

3-2016

# The Performance Comparison of Hadoop and Spark

Shengti Pan

St. Cloud State University, panshengti@hotmail.com

Follow this and additional works at: [https://repository.stcloudstate.edu/csit\\_etds](https://repository.stcloudstate.edu/csit_etds)

---

## Recommended Citation

Pan, Shengti, "The Performance Comparison of Hadoop and Spark" (2016). *Culminating Projects in Computer Science and Information Technology*. 7.

[https://repository.stcloudstate.edu/csit\\_etds/7](https://repository.stcloudstate.edu/csit_etds/7)

This Starred Paper is brought to you for free and open access by the Department of Computer Science and Information Technology at theRepository at St. Cloud State. It has been accepted for inclusion in Culminating Projects in Computer Science and Information Technology by an authorized administrator of theRepository at St. Cloud State. For more information, please contact [rswexelbaum@stcloudstate.edu](mailto:rswexelbaum@stcloudstate.edu).

# **The Performance Comparison of Hadoop and Spark**

by

Shengti Pan

A Starred Paper

Submitted to the Graduate Faculty of

St. Cloud State University

in Partial Fulfillment of the Requirements

for the Degree

Master of Science in

Computer Science

May, 2016

Starred Paper Committee:  
Jie Hu Meichsner, Chairperson  
Donald Hammes  
Jim Chen

## **Abstract**

The main focus of this paper is to compare the performance between Hadoop and Spark on some applications, such as iterative computation and real-time data processing. The runtime architectures of both Spark and Hadoop will be compared to illustrate their differences, and the components of their ecosystems will be tabled to show their respective characteristics. In this paper, we will highlight the performance comparison between Spark and Hadoop as the growth of data size and iteration counts, and also show how to tune in Hadoop and Spark in order to achieve higher performance. At the end, there will be several appendixes which describes how to install and launch Hadoop and Spark, how to implement the three case studies using java programming, and how to verify the correctness of the running results.

## **Keywords**

MapReduce, RDD, latency, sorting, rank, executor, optimization

### **Acknowledgement**

I would like to thank my advisor Dr. Meichsner for offering a lot of valuable suggestions to my work. Without her help and coordination during my laboratory work, it is difficult that my whole progress has gone so smoothly. I also thank the committee members Dr. Hammes and Dr. Chen for spending their time and energy in correcting my paper and providing beneficial suggestions. For the laboratory work, Martin Smith provides support for setting up hardware and system environment. I thank him for his time and patience.

## Table of Contents

	Page
List of Tables .....	6
List of Figures .....	7
Chapter	
1. Introduction .....	8
1.1 The Overview of Hadoop and Spark .....	8
1.2 Runtime Architecture .....	14
1.3 The Overview of Ecosystems in Spark and Hadoop .....	16
2. The Evaluation of Performance .....	18
2.1 Laboratory Environment .....	18
2.2 Laboratory Network Deployment .....	19
2.3 Case Studies for Evaluation .....	19
2.4 The Evaluation of Running Results .....	26
3. Optimization of Hadoop and Spark .....	32
3.1 Tuning in Hadoop .....	33
3.2 Tuning in Spark .....	36
4. Conclusions .....	38
References .....	39
Appendices	
A. Guide to Installing Hadoop .....	42
B. Guide to Installing Spark .....	45

	5
Chapter	Page
C. The Source Code of Case Studies .....	47

## List of Tables

Table	Page
1. The Components of Hadoop and Spark .....	13
2. The Ecosystems of Hadoop and Spark .....	17
3. Datasets for Word Count–Sorted by Keys .....	20
4. Datasets for PageRank Example .....	25
5. Running Times for the Case Study of Word Count .....	28
6. Running Times for the Case Study of Secondary Sort .....	29
7. Running Times for the Case Study of PageRank .....	29
8. Running Times for Word Count–Sorted by Keys on Spark .....	30
9. Running Times for Word Count–Sorted by Values on Spark .....	30
10. Running Times for PageRank on Spark .....	30
11. The Running Times with Default Configuration Settings .....	32
12. The Running Times with Optimization I in Hadoop .....	33
13. The Running Times with Optimization II in Hadoop .....	34
14. The Running Times with Optimization II and III in Hadoop .....	36
15. The Running Times with Optimization I in Spark .....	37
16. The Running Times with Optimization I and II in Spark .....	37

## List of Figures

Figure	Page
1. Hadoop Components .....	8
2. MapReduce Logical Data Flow .....	10
3. Logical Data Flow in Spark .....	12
4. Spark Components .....	13
5. Hadoop Architecture .....	14
6. Spark Architecture .....	16
7. The Ecosystem of Hadoop .....	16
8. The Ecosystem of Spark .....	16
9. The Network Deployment of Hadoop and Spark .....	19
10. The Data Sample of Word Count–Sorted by Keys .....	20
11. Algorithms of Word Count in Hadoop and Spark .....	21
12. The Data Sample of Word Count–Sorted by Values .....	22
13. The Data Sample of PageRank .....	25
14. The Example of Calculating the Rank Value of a Link .....	26
15. Spark Application Web UI .....	27
16. Hadoop Application Web UI .....	27



## Chapter 1: Introduction

In this chapter, an overview of Hadoop and Spark is introduced to get a basic understanding of their frameworks, including their key components and how data flows in MapReduce and Spark respectively. Next, their runtime architectures are dissected to better comprehend how an application works in Hadoop and Spark separately. In addition, their ecosystems are illustrated to show the characteristic and functionality of each element.

### 1.1 The Overview of Hadoop and Spark

Apache Hadoop is a framework for the distributed processing of big data across clusters of computers using MapReduce programming data model [1]. Figure 1 shows the four core components of Apache Hadoop: MapReduce, Hadoop Utilities, YARN (Yet Another Resource Negotiator) and HDFS (Hadoop Distributed File System).

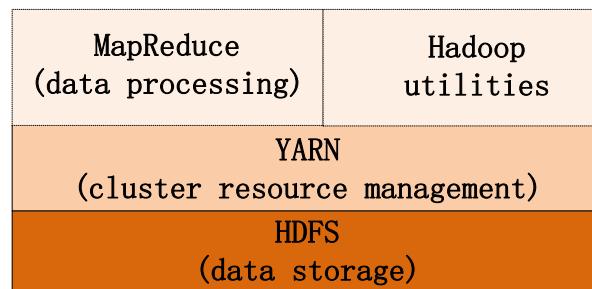


Figure 1: Hadoop Components

MapReduce is a programming model which provides support for parallel computing, locality-aware scheduling, fault-tolerance, and scalability on commodity clusters [2].

MapReduce separates the data processing into two stages: the map stage and the reduce stage.

The data flow in MapReduce in Figure 2 is as follows [3]:

1. Each split file is corresponding to a map task which is responsible for transforming input records into intermediate records [1], and the mapper outputs the result to a circular memory buffer (the default size is 100M);
2. When the data in the circular buffer is approaching a threshold size (80% of the default size), the mapper starts to spill the contents in the buffer to a spill file on the local disk; before data are written to disk, a background thread divides the data into a few partitions, the number of which is corresponding to the number of reducers. Also, during partitioning, the data are sorted by key.
3. When the data fill up the circular memory buffer during spilling, the map task is blocked until all of the contents in the buffer are emptied;
4. Once one mapper completes its output, a reducer, which is responsible for reducing a set of intermediate results that share the same key to a smaller set of results [1], starts to fetch a particular partition. This process of transferring data from the outputs of mapper to the inputs of reducer is called data shuffle, which is an all-map-to-all-reduce personalized communication [4], and Hadoop uses its internal algorithm to implement this shuffle processing.
5. After shuffling is done, the reducer starts to merge the partitions;
6. And then the reduce function is invoked to process the merged data;
7. Finally, the reduce function outputs the result on HDFS.

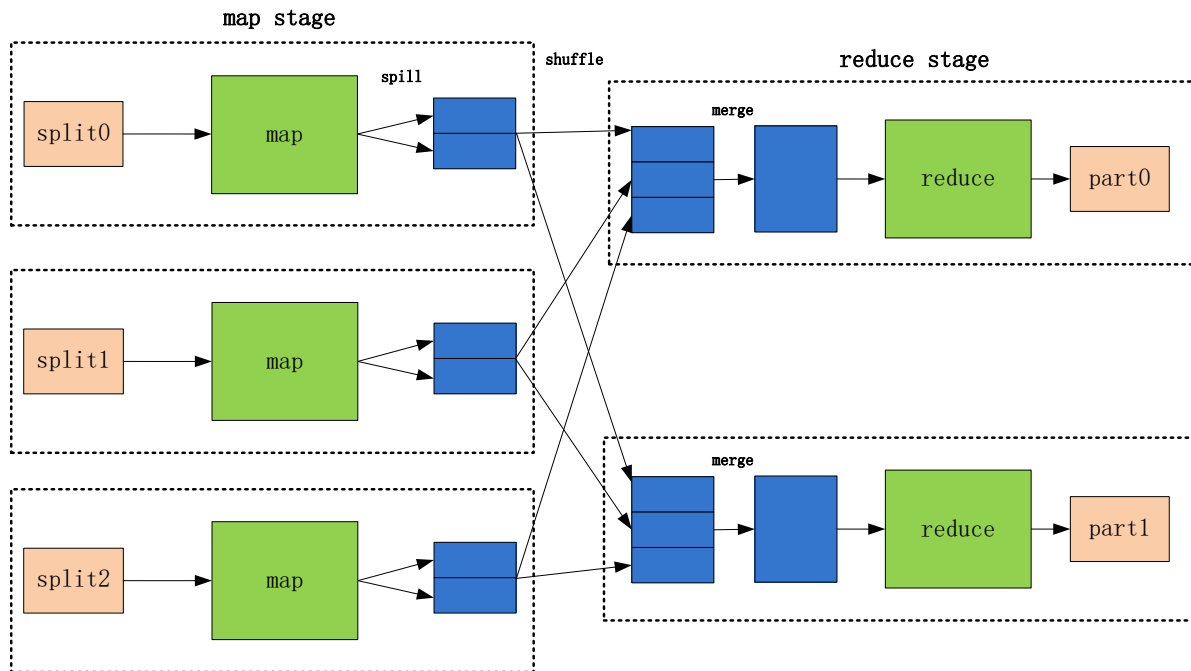


Figure 2: MapReduce Logical Data Flow [3]

YARN is a cluster resource management framework in Hadoop [5]. YARN includes two key daemons: a resource manager to schedule jobs and tasks or allocate resources across the cluster, and node managers to start and monitor containers. A container is a Java virtual machine instance, where an application or MapReduce tasks run with a given memory, CPU and other resources.

HDFS is a file system which stores big data, links data blocks logically, and streams data at high bandwidth to applications in a distributed system [6]. It separates file system metadata from application data. The former are presented on NameNode, and the later are stored on DataNode. Also, HDFS replicates data across clusters to achieve reliability in case of failure of nodes [7].

Hadoop is believed to be reliable, scalable, and fault-tolerant. It is well known that MapReduce is a good fit for applications of processing big data, but it is a poor fit for iteration

algorithms and low-latency computations because MapReduce relies on persistent storage to provide fault-tolerance, and requires the entire data set to be loaded into system before running analytical queries [8]. So that is why Spark was born.

Spark is a cluster computing framework and an engine for large-scale data processing. It constructs a distributed collections of objects, resilient distributed datasets (RDDs) in memory, and then performs a variety of operations in parallel on these datasets. Spark greatly outperforms Hadoop MapReduce by 10x in iterative machine learning tasks [9] and is up to 20x faster for iterative applications [10].

Spark is an alternative to the MapReduce framework. It is mainly used for real-time data stream processing or applied to iterative algorithms. RDDs is a distributed memory abstraction [10], and each RDD is a read-only, partitioned collection of elements across the cluster that can be operated on in parallel [9]. The immutability of a RDD means that changes to the current RDD will create a new RDD, and makes caching and sharing it easy. When operations are executed on each RDD, the number of partitions in it determines the degree of parallelism. RDDs can be created by two ways: loading datasets from external resource, such as HDFS, or parallelizing a collection in the driver program [5]. There are two types of operations in processing RDDs: transformations and actions. The operation of transformations builds a new RDD from a previous one, but actions compute a result based on an RDD, and then either return it to the driver program or save it to an external storage system. Also, transformations on RDDs are lazily evaluated [5], which means Spark does not execute those operations immediately, but only records the metadata of how to calculate the data. Once one action is invoked, Spark starts to execute all of operations. In this way, Spark decreases the

number of transfers of raw data between nodes. The logic data flow in Figure 3 is as follows

[5]:

1. An RDD is created by parallelizing an existing collection of data in the driver program or loading a dataset from the external storage system, such as HDFS or HBase;
2. And then the RDD may be transformed lazily a couple of times, which means the results are not computed at once and are just recorded to apply to the dataset;
3. Once an action is called, all of transformations are computed. Also, each time a new action is called, the entire RDD must be computed from the starting point. So intermediate results can be persisted in memory using a `cache()` or `persist()` method;
4. At last, the output is returned to the driver program.

