RESEARCH

# A Comprehensive Performance Analysis of Apache Hadoop and Apache Spark for Large Scale Data Sets Using HiBench

N Ahmed[1*†], Andre L. C. Barczak[1], Teo Susnjak[1] and Mohammed A. Rashid[2]

*Correspondence: nasim751@yahoo.com
[1]School of Natural and Computational Sciences, Massey University, Albany, 0745 Auckland, New Zealand
Full list of author information is available at the end of the article
†Equal contributor

**Abstract**

In recent times Big Data analytics has got tremendous attention and it involves storing, processing, and analysing large scale datasets. The advent of distributed computing frameworks such as Hadoop and Spark offers an efficient solution to analyse vast amounts of data. Due to the availability of an application programming interface (API) and its performance, Spark become very popular, even more popular than the MapReduce framework. Both these frameworks have more than 150 parameters and the combination of these parameters have a huge impact on cluster performance. The system default parameters help the system administrator to deploy their system applications without much effort, and they can measure their specific cluster performance with factory-set parameters. However, an open question remains: can new parameter selection improve cluster performance? In this regard, our study investigates the most impacting parameters such as input splits and shuffling, in order to compare the performance between Hadoop and Spark, using a specific cluster implemented in our department. We used a trial-and-error approach for tuning these parameters based on a large number of experiments. In order to evaluate the frameworks comparison and analysis, we select two workloads: WordCount and TeraSort. The performance metrics are carried out based on three criteria, namely execution time, throughput, and speedup. Our experimental results revealed that both system performances heavily depends on input data size and correct parameter selection. The analysis results found, unsurprisingly, that Spark has better performance as compared to Hadoop, achieving up to 2 times speedup in WordCount workload and up to 14 times in TeraSort workloads when default parameters are replaced. Finally, we conclude that the system performance depends on different parameters configuration alternatives, and they depend on the data size.

**Keywords:** HiBench; BigData; Hadoop; MapReduce; Benchmark; Spark

## 1 Introduction

Recently, Hadoop [1] got tremendous attention in the IT industry and academia for its ability to handle large amounts of data, along with extensive processing and analysis facilities. These large datasets are produced by different users and most of them are unstructured. Besides, the advent of many new applications and technologies brought much larger volumes of complex data including social media,e.g., Facebook, Twitter, YouTube, online shopping, machine data, system data, and browsing history [2]. This huge amount of digital data becomes a challenging factor in the management and analysis of data. The conventional database management tools are unable to handle this type of data [3]. Therefore, developers and

researchers needed to find out a better solution that included methods for robust and accessible storage. Big data technologies, tools, and procedures allowed organizations to speedily capture, process, and analyse large quantities of data and extract appropriate information at a reasonable cost.

Several solutions are available to handle this problem [4]. Distributed computing is one possible solution [5], and become the most efficient and fault-tolerant method for companies to store and process massive amounts of data. Among this new group of tools, MapReduce and Spark are the most commonly used cluster computing tools. They provide users with various functions using a simple application programming interface (API). MapReduce is a framework used for distributed computing that is used for parallel processing and designed purposely to write, read, and process bulky amounts of data [1, 6, 7]. This data processing framework is comprised of three stages: Map phase, Shuffle phase, and Reduce phase. In this technique, the large files are divided into several small blocks of equal sizes and distributed across the cluster for storage. MapReduce and Hadoop distributed file systems (HDFS) are a core part of the Hadoop system, so computing and storage work together across all nodes that compose a cluster of computers [8].

Apache Spark is designed based on the Hadoop and its purpose is to build a program model that "fits a wider class of applications than MapReduce while maintaining the automatic fault-tolerance" [9]. It is not only an alternative to the Hadoop framework but it also provides various functions to process real streaming data. Apart from the map and reduce functions, Spark also supports MLib1, GraphX, and Spark streaming for big data analysis. Hadoop MapReduce processing speed is slow because it requires accessing disks for reads and writes. On the other hand, Spark uses memory to store data, so it reduces the read/write cycle [1]. In this paper, we will present an empirical performance analysis between MapReduce and Spark based on an existing cluster. According to our knowledge, this is the first comparison work between Hadoop and Spark where we used large scale datasets (600GB). These new experiments show the various aspects of cluster performance applications. These new experiments are applied to test the efficiency of different jobs under MapReduce and Spark, where the datasets are repeatedly changing, and the ideal parameters need to ensure job efficiency. Multiple parameters are used to tune job performance. To make sure that our experimental results are accurate, we executed each experiment three times, getting the average execution time for each job. Besides, we selected several parameters covering different aspects suggested by MapReduce and Spark online documents. The results of the analysis will facilitate job performance tuning, as well as enhance the freedom to modify parameters if users want. The remainder of the paper is organized as follows: Section 2 presents an overview of related research, and then describes Hadoop and Spark. The gap on Map and Reduce phase is explained in Section 3. The experimental setup is presented in Section 4. In Section 5, we explain the chosen parameters and tuning approach. Section 6 presents the performance analysis of the results and finally, we conclude in section 7.

## 2 Related Work

Samadi [8] has investigated the criteria of the performance comparison between Hadoop and Spark framework. For an impartial comparison, the input data size

and configuration remained the same. For their experiment, they have used eight benchmarks of HiBench suite [10] where the input data was generated automatically for every case and size and the computation was performed several times to find out the execution time and throughput. When they deployed microbenchmark (Short and TeraSort) on both systems, Spark showed higher involvement of processor in I/Os while Hadoop mostly processes user tasks. On the other hand, Spark performance was excellent when dealing with small input sizes, for instance, micro and web search (Page Rank). Finally, they concluded that Spark is faster and very strong for processing data in-memory while Hadoop MapReduce performs map and reduce function in disk.

