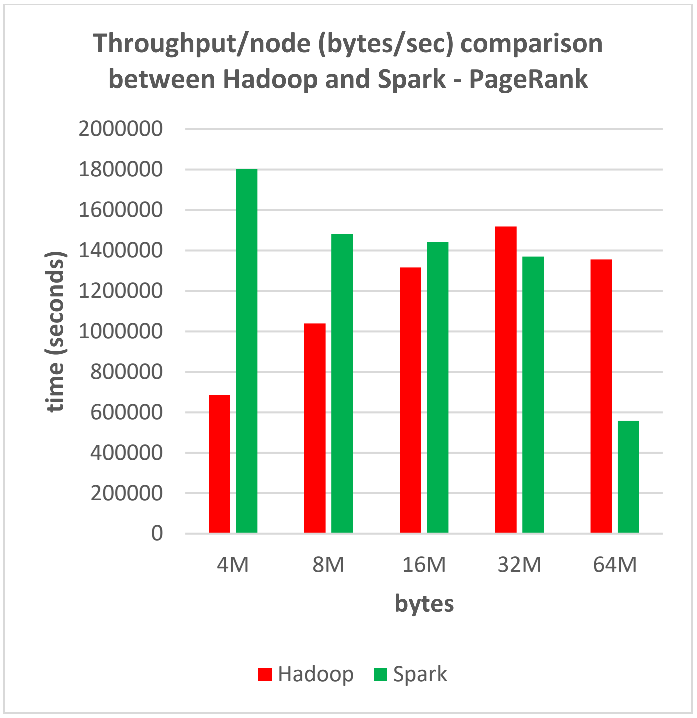
Input	**4M**	**8M**	**16M**	**32M**	**64M**
Hadoop	685382 b/s	1038822 b/s	1315590 b/s	1518732 b/s	1356494 b/s
Spark	1802734 b/s	1481630 b/s	1442548 b/s	1370024 b/s	557908 b/s

(b)

Table 4.5: (a) The throughput rate (bytes / seconds) comparison (b) The throughput / node (bytes / seconds) comparison for executing PageRank on Hadoop and Spark



(a)



(b)

Fig 4.3: (a) The throughput rate (bytes / seconds) comparison (b) The throughput / node (bytes / seconds) comparison for executing PageRank on Hadoop and Spark

4.3.1.2 WordCount

The running time (in seconds) consumption (shown in table 4.6) and performance comparison for executing WordCount on Hadoop and Spark are represented in the following graph (fig 4.4).

The range of the input data sets being used here were from 4 GB to 64 GB. It is perceptible from the graph that when the input size increases, the running time on both platforms also increases, that means the performance decreases. The WordCount benchmark executes without using iterations so the case isn’t same as PageRank where Spark used to fail behind Hadoop with very large inputs. Spark gave a better performance compared to Hadoop with a speed of up to 2 times more.
Both platforms require high memory consumptions for this benchmark and Spark consumed more than Hadoop, as shown in fig 4.5 (a). Yet, Spark has shown exceptional performance on conserving CPU resources compared to Hadoop; Hadoop consumed up to 7 times more CPU than Spark as shown in fig 4.5b. The reason behind it is that, in Hadoop, the key-value pairs being counted are written back to the disk storage whereas for Spark, they are just written back to the memory (RAM).
It is also visible from the graph 4.6, that with the varying input size, the throughput rate for Hadoop doesn’t vary that much. Despite the throughput for Spark decreases with the increasing amount of input size, it still gives better rate than Hadoop as shown in fig 4.6.

Input	**4G**	**8G**	**16G**	**32G**	**64G**
Hadoop	265.804 s	506.738 s	971.21 s	2023.44 s	3935.41 s
Spark	96.475 s	207.65 s	633.366 s	1273.112 s	2459.871 s

Table 4.6: The running time (in seconds) comparison for executing WordCount on Hadoop and Spark

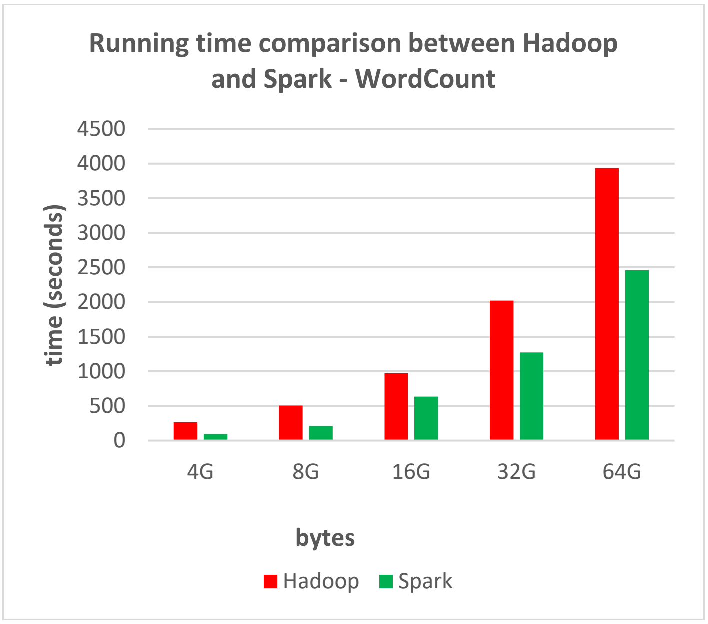


Fig 4.4: The running time (in seconds) comparison for executing WordCount on Hadoop and Spark

