Ann Oper Res (2018) 265:305–317
https://doi.org/10.1007/s10479-016-2376-0

Improved performance optimization for massive small files in cloud computing environment

Chang Choi[1] · **Chulwoong Choi**[2] · **Junho Choi**[3] · **Pankoo Kim**[1]

Published online: 17 November 2016

Abstract Hadoop uses the Hadoop distributed file system for storing big data, and uses MapReduce to process big data in cloud computing environments. Because Hadoop is optimized for large file sizes, it has difficulties processing large numbers of small files. A small file can be defined as any file that is significantly smaller than the Hadoop block size, which is typically set to 64 MB. Hadoop is optimized to store data in relatively large files, and thus suffers from name node memory insufficiency and increased scheduling and processing time when processing large numbers of small files. This study proposes a performance improvement method for MapReduce processing, which integrates the CombineFileInputFormat method and the reuse feature of the Java Virtual Machine (JVM). Existing methods create a mapper for every small file. Unlike these methods, the proposed method reduces the number of created mappers by processing large numbers of files that are combined by a single split using CombineFileInputFormat. Moreover, to improve MapReduce processing performance, the proposed method reduces JVM creation time by reusing a single JVM to run multiple mappers (rather than creating a JVM for every mapper).

Keywords Massive small files · Hadoop · MapReduce · JVM reuse · CombineFileInputFormat

✉ Pankoo Kim
pkkim@chosun.ac.kr

Chang Choi
enduranceaura@gmail.com

Chulwoong Choi
sentilemon02@gmail.com

Junho Choi
xdman@chosun.ac.kr

1 Department of Computer Engineering, Chosun University, Gwangju, South Korea

2 Department of Software Convergence Engineering, Chosun University, Gwangju, South Korea

3 Division of Undeclared Majors, Chosun University, Gwangju, South Korea

1 Introduction

With the proliferation of smartphones and the expansion of social networking services (SNSs), numerous types of data are being generated in massive quantities by a vast array of sources; this has ushered in the era of big data. Because the amount of data is so vast and the types of data are so varied, existing systems, which have been developed only very recently, are unable to effectively process big data. Thus, an appropriate new system to process big data is required (Choi et al. 2014). Cloud computing is a more advanced technology for distributed processing, e.g., a thin client and grid computing, which is implemented by means of virtualization technology for servers and storages, and advanced network functionality (Ogiela and Ogiela 2003, 2010; Choi et al. 2014). Hadoop (Zikopoulos and Eaton 2011; Shvachko et al. 2010; Bhandarkar 2010) is a representative technology that stores and processes big data. Because Hadoop is optimized for big data's large files sizes, it has difficulty processing large numbers of small files. A small file can be defined as any file that is significantly smaller than the Hadoop block size, which is usually set to 64 MB. Hadoop encounters two core problems when it attempts to process large numbers of small files. The metadata of data files is stored in an in-memory database by name node when data files are stored in Hadoop blocks in HDFS. The first problem is the lack of name node memory that occurs when large numbers of small files are stored (Zhou et al. 2015). Therefore, files in the data node cannot be stored even if memory space is sufficient. The second problem is the deterioration of scheduling and processing performance. For example, HDFS builds a mapper for each block that is stored, so 1000 mappers are needed if 1000 files are built.

In this paper, we propose a performance improvement method for MapReduce processing, which integrates the CombineFileInputFormat method and the reuse feature of the Java Virtual Machine (JVM). MapReduce processing performance is improved by allowing multiple mappers to reuse a single JVM, rather than creating a JVM for every mapper; this reduces JVM creation time.

2 Related work

2.1 MapReduce

MapReduce is a programming model (and associated implementation) for processing and generating large data sets using a parallel, distributed algorithm on a cluster (Dean and Ghemawat 2008). MapReduce is composed of a Map() procedure that performs filtering and sorting and a Reduce() method that performs a summary operation (https://en.wikipedia.org/wiki/MapReduce). MapReduce is a model for processing keyed, value-based data in parallel, and consists of two steps: the Map task, which uses input data sources to create interim results, and the Reduce task, which uses the interim results as input to obtain the final results (Choi et al. 2013), as shown in Fig. 1. The input data are divided into a plurality of data; as a result, the Map task can be carried out in a plurality of nodes. Each Map task stores the results of processing the input data allocated thereto in the local file system of each node. To provide the final results, the Reduce task receives the interim results stored in the plurality of nodes, and carries out integrated processing. If possible, task distribution is implemented so that data can be processed in the node where they are placed, in order to minimize network traffic. To this end, data are divided in consideration of the status and location of data storage (Choi et al. 2013).