In another paper, Samadi [9] proposed a virtual machine based on Hadoop and Spark to get the benefit of virtualisation. The main advantage of this virtual machine is that it can perform all operations even if the hardware fails. In this deployment, they have used Centos operating system [9] and built a Hadoop cluster based on a pseudo-distribution mode with various workloads. In their experiments, they have run the Hadoop machine on a single workstation and all other demos on its JVM. On the other hand, Spark was deployed on "Spark in MapReduce". To justify the big data framework, they have presented the results of Hadoop deployment on Amazon EC2. They have concluded that Hadoop is a better choice because Spark requires more memory resources than Hadoop. Finally, they have addressed that the cluster configuration is very important to reduce job execution time and the cluster parameter configuration must align with Mappers and Reducers.

The computational frameworks namely Apache Hadoop and Apache Spark, were investigated by [11]. In this investigation, the Apache webserver log file was taken into consideration to evaluate the comparative performance of the two frameworks. In these experiments, they have used virtualised computing resources of Okeanos based on infrastructures as a Service (IaaS) which is developed by Greek Research and Technology Network [11]. They proposed a number of applications and conducted several experiments to find out the execution time of each application. They have used various sizes of the input file and the slave nodes in order to find out the execution time. They have found that the execution time is proportional to the input data size. They have concluded that the performance of Spark is much better in most cases as compared to Hadoop. Satish and Rohan [12] have shown a comparative performance study between Hadoop MapReduce and Spark-based on the K-means algorithm. In this analysis, they have used a specific data set that supports this algorithm and considered both single and double nodes when gathering the execution time for each experiment. They have concluded that the Spark speed reaches up to three times higher than the MapReduce, though Spark performance heavily depends on sufficient memory size [13].

Xiuqin, Pen, and WU [14] have proposed a unified cloud platform, including batch processing ability over standalone tools for the log analysis. In this investigation, they have considered four different frameworks: Hadoop, Spark, and the warehouse data analysis tools Hive and Shark. They implemented two machine learning algorithms (K-means and PageRank) based on this framework with six nodes to validate the cloud platform. They have used different data sizes as inputs. In the case of K-means, as the data size increased and exceed memory size, the latency schedule, and

overall Spark performance degraded. However, the overall performance was still six times higher than Hadoop on average. On the other side, Shark shows significant performance improvement while using queries directly from disk. They have concluded that high stability, availability, and efficiency is very appropriate for batch data analysis on the cloud platform model.

Petridis [15] have investigated the most important Spark parameters and given a guideline to the developer and system administrator to select the right parameter values to instate of default parameters values based on trial-and-error methodology. Three types of case studies with different categories such as Shuffle Behavior, Compression and Serialization, and Memory Management parameters were performed in this study. They have highlighted the impact of memory allocation and serialization when the number of cores and default parallelism values change. Therefore, there are 12 parameters chosen with three benchmarking applications: sort-by-key, shuffling, and k-means. The sort-by-key experiments are used both 1000000 and 1 billion key-value where the length of this 10 and 90 bytes and the optimal degree of partition is set to 640. The highest performance improvement is increased by 25% by KryoSerializer rather than using the default Java Serializer. The Hash performance is increased at 127 second which is 30 seconds faster than default parameter and shuffle.file.buffer is increased the performance by 140 second. The rest of the parameters doesn't play an important role to improve performance. For another Shuffling experiment, they used a 400GB dataset. In this experiment, the KryoSerializer improved by 10% as compared to the default Java Serializer. The Hash shuffle performance degraded by 200 second and Tungsten-sort speed is increased by 90 second. By decreasing the buffer size from 32KB to 15KB, the system performance degraded by about 135s which is more than 10% from the primary selection. For k-means, they used two sizes of data input (100 MB and 200 MB). The number of the centre is allocated 10 and the iteration is fixed with 10 for fair time measurements. They have not found significant k-menas performance improvement by changing the parameters. Therefore, they have concluded that based on their methodology, the speedup achievement is 10-fold. However, the main challenges of Hadoop and Spark configuration parameter tuning is that due to the complicated behaviour of distributed large scale systems, the parameter selection is not always trivial for the system administrators. So, the a huge number of configuration parameters is always difficult. Inappropriate parameter values affect the overall system performance.

Based on the literature, we chose a conventional trial-and-error approach [15] and 18 important parameters under input splits and shuffle category with the extension of the data sizes to 600GB.

## 3 Difference Between Hadoop and Spark

Hadoop [16] is a very popular and effective open-source software framework that enables distributed storage, including the capability of storing a large amount of big datasets across clusters. It is designed in such a way that it can scale up from a single server to thousands of nodes. Hadoop processes large data concurrently and produces fast results. With Hadoop, the core parts are Hadoop Distributed File System (HDFS) and MapReduce. HDFS [17] splits the files into small pieces into blocks and saves them into different nodes. There are two kinds of nodes on HDFS:

data-nodes (worker) and name-nodes (master nodes) [18, 19]. All the operations including delete, read and write are based on these two types of nodes. The workflow of HDFS is like the following flow: firstly, the name-node asks for access permission. If accepted, it will turn the file name into a list of HDFS block IDs which includes the files and the data-nodes that saved the blocks related to that file. Then the ID list will be sent back to the client and the users can do further operations based on that.