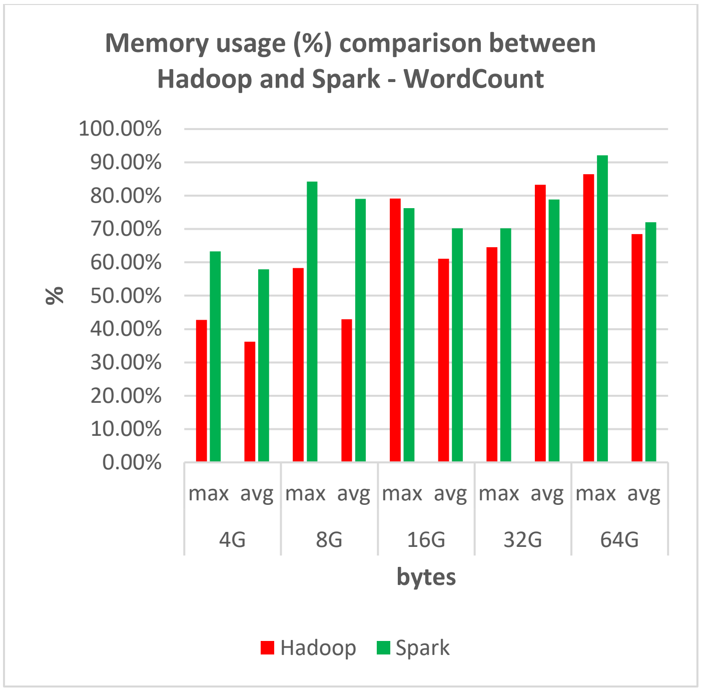
Input	**4G**		**8G**		**16G**		**32G**		**64G**	
	max	avg	max	avg	max	avg	max	avg	max	avg
Hadoop	42.75%	36.18%	58.31%	42.89%	79.11%	61.10%	64.55%	83.28%	86.43%	68.49%
Spark	63.27%	57.92%	84.22%	79.08%	76.29%	70.24%	70.18%	78.86%	92.17%	72.00%

(a)

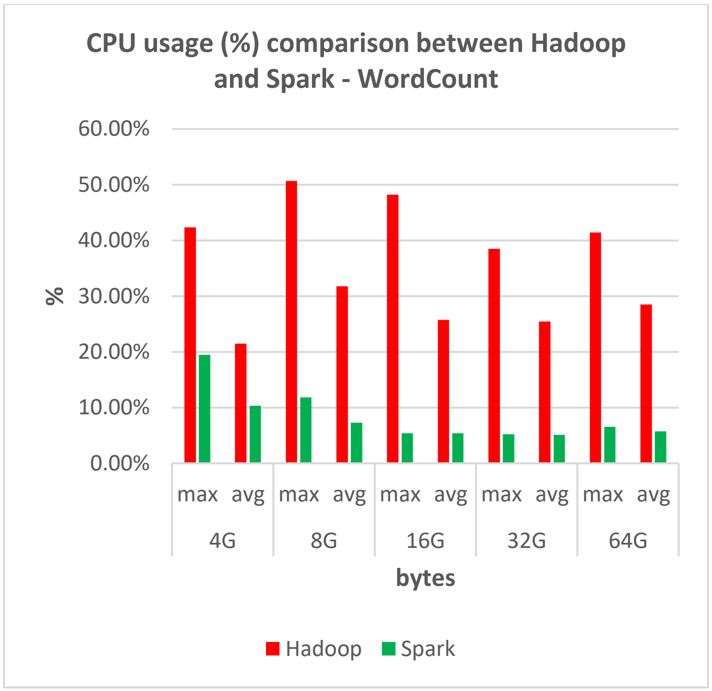
Input	**4G**		**8G**		**16G**		**32G**		**64G**	
	max	avg	max	avg	max	avg	max	avg	max	avg
Hadoop	42.35%	21.47%	50.67%	31.77%	48.19%	25.73%	38.48%	25.46%	41.43%	28.49%
Spark	19.48%	10.33%	11.81%	7.29%	5.38%	5.41%	5.23%	5.12%	6.55%	5.76%

(b)

Table 4.7: a) Maximum and average memory usage % and b) Maximum and average CPU usage % comparison for executing WordCount on Hadoop and Spark



(a)



(b)

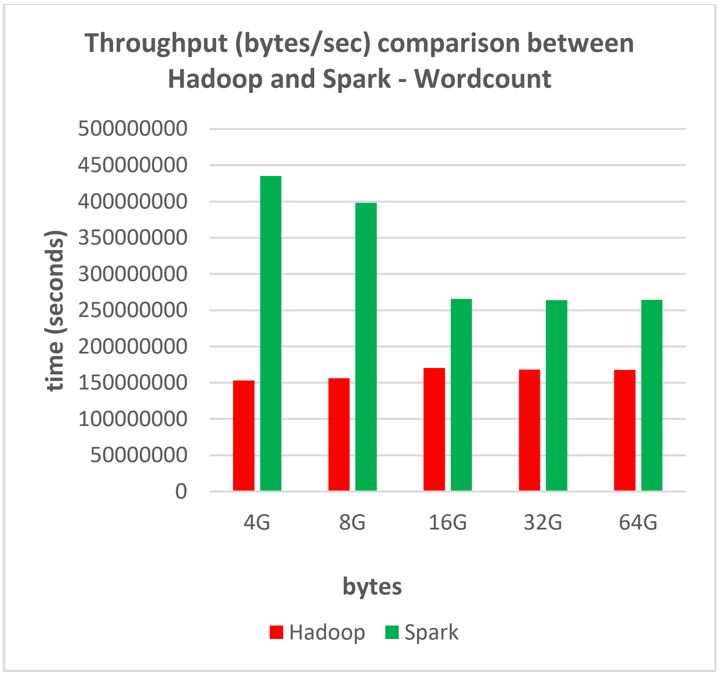
Fig 4.5: a) Maximum and average memory usage % and b) Maximum and average CPU usage % comparison for executing WordCount on Hadoop and Spark