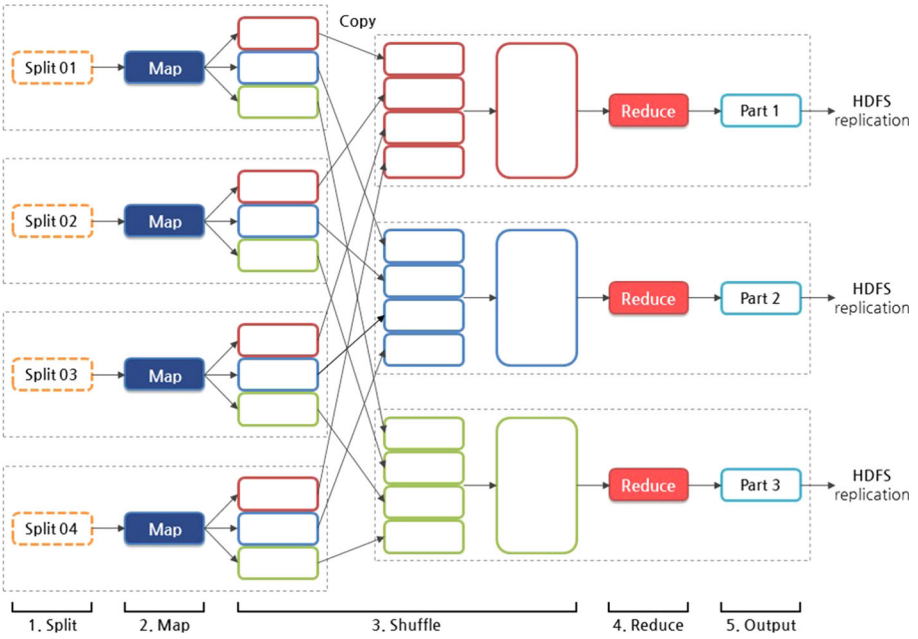

Fig. 1 MapReduce process

2.2 JVM reuse method in MapReduce

Enabling the JVM reuse feature may reduce JVM startup and shutdown overhead and improve performance, as the JVM spends less time interpreting Java bytecode. Typically, the JVM reuse feature is beneficial in cases in which a workload contains a large number of very short tasks. The JVM reuse feature can be used to modify the parameter in the Hadoop environment file. Table 1 shows the value of the JVM reuse parameter (mapred.job.reuse.jvm.num.tasks).

Table 2 shows the settings for JVM reuse. The settings indicate the number of sequential execution tasks in JVM. Default value 1 means that the JVM is not reused, and −1 means that all tasks are executed using the JVM only. Finally, a *user set* value indicates that a number of input tasks will be reused in the JVM.

Figure 2 describes the JVM reuse process in MapReduce; in this case, the *user set* value is three. In Fig. 2, three mappers (Map tasks) are performed by setting the appropriate JVM reuse value.

Table 1 JVM Reuse Parameter (mapred-site.xml)

```
<configuration>
<property>
<name>mapred.job.reuse.jvm.num.tasks</name>
<value>3</value> // Set Value
</property>
</configuration>
```

Table 2 mapred.job.reuse.jvm.num.tasks parameter set value

Setting value	Description
1(default)	Do not use the reuse JVM
−1	Unlimited USE
User set	Number of tasks to be processed on a single JVM