MapReduce [20] is a computing framework that includes two operations: Mappers and Reducers. The mappers will process files based on the map function and transfer them into the new key-value pairs [21]. Next, the new key-value pairs are assigned to different partitions and sorted based on their keys. The combiner is optional and can be recognized as a local reduces operation which allows counting the values with the same key in advance to reduce the I/O pressure. Finally, partitions will divide the intermediate key-value pairs into different pieces and transfer them to a reducer. MapReduce needs to implement one operation: shuffle. Shuffle means transferring the mapper output data to the proper reducer. After the shuffle process is finished, the reducer starts some copy threads (Fetcher) and obtains the output files of the map task through HTTP [22]. The next step is merging the output into different final files, which are then recognized as the input data of reducer. After that, the reducer processes the data based on reduce function and writes the output back to the HDFS. Figure 1 depicts a Hadoop MapReduce architecture.

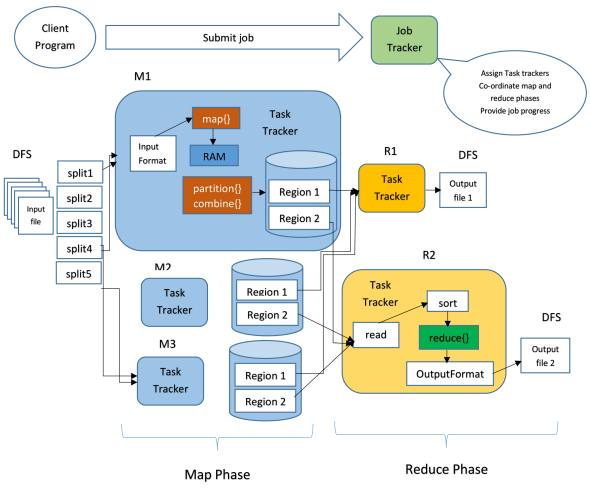



Figure 1: Hadoop MapReduce architecture [1]

Spark became an open-source project from 2010. Zahari has developed this project at UC Berkely's AMPLab in 2009 [23, 4]. Spark offers numerous advantages for developers to build big data applications. There are two important terms proposed by Spark: Resilient Distributed Datasets (RDD) and Directed Acyclic Graph (DAG). These two techniques work together perfectly and accelerate Spark up to tens of times faster than Hadoop under certain circumstances, even though normally it only achieves a performance two to three times faster than MapReduce. It supports multiple sources that have a fault tolerance mechanism, can be cached and supports

parallel operations. Besides, it can represent a single dataset with multiple partitions. When Spark runs on the Hadoop cluster, RDDs will be created on the HDFS in many formats that are supported by Hadoop, likewise text and sequence files. The DAG scheduler [24] system expresses the dependencies of RDDs. Each spark job will create a DAG and the scheduler will drive the graph into the different stages of tasks then the tasks will be launched to the cluster. The DAG will be created in both maps and reduce stages to fully express the dependencies. Figure 2 illustrates the iterative operation on RDD. Theoretically, limited Spark memory causes the performance to slow down.

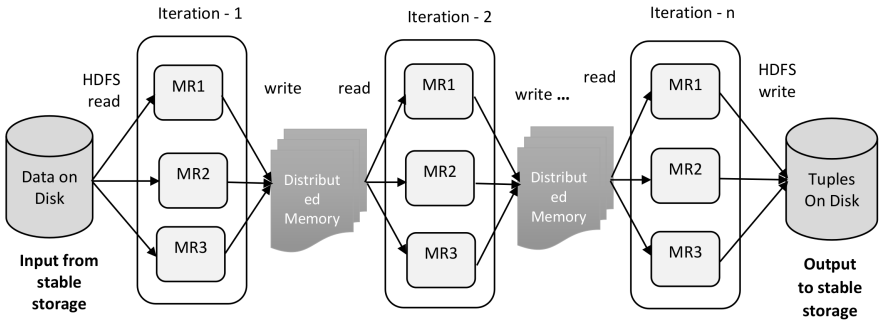



Figure 2: Spark workflow [25]

## 4 Experimental Setup

### 4.1 Cluster Architecture

In the last couple of years, many proposals came from different groups of researchers about the suitability of the framework when different types of data and different sizes are used as input in different clusters. Therefore, it become necessary to study the performance of these frameworks and understand the influence of different parameters. In these experiments, we will present our cluster performance based on MapReduce and Spark using the HiBench suite [17, 26]. In particular, we have selected two Hibench workloads out of thirteen standard workloads to represent the two types of jobs namely WordCount (aggregation job), and TeraSort (shuffle job) with large datasets.

### 4.2 Hardware and Software Specification

The experiments were deployed in our cluster. The cluster is configured with 10 nodes, 80 CPU cores and 60TB local storage. Our hardware is suitable for handling various difficult situations in Spark and MapReduce. The detailed Hadoop cluster and software specifications are presented in Table 1.