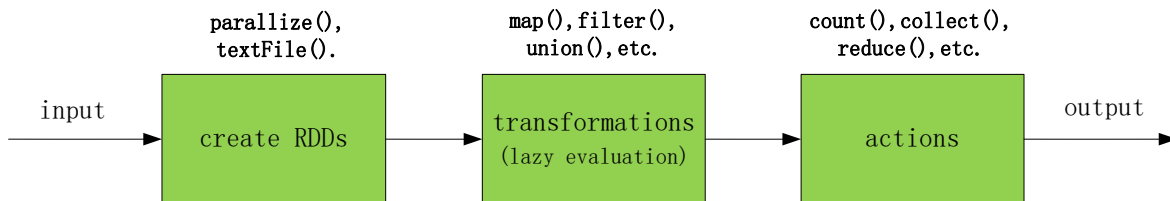


Figure 3: Logical Data Flow in Spark

Spark provides multiple optional modes for the resource management framework and the file system according to the specific context. For example, if Spark is installed on an empty set of machines, the Standalone cluster manager makes it easy to get started. But if a Hadoop system already exists, and a spark application needs to be set up to access HDFS in Hadoop, it is better to make Spark run on YARN because YARN provides support for security and better integration with its resource management policies [11]. Figure 4 shows the

architecture of a spark system, and the descriptions of the components in Spark are listed in the Table 1, which summarizes the functionalities of the components in Hadoop and Spark [1] [9].

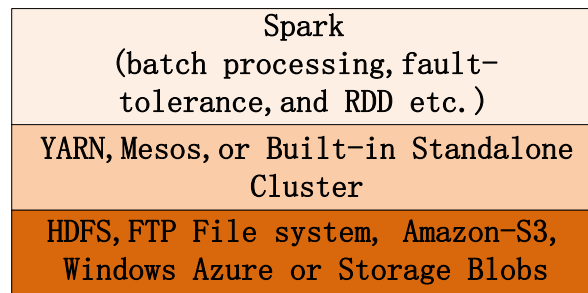


Figure 4: Spark Components

Table 1: The Components of Hadoop and Spark

Framework	Components	Description
Hadoop	Hadoop utilities	They provide support for other Hadoop modules.
	Hadoop Distributed File System (HDFS)	A filesystem aims to store a large scale data sets and provide high-throughput access to them, running on a distributed commodity hardware [1].
	Yet Another Resource Negotiator (YARN)	A cluster resource manage system that schedules jobs and manages the allocation of resources across the cluster [1].
	MapReduce	A framework for parallel processing of big data.
Spark	Spark SQL	It integrates the SQL queries into Spark programs. The Spark SQL Server can be connected through JDBC or ODBC. Also, Spark SQL is compatible with the existing Hive data warehouse [9].
	Spark Streaming	This component makes it easy to build scalable fault-tolerant streaming applications [9].
	Machine Learning Lib	MLlib helps users to create and tune the machine learning pipelines [9].
	Graphx:	This is a graph computation engine, which is used for graphs or graphs in parallel computation [9].
	Apache Spark Core Component	It includes batch processing, APIs, fault-tolerance, and Resilient Distributed Datasets (RDD).

## 1.2 Runtime Architecture

### ❖ Running a MapReduce job in Hadoop

As shown in Figure 5, first, a driver program creates a jobClient and this jobClient asks the ResourceManager for an application ID. Once it gets the ID, the jobClient copies the resources from HDFS, including the libs which the application needs to run, and the configuration files. Next, the jobClient submits the application to ResourceManager and requests the YARN scheduler to allocate a container in a NodeManager node, where the ResourceManager launches the application master that initializes the application job, and creates a map task for each split and reduce tasks. If it is a small job, the map and reduce tasks will be executed in parallel in the local node. Usually, a small job is one that owns less than 10 map tasks, only one reduce task and the size of each input file is less than the HDFS block size [3]. Otherwise, the application master will send a request to ResourceManager and ask for more containers to run MapReduce tasks.

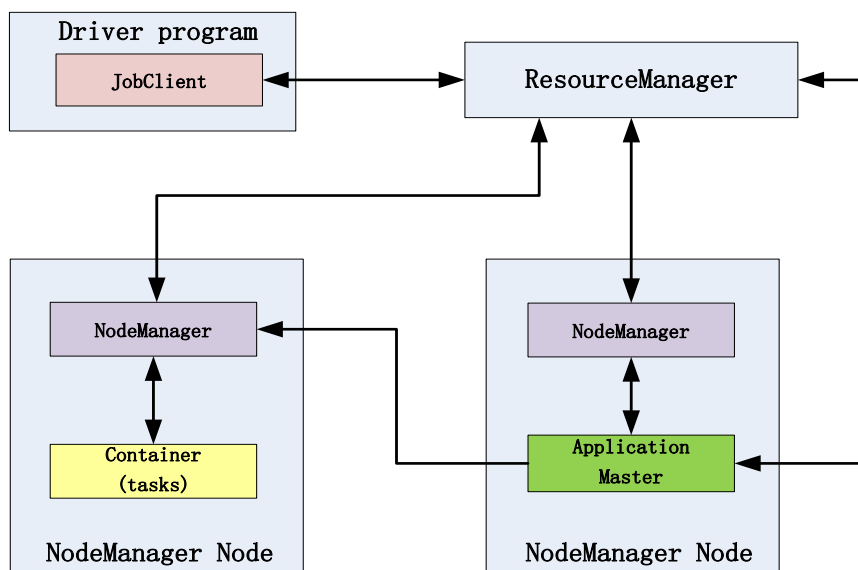


Figure 5: Hadoop Architecture [3]

### ❖ **Running a driver program in Spark**

Spark takes advantage of a master-slave architecture. A Spark application includes a central coordinator (driver) and a number of workers (executors). The driver is either the process where the `main()` function of the program runs, or when a Spark shell is launched, a driver program will be created. By default, the driver runs in the “client” mode which means the submitter starts the driver outside the cluster [9], and worker nodes are responsible for running executor processes, but the driver program can also be shipped to execute on any worker node by specifying the “cluster” mode. In addition, the driver and each executor are separated Java processes.

The driver has two duties: one duty is to convert a user program into units of tasks, which are the smallest unit of work in Spark [5]; the other is to schedule tasks on executors, which are started at the beginning of a Spark application. Executors have two roles, one is to be responsible for running the tasks and then return status to the driver; the other is to provide memory-based storage for RDDs. As shown in Figure 6: a driver program first creates `SparkContext` and connects to Cluster Manager and then the cluster manager allocates resources, such as executors, for the application. Next, the application code is sent to the executor. Finally, `SparkContext` delivers tasks to the executors to run [5].



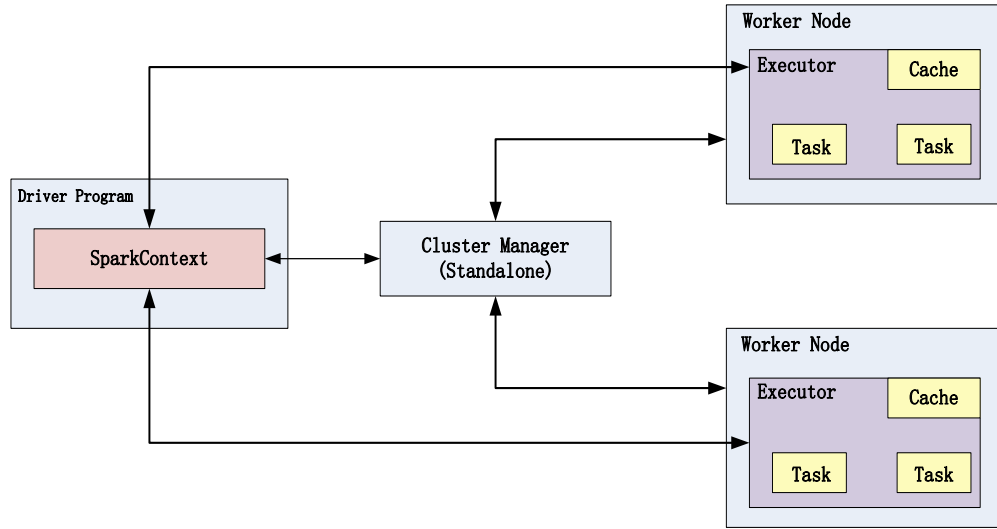


Figure 6: Spark Architecture [5]

### 1.3 The Overview of Ecosystems in Spark and Hadoop

As Hadoop and Spark are evolving, they have built their own respective ecosystems, and each of them establishes a set of businesses and their interrelationships in a common framework [12]. As a result, neither Spark nor Hadoop ecosystem is an individual product, but they are a collection of components, whose interrelationships are illustrated in Figures 7 and 8. Their descriptions and functionalities are listed in the Table 2. Currently, because Spark is younger than Hadoop, it has fewer components.

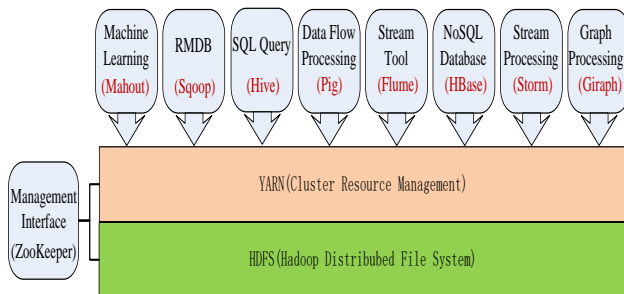


Figure 7: The Ecosystem of Hadoop

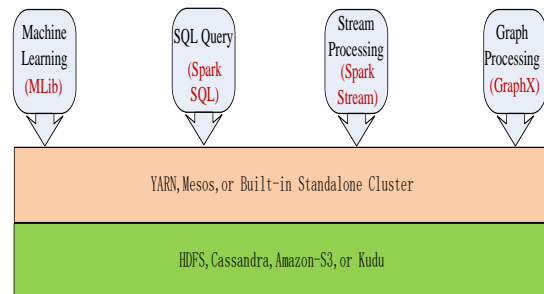


Figure 8: The Ecosystem of Spark

Table 2: The Ecosystems of Hadoop and Spark

<b>Ecosystem</b>	<b>Hadoop</b>	<b>Spark</b>
<b>Elements</b>		
Distributed File Systems	HDFS, FTP File system, Amazon-S3, Windows Azure Storage Blobs	HDFS, Cassandra, Amazon-S3, Kudu
Distributed Resource Management	YARN framework	It is adaptable. YARN, Mesos, or Built-in Standalone Manager which provides the easiest way to run applications on a cluster [5].
SQL Query	<b>HIVE:</b> A data warehouse component	SPARK SQL
Machine Learning	<b>Mahout:</b> A Machine learning component	MLib
Stream Processing	<b>Storm:</b> real-time computational engine	Spark Stream
Graph Processing	Giraph: A framework for large-scale graph processing	GraphX
Management Interface	<b>ZooKeeper:</b> A management tool for Hadoop cluster.	No support
Stream tool	<b>Flume:</b> a service for efficiently transferring streaming data into the Hadoop Distributed File System (HDFS).	No support
Pluggable to RMDB	<b>Sqoop:</b> transfer data between Relational Database Management System (RDBMS) and Hadoop	No support
Data Flow Processing	<b>Pig:</b> a high level scripting data flow language which expresses data flows by applying a series of transformations to loaded data [12].	No support
NoSQL database	<b>HBase:</b> based on BigTable, and column-oriented	No support

## Chapter 2: The Evaluation of Performance

In order to show that Spark is faster than Hadoop on performance, a laboratory work is conducted in a cluster with eight virtual machines during break (machines are relatively idle), where Hadoop and Spark are installed and deployed. Three case studies based on the same algorithm and programming language are employed to run on this cluster. The running times of each case study on Hadoop system and Spark system are tabled to exhibit the performance difference.

### 2.1 Laboratory Environment

#### ❖ Hardware configuration (8 virtual machines)

Hostname	IP	CPU	Storage
shengti1	10.59.7.151	For each IP: AMD quad-core Opteron 2348 @ 2.7GHZ	NFS raid array, WD Re 4TB 7200 RPM 64MB Cache SATA 6.0Gb/s 3.5"
shengti2	10.59.7.152		
shengti3	10.59.7.153		
shengti4	10.59.7.154		
shengti5	10.59.7.155	For each IP: AMD 12 core Opteron 6174 @ 2.2 GHZ	local disk, WD Re 1TB 7200 RPM 64MB Cache SATA 6.0Gb/s 3.5"
shengti6	10.59.7.156		
shengti7	10.59.7.157		
shengti8	10.59.7.158		
Storage size on each machine: 100G Memory size on each Machine: 6G			

#### ❖ Software configuration

Software Name	Version
Operating System	Ubuntu 12.04, 32-bit
Apache Hadoop	2.7.1
Apache Spark	1.4
JRE	Java(TM) SE Runtime Environment (build 1.8.0_66-b17)
SSH	openssh_5.9p1
virtualization platform	VMware vSphere 5.5

## 2.2 Laboratory Network Deployment

In this project, both Hadoop and Spark are deployed on 8 virtual machines. The master node IP is 10.59.7.151, and other slave nodes or workers are from 10.59.7.152 ~ 10.59.7.158. For the Hadoop system, the namenode process and the YARN cluster manager are launched on the master node, and each slave node is responsible for launching its own datanode process. For the Spark system, the master process and the built-in standalone cluster are started on the master node, each worker is responsible for launching the executor process. The network topology of the two systems is as shown in Figure 9:.

