ISSN 2277-3061
Volume15 Number 8
International journal of computer and technology

P-Codec: Parallel Compressed File Decompression Algorithm for Hadoop

Idris Hanafi[†] and Amal Abdel-Raouf[†‡]
[†]Computer Science Department, Southern Connecticut State University, USA
[‡]Computers and Systems Department, Electronics Research Institute, Giza, Egypt
hanafii2@southernct.edu, abdelraoufa1@southernct.edu

ABSTRACT

The increasing amount and size of data being handled by data analytic applications running on Hadoop has created a need for faster data processing. One of the effective methods for handling big data sizes is compression. Data compression not only makes network I/O processing faster, but also provides better utilization of resources. However, this approach defeats one of Hadoop's main purposes, which is the parallelism of map and reduce tasks. The number of map tasks created is determined by the size of the file, so by compressing a large file, the number of mappers is reduced which in turn decreases parallelism. Consequently, standard Hadoop takes longer times to process. In this paper, we propose the design and implementation of a Parallel Compressed File Decompressor (P-Codec) that improves the performance of Hadoop when processing compressed data. P-Codec includes two modules; the first module decompresses data upon retrieval by a data node during the phase of uploading the data to the Hadoop Distributed File System (HDFS). This process reduces the runtime of a job by removing the burden of decompression during the MapReduce phase. The second P-Codec module is a decompressed map task divider that increases parallelism by dynamically changing the map task split sizes based on the size of the final decompressed block. Our experimental results using five different MapReduce benchmarks show an average improvement of approximately 80% compared to standard Hadoop.

Keywords

Hadoop, MapReduce, HDFS, Compression, Parallelism;

1. Introduction

Today's flood of data is being generated at the rate of several Terabytes (TB) or even Petabytes (PB) every hour [1]. These numbers are expected to continuously increase during the next several years. Traditional databases are not suitable for big data applications because of the volume and complexity of the data. The MapReduce framework was proposed by Google in 2004 and since then has proven to be a powerful and cost-efficient method for massive parallel data processing [2]. However, processing and storing these large sets of data has become an issue; storing large amounts of raw data on servers is unconventional. An efficient solution is data compression, which has become widely adopted by professionals and researchers who manage large amounts of data. Data compression is used to reduce storage consumption and minimize input/output (I/O) time costs. Companies that receive large amounts of data every day, such as Google, Facebook, Amazon, and Twitter, use compression. As of 2014, Facebook receives 600 TB of raw data per day [3]. Assuming one server rack can hold up to 2 petabytes of data [4], Facebook would need to add a new rack of storage every 3 days without compression. According to [5], digital data is doubling in size every two years, so by 2020, it is expected to reach 44 zettabytes and without compression the cost of maintaining and storing the data would drastically increase.

Due to the increased need for compression, many different methods of optimization have been proposed to speed up the applications that process compressed data. Compression methods have been advancing with the target of achieving higher compression ratios using efficient algorithms. For example, Columnar from Vertica [6] can achieve a compression ratio of 8. Vertica is a system that includes a compression engine that can process compressed data without decompression for database management systems. RainStor [7] can achieve a compression ratio of 40 and in some cases can reach a value of 100. Parallel Database Management System (PDMS) [8] is a system that has incorporated multiple complex designs for processing and compressing data [9]. Other complex designs include the parallelization of compression while being sent over the network [10], and decompression of data while being retrieved by the system.

In contrast, Hadoop does not have any complex design to improve the processing of compressed data. The only characteristic Hadoop supports for compressed data is recognition of the data compression algorithm being processed in the system and decompression of the data based on the codec that is determined [11] [12]. Unfortunately, Hadoop has a lower degree of parallelism when it processes compressed data. As a result, Hadoop performance has been shown to be worse when processing compressed data than when processing uncompressed data [13]. With further experiments, we found that this degradation is caused by

many factors. These factors include the overhead of data decompression during the job runtime and the decreased number of map tasks that reduces parallelism in Hadoop. In this work, we introduce the Parallel Compressed file Decompressor (P-Codec) to overcome these two factors and to speed up Hadoop's current processing of compressed data. P-Codec includes two schemes:

1. A Parallel Data Decompressor (PDD) that decompresses data during the data uploading phase to the HDFS. This data decompressor is automatically activated, so any data block a node receives is decompressed. This step eliminates the decompressing overhead during the MapReduce phase.
2. A Decompressed Task Divider (DTD), which aims at increasing the degree of parallelism in Hadoop. The number of map tasks created is determined by the number of blocks [1]. Therefore, uploading decompressed data with larger sizes increases the number of blocks, and the number of map tasks increases. Consequently, parallelism increases.