All our jobs run in Spark and MapReduce. We have selected Yarn as a resource manager, which can help us to monitor the situation of each working node as well as track the details of each job with its history. We have used *Apache Ambari* to monitor and profile the selective workloads running on Spark and MapReduce. Apache Ambari is a web-based powerful cluster management tool. It supports most of the Hadoop components, including HDFS, MapReduce, Hive, Pig, Hbase, Zookeeper,

Table 1: Experimental Hadoop Cluster

| | | |
|---|---|---|
| Server Configuration | Processor | 2.9 GHz |
| | Main Memory | 64 GB |
| | Local Storage | 10 TB |
| Node Configuration | Operating System | Intel(R) Xeon(R) CPU E3-1231 v3 @ 3.40GHz |
| | Main Memory | 32 GB |
| | Number of Nodes | 10 |
| | Local Storage | 6 TB each, 60TB total |
| | CPU cores | 8 each, 80 total |
| Software | Operating System | Ubuntu 16.04.2 (GNU/Linux 4.13.0-37-generic x86 64) |
| | JDK | 1.7.0 |
| | Hadoop | 2.4.0 |
| | Spark | 2.1.0 |
| Workload | Micro Benchmarks | WordCount, and TeraSort |

Sqoop, and Hcatalog" [27]. Besides, Ambari supports the user to control the Hadoop cluster on three aspects, namely provision, management, and monitoring. We have used Ambari to monitor and profile the selected workloads running on Spark and MapReduce.

### 4.3 Workloads

As stated above, in this study we chose two workloads for the experiments [28, 29]:

**WordCount**: The wordCount workload is map-dependent and it counts the number of occurrences of separate words from text or sequence file. The input data is produced by *RandomTextWriter*. It splits into each words by using the map function and generates intermediate data for the reduce function as a key-value [30]. The intermediate results are added up, generating the final word count by the reduce function.

**TeraSort**: The TeraSort package was released by Hadoop in 2008 [31] to measure the capabilities of cluster performance. The input data is generated by the *TeraGen* function which is implemented in Java. The TeraSort function does the sorting using the MapReduce and the TeraValidate function is used to validate the output of the sorted data. For both workloads, we used up to 600 GB of synthetic input data generated using a string concatenation technique.

## 5 The Parameters of Interest and Tuning Approach

Tuning parameters in Apache Hadoop and Apache Spark is a challenging task. Every single parameter has an impact on the system performance of a cluster. The configuration of these parameters needs to be investigated according to workload, data size, and cluster architecture. We have conducted a number of experiments using Apache Hadoop and Apache Spark with different parameters setting. For this experiment, we have chosen the core MapReduce and Spark parameter setting from input splits and shuffle groups. The selected tuned parameters with their respective tuned values on the map-reduce and Spark category are shown in Tables 2 and 3.

Table 2

| Configuration Parameters Category | Hadoop | Tuned Values |
|---|---|---|
| Input Split | MapReduce.reduce.memory | 8GB |
| | mapred.reduce.task | 16,384MB, 25,600MB |
| | MapReduce.reduce.cpu.vcores | 4 |
| | mapred.min.split.size, mapred.max.split.size | 128MB (default), 256MB, 512MB, 1024MB |
| Shuffle | i/o.sort.mb | 25, 50, 75, 100 |
| | i/o.sort.factor | 512, 1024, 1536, 2047 |
| | MapReduce.reduce.shuffle.parallelcopies | 50, 100, 150, 200 |
| | MapReduce.task.io.sort.factor | 15, 30, 45, 60 |

Table 3

| Configuration Parameters Category | Spark | Tuned Values |
|---|---|---|
| Input Split | num-executors | 50 |
| | executor-cores | 4 |
| | executor-memory | 8GB |
| | spark.hadoop.MapReduce.input .fileinput-format.split.maxsize | 128MB (default), 256MB, 512MB, 1024MB |
| | spark.hadoop.MapReduce.input .fileinput-format.split.minsize | 128MB (default), 256MB, 512MB, 1024MB |
| Shuffle | spark.shuffle.file.buffer | 16k, 32k (default), 48k, 64k |
| | spark.reducer.maxSizeInFlight | 32M, 48M (default), 64M, 96M |
| | spark.hadoop.dfs.replication | 1 |
| | spark.default.parallelism | 80, 100, 200, 300 |

## 6 Results and Discussion

In this section, the results obtained after running the jobs are evaluated based on the comparison between Hadoop and Spark frameworks. We have used synthetic input data and used the same parameter configuration for a realistic comparison. Each test was repeated 3 times and the average runtime was plotted in each graph. For both frameworks, we show the execution time, throughput, and speed up in order to compare the two frameworks, and to visualise the effects of changing the default parameters.

### 6.1 Execution Time

We have conducted a number of experiments to investigate and examine the total execution time that is concerning the different parameters for each application. The execution time is affected by the input data sizes, the number of active nodes, and the application types. For the fair comparative analysis, we have fixed the same parameters such as the number of executors to 50, executor memory to 8GB, executor cores to 4.

Figures 3-a and 3-b show how MapReduce and Spark execution time depend on the size of the datasets, and on the different input splits and shuffle parameters. The execution time of MapReduce WordCount workload with the default input splits size (128MB) and shuffle parameter (*sort.mb* 100, *sort.factor* 2047) obtained better execution time for entire data sizes as compared to other parameters. The