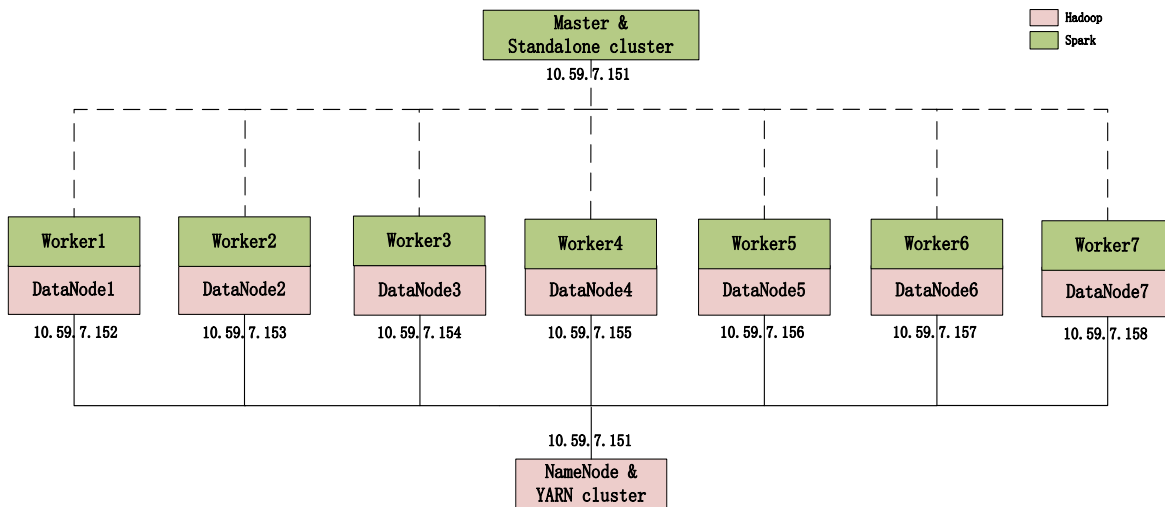


Figure 9: The Network Deployment of Hadoop and Spark

## 2.3 Case Studies for Evaluation

A couple of cases are used for evaluation in this project. Each case uses different number of iterations, but the examples of each case running on Hadoop and Spark are based on the same algorithm. In this case, with the same configurations of hardware and the default

settings of Hadoop and Spark, a resulting comparison on performance should be feasible and reasonable.

### 2.3.1 Word Count—Sorted by Keys

The classic example of word count is provided for both Hadoop and Spark, which make use of MapReduce to do the job. The data source is generated by a program which randomly picks words from a dictionary file which includes 5000 English words, and places one word on each line in the output file. The following Table 3 shows the data size used in this case study and the next case study Word Count—Sorted by Values.

Table 3: Datasets for Word Count—Sorted by Keys

Sample Name	Sample Size
wc_100M.txt	99.96MB
wc_500M.txt	499.72MB
wc_1G.txt	999.44MB
wc_2G.txt	1.95GB

#### ❖ Data Sample

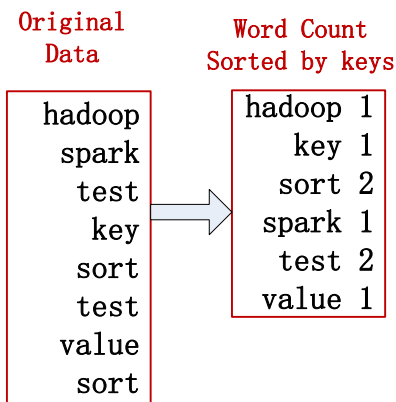


Figure 10: The Data Sample of Word Count—Sorted by Keys

### ❖ Algorithm Description

Algorithm 1 in Figure 11 shows how the Word Count case study is implemented in Hadoop. At first, the map function splits one line at a time into words using a token, and then outputs key-value pairs by the format  $\langle \langle \text{word} \rangle, 1 \rangle$  as the input data of the reduce function. Before the reduce function is called, the keys are sorted with the default dictionary order. Next, the reduce function sums up the counts for each unique word. At last, the reduce function outputs the result on HDFS.

Algorithm 2 in Figure 11 shows how the Word Count case study is implemented in Spark. At first, an RDD is created by loading data from HDFS using the function `textFile()`. Next, a few transformation functions, such as `flatMap()`, `map()`, and `reduceByKey()`, are invoked to record the metadata of how to process the actual data. At last, all of transformations are called to compute the actual immediate data once an action like the function `saveAsText()` is called.

Algorithm 1: Word Count in Hadoop	Algorithm 2: Word Count in Spark
<pre> 1: class Mapper&lt;K1, V1, K2, V2&gt; 2:   function map(K1, V1) 3:     List words = V1.splitBy(token); 4:     foreach K2 in words 5:       write(K2, 1); 6:     end for 7:   end function 8: end class  1: class Reducer&lt;K2, V2, K3, V3&gt; 2:   function reduce(K2, Iterable&lt;V2&gt; itor) 3:     sum = 0; 4:     foreach count in itor 5:       sum += count; 6:     end for 7:     write(k3, sum); 8:   end function 9: end class </pre>	<pre> 1: class WordCount 2:   function main(String[] args) 3:     file = sparkContext.textFile(filePath); 4:     JavaRDD&lt;String&gt; words = flatMap &lt;- file; 5:     JavaPairRDD&lt;String, Integer&gt; pairs = map &lt;- words; 6:     JavaPairRDD&lt;String, Integer&gt; counts = reduceByKey &lt;- pairs 7:     result = sortByKey &lt;- counts; 8:   end function 9: end class </pre>

Figure 11: Algorithms of Word Count in Hadoop and Spark

### 2.3.2 Word Count—Sorted By Values

By default, the output of the example of Word Count is sorted by key with dictionary order. In the following example in Figure 12, the default sorting function will be overridden by an integer sorting function to sort values. As shown in the Figure 12, there are three jobs which complete the whole program. The first job outputs the same immediate data as what the first case study does. The second job swaps key-value pairs, and then uses the integer sorting function to sort frequencies. At last, the third job is to group the immediate data by frequency and then sort by words.

#### ❖ Data Sample

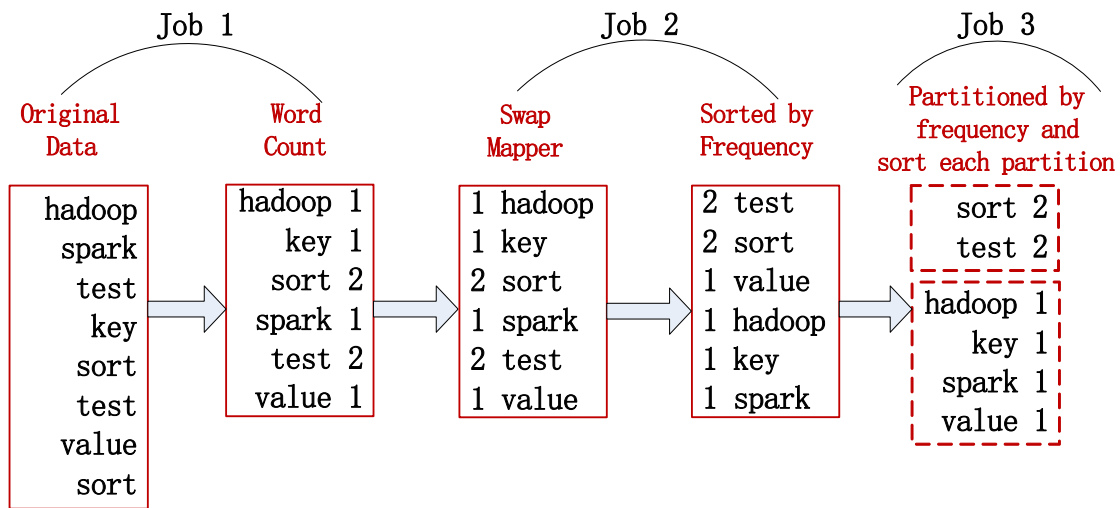


Figure 12: The Data Sample of Word Count—Sorted by Values

#### ❖ Algorithm Description

1. Based on the case study of word count, swap the key and the value of Mapper;
2. Overwrite the comparator of Mapper, and sort frequency with integer type;
3. Partition the running result by frequency;
4. Swap the key and the value of each partition;

5. Sort each partition by words;
6. Merge the partition;
7. Finally, output the result.

### **2.3.3 Iterative algorithm**

Here, PageRank is chosen to show the difference of performance between Hadoop and Spark because of the following reasons:

- 1) the implementation of PageRank algorithm is involved in multiple iterations of computation;
- 2) in Hadoop, for each iteration computation, MapReduce always writes immediate data back to HDFS (Hadoop distributed file system) after a map or reduce action.
- 3) However, Spark processes immediate data in an in-memory cache.

In this case, for such a specific application, it is obviously expected that Spark would completely overwhelm Hadoop on performance.

#### **❖ Algorithm Description**

Because this paper mainly focus on the comparison of performance between Spark and Hadoop, a common PageRank algorithm will be used in this case study.

PageRank appears along with the development of web search engines. It is considered as the probability that a user, who is given a random page and clicks links at random all the time, eventually gets bored and jumps to another page at random [14]. As result, it is used to calculate a quality ranking for each page in the link structure of the web, and thus improves the precision of searching results. This is the way that Google searching engine evaluates the quality of web pages.



Basically, the core idea of PageRank is as follows [14]:

- I. If a page is linked by a large number of other pages, it is much more important than a page linked by a few others, and this page also owns higher rank value;
- II. If a page is linked by another page with higher rank value, its rank value is improved;
- III. The final goal of this algorithm is to find stable ranks for all of links after multiple iterations.

In this case study, a PageRank value will be calculated by the following formula [15], which was proposed by the Google founders Brin and Page in 1998:

$$R_i = d * \sum_{j \in S} (R_j / N_j) + (1 - d)$$

$R_i$  : The PageRank value of the link i

$R_j$  : The PageRank value of the link j

$N_j$  : The number of outgoing links of link j pointing to its neighbor links

$S$  : The set of links that point to the link i

$d$  : The damping factor (usually,  $d = 0.85$ )

#### **2.3.4 Sample data preparation and running results**

The sample datasets in the Table 4 are used to evaluate the performances of the PageRank application respectively running in Hadoop and Spark. All of the data source are from <http://snap.stanford.edu/data/> [16].

Table 4: Datasets for PageRank Example

Sample Name	Sample Size	Nodes	Edges
web-NotreDame.txt	20.56MB	325,729	1,497,134
web-Google.txt	73.6MB	875,713	5,105,039
as-Skitter.txt	142.2MB	1,696,415	11,095,298

## ❖ Data Sample

The data sample shown in Figure 13 is from web-Google.txt. Each line represents a directed edge. The left column represents starting link points, and the right column is the ending link points.

From LinkId	To LinkId
0	11342
0	824020
0	867932
11342	0
11342	27469
11342	23689
11342	867932
824020	0
824020	91807

Figure 13: The Data Sample of PageRank

Let us take some sample data as an example to show how PageRank values are calculated. Initially, the PageRank value of each link is equal to 1.0, and then their final values are calculated as follows:

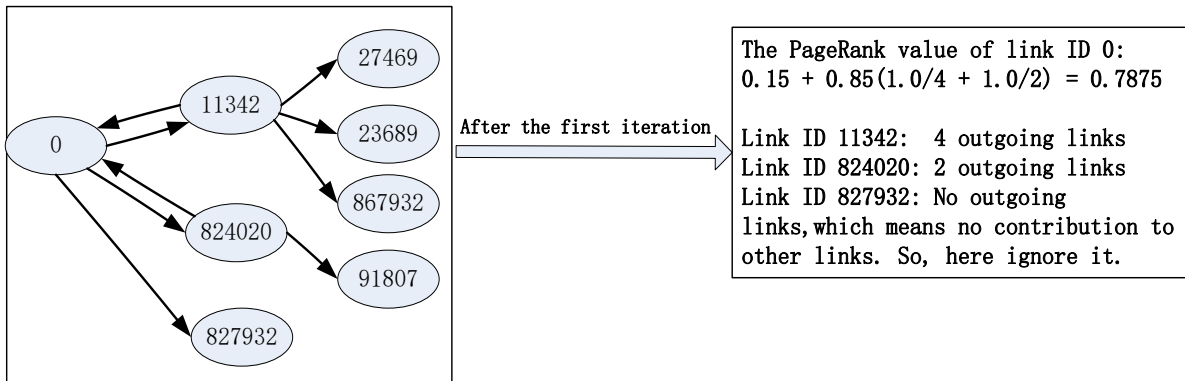


Figure 14: The Example of Calculating the Rank Value of a Link

## 2.4 The Evaluation of Running Results

### 2.4.1 Performance Measurement and Metrics

For the three case studies above, their performance in Hadoop and in Spark are compared by the running times. To keep a fair comparison, we guarantee the following metrics which are applied to Hadoop and Spark:

- ❖ Hadoop and Spark platforms run on the same cluster machines;
- ❖ Both Hadoop and Spark use HDFS as the file storage system;
- ❖ Case studies implemented in Hadoop and Spark are based on the same programming language and algorithm;
- ❖ At last, we take advantage of Hadoop Application Web UI and Spark Application Web UI where the Start Time and Finish Time are listed to calculate the elapsed time of a Hadoop or Spark application, as shown in the Figures 15 and 16:

Running Applications							
Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
<b>Completed Applications</b>							
Application ID	Name	Cores	Memory per Node	Submitted Time	User	State	Duration
app-20160124174056-0002	PageRank	14	3.0 GB	2016/01/24 17:40:56	student	FINISHED	1.1 min
app-20160124173957-0001	WordCount Secondary Sort	14	3.0 GB	2016/01/24 17:39:57	student	FINISHED	23 s
app-20160124173913-0000	WordCount	14	3.0 GB	2016/01/24 17:39:13	student	FINISHED	21 s

Figure 15: Spark Application Web UI

Show 20 entries							
ID	User	Name	Application Type	Queue	StartTime	FinishTime	State
application_1453677815715_0002	student	WCSortValues for wc_500M.txt	MAPREDUCE	default	Sun Jan 24 17:27:23 -0600 2016	Sun Jan 24 17:28:41 -0600 2016	FINISHED
application_1453677815715_0001	student	WordCount for wc_500M.txt	MAPREDUCE	default	Sun Jan 24 17:25:02 -0600 2016	Sun Jan 24 17:26:26 -0600 2016	FINISHED

Figure 16: Hadoop Application Web UI

Since the goal of this paper is not to focus on how to achieve the convergence of PageRank, 15 iterations are applied to each dataset in Hadoop and Spark respectively to just make sure we are able to evaluate the time difference between Hadoop and Spark.