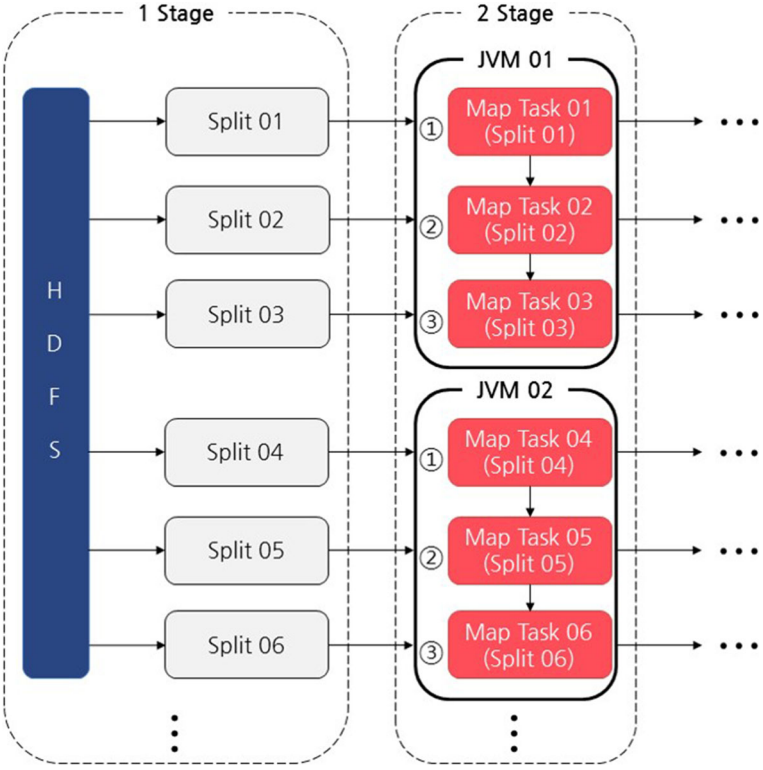

Fig. 2 JVM reuse process in MapReduce (Set Value: 3)

2.3 CombineFileInputFormat method

CombineFileInputFormat works efficiently with small files, which enables FileInputFormat to create a split for each file. CombineFileInputFormat packs many files into each split, so each mapper has more to process. CombineFileInputFormat can also provide benefits when processing large files. Basically, it decouples the amount of data that a mapper consumes from the block size of the files in HDFS (http://www.ibm.com/developerworks/library/bd-hadoopcombine/). Figure 3 describes the Combine File Input Format process. In Fig. 3, files are integrated until their combined size reaches 64 MB, and split files are performed first. Therefore, the number of mappers is reduced and performance is improved.

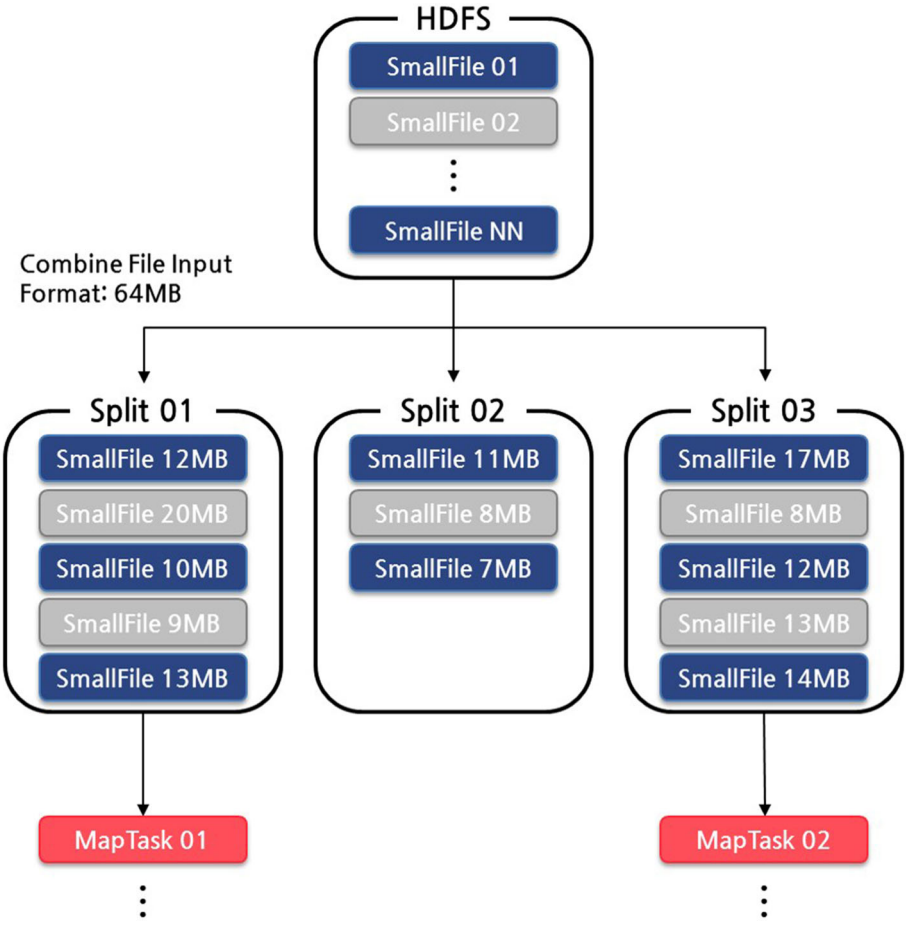

Fig. 3 CombineFileInputFormatProcess (64 MB)

2.4 The problems with small files in Hadoop

In this paper, a small file is defined as any file that is significantly smaller than the Hadoop block size, which is usually set to 64 MB (such as in the example in Fig. 4).