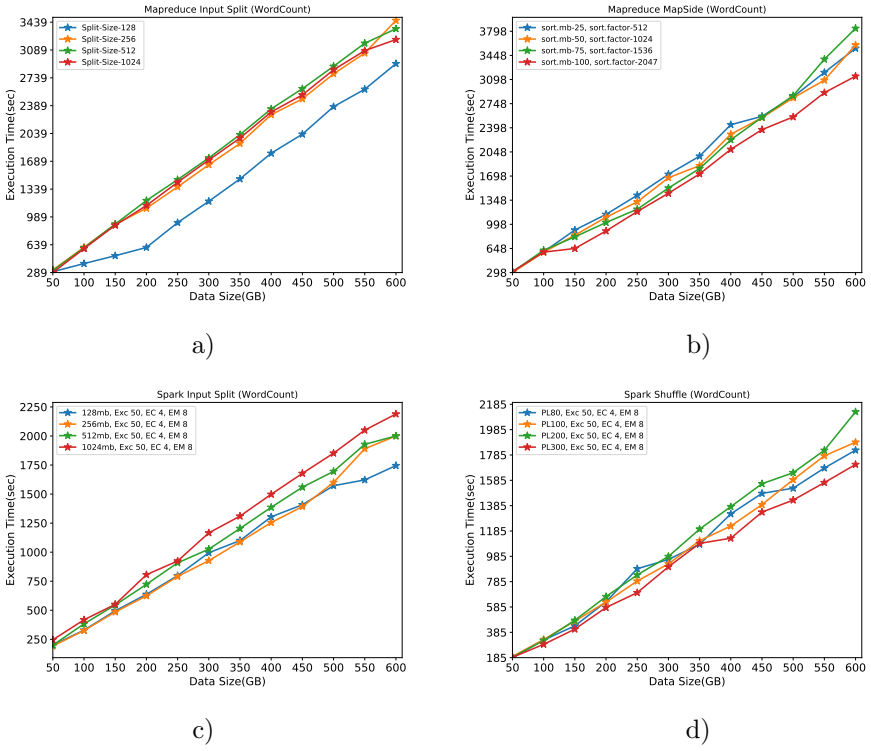



Figure 3: The performance of the WordCount application with a varied number of input splits and shuffle tasks.

possible reason of Hadoop Map and Reduce function behave differently because its longer execution time and overlooked container initialization overhead for specific workload types. This analysis suggests that the default parameter is more suitable for our cluster when using data sizes from 50GB to 600GB. In fig.3-c the default input splits of Spark is 128MB. Previously, we have mentioned that the number of executors, executor memory, and executor cores are fixed. From the above fig.3-c, we see that the execution time of input split size 256MB outperforms the default set up until 450GB data sizes. In fact, the default splits size (128MB) is more efficient when data size is larger than 450GB. Notably, we can see that the default parameter shows better execution performance when the data set reaches 500GB onwards. The new parameter value can improve the processing efficiency by 2.2% higher than the default value (128MB). Table 4 presents the experimental data of WordCount workload between MapReduce and Spark while the default parameters are changing.

For Spark shuffle parameter, we have chosen the default serializer, that is *JavaSerializer*. In this category, the serializer is PL100 object [32]. We can see from figure 3-d that the improvement rate is significantly increased when we set PL value to 300. It is evident that the best performance is achieved for sizes larger than 400GB. Also, it shows that tuned PL value 300 can achieve a 3% higher improvement for the rest of the data sizes. Consequently, we can conclude that input splits can be

considered as an important factor to enhance the efficiency of Spark WordCount jobs when executing small datasets.

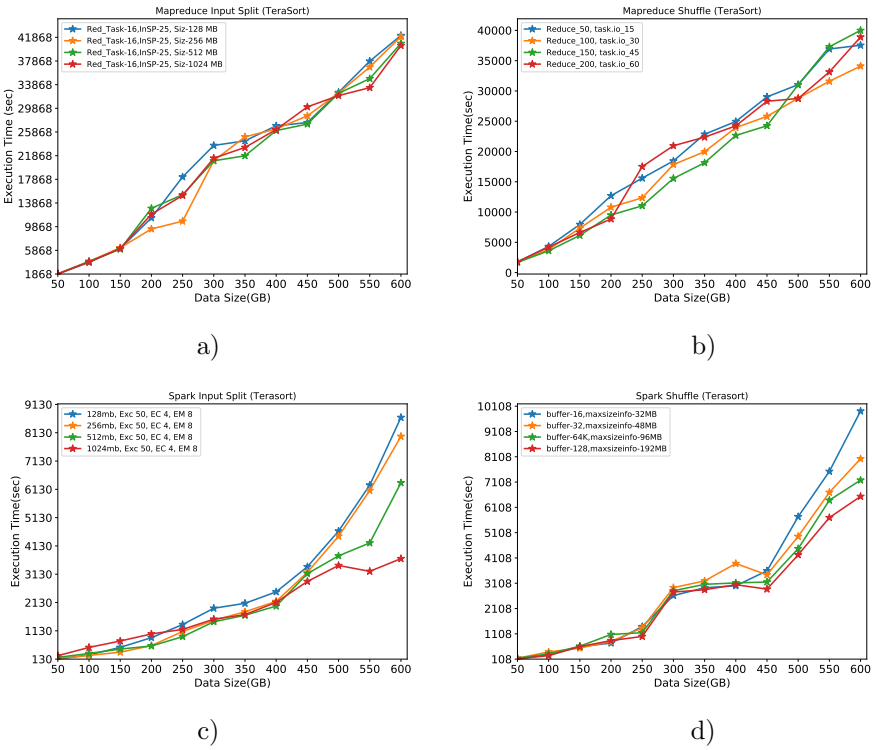


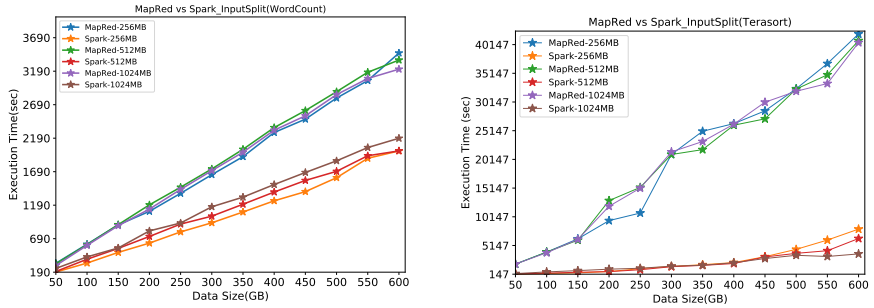

Figure 4: The performance of the TeraSort application with a varied number of input splits and shuffle tasks.