Input	**4G**	**8G**	**16G**	**32G**	**64G**
Hadoop	152976444 b/s	156412030 b/s	170212474 b/s	168234452 b/s	167667208 b/s
Spark	435189006 b/s	398235676 b/s	265501920 b/s	264073180 b/s	264500982 b/s

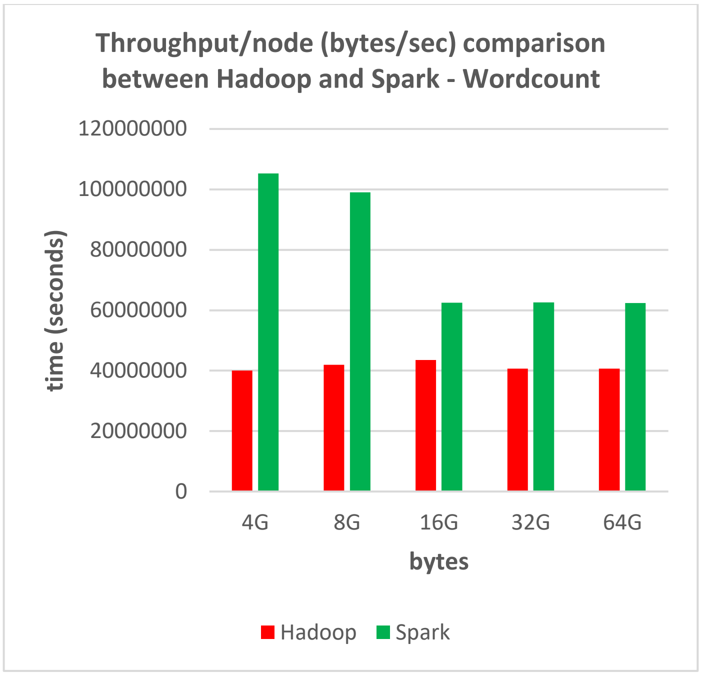
(a)

Input	**4G**	**8G**	**16G**	**32G**	**64G**
Hadoop	39986342 b/s	41919056 b/s	43501474 b/s	40688312 b/s	40695116 b/s
Spark	105235790 b/s	99005321 b/s	62456680 b/s	62553124 b/s	62434570 b/s

Table 4.8: (a) The throughput rate (bytes / seconds) comparison (b) The throughput / node (bytes / seconds) comparison for executing WordCount on Hadoop and Spark



(a)



(b)

Fig 4.6: (a) The throughput rate (bytes / seconds) comparison (b) The throughput / node (bytes / seconds) comparison for executing WordCount on Hadoop and Spark

4.3.1.3 TeraSort

The running time (in seconds) consumption (shown in table 4.9) and performance comparison for executing **TeraSort** on Hadoop and Spark are represented in the following graph (fig 4.7).
The range of the input data sets being used here were from 80 million to 1280 million records and the sizes of the input were ranged from 8 GB to almost 128 GB where the size of each record was 100 bytes. It is observable from the graph that when the input size small, then both Spark and Hadoop have almost the same performance, but Spark starts to perform faster when the input size crosses past 160 million records. Spark gives almost 2x faster running time when the input size is around 320 million records.
In purpose of memory utilization, both platforms consume almost the same amount (fig 4.8 (a)). In case of maximum CPU resource consumption, Spark performs better than Hadoop (fig 4.8 (b)). Hadoop consumes around 12x more maximum CPU resource while the input size reaches 128 GB records. Spark gives higher throughput rate than Hadoop when the input size becomes larger than 160 MB which is shown in fig 4.9.

Input	**80M**	**160M**	**320M**	**640M**	**1.28G**
Hadoop	85.098 s	157.431 s	507.112 s	1215.1022 s	3334.56 s
Spark	81.287 s	129.667 s	323.379 s	641.247 s	2990.344 s

Table 4.9: The running time (in seconds) comparison for executing TeraSort on Hadoop and Spark

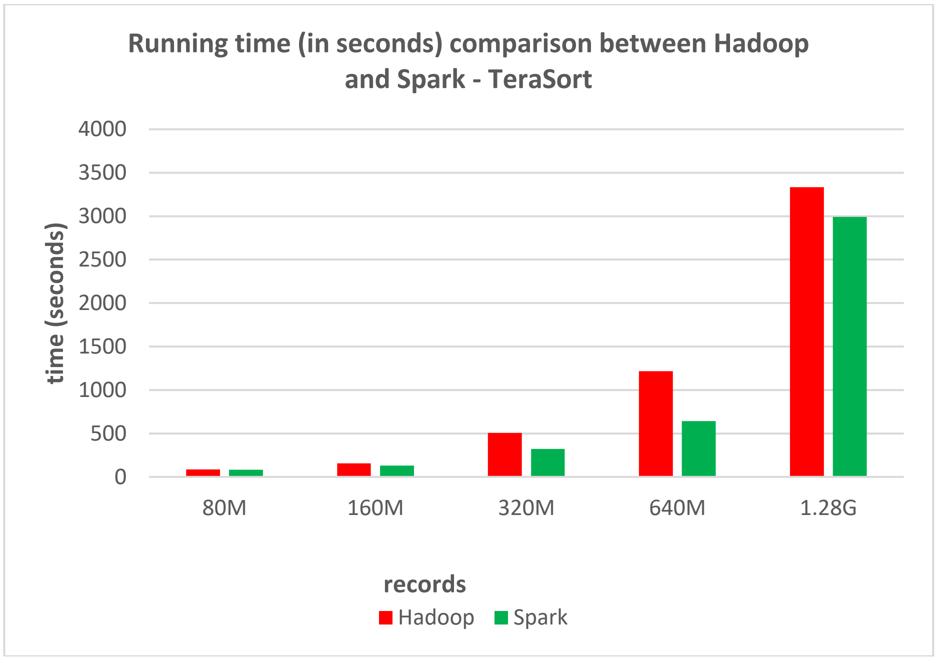


Fig 4.7: The running time (in seconds) comparison for executing TeraSort on Hadoop and Spark