#### 2.4.2 The Comparison of Running Results

In this project, each case study was repeated more than 10 times of testing to obtain the average running results. Sometimes because of unstable network traffic, there are a few seconds of error band for a small job, or tens of seconds of error band for a big job. Tables 5, 6, and 7 show the average running time comparison based on different sizes of data for each case study in Hadoop and in Spark. The observations are as follows:

- ❖ In the first case study Word Count, for a small data size which is less than the default block size of 128MB, there is a stable performance ratio between Spark and Hadoop because the data is processed in the local node no matter whether it is in Hadoop or in Spark. However, as the growth of data size, i.e., more split blocks are generated, there is an increasing performance ratio between Spark and Hadoop.

Here performance ratio (PR) is defined as:

$$PR = \frac{\text{The running time of a given data size in Spark}}{\text{The running time of the same data size in Hadoop}}$$

- ❖ In the second case study, Word Count–Sorted by Values, the performance ratio value is bigger than that in the first case study because there is more than one iteration.
- ❖ When multiple iterations, such as 15 iterations, apply to Hadoop and Spark respectively, Spark has a compelling performance enhancement when being compared with Hadoop.

Table 5: Running Times for the Case Study of Word Count

<b>Data size</b>	100 MB	500 MB	1 GB	2 GB
<b>System</b>				
Hadoop	58 secs	1 min 12 secs	1 min 48 secs	2 mins 25 secs
Spark	16 secs	23 secs	25 secs	30 secs
	PR=3.63	PR=3.13	PR=4.32	PR=4.83

Table 6: Running Times for the Case Study of Secondary Sort

<b>Data size</b>	100 MB	500 MB	1 GB	2GB
<b>System</b>				
Hadoop	1 min 43 secs	1 min 56 secs	2 mins 27 secs	3 mins 2 secs
Spark	12 secs	22 secs	23 secs	30 secs
	PR=8.58	PR=5.27	PR=6.39	PR=6.07

Table 7: Running Times for the Case Study of PageRank

<b>Data size</b>	20.56 MB NotreDame	67.02 MB Google	145.62 MB as-skitter
<b>System</b>			
Hadoop	7 mins 22 secs	15 mins 3 secs	38 mins 51 secs
Spark	37 secs	1 min 18 secs	2 mins 48 secs
	PR = 11.95	PR = 11.58	PR=13.86

As mentioned before, Spark utilizes memory-based storage for RDDs but MapReduce in Hadoop processes disk-based operations, so it stands to reason that the performance of Spark outperforms that of Hadoop. However, Spark allows to limit the memory usage of each executor by assigning `spark.executor.memory` to a proper value, e.g., 2 GB. Therefore, as the memory usage limit is varied between 1 and 3 GB on each executor, comparable running results are listed in the Table 8, 9, and 10 (Time Unit: seconds), and we have the following observations:

- ❖ For small size of data or fewer iterations, increasing memory does not contribute to the improvement of performance, as shown in Tables 8 and 9.

- ❖ As the growth of data size and multiple iterations are executed, there is a significant performance improvement with the increasing memory usage by setting the value of spark.executor.memory, as shown in Table 9.

Table 8: Running Times for Word Count–Sorted by Keys on Spark

<b>Data size</b> <b>Memory usage (GB)</b>	100 MB	500MB	1GB	2GB
3	16 secs	24 secs	25 secs	29 secs
2	16 secs	24 secs	24 secs	30 secs
1	16 secs	23 secs	25 secs	31 secs

Table 9: Running Times for Word Count–Sorted by Values on Spark

<b>Data size</b> <b>Memory usage (GB)</b>	100 MB	500 MB	1 GB	2 GB
3	16 secs	24 secs	25 secs	31 secs
2	17 secs	23 secs	27 secs	29 secs
1	15 secs	23 secs	24 secs	31 secs

Table 10: Running Times for PageRank on Spark

<b>Data size</b> <b>Memory usage (GB)</b>	20.56 MB NotreDame	67.02 MB Google	145.62 MB as-skitter
3	36 secs	78 secs	162 secs
2	37 secs	78 secs	180 secs
1	33 secs	114 secs	780 secs

Based on the same memory usage, Spark still performs better than Hadoop (The default memory for a map task is 1GB [1]). The reasons mainly result from the following factors [17]:

- 1) Spark workloads have a higher number of disk accesses per second than Hadoop's;
- 2) Spark has a better memory bandwidth utilization than Hadoop;
- 3) Spark achieves higher IPCs than Hadoop;

Also, in Spark, task scheduling is based on an event-driven mode but Hadoop employs heartbeat to tracking tasks, which periodically causes a few seconds delays [18].

Moreover, in Hadoop, there is overhead to complete the minimal requirements of job setup, starting tasks, and cleaning up because of the minimum overhead of the Hadoop software stack [10]. For some applications involved in iterative algorithm, Hadoop is totally overwhelmed by Spark because multiple jobs in Hadoop cannot share data and have to access HDFS frequently [19].



### Chapter 3: Optimization of Hadoop and Spark

Hadoop provides over hundreds of default parameter configuration settings for clusters and applications [20], and Spark allows users to customize dozens of properties for application tuning. Changing some default values may have an impact on other performance because of their interconnections. For example, in Hadoop, assume that given proper map tasks, if the number of reducers is set to a higher value, tasks will be processed in parallel in the cluster but the process of shuffling data between mapper and reducer will cause a lot of overhead because the shuffle phase includes network transferring, in-memory merging and any on-disk merging. However, if the number of reducers is set to one, the bandwidth of the network will limit heavy data transfer and degrade the whole performance of the application. So, optimization of parameter or property configuration settings in Hadoop and Spark not only depends on application characteristics but also the hardware environment itself on which the cluster runs.

Here, we take the first case study Word Count–Sorted by Keys as an example of tuning some parameter configuration settings to see how its performance will be improved step by step when running on Hadoop and Spark respectively. In order to make a comparison between each step, the running time on Hadoop and Spark respectively with default configuration settings is shown as follows:

Table 11: The Running Times with Default Configuration Settings

<b>Data size</b>	100 MB	500 MB	1 GB	2 GB
<b>System</b>				
Hadoop	55 secs	1 min 44 secs	1 min 59 secs	3 mins 53 secs
Spark	18 secs	26 secs	28 secs	34 secs

### 3.1 Tuning in Hadoop

#### ❖ Optimization I: data Compression

By default, Hadoop provides support for three types of data compression: gzip, bzip2 and LZO, and each of them has different compression ratios and speeds [3].

In this case study, a better performance is achieved for a bigger size of data by setting the following factors:

- a) `mapreduce.map.output.compress = true` (false by default)
- b) `mapreduce.map.output.compress.codec = gzip`

For a small size of data, up to 2GB data, the performance is degraded due to the cost of compressing and decompressing data as shown below:

Table 12: The Running Times with Optimization I in Hadoop

<b>Data size</b>	100 MB	500 MB	1 GB	2 GB
<b>System</b>				
Hadoop	1 min 3 secs	2 mins 4 secs	2 mins 10 secs	3 mins 23 secs

#### ❖ Optimization II: Change memory management and YARN parameter configuration

From the running results with default configuration settings, we have two observations in Hadoop:

- 1) The map stage dominates the total running time;
- 2) With the growth of data, the cost of reducer stage becomes bigger;

During a map stage, the mapper output is first buffered in memory and then spilled to disk after a threshold value is exceeded. The cost of outputting mapper results to disk is decided by two factors: `mapreduce.task.io.sort.mb`, the size of in-memory buffer, and `mapreduce.map.sort.spill.percent`, the threshold of the buffer before its content is spilled to disk. Therefore, we first optimize the map stage through modifying the following factors in `mapred-site.xml`:

- a) Enlarge the size of the circular memory buffer to 500M(100M by default) by setting the property `mapreduce.task.io.sort.mb`;
- b) Set the number of shuffling copies in parallel to 20 (5 by default) by setting the property `mapreduce.reduce.shuffle.parallelcopies`;
- c) Reduce data spilling by setting `mapreduce.map.sort.spill.percent = 0.95` (0.8 by default).

After running the first case Word Count–Sorted by Keys, a better performance improvement based on the default configuration is achieved as shown in the following Table. But for the data size which is less than the block size such as 100M, the running time is stable (sometimes a few seconds of error band) because the data is always processed in the local node.

Table 13: The Running Times with Optimization II in Hadoop

<b>Data size</b>	100 MB	500 MB	1 GB	2 GB
<b>System</b> Hadoop	53 secs	1 min 18 secs	1 min 43 secs	2 min 27 secs

### ❖ **Optimization III: Change data partitioning configuration**

The number of reduce tasks is one of the most basic Hadoop MapReduce parameters, so optimizing the number of reducers will contribute to performance improvement [21]. In Hadoop, by default there is a single reducer [3], namely a single partition.

Increasing the number of reducers improves load balancing and lowers the cost of failures, but increases the cost of network communication between nodes. The right number of reducers is suitable when it is about  $0.95$  or  $1.75 * (\text{Number of Nodes} * \text{Number of Maximum containers per node})$

The factor of  $0.95$  means that all of reduces can launch immediately and start transferring map outputs as the maps finish [1]. The factor of  $1.75$  means that the faster nodes will finish their first round of reduces and launch a second wave of reduces doing a much better job of load balancing [1].

In our laboratory environment, the memory size of each node is 6G, but the free memory for each work node is less than 3.5G after starting some necessary processes or daemons, such as a NodeManager process and a DataNode process in Hadoop, and a worker process in Spark. In the configuration, we set

`mapreduce.map.memory.mb = 1204M` for each mapper container

and

`mapreduce.reduce.memory.mb = 2560M` for each reducer container.

As a result, there are at most two mapper containers or one reducer container which are able to be launched in a node. So, a suitable reducer number for the case

studies should be about 6~8 or 12~14. However, based on the configuration of optimization II, different reducer numbers are tried and the best performance improvement achieved is showed in the following table, where the reducer number is listed in the parenthesis:

Table 14: The Running Times with Optimization II and III in Hadoop

<b>Data size</b>	100 MB	500 MB	1 GB	2 GB
<b>System</b>				
Hadoop	53 secs (reducer = 1)	1 min 16 secs (reducer = 1)	1 min 34 secs (reducer = 6)	2 mins 23 secs (reducer = 13)

### 3.2 Tuning in Spark

Spark provides support for three types of data compression: lz4, lzf, and snappy [9].

By default, the codec used to compress RDD partitions is snappy.

#### ❖ Optimization I: Change memory management

By default, the number of cores used for the driver process is 1, and its memory usage is set to 1GB. Also, the amount of memory used for each executor process is 1GB [9]. In order to take full advantage of in-memory processing to improve the computing efficiency, it is important to allocate enough memory needed for RDDs to avoid possible slowdown of the execution [22]. Here, when we set

```
spark.driver.cores = 2,
```

```
spark.driver.memory = 2G,
```

```
and spark.executor.memory = 3G,
```

the best performance is achieved as follows:

Table 15: The Running Times with Optimization I in Spark

<b>Data size</b>	100 MB	500 MB	1 GB	2 GB
<b>System</b>				
Spark	15 secs	25 secs	26 secs	31 secs

#### ❖ Optimization II: data partition

In this project, only one partition is output for each case study running in Hadoop or Spark. But this way will degrade the performance because of network bottleneck when there are a number of map tasks. Therefore, a proper partitioning will contribute to the performance improvement. In Spark, the number of partitions of each RDD is decided by the number of map tasks, which means that each mapper output is corresponding to one partition of an RDD. Repartitioning data in Spark is a fairly expensive operation [5], so a better performance is achieved by the default partitions based on the optimization I.

Table 16: The Running Times with Optimization I and II in Spark

<b>Data size</b>	100 MB	500 MB	1 GB	2 GB
<b>System</b>				
Spark	16 secs	23 secs	24 secs	29 secs

## Chapter 4: Conclusions

From the laboratory work, we find that Spark totally overshadows Hadoop on performance in all of case studies, especially those involved in the iterative algorithm. We conclude that several factors can give a rise to a significant performance difference. First of all, Spark pipelines RDDs transformations and keeps persistent RDDs in memory by default, but Hadoop mainly concentrates on high throughput of data rather than on job execution performance [23] such that MapReduce results in overheads due to data replication, disk I/O, and serialization, which can dominate application execution times. Also, in order to achieve fault-tolerance efficiently, RDDs provide a coarse-grained transformations rather than fine-grained updates to shared state or data replication across cluster [10], which means Spark builds the lineage of RDDs through transformations rather than the actual data. For example, if a partition of an RDD is missing, the RDD can retrieve the information about how it was originated from other RDDs. Last but not least, Spark has more optimizations, such as the number of disk accesses per second, memory bandwidth utilization and IPC rate, than Hadoop, so that it provides a better performance.

Spark is generally faster than Hadoop because it is at the expense of significant memory consumption. But Spark is not a good fit for applications that make asynchronous fine-grained updates to shared state [10]. Also, if we do not have sufficient memory and the speed is not a demanding requirement, Hadoop is a better choice. For those applications which are time sensitive or involved in iterative algorithms and there is abundant memory available, Spark is sure to be the best fit.