Figure 4-a is comparing MapReduce TeraSort workloads based on input splits that include default parameters. In this analysis, we have set (Red_Task and InSp) value fixed with default split size 128MB. We have changed the parameter values and tested whether the size of the splits can keep the impact on the runtime.So, for this reason, we have selected three different sizes: 256MB, 512MB, and 1024MB. We have observed that with split size of 256MB, the execution performance is increased by around 2% in datasets with up to 300GB. On the contrary, when the data sizes are larger than 300GB, the default size outperforms splits size 512MB. Moreover, we have noticed that the improvement rates are almost similar when the data sizes are smaller than 200GB.

Figure 4-b illustrates the execution performance with MapReduce shuffle parameter for TeraSort workload. We have seen that the average execution time behaves linearly for sizes up to 450GB when the parameter change to (*Reduce_150* and *task.io_45*) as compared to the default configuration (*Reduce_100* and *task.io_30*). Besides, we have noticed that the default configuration is outperforming all other settings when the data sizes are larger than 450GB. So, we can conclude that by changing the shuffled value, the system execution performance improves by 1%. In general, this is very unlikely that the default size has optimum performance for larger data sizes.

Figure 4-c illustrates the Spark input split parameter execution performance analysis for the TeraSort workload. The Spark executor memory, number of executors, and executor memory are fixed while we change the block size to measure the execution performance. Apart from the default block size (128MB), there are 3 pairs (256MB, 512MB, and 1024MB) of block size is taken into this consideration. Our results revealed that the block size 512 MB and 1024MB present better runtime for sizes up to 500GB data size. Also, we have observed that a significant performance improvement achieved by the 1024 block size which is 4% when the data size is larger than 500GB. Thus, we can conclude that by adding the input splits block size for large scale data size, Spark performance can be increased.

Figure 4-d shows Spark shuffle behaviour performance for TeraSort workloads. We have taken two important default parameters (*buffer*=32, *spark.reducer.maxSizeIn Flight*=48MB) into our analysis. We have found that when the buffer and *maxSizeInFlight* are increased by 128 and 192, the execution performance increased proportionally up to 600GB data sizes. Our results show that the default execution is equal with a tested value of up to 200GB data sizes. The possible reason for this performance improvement is the larger number of splits size for different executors. Table 5 presents the experimental data of the TeraSort workload between MapReduce and Spark while the default parameters are changing.




a) b)

Figure 5: The comparison of Hadoop and Spark with WordCount and TeraSort workload with varied input splits and shuffle tasks.

Figure 5-a illustrates the comparison between Spark and MapReduce for WordCount and TeraSort workloads after applying the different input splits. We have observed that Spark with WordCount workloads shows higher execution performance by more than 2 times when data sizes are larger than 300GB for WordCount workloads. For the smaller data sizes, the performance improvement gap is around 10 times. Fig. 5-b shows a TeraSort workload for MapReduce and Spark. We can see that Spark execution performance is linear and proportionally larger as the data size increase. Also, we noticed that the MapReduce job is not as linear as Spark. The possible reason could be unavoidable job action on the clusters. So, we conclude that MapReduce has slower data sharing capabilities and a longer time to the read-write operation than Spark [4].

Table 4: The best execution time of MapReduce and Spark with WordCount workload

| | split sizes (MB) | execution time (sec) |
|---|---|---|
| MapReduce input splits (WordCount) | 128 | 2376 |
| Spark input splits (WordCount) | 256 | 1392 |
| MapReduce shuffle (WordCount) | 100 | 2371 |
| Spark shuffle (WordCount) | 300 | 1334 |

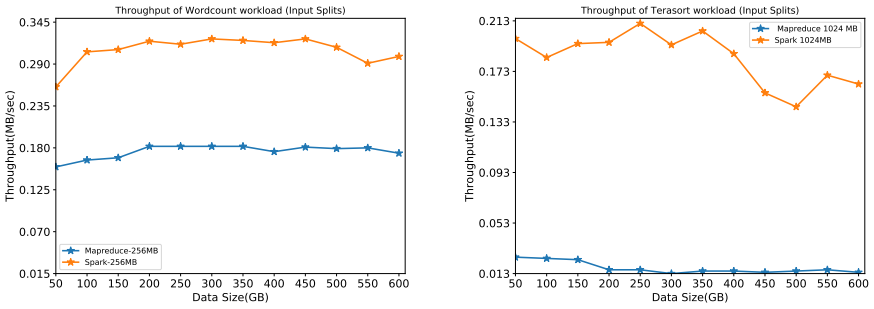
Table 5: The best execution time of MapReduce and Spark with Terasort workload

| | split sizes (MB) | execution time (sec) |
|---|---|---|
| MapReduce input splits (TeraSort) | 256 | 21014 |
| Spark input splits (TeraSort) | 512 & 1024 | 3780 & 3439 |
| MapReduce shuffle (TeraSort) | 150 & 45 | 24250 |
| Spark shuffle (TeraSort) | 128 & 192 | 6540 |

## 6.2 Throughput

A general definition for the term *throughput* is: the amount of information a system can process in a given period. The throughput is calculated likewise, the total input data size over the execution time.