The rest of the paper is organized as follows. We present the background and related work in Section 2. Then the problem description is introduced in Section 3. Section 4 describes our proposed method, P-Codec, together with its design and implementation details. An experimental evaluation of the algorithm is presented in Section 5. Finally, we provide the conclusions and future work in Section 6.

2. Background and Related Works

Hadoop is a scalable data processing tool that enables one to process a massive volume of data in parallel using many low-end computers. Hadoop includes two parts: the HDFS and MapReduce [1]. MapReduce activates multiple nodes, each of which concurrently processes a certain chunk of a file. Before any MapReduce job can be executed, the data file must be uploaded to the HDFS. In each part of Hadoop, there contains a file split [1]. In the HDFS there is a physical file split and the MapReduce contains a logical file split. The physical file split is performed on the file when it is moved from the local machine to the HDFS. Then the file is stored as blocks which are distributed to nodes called datanodes [14]. By default, in Hadoop 2.x the block size is 128 Megabytes (MB) [1]. On the other hand, there is a logical file split for the MapReduce section. This logical split divides the data of each block into map tasks, assigns them to the datanodes, and then the given datanodes execute the reduce functions [2]. Although Hadoop does support other methods of retrieving map task split sizes–such as split sizes by lines of a text file-but by default, the map task has a size of the 128 MB [1]. To reduce the network hops and to follow the rack awareness scheme, Hadoop YARN executes the map tasks on nodes where the block resides [15]. With this design of multiple datanodes working on separate chunks (map tasks) of the file, Hadoop achieves a high degree of parallelism.

Figure 1 shows the procedure of a one Gigabyte (GB) or 1024 MB file that is processed by Hadoop for a MapReduce job. In the first step (a), the file is uploaded to the HDFS. When uploading the file to the HDFS, the file is split into 128 MB blocks, thereafter, creating eight (1024/128) blocks stored in different datanodes. Once the file is uploaded to the HDFS, the file is ready to be processed by MapReduce. Step (b) shows the second step when MapReduce processes the file, and the Resource Manager splits the blocks map tasks for the datanodes to execute.

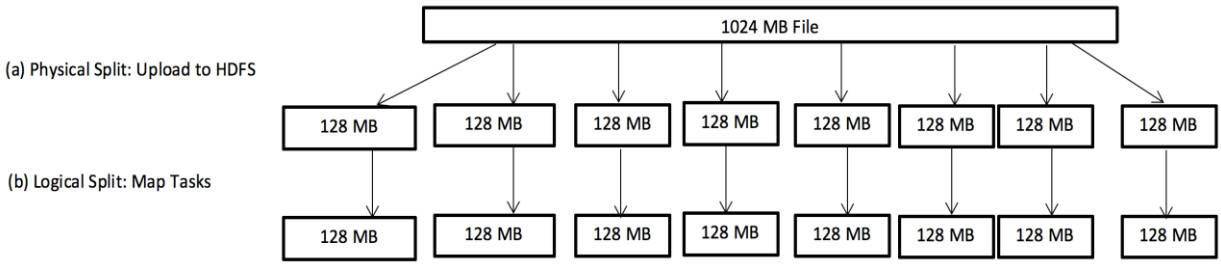

Figure 1: The stages of a file being processed in Hadoop.