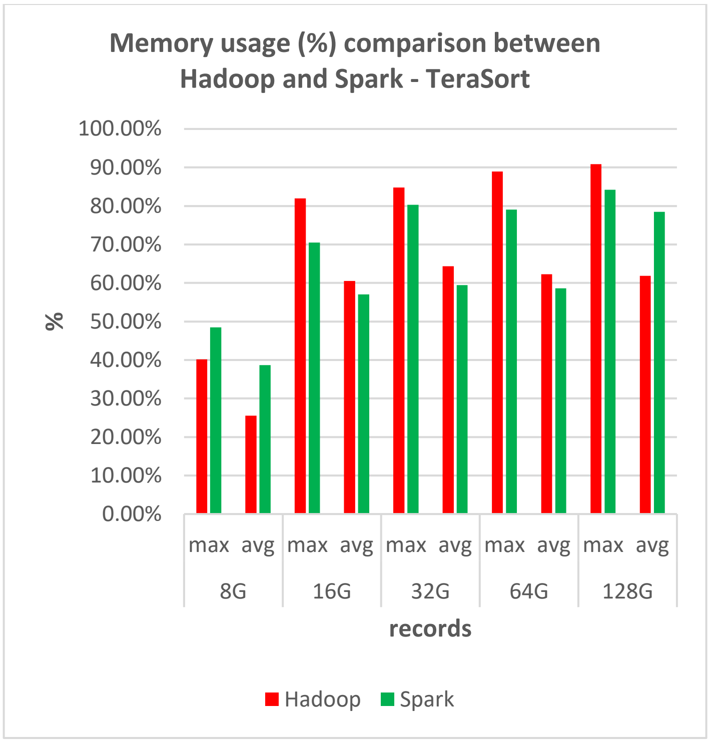
Input	**8G**		**16G**		**32G**		**64G**		**128G**	
	max	avg	max	avg	max	avg	max	avg	max	avg
Hadoop	40.21%	25.58%	81.93%	60.57%	84.77%	64.39%	88.93%	62.30%	90.84%	61.86%
Spark	48.47%	38.67%	70.48%	57.06%	80.29%	59.41%	79.04%	58.64%	84.25%	78.46%

(a)

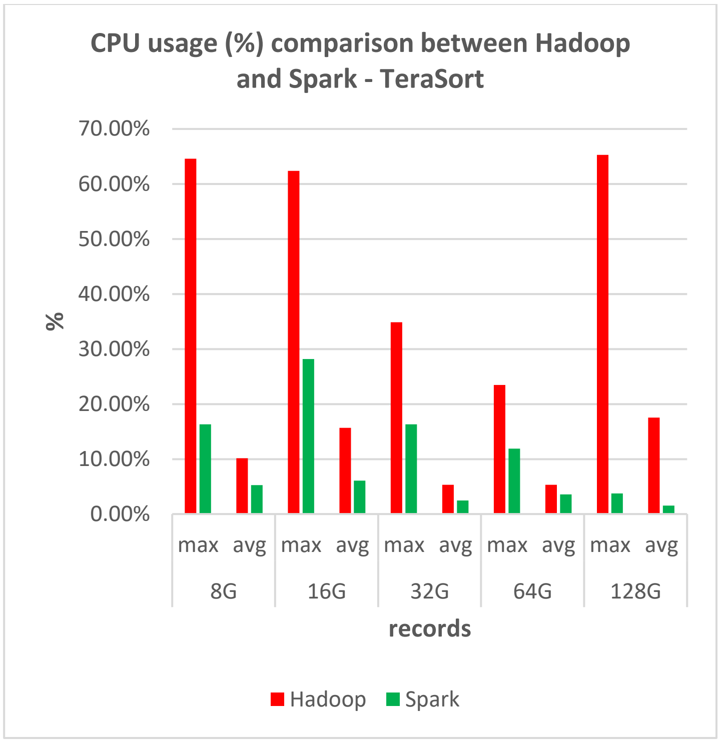
Input	**8G**		**16G**		**32G**		**64G**		**128G**	
	max	avg	max	avg	max	avg	max	avg	max	avg
Hadoop	64.58%	10.13%	62.37%	15.68%	34.86%	5.29%	23.47%	5.32%	65.27%	17.52%
Spark	16.29%	5.24%	28.17%	6.09%	16.29%	2.46%	11.87%	3.60%	3.76%	1.53%

(b)

Table 4.10: a) Maximum and average memory usage % and b) Maximum and average CPU usage % comparison for executing TeraSort on Hadoop and Spark



(a)



(b)