For this analysis, we consider the best results from each category. We have observed that MapReduce throughput performance for the TeraSort workload is decreasing slightly as the data size crosses beyond 200GB. Besides, for the WordCount workload, the MapReduce throughput is almost linear. As far as the Spark TeraSort workload, it can be observed that the throughput is not linear but for the WordCount workload, the throughput is linear. In this analysis, the main focus was to present the throughput difference between WordCount and TeraSort workload for MapReduce and Spark. We found that WordCount workload remains almost stable for most of the data sizes and concerning TeraSort workload MapReduce remain stable than Spark (see Figure 6).




a) b)

Figure 6: Throughput of WordCount and TeraSort workload.

## 6.3 Speedup

Figures 7(a, b, c) show the Spark's speed up compared to MapReduce. Figures 7(a and b) depicts individual workload speedup. The best results are taken into this consideration from each category in order to get a speedup. From the above figures,

we can see that as the data size increases, WordCount workload speedup decreases with some non-linearity. Besides, we can see that the TeraSort speedup decreases when data reaches sizes larger than 300GB. Notably, as the data size increases to more than 500GB for both workloads, the speedup starts to increase. Figure 7(c) illustrates the speedup comparison between the workloads. It can be seen that the TeraSort workload outperforms WordCount workload and achieves an all-time maximum speedup of around 14 times.

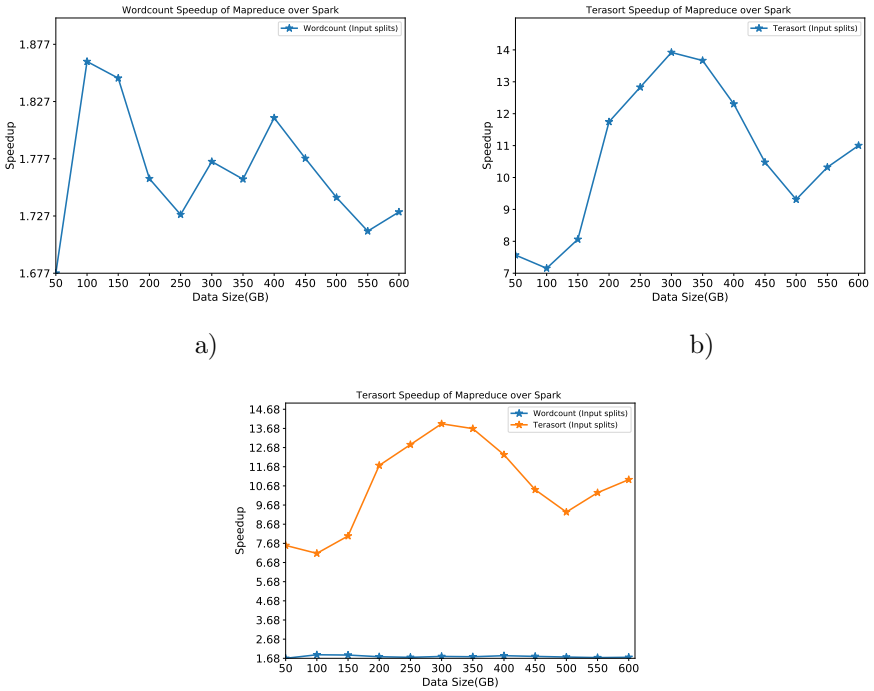



Figure 7: Spark over MapReduce speedup on input splits and shuffle.

## 7 Conclusion

In this article, the empirical performance analysis between Hadoop and Spark based on a large scale dataset is presented. We have executed WordCount and Terasort workloads and 18 different parameter values by replacing them with default set-up. In order to investigate the execution performance, we have used a trial-and-error approach for tuning these parameters including a large number of experiments. Our experimental results depict that both system performance heavily depends on input data size and right parameter selection. We have found that Spark has better performance as compared to Hadoop by 2 times with WordCount workload and 14 times with TeraSort workloads respectively when default parameters replaced with new values. Moreover, the throughput and speedup results show that Spark is more stable and faster than Hadoop because of Spark data processing ability in memory instate of a store in disk for the map and reduce function. As part of our future work, we plan to add and investigate 15 HiBench workloads, consider more parameters

under resource utilization, parallelisation, and all other aspects including practical data sets. In fact, the main focus would be to analyse the job performance for MapReduce and Spark when several parameter configurations replace the default values.

**Author's contributions**
Ahmed was the main contributor of this work. He has done an initial literature review, data collection, experiments, prepare results, and drafted the manuscript. Andre worked closely with Ahmed to review, analyze, and manuscript preparation. Teo and Rashid helped to improve the final paper.

**Acknowledgements**
The authors acknowledge Sibgat for his valuable suggestion.

**Availability of data and materials**
The data that support the findings of this study are available from the corresponding author upon reasonable request.

**Ethics approval and consent to participate**
Not applicable.

**Consent for publication**
Not applicable.

**Competing interests**
The authors declare that they have no competing interests.

**Funding**
This work was not funded.

**Author details**
[1]School of Natural and Computational Sciences, Massey University, Albany, 0745 Auckland, New Zealand. [2]School of Natural and Computational Sciences, Massey University,, 0745 Auckland, New Zealand. [3]School of Natural and Computational Sciences, Massey University,, 0745 Auckland, New Zealand. [4]Department of Mechanical and Electrical Engineering, Massey University,, 0745 Auckland, New Zealand.