In the literature, there is some work dealing with compression in the Hadoop framework. Proposals include data compressions for map input data, map output data, and reduce output data as well as other more complex compression algorithms. Some compression schemes target specific file formats like the Optimized Row Columnar (ORC), which improves compressed data processing [16], and is designed for structured data and runs on Hive. Hive works on top of Hadoop and focuses on data queries [17]. Ding et al. in [18] designed a row-column hybrid page file storage structure on top of HDFS (RCHP) to be served as a storage model for compressed structured big data. In this work, a parallel data compression strategy is proposed to compress the column data in each partition of the input data by selecting the most suitable compression algorithm which will then be stored on HDFS in the form of RCHP files. With this RCHP file structure, the author provided a

parallel query strategy, which can execute a query directly on compressed RCHP data files [18]. Although this approach improved the performance of processing structured data, it is limited to a specific file structure.

Other works target the HDFS and MapReduce modules in Hadoop; one example is IC-Data [19]. While IC-Data only processes uncompressed data, it compresses the data upon uploading the file to the HDFS by using the Network Overlapped Compression (NOC) scheme. IC-Data also proposed a Compression Awareness Storage (CAS) scheme that increases parallelism by reducing the block size. For the MapReduce module, there is the Compressed Task Divider, which normalizes the amount of map tasks by splitting the blocks based on compressed sizes. IC-Data improves job runtimes by 66% [19]. In the following section we show that our proposed method, P-Codec outperforms IC-Data as it improves the job runtime on Hadoop by 80%.

3. Problem Description

A Codec is a compression-decompression algorithm implementation. Currently, Hadoop processes compressed data by recognizing the file's codec type and decompressing the file based on the codec identified [11]. Moreover, Hadoop is limited to support few types of codecs, the most common of which are: GZip, BZip2, Snappy, and LZO [1] [12]. Table 1 lists the description of the most common codecs Hadoop supports and their characteristics. Most importantly, some algorithms are capable of being split while others are not. When a compressed file is uploaded to the HDFS, the compressed file is either splitted or not based on whether the file compression type is splittable. Based on our observations, splittable codecs are more suitable to use with MapReduce.

Table 1: The four codecs Hadoop supports.

Codecs	Degree of Compression	Compression Speed	Splittable	Algorithm
GZip	Medium	Medium	No	Deflate
BZip2	High	Slow	Yes	BZip2
Snappy	Medium	Fast	No	Snappy
LZO	Medium	Fast	No, unless Indexed	LZO

To process a codec that is not splittable in Hadoop, such as GZip or Snappy, Hadoop would upload the file regularly to the HDFS, but only one block will contain this complete file and only one datanode will store this block. Then when a MapReduce job is attempted to run on this non-splittable compressed file, only one map task will be assigned. Since only one datanode is to process the task, parallelism is abandoned in Hadoop. Figure 2 shows the example of a 1 GB file compressed using GZip (resulting in approximately 5 MB block size) being processed in Hadoop for a MapReduce job. As seen on figure 2 (b), the 1 GB file is only being processed by one map task.

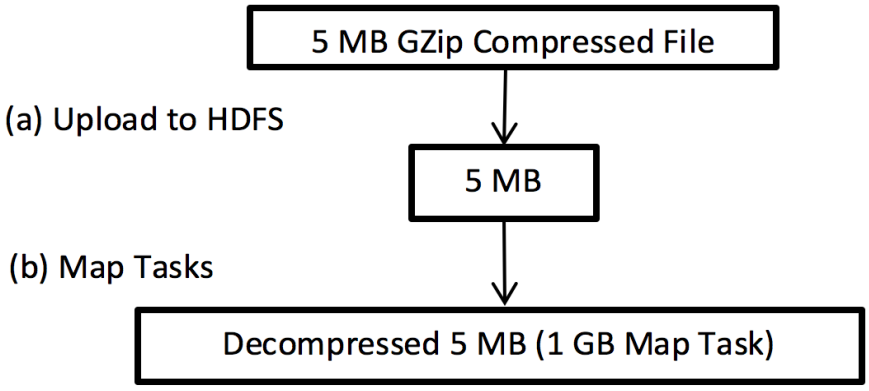

Figure 2: The stages of a of a non-splittable GZip-compressed 1 GB file being processed by Hadoop for a MapReduce job.