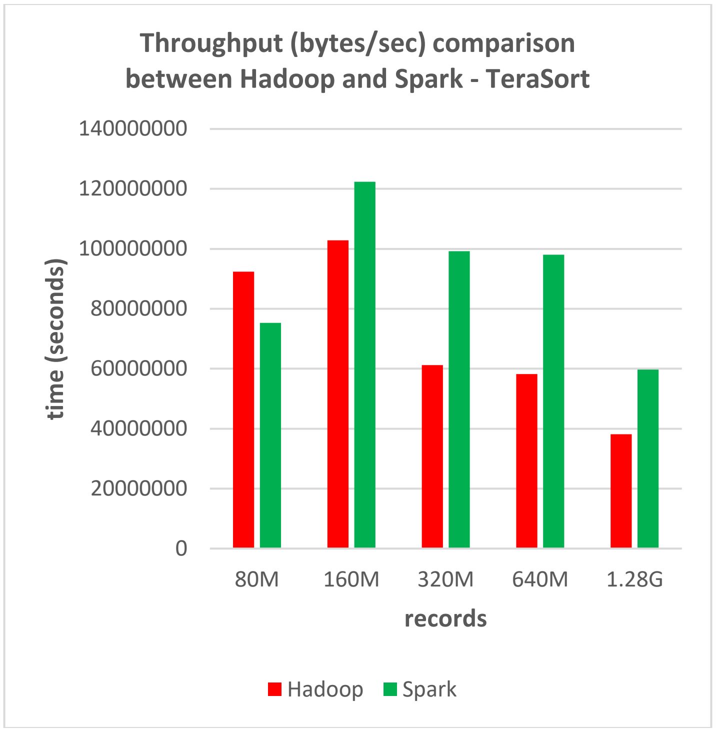
Fig 4.8: a) Maximum and average memory usage % and b) Maximum and average CPU usage % comparison for executing TeraSort on Hadoop and Spark

Input	**80M**	**160M**	**320M**	**640M**	**1.28G**
Hadoop	92347564 b/s	102893402 b/s	61189234 b/s	58199162 b/s	38192834 b/s
Spark	75290346 b/s	122390178 b/s	99236548 b/s	98046290 b/s	59667316 b/s

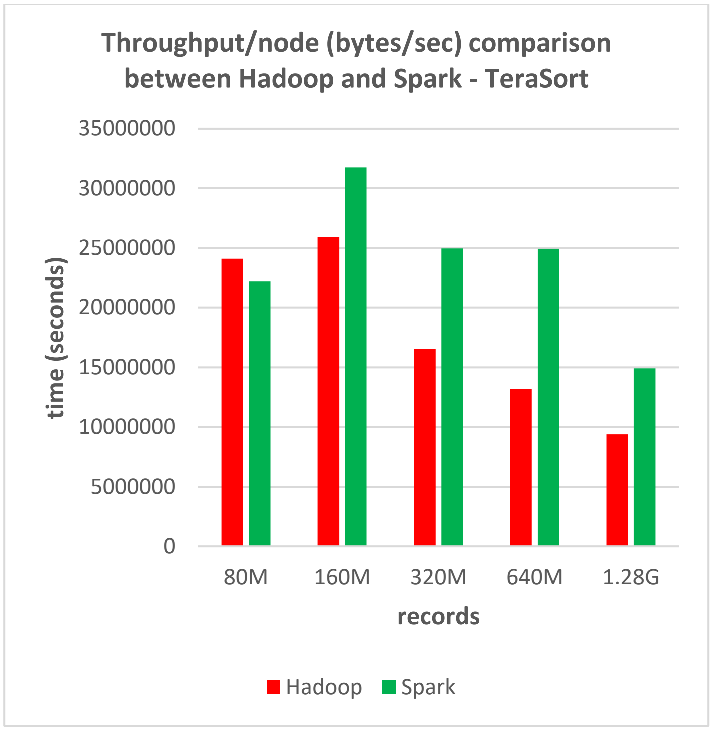
(a)

Input	**80M**	**160M**	**320M**	**640M**	**1.28G**
Hadoop	24112782 b/s	25911782 b/s	16513298 b/s	13182456 b/s	9387524 b/s
Spark	22200236 b/s	31765322 b/s	24973828 b/s	24941124 b/s	14919028 b/s

Table 4.11: (a) The throughput (bytes / seconds) comparison (b) The throughput / node (bytes / seconds) comparison for executing TeraSort on Hadoop and Spark



(a)



(b)

Fig 4.9: (a) The throughput (bytes / seconds) comparison (b) The throughput / node (bytes / seconds) comparison for executing TeraSort on Hadoop and Spark

4.3.1.4 K-means:

The running time (in seconds) consumption (shown in table 4.12) and performance comparison for executing **K-means** on Hadoop and Spark are represented in the following graph (fig 4.10).

K-means, a machine learning algorithm, where the input data points tend to form a cluster with the closest centroid available near them; the process is repeated again, these points allocated in the clusters create new centroids until it assembles. This algorithm is particularly appropriate for Spark rather than Hadoop because for Hadoop, every time these intermediate results have to be written back into disk for storage while Spark just stores the results into memory which saves time.

The range of the input data sets being used here were from 100k samples to 1.6 million samples and the sizes of the input were ranged from 10 GB to almost 160 GB. It is observable from the graph 4.10 that Spark gave far better performance than Hadoop with almost 6x faster speed.

Nevertheless, this benefit of performance for Spark is constrained by the memory (fig 4.11 (a)). After the input goes beyond 100 million samples the speedup of Spark which was 6x more than Hadoop, now becomes just almost 1.90 x faster than Hadoop.