The metadata of data files is stored in an in-memory database by name node when data files are stored in Hadoop blocks in HDFS. The first problem is the lack of memory for name nodes when many small files are stored. Therefore, files in data nodes cannot be stored even if there is sufficient memory space. Another problem is deterioration in scheduling and processing performance. For example, HDFS builds a mapper for each block that is stored, so 1000 mappers are needed if 1000 files are built (Fig. 5).

3 Improved performance optimization for massive small files

In this section, we describe a method for improved performance optimization when processing large numbers of small files, as shown in Fig. 6. This method uses the CombineFileInput-Format class and the reuse feature of the JVM.

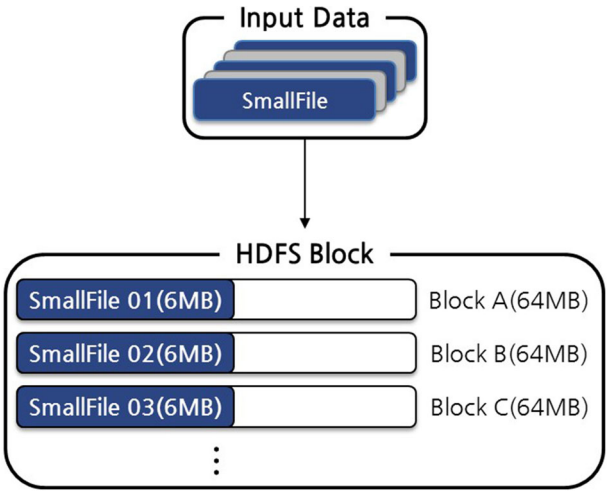

Fig. 4 SmallFile in Hadoop

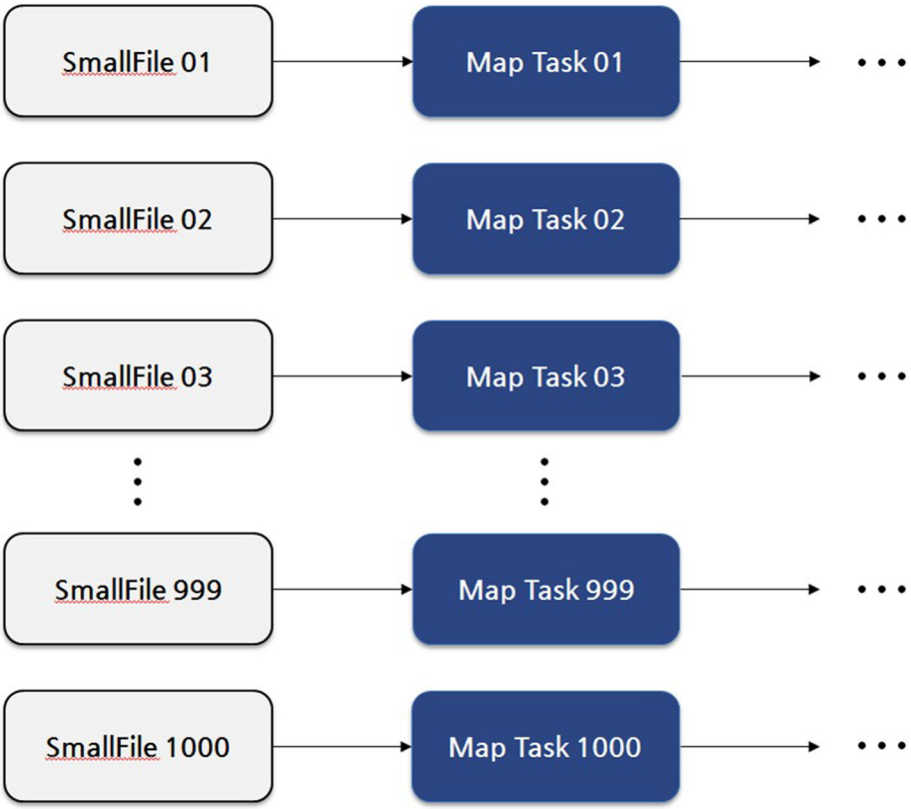

Fig. 5 Problem of increasing time in Hadoop

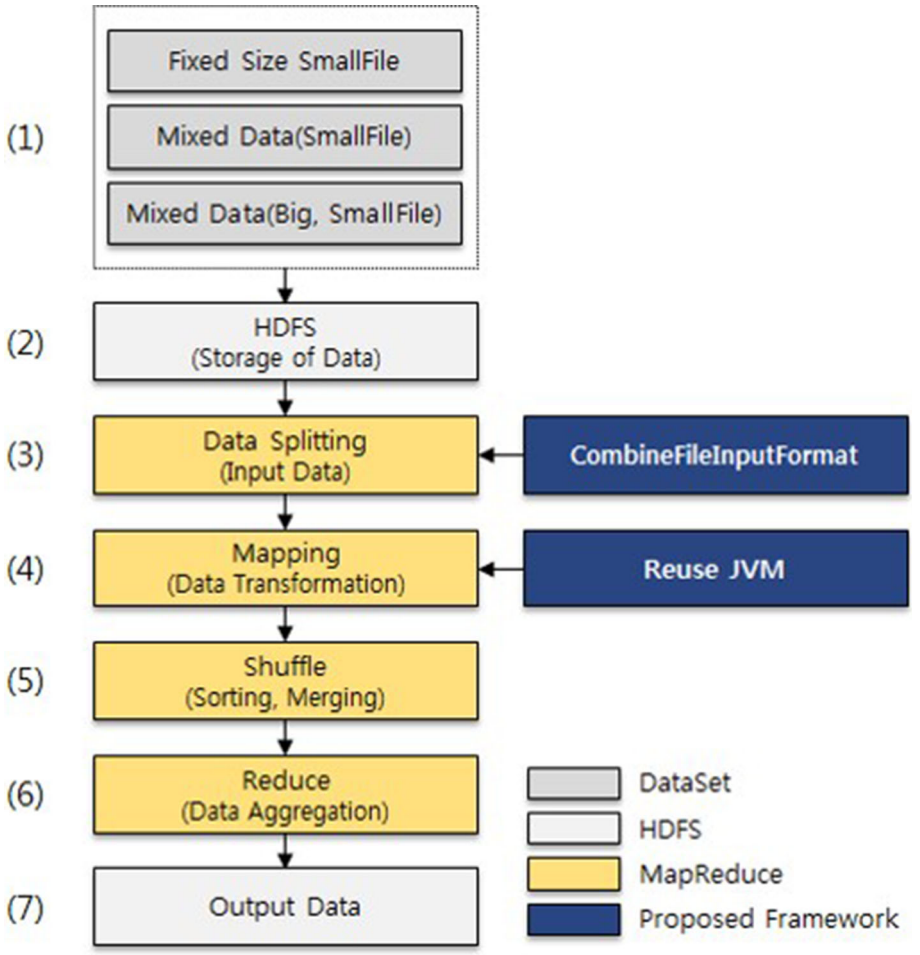

Fig. 6 Proposed method based on CombineFileInputFormat and JVM reuse

The process of the proposed method is as follows:

1. Various types of files are used to create a dataset; these include small files of fixed sizes, small files of mixed sizes, and small and large files of mixed sizes.
2. Dataset of (1) is stored in HDFS blocks.
3. Data blocks are loaded in HDFS, and these blocks are transferred to the map task by data splits according to the size specified for CombineFileInputFormat.
4. Map task transforms the files of (3) based on JVM reuse settings.
5. The output of the map task is sorted and merged.
6. The values are extracted by the user after a shuffle process.
7. The output data is stored after a reducing process in HDFS.

3.1 Applied method of CombineFileInputFormat

Hadoop can create a split for each input file. However, the CombineFileInputFormat method can also process a split that contains several files. By processing large numbers of files combined into a single split, CombineFileInputFormat reduces the number of mappers that must be created.

In line 5, the file size is set to 128 MB. The RecordReader method performs pair (Key, Value) formatting for the mapper based on the input format of smallcombinewritable in line 7. isSplitable is the method used for the input format class in line 15. The file is divided by block if this method is true.

3.2 Applied method of CombineFileInputFormat

The JVM reuse function improves MapReduce processing performance by allowing multiple mappers to reuse a single JVM, rather than creating a JVM for every mapper; this reduces JVM creation time.

The mapred.job.tracker parameter stores the address of the job tracker in line 8, and the data node requests the MapReduce task. The mapred.job.reuse.num.tasks parameter must be set for JVM reuse in lines 12 and 13.