Processing a codec that is splittable by Hadoop, such as BZip2 or indexed LZO, would also result in a reduced degree of parallelism. For example, a 20 GB (20480 MB) size file, processed by standard Hadoop would result in 160 (20480/128) map tasks of 128 MB size each. When the 20 GB file is compressed with indexed LZO, the result is approximately a 7168 MB size file. Processing the compressed file with standard Hadoop will result in 56 (7168/128) map tasks, which means instead of 160 map tasks there will only be one-third of that amount of map tasks. Standard Hadoop suffers from this problem when dealing with compressed files, as compression lowers the file size. The file size is correlated to the number of map tasks and the degree of parallelism; the larger the file, the larger the number of the map tasks and the higher the degree of parallelism. Additionally, standard Hadoop suffers from the decompression overhead during the map tasks

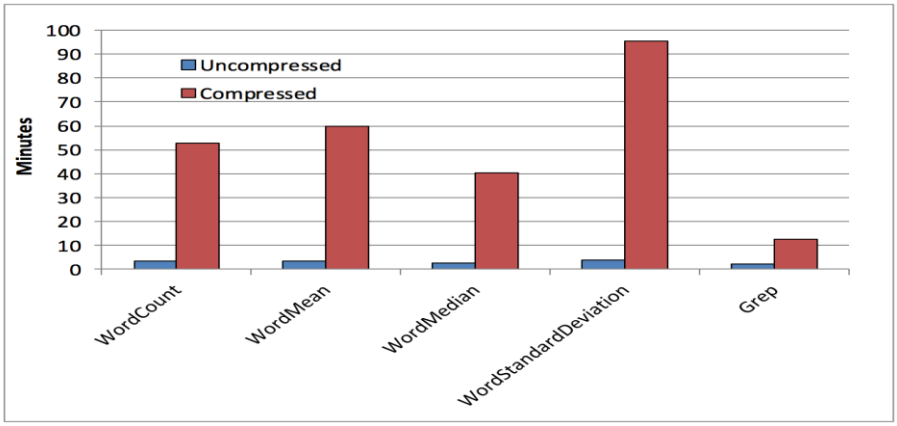

Figure 3: Processing time of a 20 GB size compressed and uncompressed file using 5 different benchmarks.

runtime. Figure 3 shows the results of processing a compressed 20 GB size file by Hadoop versus processing an uncompressed 20 GB size file using five MapReduce benchmarks. Processing compressed data using standard Hadoop results in a 60-85% longer runtime than processing uncompressed data.

The motivation of our work was to eliminate the decompression overhead during run time by having the file decompressed early while uploading to HDFS. However, this approach slows down data transfer across the network due to the larger size of the files after decompression. We conducted an experiment to compare the time it takes to transfer the standard block size (128 MB) across the network versus the time it takes to decompress the same size of a map task by different compression algorithms. Figure 4 shows the time comparison results which indicate that transferring uncompressed data is faster than decompressing the default map task size using all the codecs Hadoop supports. The following section illustrates how we use this fact to implement the P-Codec to improve the Hadoop performance.

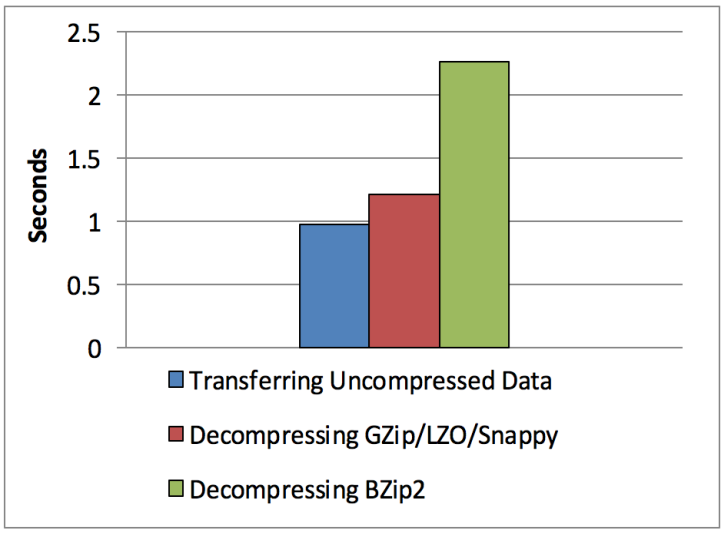

Figure 4: Time comparison between block transfer over the network and the block decompression.