**References**

1. Apache Hadoop Documentation 2014. http://hadoop.apache.org/
2. Verma, A., Mansuri, A.H., Jain, N.: Big data management processing with hadoop mapreduce and spark technology: A comparison. In: 2016 Symposium on Colossal Data Analysis and Networking (CDAN), pp. 1–4 (2016). IEEE
3. Management Association, I.R.: Big Data: Concepts, Methodologies, Tools, and Applications. IGI Global, United States (2016)
4. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., Mccauley, M., Franklin, M., Shenker, S., Stoica, I.: Fast and interactive analytics over hadoop data with spark. Usenix Login **37**, 45–51 (2012)
5. Karimov, J., Rabl, T., Katsifodimos, A., Samarev, R., Heiskanen, H., Markl, V.: Benchmarking distributed stream data processing systems. In: 2018 IEEE 34th International Conference on Data Engineering (ICDE), pp. 1507–1518 (2018). IEEE
6. Dean, J., Ghemawat, S.: Mapreduce: simplified data processing on large clusters. Communications of the ACM **51**(1), 107–113 (2008)
7. Wang, G., Butt, A.R., Pandey, P., Gupta, K.: Using realistic simulation for performance analysis of mapreduce setups. In: Proceedings of the 1st ACM Workshop on Large-Scale System and Application Performance, pp. 19–26 (2009)
8. Samadi, Y., Zbakh, M., Tadonki, C.: Comparative study between hadoop and spark based on hibench benchmarks. In: 2016 2nd International Conference on Cloud Computing Technologies and Applications (CloudTech), pp. 267–275 (2016). IEEE
9. Samadi, Y., Zbakh, M., Tadonki, C.: Performance comparison between hadoop and spark frameworks using hibench benchmarks. Concurrency and Computation: Practice and Experience **30**(12), 4367 (2018)
10. Wang, L., Zhan, J., Luo, C., Zhu, Y., Yang, Q., He, Y., Gao, W., Jia, Z., Shi, Y., Zhang, S.: Bigdatabench: A big data benchmark suite from internet services. In: 2014 IEEE 20th International Symposium on High Performance Computer Architecture (HPCA), pp. 488–499 (2014). IEEE
11. Mavridis, I., Karatza, E.: Log file analysis in cloud with apache hadoop and apache spark (2015)
12. Gopalani, S., Arora, R.: Comparing apache spark and map reduce with performance analysis using k-means. International journal of computer applications **113**(1) (2015)
13. Gu, L., Li, H.: 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, pp. 721–727 (2013). IEEE
14. Lin, X., Wang, P., Wu, B.: Log analysis in cloud computing environment with hadoop and spark. In: 2013 5th IEEE International Conference on Broadband Network & Multimedia Technology, pp. 273–276 (2013). IEEE
15. Petridis, P., Gounaris, A., Torres, J.: Spark parameter tuning via trial-and-error. In: INNS Conference on Big Data, pp. 226–237 (2016). Springer

16. Landset, S., Khoshgoftaar, T.M., Richter, A.N., Hasanin, T.: A survey of open source tools for machine learning with big data in the hadoop ecosystem. Journal of Big Data **2**(1), 24 (2015)
17. HiBench Benchmark Suite. https://github.com/intel-hadoop/HiBench
18. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10 (2010). IEEE
19. Luo, M., Yokota, H.: Comparing hadoop and fat-btree based access method for small file i/o applications. In: International Conference on Web-Age Information Management, pp. 182–193 (2010). Springer
20. Taylor, R.C.: An overview of the hadoop/mapreduce/hbase framework and its current applications in bioinformatics. In: BMC Bioinformatics, vol. 11, p. 1 (2010). Springer
21. Vohra, D.: Practical Hadoop Ecosystem: A Definitive Guide to Hadoop-Related Frameworks and Tools. Apress, California (2016)
22. Lee, K.-H., Lee, Y.-J., Choi, H., Chung, Y.D., Moon, B.: Parallel data processing with mapreduce: a survey. AcM sIGMoD Record **40**(4), 11–20 (2012)
23. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. HotCloud **10**, 95 (2010)
24. Kannan, P.: Beyond hadoop mapreduce apache tez and apache spark. San Jose State University. URL: http://www. sjsu. edu/people/robert. chun/courses/CS259Fall2013/s3/F. pdf ( 02.08. 2016) (2015)
25. Spark Core Programming. https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm
26. HiBench Benchmark Suit. https://github.com/intel-hadoop/HiBench
27. Ambari. https://ambari.apache.org/
28. Huang, S., Huang, J., Dai, J., Xie, T., Huang, B.: The hibench benchmark suite: Characterization of the mapreduce-based data analysis. In: 2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010), pp. 41–51 (2010). IEEE
29. Chen, C.-O., Zhuo, Y.-Q., Yeh, C.-C., Lin, C.-M., Liao, S.-W.: Machine learning-based configuration parameter tuning on hadoop system. In: 2015 IEEE International Congress on Big Data, pp. 386–392 (2015). IEEE
30. Xiang, L.-H., Miao, L., Zhang, D.-F., Chen, F.-P.: Benefit of compression in hadoop: A case study of improving io performance on hadoop. In: Proceedings of the 6th International Asia Conference on Industrial Engineering and Management Innovation, pp. 879–890 (2016). Springer
31. O'Malley, O.: Terabyte sort on apache hadoop. Report, Yahoo! (2008). http://sortbenchmark.org/YahooHadoop.pdf
32. Spark Configuration. https://spark.apache.org/docs/latest/configuration.html