3.3 Proposed framework

Figure 7 shows the proposed framework for improving performance. The process of the proposed framework is as follows:

1. A large number of small files are loaded into HDFS.
2. The size of the loaded dataset is set according to a split based on CombineFileInputFormat (64 MB).
3. The split is transferred to a mapper.
4. Three mappers are executed by setting the appropriate JVM reuse value.

4 Improved performance optimization for massive small files

4.1 Experiment environment

This study compares the performance of the proposed method against existing methods based on CombineFileInputFormat and JVM reuse. Table 3 shows the experimental environments, which contained six servers.

The mixed files consisted of large and small files. The experimental dataset, which was created using the split command in Linux, contained the contents of large and small files. Table 4 shows the experimental dataset; its total size was 11 GB, and it consisted of 4, 5, and 6 MB files. The 4MB files were categorized as small files. The experimental dataset contained three-to-seven ratios of large/small files, or three-to-seven ratios of small/large files (Tables 5, 6).

4.2 Performance evaluation

In this study, three types of experiments were performed. The first test involved searching for the optimal block size in CombineFileInputFormat. This test consisted of six block sizes,

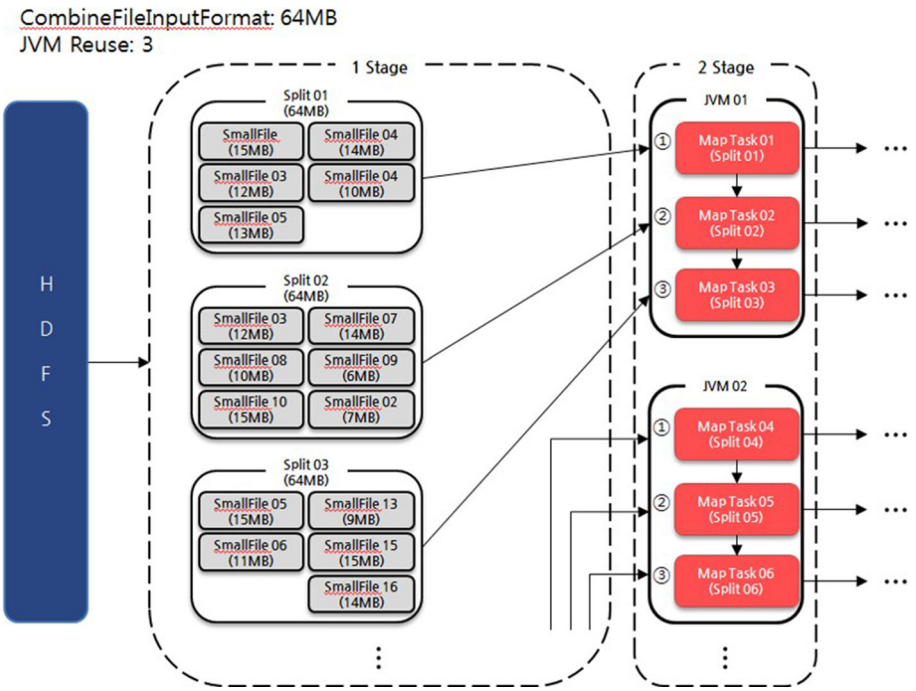

Fig. 7 Proposed framework

Table 3 Combine Class based on CombineFileInputFormat

```
1  public class smallcombinefileinputformat
2  extends CombineFileInputFormat<smallcombinewritable, Text> {
3  public smallcombinefileinputformat (){
4     super();
5        setMaxSplitSize(134217728); // setting value of 128MB
6  }
7  public RecordReader<smallcombinewritable,Text>
8  createRecordReader(InputSplit split,
9       TaskAttemptContext context) throws IOException {
10         return new
11         CombineFileRecordReader<smallcombinewritable,
12         Text>((CombineFileSplit)split, context,
13         smallcombinerecordreader.class);
14      }
15  protected boolean isSplitable(JobContext context, Path file){
16    return false;
17    }
18 }
```

e. g., 32, 66, 128, 256, 512, and 1024 MB. Figure 8 shows the results. In Fig. 8, the 256 MB block size is the optimal result for CombineFileInputFormat.