4. P-Codec Design and Implementation

We propose the design and implementation of a Parallel Compressed file Decompressor (P-Codec). P-Codec consists of two modules; the first module works on top of the HDFS part of Hadoop and the second module works on top of MapReduce. P-Codec is driven by the notion of reducing the runtime of a job by removing the burden of decompressing the blocks during the MapReduce phase. P-Codec has been developed on Hadoop's latest version 2.7.1. This required modifications in DFSOutputStream, DFSInputStream, PacketReceiver, BlockReader, and FileInputFormat. Figure 5 shows the architecture of P-Codec.

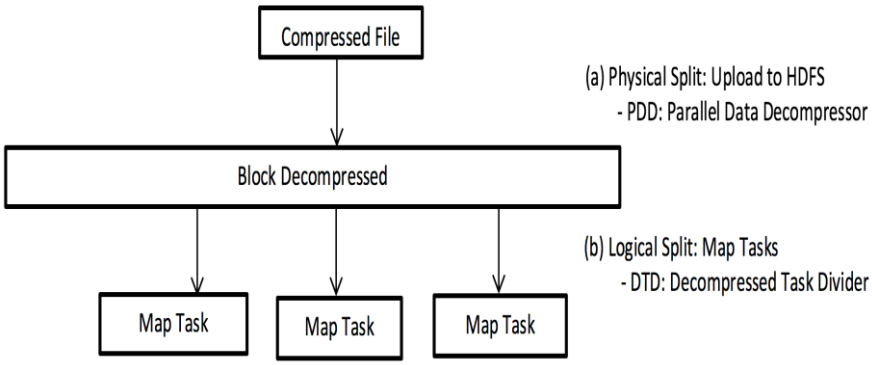

Figure 5: The stages of a file being processed in Hadoop with P-Codec

4.1. Parallel Data Decompressor (PDD)

This new component is added to the current model of Hadoop's HDFS. When Hadoop uploads compressed files to the HDFS, there are two cases. The first case is the file is compressed using a non-splittable codec (GZip, Snappy, and unindexed LZO) and uploaded to the HDFS; it will always be stored in one block. The second case is when Hadoop uploads splittable compressed files (BZip2 and indexed LZO), which results in a fewer number of blocks than when processing an uncompressed file. As illustrated in section 3 and shown in figure 4, we observed that decompressing files can take longer than transferring the uncompressed file over the network. Under this premise, we wanted to incorporate the idea of decompressing the file only once to eliminate the need to decompress the blocks every time a MapReduce job is executed.

Figure 5 (a) shows the implementation of a parallel data decompressor (PDD) module which decompresses any compressed file that is being uploaded to the HDFS. Thus the decompression is only done once. The data decompression is performed at the same time when the master node is sending out packets of the file to the datanodes. This parallelized approach removes the overhead of data decompression after receiving the blocks. Since uploading the data to the HDFS and decompression is done in parallel, the overall time to upload the file to the HDFS is determined by the longest time of a block decompression time and the block upload time.

4.2. Decompressed Task Divider (DTD)

When uploading compressed files, the file size decreases and causes Hadoop to use less blocks. Since the number of map tasks is determined by the number of blocks [1], the job runtime will decrease when there are more blocks and vice versa [20]. For non-splittable compressed files, there will be only one block and one mapper that result in a longer runtime. For splittable codecs, there will be a decrease in the number of blocks with a corollary of less number of map tasks and less parallelism. To compensate for this shortage of map tasks for compressed files, we have developed a Decompressed Task Divider (DTD) to increase the number of of map tasks. As shown in figure 5 the blocks have been decompressed in the HDFS, meaning that any block can be splitted into smaller map tasks even if the uploaded file in the HDFS was compressed with a non-splittable codec. Figure 5 (b) shows the decompressed task divider (DTD) that dynamically changes the map task split size based on the size of the final decompressed block. According to Hadoop's MapReduce implementation and guide [1] [21], Hadoop runs at its best when the number of maps tends to be around 10-100 maps per node [19]. Upon choosing the map task split size, we used the guide's approach to determine the number of maps per node using the formula:

$N = \frac{D}{10}$ if the file size <10 GB

$N = \frac{D}{100}$ if the file size >=10 GB

N is the number of maps per node and D is the decompressed file size. As a result, this DTD module increases the number of mappers and thus increases the degree of parallelism for compressed files using both splittable and non-splittable codecs in Hadoop.