## References

- [1] *MapReduce Tutorial*, 2015, <http://hadoop.apache.org/>.
- [2] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: cluster computing with working sets," in *Proceedings of the 2nd USENIX Conference on Hot Topics in Cloud Computing (HotCloud'10)*, USENIX Association, Berkeley, CA, 2010, p. 10-10.
- [3] T. White, *Hadoop: The Definitive Guide* (Fourth edition). Sebastopol, CA: O'Reilly Media, 2015.
- [4] F. Ahmad, S. Lee, M. Thottethodi, and T. Vijaykumar, "MapReduce with communication overlap (MaRCO)," *Journal of Parallel and Distributed Computing*, pp. 608-620, n.d.
- [5] H. Karau, A. Konwinski, P. Wendell, and M. Zaharia, *Learning Spark*. Sebastopol, CA: O'Reilly Media, 2015.
- [6] 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.
- [7] H. Ahmed, M. A. Ismail, and M. F. Hyder, "Performance optimization of Hadoop cluster using Linux services," in *Multi-Topic Conference (INMIC), 2014 IEEE 17th International*, pp.167-172, December 8-10, 2014.
- [8] B. Li, E. Mazur, Y. Diao, A. McGregor, and P. Shenoy, "A platform for scalable one-pass analytics using MapReduce," in *Proceedings of the 2011 International Conference on Management of Data-SIGMOD '11*.
- [9] *Spark Overview*, 2015, <http://spark.apache.org/>.
- [10] 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 *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI'12)*, USENIX Association, Berkeley, CA, 2012, pp. 2-2.
- [11] 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.



- [12] E. Yu and S. Deng, "Understanding software ecosystems: A strategic modeling approach," in *Proceedings of the Workshop on Software Ecosystems 2011*, 746(IWSECO2011), 2011, p. 6-6.
- [13] S. Brin and L. Page, The anatomy of a large-scale hypertextual web search engine, December 3, 2015, <http://infolab.stanford.edu/~backrub/google.html>.
- [14] G. Rumi, C. Colella, and D. Ardagna, "Optimization techniques within the Hadoop ecosystem: a survey," in *2014 16th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, September 22-25, 2014, pp. 437-444.
- [15] M. Bianchini, M. Gori, and F. Scarselli, "Inside PageRank," *ACM Trans. Inter. Tech. TOIT ACM Transactions on Internet Technology*, pp. 92-128, 2005.
- [16] Stanford University, "Stanford large network dataset collection," 2009, <http://snap.stanford.edu/>, November 10, 2015.
- [17] 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.
- [18] X. Lin, P. Wang, and B. Wu, "Log analysis in cloud computing environment with Hadoop and Spark," in *2013 5th IEEE International Conference on Broadband Network & Multimedia Technology (IC-BNMT)*, November 17-19, 2013, pp. 273-276.
- [19] 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.
- [20] B. Mathiya and V. Desai, "Apache Hadoop yarn parameter configuration challenges and optimization," in *2015 International Conference on Soft-Computing and Networks Security (ICSNS)*.
- [21] 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.
- [22] K. Wang and M. M. H. Khan, "Performance prediction for Apache Spark platform," in *2015 IEEE 12th International Conference on Embedded Software and Systems (ICISS)*, *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.

- [23] 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.

## Appendix A: Guide to Installing Hadoop

This project is conducted under the software configuration in Chapter 2.1 Laboratory Environment. Therefore, before installing Hadoop 2.7.1, make sure that the operating system Ubuntu 14.0.2 and all of other software are already set up, and then follow the steps one by one:

- ❖ Step 1: Download Hadoop-2.7.1 and unpack it.

```
student@shengti1:~$ wget http://mirrors.ibiblio.org/apache/hadoop/common/hadoop-2.7.1/hadoop-2.7.1.tar.gz
```

```
student@shengti1:~$ tar -xvf Hadoop-2.7.1
```

- ❖ Step 2: Edit /etc/profile and set HADOOP\_PREFIX, JAVA\_HOME and HADOOP\_CLASSPATH.

```
HADOOP_PREFIX=/home/student/hadoop-2.7.1
export HADOOP_PREFIX
export JAVA_HOME=/usr/lib/jvm/java-8-oracle
export PATH=${JAVA_HOME}/bin:${PATH}
export HADOOP_CLASSPATH=${JAVA_HOME}/lib/tools.jar
```

- ❖ Step 3: Set up passphraseless SSH on the master machine.

```
student@shengti1:~/hadoop-2.7.1$ ssh-keygen -t dsa -P '' -f ~/.ssh/id_dsa
```

```
student@shengti1:~/hadoop-2.7.1$ cat ~/.ssh/id_dsa.pub >> ~/.ssh/authorized_keys
```

- ❖ Step 4: copy id\_dsa.pub on the master machine to all other slave machines and make sure that the master machine can access all of slave machines through SSH service without password.

```
student@shengti1:~/hadoop-2.7.1$ scp ~/.ssh/id_dsa.pub shengti2:~/.ssh/authorized_keys
```

- ❖ Step 5: Set up Hadoop cluster environment by editing the configuration files core-site.xml, hdfs-site.xml, mapred-site.xml, yarn-site.xml and slaves under the directory \$HADOOP\_HOME/etc/hadoop/.

```
<configuration>
  <property>
    <name>fs.defaultFS</name>
    <value>hdfs://10.59.7.151:9000</value>
  </property>
</configuration>
```

core-site.xml

```
<configuration>
  <property>
    <name>dfs.replication</name>
    <value>3</value>
  </property>
</configuration>
```

hdfs-site.xml

```
<configuration>
  <property>
    <name>mapreduce.framework.name</name>
    <value>yarn</value>
  </property>
</configuration>
```

yarn-site.xml

```
shengti2
shengti3
shengti4
shengti5
shengti6
shengti7
shengti8
```

slaves

```
<configuration>
  <property>
    <name>yarn.nodemanager.aux-services</name>
    <value>mapreduce_shuffle</value>
  </property>
  <property>
    <name>yarn.nodemanager.aux-services.mapreduce.shuffle.class</name>
    <value>org.apache.hadoop.mapred.ShuffleHandler</value>
  </property>
  <property>
    <name>yarn.resourcemanager.hostname</name>
    <value>10.59.7.151</value>
  </property>
</configuration>
```

mapred-site.xml

- ❖ Step 6: Copy the entire HADOOP\_HOME fold to the same path of each slave machine

```
student@shengti1:~$ scp -r hadoop-2.7.1/ shengti2:/home/student/
```

- ❖ Step 7: Start Hadoop Cluster in the \$HADOOP\_HOME

```
student@shengti1:~/hadoop-2.7.1$ sbin/start-all.sh
```

- ❖ Step 8: Verify if all daemons are launched on the master node and slave nodes.

When running the command “jps”, if we can see NameNode and ResourceManager processes listed as below, it shows that the master node works successfully.

When opening the URL like `http://10.59.7.151:50070/`, if we can see there are 7 live nodes, it shows that all slave nodes work successfully.

```
student@shengti1:~/hadoop-2.7.1$ jps
6147 SecondaryNameNode
10792 RunJar
10985 Jps
10300 ResourceManager
5901 NameNode
student@shengti1:~/hadoop-2.7.1$
```

## Summary

Security is off.

Safemode is off.

2363 files and directories, 2285 blocks = 4648 total filesystem object(s).

Heap Memory used 89.77 MB of 253 MB Heap Memory. Max Heap Memory is 1.3 GB.

Non Heap Memory used 64.17 MB of 65.56 MB Committed Non Heap Memory. Max Non Heap Memory is -1 B.

Configured Capacity:	688.13 GB
DFS Used:	12.31 GB (1.79%)
Non DFS Used:	35.69 GB
DFS Remaining:	640.13 GB (93.02%)
Block Pool Used:	12.31 GB (1.79%)
DataNodes usages% (Min/Median/Max/stdDev):	1.24% / 1.86% / 2.00% / 0.24%
<b>Live Nodes</b>	7 (Decommissioned: 0)
<b>Dead Nodes</b>	0 (Decommissioned: 0)
<b>Decommissioning Nodes</b>	0
<b>Total Datanode Volume Failures</b>	0 (0 B)

## Appendix B: Guide to Installing Spark

Before installing Spark-1.4.1, make sure that the operating system Ubuntu 14.0.2 and all of other software mentioned in Chapter 2.1 Laboratory Environment are already set up, and then follow the steps one by one:

- ❖ Step 1: Download Spark-1.4.1 and unpack it.

```
student@shengti1:~$ wget http://d3kbcqa49mib13.cloudfront.net/spark-1.4.1-bin-hadoop2.6.tgz
```

```
student@shengti1:~$ tar -xvf spark-1.4.1-bin-hadoop2.6.tgz
```

- ❖ Step 2: Because in this laboratory work, Spark needs to access HDFS and use the Standalone Manager to schedule resources and tasks, we add the following two properties to `$SPARK_HOME/conf/spark-env.sh`:

```
# - HADOOP_CONF_DIR, to point Spark towards Hadoop configuration files
HADOOP_CONF_DIR=/home/student/hadoop-2.7.1/etc/hadoop
# - SPARK_MASTER_IP, to bind the master to a different IP address or hostname
SPARK_MASTER_IP=10.59.7.151
```

- ❖ Step 3: Configure the slave file

```
# A Spark Worker will be started on each of the machines listed below.
shengti2
shengti3
shengti4
shengti5
shengti6
shengti7
shengti8
```

- ❖ Step 4: Copy the entire `SPARK_HOME` fold to the same path of each slave machine

```
student@shengti1:~$ scp -r spark-1.4.1-bin-hadoop2.6/ shengti2:/home/student/
```

- ❖ Step 5: Start Spark Cluster in the `$SPARK_HOME`

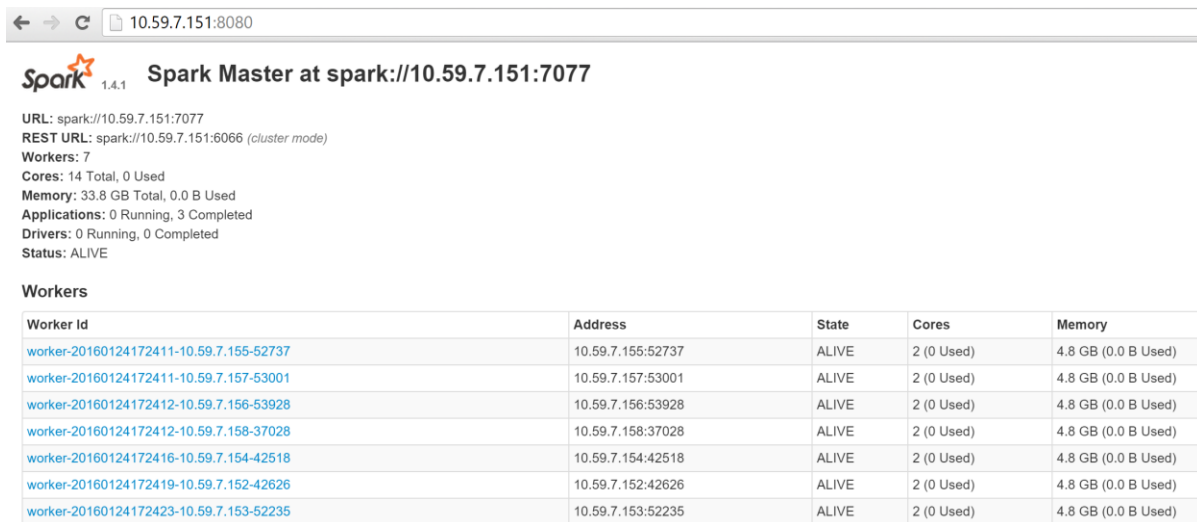
```
student@shengti1:~/spark-1.4.1-bin-hadoop2.6$ sbin/start-all.sh
```

- ❖ Step 6: Verify if all daemons are launched on the master node and woker nodes.

When running the command “jps”, if we can see a Master process listed as below, it shows that the master node works successfully.

```
student@shengti1:~/spark-1.4.1-bin-hadoop2.6$ jps
29190 Master
26907 Jps
student@shengti1:~/spark-1.4.1-bin-hadoop2.6$
```

When opening the URL like <http://10.59.7.151:8080/>, if we can see there are 7 worker nodes listed as below, it shows that all worker nodes works successfully.