Performance improved as the number of small files increased. Moreover, using small files produced better results than using mixed-size files, because larger files already had the same

Table 4 Environment setting file for JVM reuse

```
1  <?xml version="1.0"?>
2  <?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
3
4  <!-- Put site-specific property overrides in this file. -->
5
6  <configuration>
7      <property>
8          <name>mapred.job.tracker</name>
9          <value>oracle:9001</value>
10     </property>
11     <property>
12         <name>mapred.job.reuse.jvm.num.tasks</name>
13         <value>-1</value>
14     </property>
15 </configuration>
```

Table 5 Experiment environment (6 Servers)

List	Description
H/W	
CPU(core)	Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz
RAM	2 GB
HDD	200 GB
S/W	
Virtual machine	VMware 8.0.2
O.S	CentOS 6.6
Hadoop version	Hadoop 1.2.1

Table 6 Data set (11 GB)

List		Files
Big files		22
Small files		
	4 MB	2879
	5 MB	2305
	6 MB	1922
	Mixed files (4, 5 and 6 MB)	2374
Mixed data (big & small files)		
	Big file (7) : Small file (3)	877
	Big file (3) : Small file (7)	2024

file size. In this experiment, using 4 MB files produced the best results in CombineFileInputFormat. The next test involved searching for the optimal JVM reuse setting. This test consisted of 11 setting values, from −1 to 11. Figure 9 shows the results. A value of −1 produced the best JVM reuse results in the test depicted in Fig. 9. All of the results were improved, because the setting for the number of files was important.

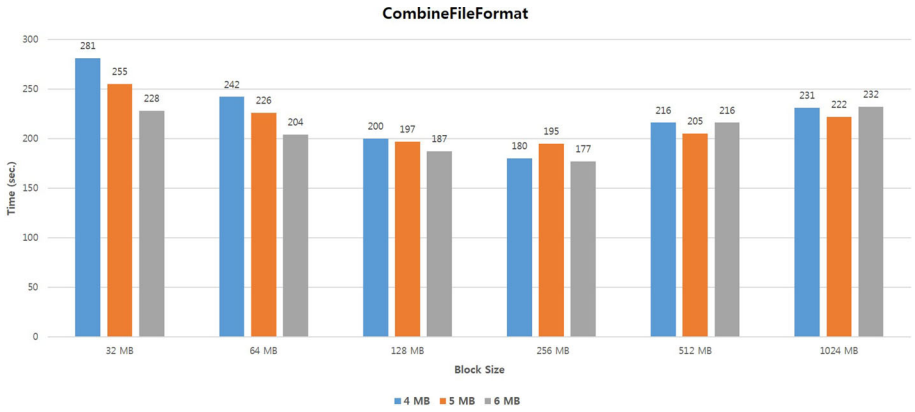

Fig. 8 Result of process time in CombineFileFormat

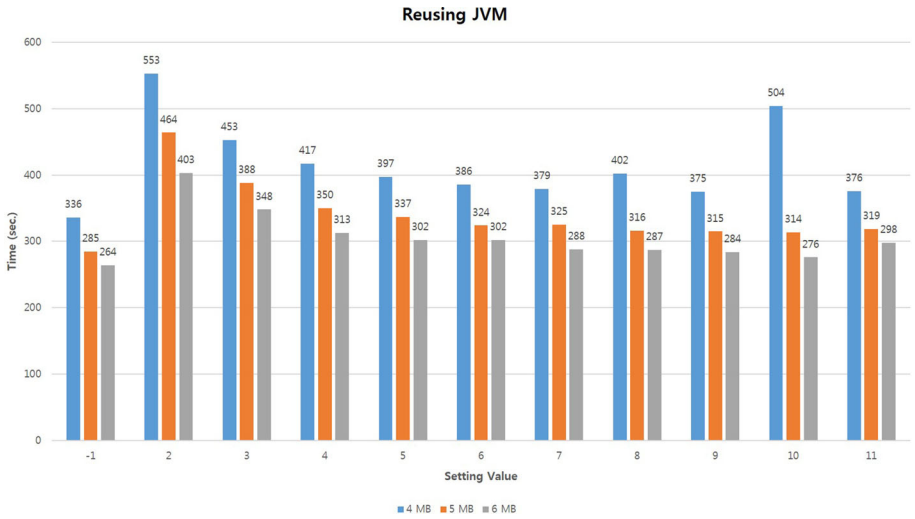

Fig. 9 Result of process time in JVM reuse

The most important test compared the processing time required by the proposed method and the other methods. In this experiment, a value of -1 was used as the JVM reuse setting, and a block size of 256 MB was used in CombineFileInputFormat. The processing times required by the proposed method and the other methods (CombineFileInputFormat, JVM reuse, and MapReduce) were evaluated; the results are shown in Fig. 10. The proposed method produced good results.