5. Experimental Results

The experiments were conducted using Hadoop's latest version 2.7.1 on Ubuntu 14.10. Our test cluster contains one master node and 20 datanodes, 17 of which are identical and the other 3 are different from each other, resulting in a total of 4 heterogeneous systems. Table 2 is a breakdown of the cluster used.

Table 2: Breakdown of the cluster

Amount	SSD	RAM	Processors
1	500 GB	32 GB	3.6 GHz intel i5
1	250 GB	16 GB	3.2 GHz intel i5
1	120 GB	8 GB	3.2 GHz intel i5
17	120 GB	4 GB	2.5 GHz intel i5

The job execution time is used as the performance metric in our experiments. To show the effectiveness of P-Codec, we calculated the job execution time of five different benchmarks and compared the results with the job execution time calculated using standard Hadoop. The benchmarks we used include: WordCount, WordMean, WordStandardDeviation, WordMedian, which are I/O-intensive and Grep, which is CPU-intensive. We executed these benchmarks using 3 different file sizes: 1 GB, 10 GB, and 20 GB. Figure 6 shows the comparison results between standard Hadoop and P-Codec on all 3 files compressed using GZip and BZip2. The results indicate that P-Codec is faster than standard Hadoop.

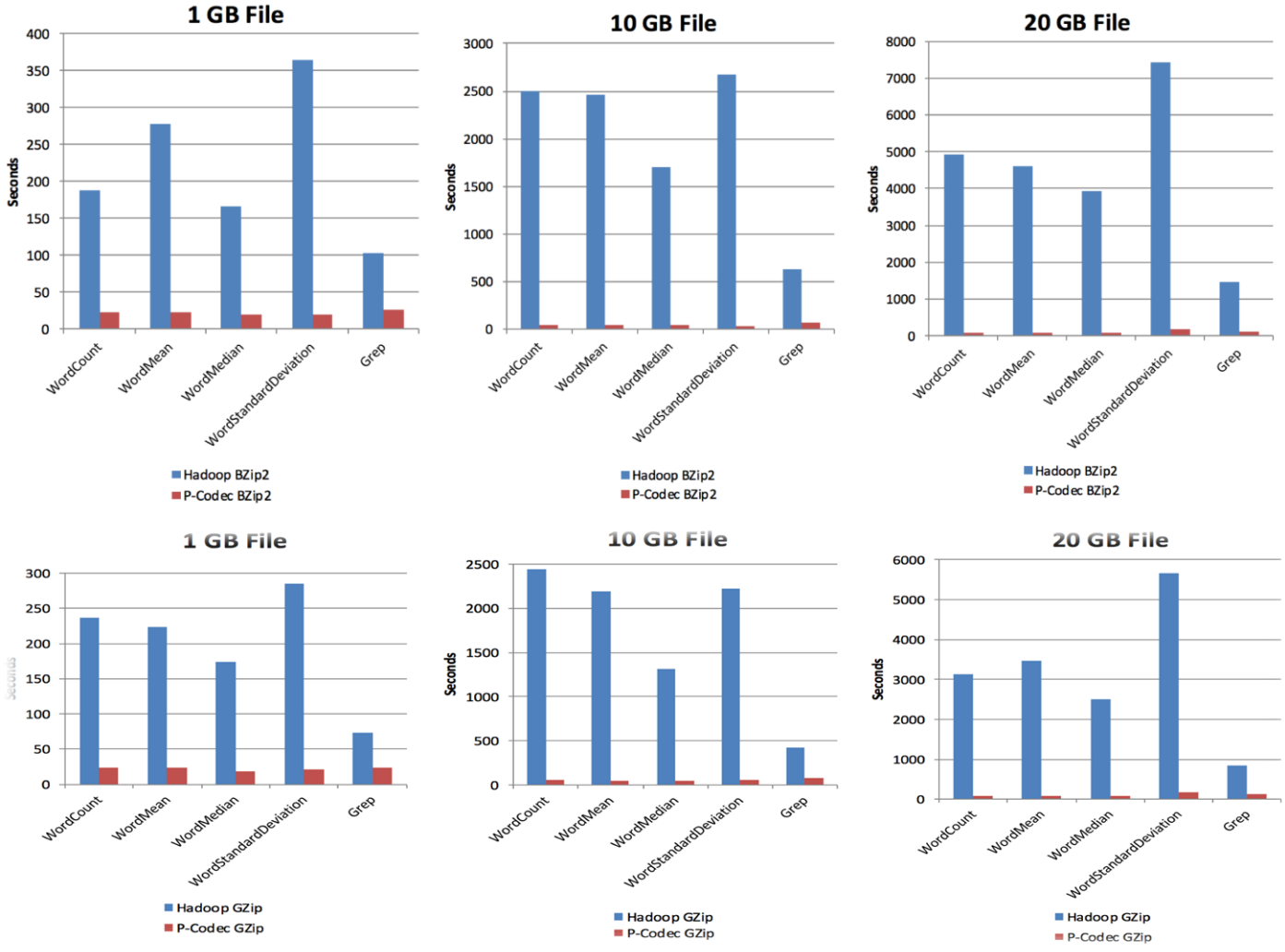

Figure 6: Tme comparison between Hadoop and P-Codec processing of files compressed with GZip and BZip2.

Table 3 and Table 4 show the performance improvement in percentage using P-Codec in a tabular format for both GZip and BZip2. Based on these percentages, the average improvement of the performance using P-Codec is approximately 80%.

Table 3: The job runtime improvement with P-Codec for GZip.

Benchmark	1GB	10GB	20GB
WordCount	80.13%	87.85%	87.13%
WordMean	79.17%	87.71%	87.32%
WordMedian	79.28%	86.55%	87.05%
WordStdDeviation	82.72%	87.51%	88.70%