Spark 1.4.1 Spark Master at spark://10.59.7.151:7077

URL: spark://10.59.7.151:7077  
 REST URL: spark://10.59.7.151:6066 (cluster mode)  
 Workers: 7  
 Cores: 14 Total, 0 Used  
 Memory: 33.8 GB Total, 0.0 B Used  
 Applications: 0 Running, 3 Completed  
 Drivers: 0 Running, 0 Completed  
 Status: ALIVE

**Workers**

Worker Id	Address	State	Cores	Memory
<a href="#">worker-20160124172411-10.59.7.155-52737</a>	10.59.7.155:52737	ALIVE	2 (0 Used)	4.8 GB (0.0 B Used)
<a href="#">worker-20160124172411-10.59.7.157-53001</a>	10.59.7.157:53001	ALIVE	2 (0 Used)	4.8 GB (0.0 B Used)
<a href="#">worker-20160124172412-10.59.7.156-53928</a>	10.59.7.156:53928	ALIVE	2 (0 Used)	4.8 GB (0.0 B Used)
<a href="#">worker-20160124172412-10.59.7.158-37028</a>	10.59.7.158:37028	ALIVE	2 (0 Used)	4.8 GB (0.0 B Used)
<a href="#">worker-20160124172416-10.59.7.154-42518</a>	10.59.7.154:42518	ALIVE	2 (0 Used)	4.8 GB (0.0 B Used)
<a href="#">worker-20160124172419-10.59.7.152-42626</a>	10.59.7.152:42626	ALIVE	2 (0 Used)	4.8 GB (0.0 B Used)
<a href="#">worker-20160124172423-10.59.7.153-52235</a>	10.59.7.153:52235	ALIVE	2 (0 Used)	4.8 GB (0.0 B Used)

## Appendix C: The Source Code of Case Studies

### ❖ Source Code: WordCountHadoop.java

```

1. /**
2.  * Copyright [2015] [Shengti Pan]
3.  *
4.  * Licensed under the Apache License, Version 2.0 (the "License");
5.  * you may not use this file except in compliance with the License.
6.  * You may obtain a copy of the License at
7.  *
8.  * http://www.apache.org/licenses/LICENSE-2.0
9.  *
10. * Unless required by applicable law or agreed to in writing, software
11. * distributed under the License is distributed on an "AS IS" BASIS,
12. * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
13. * See the License for the specific language governing permissions and
14. * limitations under the License.
15. */
16.
17.
18. import java.io.IOException;
19. import java.util.StringTokenizer;
20. import org.apache.hadoop.conf.Configuration;
21. import org.apache.hadoop.fs.Path;
22. import org.apache.hadoop.io.IntWritable;
23. import org.apache.hadoop.io.Text;
24. import org.apache.hadoop.mapreduce.Job;
25. import org.apache.hadoop.mapreduce.Mapper;
26. import org.apache.hadoop.mapreduce.Reducer;
27. import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
28. import org.apache.hadoop.mapreduce.lib.output.*;
29. import org.apache.hadoop.io.compress.*;
30.
31. /**
32.  * This Hadoop program is to implement counting the frequency
33.  * of words in a text file which is stored in HDFS.
34.  */
35.
36. public class WordCountHadoop {
37.
38.     private final static String rootPath = "/user/hadoop/";
39.

```



```
40. //map each word to a value one
41. public static class TokenizerMapper extends
42.     Mapper<Object, Text, Text, IntWritable> {
43.     private final static IntWritable one = new IntWritable(1);
44.     private Text word = new Text();
45.     public void map(Object key, Text value, Context context)
46.         throws IOException, InterruptedException {
47.         StringTokenizer itr = new StringTokenizer(value.toString());
48.         while (itr.hasMoreTokens()) {
49.             word.set(itr.nextToken());
50.             context.write(word, one);
51.         }
52.     }
53. }
54.
55. //reduce values by a unique word
56. public static class IntSumReducer extends
57.     Reducer<Text, IntWritable, Text, IntWritable> {
58.     private IntWritable result = new IntWritable();
59.     public void reduce(Text key, Iterable<IntWritable> values,
60.         Context context) throws IOException, InterruptedException {
61.         int sum = 0;
62.         for (IntWritable val : values) {
63.             sum += val.get();
64.         }
65.         result.set(sum);
66.         context.write(key, result);
67.     }
68. }
69.
70. public static void main(String[] args) throws Exception {
71.     //this program accepts two parameters by default;
72.     //if there is a third paramter, it is treated as the number of the reducers
73.     if(args.length < 2 || args.length > 3){
74.         System.out.println("Usage: wc.jar <input file> <output file> or");
75.         System.out.println("wc.jar <input file> <output file> <reduce number>");
76.         System.exit(1);
77.     }
78.
79.     //set up Hadoop configuration
80.     Configuration conf = new Configuration();
81.
82.     //set the compression format for map output
83.     conf.setBoolean(Job.MAP_OUTPUT_COMPRESS,true);
```

```

84.     conf.setClass(Job.MAP_OUTPUT_COMPRESS_CODEC,GzipCodec.class,
85.                 CompressionCodec.class);
86.
87.     //create a Hadoop job
88.     Job job = Job.getInstance(conf, "WordCount for " + args[0]);
89.     job.setJarByClass(WordCountHadoop.class);
90.     job.setMapperClass(TokenizerMapper.class);
91.     job.setCombinerClass(IntSumReducer.class);
92.     job.setReducerClass(IntSumReducer.class);
93.     job.setOutputKeyClass(Text.class);
94.     job.setOutputValueClass(IntWritable.class);
95.
96.     //set the number of reducers. By default, No. of reducers = 1
97.     if (args.length == 3) {
98.         job.setNumReduceTasks(Integer.parseInt(args[2]));
99.     }
100.    FileInputFormat.addInputPath(job, new Path(rootPath + "input/"
101.        + args[0]));
102.    FileOutputFormat
103.        .setOutputPath(job, new Path(rootPath + args[1]));
104.    System.exit(job.waitForCompletion(true) ? 0 : 1);
105. }
106. }

```

#### ❖ Source Code : WordCountSpark.java

```

1.  /**
2.   * Copyright [2015] [Shengti Pan]
3.   *
4.   * Licensed under the Apache License, Version 2.0 (the "License");
5.   * you may not use this file except in compliance with the License.
6.   * You may obtain a copy of the License at
7.   *
8.   * http://www.apache.org/licenses/LICENSE-2.0
9.   *
10.  * Unless required by applicable law or agreed to in writing, software
11.  * distributed under the License is distributed on an "AS IS" BASIS,
12.  * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
13.  * implied.
14.  * See the License for the specific language governing permissions and
15.  * limitations under the License.

```

```
15. */
16.
17. import scala.Tuple2;
18. import org.apache.spark.SparkConf;
19. import org.apache.spark.api.java.JavaPairRDD;
20. import org.apache.spark.api.java.JavaRDD;
21. import org.apache.spark.api.java.JavaSparkContext;
22. import org.apache.spark.api.java.function.FlatMapFunction;
23. import org.apache.spark.api.java.function.Function2;
24. import org.apache.spark.api.java.function.PairFunction;
25. import org.apache.spark.serializer.KryoRegistrar;
26. import java.util.Arrays;
27. import java.util.List;
28. import java.util.regex.Pattern;
29.
30.
31. /**
32.  * This spark program is to implement counting the frequency
33.  * of words in a text file which is stored in HDFS.
34.  */
35.
36. public class WordCountSpark {
37.     private static int num = 0;//the partition number
38.     //the path to access HDFS
39.     private final static String rootPath = "hdfs://10.59.7.151:9000/user/hadoop/";
40.
41.     public static void main(String[] args) throws Exception {
42.         //this program accepts two parameters by default;
43.         //if there is a third paramter, it is treated as the
44.         parameter of a partition number
45.         if(args.length < 2 || args.length > 3){
46.             System.out.println("Usage: wc.jar <input file> <output file> or");
47.             System.out.println("wc.jar <input file> <output file> <partition number>");
48.             System.exit(1);
49.         }
50.         if(args.length == 3)
51.             num = Integer.parseInt(args[2]);
52.
53.         //set up the configuration and context of this spark application
54.         SparkConf sparkConf = new SparkConf().setAppName("WordCountSpark");
55.         JavaSparkContext spark = new JavaSparkContext(sparkConf);
56.
57.         //words are split by space
58.         JavaRDD<String> textFile = spark
```

```

59.         .textFile(rootPath + "input/"
60.             + args[0]);
61.     JavaRDD<String> words = textFile
62.         .flatMap(new FlatMapFunction<String, String>() {
63.             public Iterable<String> call(String s) {
64.                 return Arrays.asList(s.split(" "));
65.             }
66.         });
67.
68.     //map each word to the value 1
69.     JavaPairRDD<String, Integer> wordsMap = words
70.         .mapToPair(new PairFunction<String, String, Integer>() {
71.             public Tuple2<String, Integer> call(String s) {
72.                 return new Tuple2<String, Integer>(s, 1);
73.             }
74.         });
75.
76.     //reduce the value of a unique word
77.     JavaPairRDD<String, Integer> freqPair = wordsMap
78.         .reduceByKey(new Function2<Integer, Integer, Integer>() {
79.             public Integer call(Integer a, Integer b) {
80.                 return a + b;
81.             }
82.         });
83.
84.     //if num == 0, using the default partition rule
85.     if(num == 0)
86.         freqPair.sortByKey().map(x -> x._1 + "\t" + x._2).
87.             saveAsTextFile(rootPath + args[1]);
88.     else
89.         //else manually assign the partition number
90.         freqPair.repartition(num).
91.             sortByKey().map(x -> x._1 + "\t" + x._2).
92.             saveAsTextFile(rootPath + args[1]);
93.
94.     //terminate the spark application
95.     spark.stop();
96. }
97. }

```

❖ Source Code : WCHadoopSortValues.java

1. /\*\*
2. \* This Hadoop program is to implement a secondary sort

```

3.  * of the WordCount example, which means that the final
4.  * result is not only sorted by frequency, but also sorted
5.  * by words.
6.  */
7.
8.  import java.util.*;
9.  import java.io.*;
10. import org.apache.hadoop.fs.FileSystem;
11. import org.apache.hadoop.io.*;
12. import org.apache.hadoop.conf.Configuration;
13. import org.apache.hadoop.fs.Path;
14. import org.apache.hadoop.mapreduce.lib.input.*;
15. import org.apache.hadoop.mapreduce.lib.output.*;
16. import org.apache.hadoop.mapreduce.*;
17. import org.apache.hadoop.mapreduce.Partitioner;
18. import org.apache.hadoop.mapreduce.lib.partition.HashPartitioner;
19. import org.apache.hadoop.mapreduce.lib.map.InverseMapper;
20.
21. public class WCHadoopSortValues {
22.
23.     public static String rootPath = "/user/hadoop/";
24.
25.     public static class TokenizerMapper extends
26.         Mapper<Object, Text, Text, IntWritable> {
27.
28.         private final static IntWritable one = new IntWritable(1);
29.         private Text word = new Text();
30.
31.         // map each word to a value one
32.         public void map(Object key, Text value, Context context)
33.             throws IOException, InterruptedException {
34.             StringTokenizer itr = new StringTokenizer(value.toString());
35.             while (itr.hasMoreTokens()) {
36.                 word.set(itr.nextToken());
37.                 context.write(word, one);
38.             }
39.         }
40.     }
41.
42.     // calculate the frequency of a unique word via reduce
43.     public static class IntSumReducer extends
44.         Reducer<Text, IntWritable, Text, IntWritable> {
45.         private IntWritable result = new IntWritable();
46.

```

```

47.     public void reduce(Text key, Iterable<IntWritable> values,
48.         Context context) throws IOException, InterruptedException {
49.         int sum = 0;
50.         for (IntWritable val : values) {
51.             sum += val.get();
52.         }
53.         result.set(sum);
54.         context.write(key, result);
55.     }
56. }
57.
58. // construct a map with a composite key, such as ((hadoop,1),null);
59. public static class SecondaryMapper extends
60.     Mapper<IntWritable, Text, CompositeKey, Text> {
61.     private Text word = new Text();
62.
63.     public void map(IntWritable value, Text key, Context context)
64.         throws IOException, InterruptedException {
65.         context.write(new CompositeKey((Text) key, value), word);
66.     }
67. }
68.
69. // implement a comparator for the comparison between two integers
70. private static class IntWritableComparator extends IntWritable.Comparator {
71.     public int compare(WritableComparable a, WritableComparable b) {
72.         return -super.compare(a, b);
73.     }
74.
75.     public int compare(byte[] b1, int s1, int l1, byte[] b2, int s2, int l2) {
76.         return -super.compare(b1, s1, l1, b2, s2, l2);
77.     }
78. }
79.
80. public static void main(String[] args) throws Exception {
81.     Configuration conf = new Configuration();
82.     Job job = Job.getInstance(conf, "WordCount for " + args[0]);
83.
84.     // save the immediate result into a temp file
85.     Path tempDir = new Path("temp_wc_" + System.currentTimeMillis());
86.     job.setJarByClass(WCHadoopSortValues.class);
87.     job.setMapperClass(TokenzierMapper.class);
88.     job.setCombinerClass(IntSumReducer.class);
89.     job.setReducerClass(IntSumReducer.class);
90.     job.setOutputKeyClass(Text.class);

```

```

91.     job.setOutputValueClass(IntWritable.class);
92.     FileInputFormat.addInputPath(job, new Path(rootPath + "input/"
93.         + args[0]));
94.     FileOutputFormat.setOutputPath(job, tempDir);
95.     job.setOutputFormatClass(SequenceFileOutputFormat.class);
96.
97.     // order by frequency
98.     if (job.waitForCompletion(true)) {
99.         Job job2 = new Job(conf, "sorted by frequency");
100.        job2.setJarByClass(WCHadoopSortValues.class);
101.
102.        FileInputFormat.addInputPath(job2, tempDir);
103.        job2.setInputFormatClass(SequenceFileInputFormat.class);
104.
105.        job2.setMapperClass(InverseMapper.class);
106.        FileOutputFormat.setOutputPath(job2, new Path(args[1]));
107.
108.        job2.setOutputKeyClass(IntWritable.class);
109.        job2.setOutputValueClass(Text.class);
110.        job2.setSortComparatorClass(IntWritableComparator.class);
111.        FileSystem.get(conf).deleteOnExit(tempDir);
112.        tempDir = new Path("temp_wc_" + System.currentTimeMillis());
113.        FileOutputFormat.setOutputPath(job2, tempDir);
114.        job2.setOutputFormatClass(SequenceFileOutputFormat.class);
115.
116.        // order by word
117.        if (job2.waitForCompletion(true)) {
118.            Job job3 = new Job(conf, "sorted by word");
119.            job3.setJarByClass(WCHadoopSortValues.class);
120.
121.            FileInputFormat.addInputPath(job3, tempDir);
122.            job3.setInputFormatClass(SequenceFileInputFormat.class);
123.            job3.setMapperClass(SecondaryMapper.class);
124.
125.            // set parameters for the job3, such as partitioner and
126.            // comparator
127.            job3.setMapOutputKeyClass(CompositeKey.class);
128.            job3.setPartitionerClass(KeyPartitioner.class);
129.            job3.setSortComparatorClass(CompositeKeyComparator.class);
130.            job3.setGroupingComparatorClass(KeyGroupingComparator.class);
131.
132.            FileOutputFormat.setOutputPath(job3, new Path(rootPath
133.                + args[1]));
134.            job3.setOutputKeyClass(IntWritable.class);