Input	**50M**	**100M**	**200M**	**400M**	**800M**
Hadoop	333.284 s	858.454 s	1357.835 s	2755.113 s	6558.226 s
Spark	82.307 s	137.926 s	297.271 s	2322.208 s	2978.114 s

Table 4.12: The running time (in seconds) comparison for executing K-means on Hadoop and Spark

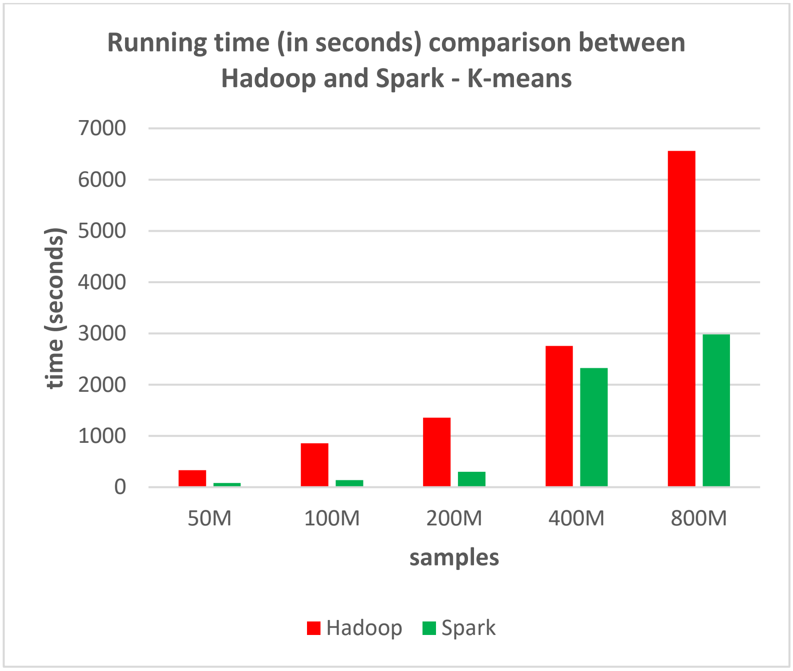


Fig 4.10: The running time (in seconds) comparison for executing K-means on Hadoop and Spark

In purpose of maximum memory utilization, for Spark, it is just the half of Hadoop when the input sizes are 50M to 100M. Spark consumes almost 100% of it during the input of 400M and beyond (fig 4.11 (a)) because at this point no newer RDDs can’t be created by Spark. Spark is better in saving more CPU resources than Hadoop when the input size is smaller (fig 4.11 (b)).
Spark gives higher throughput rate of up to almost 6x times than Hadoop when the input size is lesser than 200M samples which is shown in fig 4.12. When it goes beyond, the throughput rate for Spark drives down.

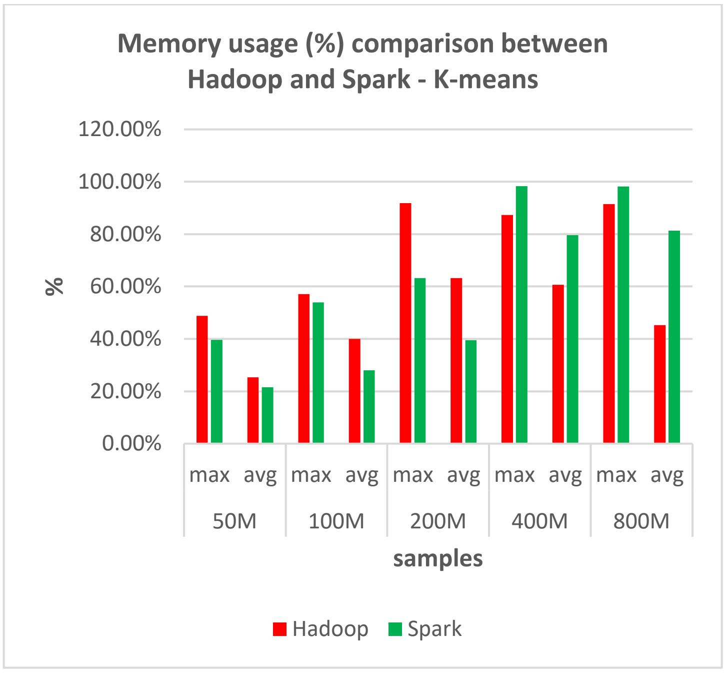
Input	**50M**		**100M**		**200M**		**400M**		**800M**	
	max	avg	max	avg	max	avg	max	avg	max	avg
Hadoop	48.75%	25.38%	57.11%	39.97%	91.83%	63.18%	87.38%	60.72%	91.48%	45.22%
Spark	39.66%	21.57%	53.94%	28.05%	63.24%	39.55%	98.27%	79.58%	98.26%	81.38%

(a)

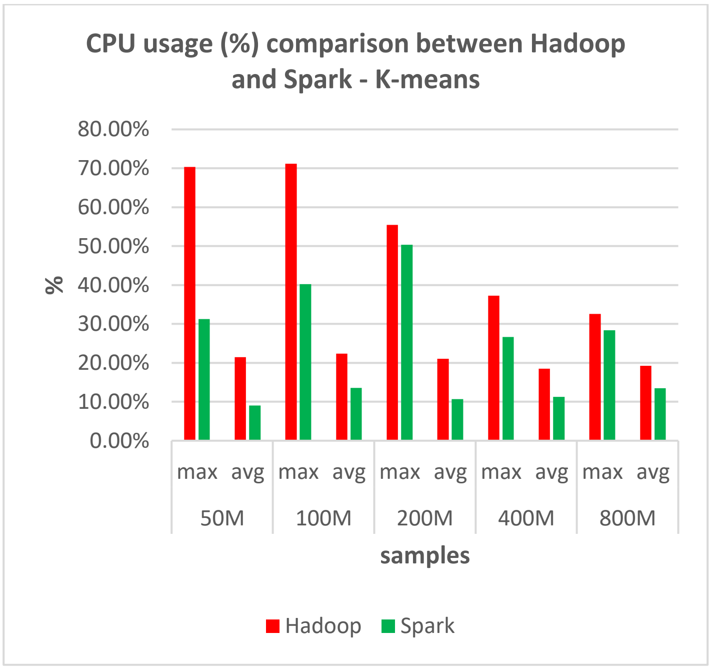
Input	50M		100M		200M		400M		800M	
	max	avg	max	avg	max	avg	max	avg	max	avg
Hadoop	70.35%	21.51%	71.19%	22.38%	55.48%	21.07%	37.24%	18.48%	32.56%	19.24%
Spark	31.24%	9.07%	40.21%	13.57%	50.39%	10.66%	26.69%	11.27%	28.38%	13.49%