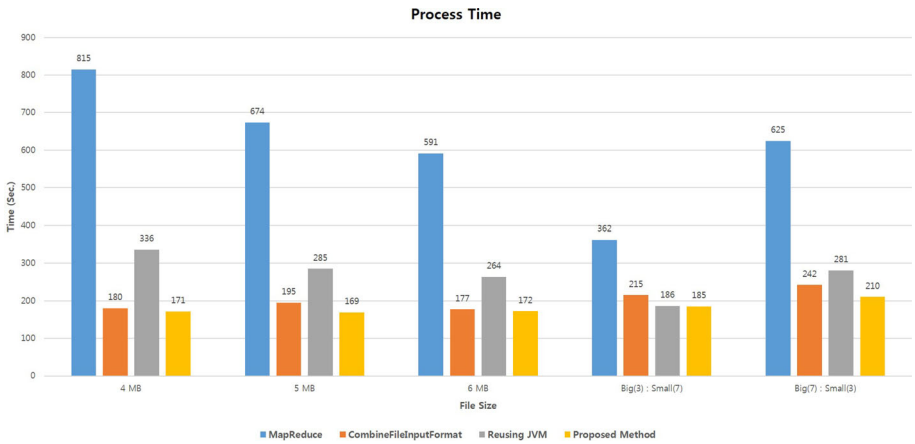

Fig. 10 Result of proposed method

5 Conclusion

In this paper, we proposed a MapReduce performance improvement method that integrates the CombineFileInputFormat method and JVM reuse feature. Previous methods need a significant amount of preprocessing, and require Hadoop resources such as the distributed cache method. However, the proposed method does not require preprocessing, and additional Hadoop resources were not used. In addition, the proposed method requires less processing time. Furthermore, the proposed method produced good results when we compared it against other methods (CombineFileInputFormat, JVM reuse and MapReduce). In future work, we plan to improve the algorithms in CombineFileInputFormat and JVM reuse, in order to optimize them for data size. In addition, we plan to support the Combine Class and RecordReader method in the Java library.

Acknowledgement This research was supported by Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Education (2015R1D1A3A01019642) and Basic Science Research Program through the National Research Foundation of Korea(NRF) funded by the Ministry of Science, ICT & Future Planning (2015R1C1A1A02037515).

References

Bhandarkar, M. (2010). MapReduce programming with apache Hadoop. Parallel & Distributed Processing (IPDPS), 2010 IEEE International Symposium on. IEEE, 2010.

Choi, C., Choi, J., & Kim, P. (2014). Ontology based access control model for security policy reasoning in cloud computing. *Journal of Supercomputing*, *67*(3), 711–722.

Choi, J., Choi, C., Ko, B., & Kim, P. (2014). A method of DDoS attack detection using HTTP packet pattern and rule engine in cloud computing environment. *Journal of Soft Computing*, *18*(9), 1697–1703.

Choi, J., Choi, C., Yim, K., Kim, J., & Kim, P. (2013). Intelligent recongurable method of cloud computing resources for multimedia data delivery. *Journal of Informatica*, *24*(3), 381–394.

Dean, J., & Ghemawat, S. (2008). MapReduce: Simplified data processing on large clusters. *Communications of the ACM*, *51*(1), 107–113.

Heger, D. (2013). Hadoop performance tuning-a pragmatic & iterative approach. *CMG Journal*, *4*, 97–113.

Ogiela, M.R., & Ogiela, U. (2003). *Linguistic Approach to Cryptographic Data Sharing*. In The 2nd International Conference on Future Generation Communication and Networking(FGCN) (Vol. 1, pp. 377–380), December 2008.

Ogiela, M. R., & Ogiela, U. (2010). Grammar encoding in DNA-like secret sharing infrastructure. *Lecture Notes in Computer Science*, *6059*, 175–182.

Shvachko, K., et al. (2010). The hadoop distributed file system. Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on. IEEE, 2010.

Zhou, F., Pham, H., Yue, J., Zou, H., & Yu, W., (2015). *SFMapReduce: An Optimized MapReduce Framework for Small Files, Networking, Architecture and Storage (NAS)*. In 2015 IEEE International Conference on, (pp. 23–32), August 2015.

Zikopoulos, P., & Eaton, C. (2011). *Understanding big data: Analytics for enterprise class hadoop and streaming data*. New York: McGraw-Hill Osborne Media.