```

```

135.         job3.setOutputValueClass(Text.class);
136.
137.         System.exit(job3.waitForCompletion(true) ? 0 : 1);
138.     }
139. }
140.     FileSystem.get(conf).deleteOnExit(tempDir);
141. }
142. }
143.
144. // partitioned by frequency
145. class KeyPartitioner extends Partitioner<CompositeKey, Text> {
146.     HashPartitioner<IntWritable, Text> hashPartitioner =
147.         new HashPartitioner<IntWritable, Text>();
148.     IntWritable newKey = new IntWritable();
149.
150.     @Override
151.     public int getPartition(CompositeKey key, Text value, int numReduceTasks) {
152.
153.         try {
154.             return hashPartitioner.getPartition(key.getFrequency(), value,
155.                 numReduceTasks);
156.         } catch (Exception e) {
157.             e.printStackTrace();
158.             return (int) (Math.random() * numReduceTasks);
159.         }
160.     }
161. }
162.
163. // group words together by frequency order
164. class KeyGroupingComparator extends WritableComparator {
165.     protected KeyGroupingComparator() {
166.
167.         super(CompositeKey.class, true);
168.     }
169.
170.     @SuppressWarnings("rawtypes")
171.     @Override
172.     public int compare(WritableComparable w1, WritableComparable w2) {
173.         CompositeKey key1 = (CompositeKey) w1;
174.         CompositeKey key2 = (CompositeKey) w2;
175.         return key2.getFrequency().compareTo(key1.getFrequency());
176.     }
177. }
178.

```



```
179. // comparison between composite keys
180. class CompositeKeyComparator extends WritableComparator {
181.     protected CompositeKeyComparator() {
182.         super(CompositeKey.class, true);
183.     }
184.
185.     @SuppressWarnings("rawtypes")
186.     @Override
187.     public int compare(WritableComparable w1, WritableComparable w2) {
188.
189.         CompositeKey key1 = (CompositeKey) w1;
190.         CompositeKey key2 = (CompositeKey) w2;
191.         int cmp = key2.getFrequency().compareTo(key1.getFrequency());
192.         if (cmp != 0)
193.             return cmp;
194.         return key1.getWord().compareTo(key2.getWord());
195.     }
196. }
197.
198. // construct a composite key class
199. class CompositeKey implements WritableComparable<CompositeKey> {
200.     private Text word;
201.     private IntWritable frequency;
202.
203.     public CompositeKey() {
204.         set(new Text(), new IntWritable());
205.     }
206.
207.     public CompositeKey(String word, int frequency) {
208.
209.         set(new Text(word), new IntWritable(frequency));
210.     }
211.
212.     public CompositeKey(Text w, IntWritable f) {
213.         set(w, f);
214.     }
215.
216.     public void set(Text t, IntWritable n) {
217.         this.word = t;
218.         this.frequency = n;
219.     }
220.
221.     @Override
222.     public String toString() {
```

```
223.     return (new StringBuilder()).append(frequency).append(' ').append(word)
224.         .toString();
225.     }
226.
227.     @Override
228.     public boolean equals(Object o) {
229.         if (o instanceof CompositeKey) {
230.             CompositeKey comp = (CompositeKey) o;
231.             return word.equals(comp.word) && frequency.equals(comp.frequency);
232.         }
233.         return false;
234.     }
235.
236.     @Override
237.     public void readFields(DataInput in) throws IOException {
238.         word.readFields(in);
239.         frequency.readFields(in);
240.     }
241.
242.     @Override
243.     public void write(DataOutput out) throws IOException {
244.         word.write(out);
245.         frequency.write(out);
246.     }
247.
248.     @Override
249.     public int compareTo(CompositeKey o) {
250.         int result = word.compareTo(o.word);
251.         if (result != 0) {
252.             return result;
253.         }
254.         return result = frequency.compareTo(o.frequency);
255.     }
256.
257.     public Text getWord() {
258.         return word;
259.     }
260.
261.     public IntWritable getFrequency() {
262.         return frequency;
263.     }
264. }
```

## ❖ Source Code : WCSparkSortValues.java

```

1.  /**
2.   * This Spark program is to implement a secondary sort
3.   * of the WordCount example, which means that the final
4.   * result is not only sorted by frequency, but also sorted
5.   * by words.
6.   */
7.
8.  import scala.Tuple2;
9.  import org.apache.spark.SparkConf;
10. import java.util.*;
11. import org.apache.spark.api.java.*;
12. import org.apache.spark.Partitioner;
13. import org.apache.spark.HashPartitioner;
14. import org.apache.hadoop.conf.Configuration;
15. import org.apache.spark.api.java.function.FlatMapFunction;
16. import org.apache.spark.api.java.function.Function2;
17. import org.apache.spark.api.java.function.PairFunction;
18. import org.apache.hadoop.fs.*;
19. import java.io.Serializable;
20. import java.util.regex.Pattern;
21.
22.
23. public final class WCSparkSortValues {
24.     private static final Pattern SPACE = Pattern.compile(" ");
25.     private static String rootPath = "hdfs://10.59.7.151:9000/user/hadoop/";
26.
27.     public static void main(String[] args) throws Exception {
28.         SparkConf sparkConf = new SparkConf().setAppName("WordCount in Spark");
29.         JavaSparkContext spark = new JavaSparkContext(sparkConf);
30.
31.         FileSystem fs = FileSystem.get(spark.hadoopConfiguration());
32.         JavaRDD<String> textFile = spark.textFile(rootPath + "/input/" + args[0]);
33.         //load data for HDFS and split each line by space
34.         JavaRDD<String> words = textFile
35.             .flatMap(new FlatMapFunction<String, String>() {
36.                 public Iterable<String> call(String s) {
37.                     return Arrays.asList(s.split(" "));
38.                 }
39.             });
40.         //map each word to the value one
41.         JavaPairRDD<String, Integer> pairs = words
42.             .mapToPair(new PairFunction<String, String, Integer>() {

```

```

43.         public Tuple2<String, Integer> call(String s) {
44.             return new Tuple2<String, Integer>(s, 1); }
45.     });
46.
47.     //reduce by key, namely, the words.
48.     JavaPairRDD<String, Integer> counts = pairs
49.         .reduceByKey(new Function2<Integer, Integer, Integer>()
50.             {
51.                 public Integer call(Integer a, Integer b) { return a + b; }
52.             });
53.
54.     //sort by key
55.     JavaPairRDD<String, Integer> sortedByKeyList = counts.sortByKey(true);
56.
57.     //reverse key-to-value to value-to-key
58.     JavaPairRDD<Tuple2<Integer, String>, Integer> countInKey = sortedByKeyList
59.         .mapToPair(a -> new Tuple2(
60.             new Tuple2<Integer, String>(a._2, a._1), null));
61.
62.     //construct a composite RDD pair and also group by frequency
63.     JavaPairRDD<Tuple2<Integer, String>, Integer> groupAndOrderByvalues
64.     = countInKey.repartitionAndSortWithinPartitions(new MyPartitioner(1),
65.         new TupleComparator());
66.
67.     //extract the key of the composite RDD pair
68.     JavaRDD<Tuple2<Integer,String>> data = groupAndOrderByvalues.keys();
69.
70.     //convert JavaPairRDD to JavaRDD
71.     JavaPairRDD<Integer,String> results = JavaPairRDD.fromJavaRDD(data);
72.
73.     //make sure only one output and also format it
74.     results.repartition(1).map(s -> s._1 + "\t" + s._2)
75.         .saveAsTextFile(rootPath + args[1]);
76.
77.     //stop the spark application
78.     spark.stop();
79. }
80. }
81.
82. class TupleStringComparator implements
83.     Comparator<Tuple2<Integer, String>>, Serializable {
84.     @Override
85.     public int compare(Tuple2<Integer, String> tuple1,
86.         Tuple2<Integer, String> tuple2) {

```

```

87.     return tuple1._2.compareTo(tuple2._2);
88. }
89. }
90.
91. //construct a practitioner by frequency
92. class MyPartitioner extends Partitioner {
93.     private int partitions;
94.     public MyPartitioner(int partitions) {
95.         this.partitions = partitions;
96.     }
97.
98.     @Override
99.     public int getPartition(Object o) {
100.         Tuple2 newKey = (Tuple2) o;
101.         return (int) newKey._1 % partitions;
102.     }
103.
104.     @Override
105.     public int numPartitions() {
106.         return partitions;
107.     }
108. }
109.
110. //construct a key comparator in a composite RDD
111. class TupleComparator implements Comparator<Tuple2<Integer, String>>,
112.     Serializable {
113.     @Override
114.     public int compare(Tuple2<Integer, String> tuple1,
115.         Tuple2<Integer, String> tuple2) {
116.         return tuple2._1 - tuple1._1;
117.     }
118. }

```

❖ Source Code : PageRankHadoop.java

```

1. /**
2.  * This Hadoop program is to implement a PageRank algorithm,
3.  * which was proposed by the Google founders Brin and Page.
4.  * The datasouce is from SNAP (https://snap.stanford.edu/).
5.  */
6.
7. import java.io.IOException;
8. import java.text.*;
9. import java.util.*;

```

```

10. import org.apache.hadoop.io.*;
11. import org.apache.hadoop.mapreduce.Mapper;
12. import org.apache.hadoop.mapreduce.Reducer;
13. import org.apache.hadoop.conf.Configuration;
14. import org.apache.hadoop.fs.FileSystem;
15. import org.apache.hadoop.fs.Path;
16. import org.apache.hadoop.mapreduce.lib.input.*;
17. import org.apache.hadoop.mapreduce.lib.output.*;
18. import org.apache.hadoop.mapreduce.Job;
19.
20. public class PageRankHadoop {
21.     // utility attributes
22.     public static NumberFormat NF = new DecimalFormat("00");
23.     public static String LINKS_SEPARATOR = "|";
24.
25.     // configuration values
26.     public static Double DAMPING = 0.85;
27.     public static int ITERATIONS = 1;
28.     public static String INPUT_PATH = "/user/hadoop/input/";
29.     public static String OUTPUT_PATH = "/user/hadoop/";
30.
31.     // A map task of the first job: transfer each line to a map pair
32.     public static class FetchNeighborsMapper extends
33.         Mapper<LongWritable, Text, Text, Text> {
34.         public void map(LongWritable key, Text value, Context context)
35.             throws IOException, InterruptedException {
36.             //skip the comment line with #
37.             if (value.charAt(0) != '#') {
38.                 int tabIndex = value.find("\t");
39.                 String nodeA = Text.decode(value.getBytes(), 0, tabIndex);
40.                 String nodeB = Text.decode(value.getBytes(), tabIndex + 1,
41.                     value.getLength() - (tabIndex + 1));
42.                 context.write(new Text(nodeA), new Text(nodeB));
43.             }
44.         }
45.     }
46.
47.     // A reduce task of the first job: fetch the neighbor's links
48.     // and set the initial value of fromLinkId 1.0
49.     public static class FetchNeighborsReducer extends
50.         Reducer<Text, Text, Text, Text> {
51.         public void reduce(Text key, Iterable<Text> values, Context context)
52.             throws IOException, InterruptedException {
53.             boolean first = true;

```

```

54.     String links = "1.0\t";
55.     int count = 0;
56.     for (Text value : values) {
57.         if (!first)
58.             links += ",";
59.         links += value.toString();
60.         first = false;
61.         count++;
62.     }
63.     context.write(key, new Text(links));
64. }
65.
66. }
67.
68. // A map task of the second job:
69. public static class CalculateRankMapper extends
70.     Mapper<LongWritable, Text, Text, Text> {
71.     public void map(LongWritable key, Text value, Context context)
72.         throws IOException, InterruptedException {
73.
74.         int tIdx1 = value.find("\t");
75.         int tIdx2 = value.find("\t", tIdx1 + 1);
76.
77.         // extract tokens from the current line
78.         String page = Text.decode(value.getBytes(), 0, tIdx1);
79.         String pageRank = Text.decode(value.getBytes(), tIdx1 + 1, tIdx2
80.             - (tIdx1 + 1));
81.
82.         // Skip pages with no links.
83.         if (tIdx2 == -1)
84.             return;
85.
86.         String links = Text.decode(value.getBytes(), tIdx2 + 1,
87.             value.getLength() - (tIdx2 + 1));
88.         String[] allOtherPages = links.split(",");
89.         for (String otherPage : allOtherPages) {
90.             Text pageRankWithTotalLinks = new Text(pageRank + "\t"
91.                 + allOtherPages.length);
92.             context.write(new Text(otherPage), pageRankWithTotalLinks);
93.         }
94.
95.         // put the original links so the reducer is able to produce the
96.         // correct output