(b)

Table 4.13: a) Maximum and average memory usage % and b) Maximum and average CPU usage % comparison for executing K-means on Hadoop and Spark



(a)



(b)

Fig 4.11: a) Maximum and average memory usage % and b) Maximum and average CPU usage % comparison for executing K-means on Hadoop and Spark

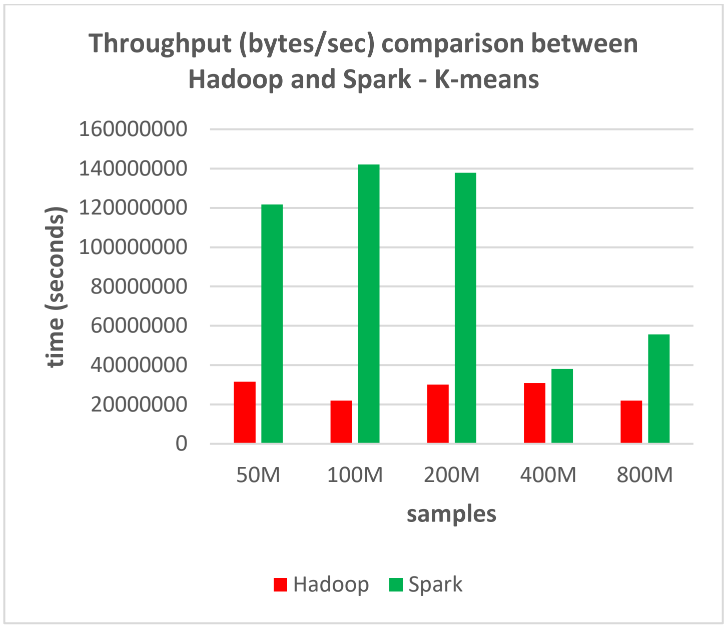
Input	50M	100M	200M	400M	800M
Hadoop	31515234 b/s	21892562 b/s	30124346 b/s	30937888 b/s	21904508 b/s
Spark	121756238 b/s	142117894 b/s	137782398 b/s	38018824 b/s	55597620 b/s

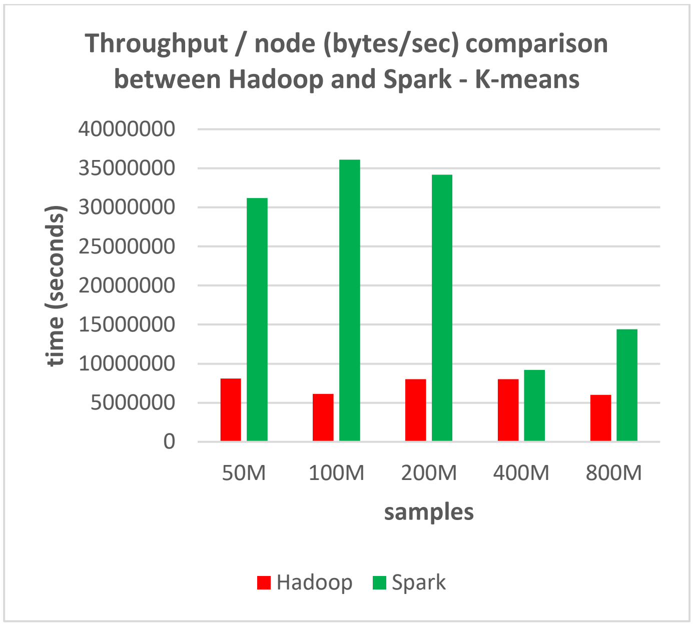
(a)

Input	50M	100M	200M	400M	800M
Hadoop	8089344 b/s	6145982 b/s	8000986 b/s	8009038 b/s	6012318 b/s
Spark	31190228 b/s	36112232 b/s	34171238 b/s	9213454 b/s	14389470 b/s

(b)

Table 4.14: (a) The throughput (bytes / seconds) comparison (b) The throughput / node (bytes / seconds) for executing K-means on Hadoop and Spark





(a) (b)

Fig 4.12: (a) The throughput (bytes / seconds) comparison (b) The throughput / node (bytes / seconds) for executing K-means on Hadoop and Spark

Chapter 5
Conclusion

5.1 Conclusion

In this experiment, I conducted the execution of benchmarks – web search benchmark PageRank, micro benchmark WordCount, micro benchmark TeraSort and machine learning benchmark K-means on two leading distributed big data analytics platform – Spark and Hadoop, for analyzing different runtime aspects and accordingly present a comparative performance study between them.

According to the outcomes of the experiment, I found that Spark performs quicker than Hadoop because it uses in-memory computation. For machine learning such as K-means and for web search such as PageRank, Spark is very effective because it is designed to implement iterative applications. With smaller input size Spark runs faster for PageRank but due to the in-memory calculation restriction it starts falling off behind Hadoop with 32 million pages input and more. It means that Spark is good for PageRank benchmark with smaller input size whereas Hadoop gives better performance with larger input size in this case. For Spark, the amount of maximum memory usage has also touched 100%.