```

```

97.     context.write(new Text(page), new Text(PageRankHadoop.LINKS_SEPARAT
OR
98.         + links));
99.     }
100. }
101.
102. public static class CalculateRankReducer extends
103.     Reducer<Text, Text, Text, Text> {
104.     public void reduce(Text key, Iterable<Text> values, Context context)
105.         throws IOException, InterruptedException {
106.         String links = "";
107.         double sumShareOtherPageRanks = 0.0;
108.
109.         for (Text value : values) {
110.
111.             String content = value.toString();
112.
113.             //check if a linke has an appending 'links' string
114.             if (content.startsWith(PageRankHadoop.LINKS_SEPARATOR)) {
115.                 links += content.substring(PageRankHadoop.LINKS_SEPARATOR
116.                     .length());
117.             } else {
118.                 String[] split = content.split("\\t");
119.
120.                 // extract tokens
121.                 double pageRank = Double.parseDouble(split[0]);
122.                 if (split[1] != null && !split[1].equals("null")) {
123.                     int totalLinks = Integer.parseInt(split[1]);
124.
125.                     // calculate the contribution of each outgoing link
126.                     // of the current link
127.                     sumShareOtherPageRanks += (pageRank / totalLinks);
128.                 }
129.             }
130.
131.         }
132.         //get the final page rank of the current link
133.         double newRank = PageRankHadoop.DAMPING * sumShareOtherPageRanks
134.             + (1 - PageRankHadoop.DAMPING);
135.         //ignore the link which has no outgoing links
136.         if (newRank > 0.15000000000000002
137.             && !key.toString().trim().equals(""))
138.             context.write(key, new Text(newRank + "\t" + links));

```



```

139.     }
140. }
141.
142. // A map task of the third job for sorting
143. public static class SortRankMapper extends
144.     Mapper<LongWritable, Text, Text, DoubleWritable> {
145.     public void map(LongWritable key, Text value, Context context)
146.         throws IOException, InterruptedException {
147.
148.         int tIdx1 = value.find("\t");
149.         int tIdx2 = value.find("\t", tIdx1 + 1);
150.
151.         // extract tokens from the current line
152.         String page = Text.decode(value.getBytes(), 0, tIdx1);
153.         double pageRank = Double.parseDouble(Text.decode(value.getBytes(),
154.             tIdx1 + 1, tIdx2 - (tIdx1 + 1)));
155.         context.write(new Text(page), new DoubleWritable(pageRank));
156.     }
157.
158. }
159.
160. public static void main(String[] args) throws Exception {
161.
162.     if(args.length == 6){
163.         //set the iteration numbers
164.         if(args[0].equals("-c"))
165.             PageRankHadoop.ITERATIONS = Math.max(Integer.parseInt(args[1]), 1);
166.         else printHelp();
167.         //set input path
168.         if(args[2].equals("-i"))
169.             PageRankHadoop.INPUT_PATH = PageRankHadoop.INPUT_PATH + args[3
170.         ];
171.         else printHelp();
172.         //set output path
173.         if(args[4].equals("-o"))
174.             PageRankHadoop.OUTPUT_PATH = PageRankHadoop.OUTPUT_PATH + a
175.             rgs[5];
176.         else printHelp();
177.     }else{
178.         printHelp();
179.     }
180.
181.     String inPath = null;
182.     String lastOutPath = null;

```

```

181. PageRankHadoop pagerank = new PageRankHadoop();
182.
183. System.out.println("Start to fetch neighbor links ...");
184. boolean isCompleted = pagerank.job("fetchNeighborLinks",
185.     FetchNeighborsMapper.class, FetchNeighborsReducer.class,
186.     INPUT_PATH, OUTPUT_PATH + "/iter00");
187. if (!isCompleted) {
188.     System.exit(1);
189. }
190.
191. for (int runs = 0; runs < ITERATIONS; runs++) {
192.     inPath = OUTPUT_PATH + "/iter" + NF.format(runs);
193.     lastOutPath = OUTPUT_PATH + "/iter" + NF.format(runs + 1);
194.     System.out.println("Start to calculate rank [" + (runs + 1) + "/"
195.         + PageRankHadoop.ITERATIONS + "] ...");
196.     isCompleted = pagerank.job("jobOfCalculatingRanks",
197.         CalculateRankMapper.class, CalculateRankReducer.class,
198.         inPath, lastOutPath);
199.     if (!isCompleted) {
200.         System.exit(1);
201.     }
202. }
203.
204. System.out.println("Start to sort ranks ...");
205. isCompleted = pagerank.job("jobOfSortingRanks", SortRankMapper.class,
206.     SortRankMapper.class, lastOutPath, OUTPUT_PATH + "/result");
207. if (!isCompleted) {
208.     System.exit(1);
209. }
210.
211. System.out.println("All jobs done!");
212. System.exit(0);
213. }
214.
215. public boolean job(String jobName, Class m, Class r, String in, String out)
216.     throws IOException, ClassNotFoundException, InterruptedException {
217.     Configuration conf = new Configuration();
218.     Job job = Job.getInstance(conf, jobName);
219.     job.setJarByClass(PageRankHadoop.class);
220.
221.     // input / mapper
222.     FileInputFormat.addInputPath(job, new Path(in));
223.     job.setInputFormatClass(TextInputFormat.class);
224.     if (jobName.equals("jobOfSortingRanks")) {

```

```

225.     job.setOutputKeyClass(Text.class);
226.     job.setMapOutputValueClass(DoubleWritable.class);
227. } else {
228.     job.setMapOutputKeyClass(Text.class);
229.     job.setMapOutputValueClass(Text.class);
230. }
231.
232.     job.setMapperClass(m);
233.
234.     // output / reducer
235.     FileOutputFormat.setOutputPath(job, new Path(out));
236.     job.setOutputFormatClass(TextOutputFormat.class);
237.
238.     if (jobName.equals("jobOfSortingRanks")) {
239.         job.setOutputKeyClass(Text.class);
240.         job.setMapOutputValueClass(DoubleWritable.class);
241.     } else {
242.         job.setOutputKeyClass(Text.class);
243.         job.setOutputValueClass(Text.class);
244.         job.setReducerClass(r);
245.     }
246.
247.     return job.waitForCompletion(true);
248.
249. }
250.
251. //Print help message if the user does know how to run the program
252. public static void printHelp() {
253.     System.out.println("Usage: PageRank.jar -c <iterations>
254.                         -i <input file>
255.                         -o <output file> \n");
256. }
257. }

```

❖ Source Code : PageRankSpark.java

```

1. /**
2.  * Copyright [2015] [Shengti Pan]
3.  *
4.  * Licensed under the Apache License, Version 2.0 (the "License");
5.  * you may not use this file except in compliance with the License.
6.  * You may obtain a copy of the License at
7.  *
8.  * http://www.apache.org/licenses/LICENSE-2.0

```

```

9.  *
10. * Unless required by applicable law or agreed to in writing, software
11. * distributed under the License is distributed on an "AS IS" BASIS,
12. * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or
    implied.
13. * See the License for the specific language governing permissions and
14. * limitations under the License.
15. */
16.
17. import scala.Tuple2;
18. import com.google.common.collect.Iterables;
19. import org.apache.spark.SparkConf;
20. import org.apache.spark.api.java.JavaPairRDD;
21. import org.apache.spark.HashPartitioner;
22. import org.apache.spark.storage.StorageLevel;
23. import org.apache.spark.api.java.JavaRDD;
24. import org.apache.spark.api.java.JavaSparkContext;
25. import org.apache.spark.api.java.function.*;
26. import java.util.*;
27. import java.util.regex.Pattern;
28.
29. /**
30. * This Spark program is to implement a PageRank algorithm,
31. * which was proposed by the Google founders Brin and Page.
32. * The datasource is from SNAP (https://snap.stanford.edu/).
33. */
34.
35. public final class PageRankSpark {
36.   private static final Pattern SPACES = Pattern.compile("\\t+");
37.   private static final String ROOT_PATH = "hdfs://10.59.7.151:9000/user/hadoop/";
38.   private static final double DAMPING_FACTOR = 0.85d;
39.
40.   public static void main(String[] args) throws Exception {
41.     if (args.length < 3) {
42.       System.err.println("Usage: PageRankSpark <file> <iteration number> <output>");
43.       System.exit(1);
44.     }
45.
46.     SparkConf sparkConf = new SparkConf().setAppName("PageRankSpark");
47.     JavaSparkContext ctx = new JavaSparkContext(sparkConf);
48.     JavaRDD<String> lines = ctx.textFile(ROOT_PATH + "/input/" + args[0], 1);
49.
50.     //filter the data and ignor the comment line
51.     final JavaRDD<String> data = lines.filter(new Function<String, Boolean>(){

```

```

52.     public Boolean call(String s) {
53.         return !s.startsWith("#");
54.     }
55. });
56.
57. //load all links and fetch their neighbors.
58. JavaPairRDD<String, Iterable<String>> links = data.mapToPair(
59.     new PairFunction<String, String, String>() {
60.         @Override
61.         public Tuple2<String, String> call(String s) {
62.             String[] parts = SPACES.split(s);
63.             return new Tuple2<String, String>(parts[0], parts[1]);
64.         }
65.     }).groupByKey().persist(StorageLevel.MEMORY_ONLY());
66.
67. //initialize the rank value of each link to 1.0
68. JavaPairRDD<String, Double> ranks = links.mapValues(new Function<Iterable<String>,
69.     Double>() {
70.         @Override
71.         public Double call(Iterable<String> rs) {
72.             return 1.0;
73.         }
74.     });
75.
76. //calculate and update the ranks in multiple iterations
77. for (int current = 0; current < Integer.parseInt(args[1]); current++) {
78.     //calculate the contributions to its outgoing links of the current link.
79.     JavaPairRDD<String, Double> contribs = links.join(ranks).values()
80.     .flatMapToPair(new PairFlatMapFunction<Tuple2<Iterable<String>, Double>,
81.         String, Double>() {
82.         @Override
83.         public Iterable<Tuple2<String, Double>>
84.             call(Tuple2<Iterable<String>, Double> s) {
85.                 int urlCount = Iterables.size(s._1);
86.                 List<Tuple2<String, Double>> results = new
87.                     ArrayList<Tuple2<String, Double>>();
88.                 for (String n : s._1) {
89.                     results.add(new Tuple2<String, Double>(n, s._2() / urlCount));
90.                 }
91.                 return results;
92.             }
93.         });
94.

```

```
95. //get the final rank of the current link
96. ranks = contribs.reduceByKey(new Sum()).mapValues(new Function<Double, Double>()
97. {
98.     @Override
99.     public Double call(Double sum) {
100.         return (1.0 - DAMPING_FACTOR) + sum * DAMPING_FACTOR;
101.     }
102. });
103. }
104.
105. //sort and format the result
106. ranks.sortByKey().repartition(1).map(x -> x._1 + "\t" + x._2)
107.     .saveAsTextFile(ROOT_PATH + args[2]);
108. ctx.stop();
109. }
110.
111. private static class Sum implements Function2<Double, Double, Double> {
112.     @Override
113.     public Double call(Double a, Double b) { return a + b; }
114. }
115. }
```

## Appendix D: Verify the Correctness of Running Results

In order to make sure the correctness of programs and running results, we make use of Sqoop to export the running results on HDFS into Mysql database. Sqoop is one of components in Hadoop ecosystem, and it is able to transfer data on HDFS into tables in relational databases. In this case, it is easy to check the correctness and consistency of the records between two tables which respectively save the running results of Hadoop and Spark. Here, we take the second case study as an example to show how to verify the data step by step.

### ❖ Step 1: Mark the running results

Before exporting the running results on HDFS into Mysql, we first mark each row of the running results with a sequential number as follows:

```
GNU nano 2.2.6      File: marked-part-00000
1      22356  resonance
2      22355  invalid
3      22325  fragile
4      21991  head
5      11579  wretchedness
6      11550  precarious
7      11546  evert
8      11516  autonomous
9      11495  denomination
10     11492  scribe
11     11490  superintend
12     11484  transfigure
```

### ❖ Step 2: Create two tables for saving the marked running results of Hadoop and Spark.

```
mysql> create table hadoop_wc_sorted(seq_no int not null, freq int not null, word
varchar(50) not null, primary key(word));
Query OK, 0 rows affected (0.05 sec)

mysql> create table spark_wc_sorted(seq_no int not null, freq int not null, word
varchar(50) not null, primary key(word));
Query OK, 0 rows affected (0.02 sec)
```

### ❖ Step 3: Export the marked running results into tables in Mysql using Sqoop.

```
student@shengti1:~/sqoop-1.4.6$ bin/sqoop export --connect jdbc:mysql://10.59.7
.151:3306/hadoop --table hadoop_wc_sorted --m 1 --export-dir /user/hadoop/outpu
t_hadoop_wc_sort_values_500M/marked-part-r-00000 --input-fields-terminated-by '
\t' --username root --password stpan
```

- ❖ Step 4: Browse the data of tables in Mysql after exporting data is done.

```
Database changed
mysql> select * from hadoop_wc_sorted order by freq desc, word asc;
mysql> select * from spark_wc_sorted order by freq desc, word asc;
```

- ❖ Step 5: For all rows, verify that the sequence number on successive rows differ by one.

```
mysql> set @last_id := -1;
Query OK, 0 rows affected (0.00 sec)

mysql> select seq_no, freq, word, result
-> from
-> (select a.*, @last_id, if(@last_id < 0, null, seq_no-@last_id) as result
, @last_id :=seq_no from spark_wc_sorted a order by freq desc, word asc)
-> as tmp;
```

- ❖ Step 6: At last, check if the two tables have the same running results.

```
mysql> select * from spark_wc_sorted s, hadoop_wc_sorted h where s.seq_no=h
seq_no and s.freq=h.freq and s.word=h.word;
```