Spark performed faster with larger input on WordCount and TeraSort benchmark. Nevertheless, it occupied higher memory resources compared to Hadoop and also its performance slowed down than Hadoop when the input size increased, and the system failed to provide enough memory space.

Hence, I found out that Spark is a good platform for executing iterative applications where the execution time would be considered a sensitive criterion and with not gigantic input datasets. Though normally Spark performs faster than Hadoop the cost pays off at consuming a substantial amount of memory. If enough memory is not available and also the speed is not a major requirement, it's better to choose Hadoop over Spark. On condition that sufficient disk space is available to deal with the input data and intermediate results, Hadoop can perform better.

References

[1] Apache spark document. http://spark.apache.org/docs/1.3.1/.

[2] Spark. [Online]. Available: http://spark-project.org/

[3] J. Tao, Q. Zhang, R. Hou, L. Chai, S. A. Mckee, J. Zhen, and N. Sun, "Understanding the behavior of in-memory computing workloads," in 2014 IEEE International Symposium on Workload Characterization (IISWC), October 26-28, 2014, pp. 22-30.

[4] Hibench. https://github.com/intel-hadoop/HiBench.

[5] Introduction to yarn and mapreduce 2. http://www.slideshare.net/cloudera/introduction-to-yarn-and-mapreduce-2.

[6] Powered by hadoop. http://wiki.apache.org/hadoop/PoweredBy

[7] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox, “Twister: A runtime for iterative MapReduce,” in Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing, 2010, pp. 810–818.

[8] Powered by spark. https://cwiki.apache.org/confluence/display/SPARK/Powered+By+Spark.

[9] Spark streaming programming guide. http://spark.apache.org/docs/latest/streaming-programming-guide.html.

[10] N. Islam, S. Sharmin, M. Wasi-ur-Rahman, X. Lu, D. Shankar, D. K. Panda, “Performance characterization and acceleration of in-memory file systems for Hadoop and Spark applications on HPC clusters,” in 2015 IEEE International Conference on Big Data (Big Data), October 29, 2015-November 1, 2015, pp. 243-252.

[11] L. Page, S. Brin, R. Motwani, and T. Winograd, “The PageRank citation ranking: Bringing order to the Web,” Technical Report, Stanford InfoLab, 1999.

[12] Apache hadoop document. http://hadoop.apache.org/docs/r2.7.1/index.html , June 2015.

[13] K.-H. Lee, Y.-J. Lee, H. Choi, Y. D. Chung, and B. Moon, “Parallel data processing with MapReduce: a survey,” ACM SIGMOD Record, vol. 40, no. 4, pp. 11–20, 2012.

[14] K. Shvachko, K. Hairong, S. Radia, and R. Chansler, “The Hadoop Distributed File System,” in 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp.1-10, May 3-7, 2010.

[15] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. OSDI’04: Sixth Symposium on Operating System Design and Implementation, December 2004.

[16] Lei Gu and Huan Li. Memory or time: Performance evaluation for iterative operation on hadoop and spark. IEEE International Conference, 2013.

[17] Satish Gopalani and Rohan Arora. Comparing apache spark and map reduce with performance analysis using k-means. International Journal of Computer Applications, 113(1), March 2015.

[18] Josh James. How much data is created every minute? https://www.domo.com/blog/2012/06/how-much-data-is-created-every-minute/ , June 2012.

[19] Josh James. Data never sleeps 3.0. https://www.domo.com/blog/2015/08/data-never-sleeps-3-0/ , August 2015.

[20] Josh James. Data never sleeps 2.0. https://www.domo.com/blog/2014/04/data-never-sleeps-2-0/ , August 2014.

[21] L. Gu and H. Li, “Memory or time: performance evaluation for iterative operation on Hadoop and Spark,” in 2013 IEEE 10th International Conference on High Performance Computing and

Communications & 2013 IEEE International Conference on Embedded and Ubiquitous Computing (HPCC_EUC), November 13-15, 2013, pp. 721-727.

[22] J. Zhan, "Big data benchmarks, performance optimization, and emerging hardware," 4th and 5th Workshops, BPOE 2014, Salt Lake City, March 1, 2014 and Hangzhou, China, September 5, 2014, Revised selected papers.

[23] Vinod Kumar Vavilapalli, Arun C. Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, Bikas Saha, Carlo Curino, Owen O'Malley, Sanjay Radia, Benjamin Reed, and Eric Baldeschwieler. Apache hadoop yarn: Yet another resource negotiator. In Proceedings of the 4th Annual Symposium on Cloud Computing, SOCC '13, pages 5:1–5:16, New York, NY, USA, 2013. ACM.

[24] K. Wang and M. M. H. Khan, "Performance prediction for Apache Spark platform," in 2015 IEEE 12th International Conference on Embedded Software and Systems (ICESS), 2015 IEEE 17th International Conference on High Performance Computing and Communications (HPCC), 2015 IEEE 7th International Symposium on Cyberspace Safety and Security (CSS), August 24-26, 2015, pp. 166-173.

[25] J. Yan, X. Yang, R. Gu, C. Yuan, and Y. Huang, "Performance optimization for short MapReduce job execution in Hadoop," in 2012 Second International Conference on Cloud and Green Computing (CGC), November 1-3, pp. 688-694.

[26] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A fault-tolerant abstraction for in-memory cluster computing," In NSDI, April 2012.

[27] M. Zaharia, M. Chowdhury, S. S. Michael J. Franklin, and I. Stoica, "Spark: Cluster computing with working sets," In HotCloud, June 2010.

[28] J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," In OSDI, 2004.