Grep	67.53%	71.28%	85.10%

Table 4: The job runtime improvement with P-Codec for BZip2.

Benchmark	1GB	10GB	20GB
WordCount	87.88%	88.27%	88.02%
WordMean	82.04%	88.36%	88.20%
WordMedian	88.50%	87.62%	88.12%
WordStdDeviation	84.55%	88.61%	87.66%
Grep	74.39%	88.64%	80.94%

6. Conclusion

The need for data compression has been increasing due to the rise of Big Data. Data Storage warehouses have already incorporated compression algorithms, but the systems that process compressed data have not been improved to keep up with the increased usage of compressed data. Current Hadoop lacks a sophisticated design to process compressed data. Our experimental observations show that Hadoop results in a 60-85% longer job runtime for processing compressed data versus processing uncompressed data. In this work, we proposed and developed P-Codec on top of Hadoop to improve its performance processing compressed data. P-Codec consists of 2 main components: a parallelized data decompression (PDD) module and a decompressed map task divider (DTD) module. Using these two components, we improved the job runtime of compressed files by approximately 80%. The use of P-Codec, then, allows users to achieve both performance time and storage savings.

REFERENCES

[1] White, T. (2012). “Hadoop: The definitive guide.” "O'Reilly Media, Inc.".
[2] Dean, J. and Ghemawat, S. 2004. Mapreduce: Simplified Data Processing on Large Clusters. OSDI ‘04. Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation, 2004, pp.137-150
[3] Morgan, T. P. 2014. “How Facebook Compresses it’s 300 PB Data Warehouses.” [Online]. Available: http://www.enterprisetech.com/2014/04/11/facebook-compresses-300-pb-data-warehouse/. [Accessed 10 April 2016].
[4] Miller, R. 2013. “Facebook Builds Exabyte Data Centers for Cold Storage.” [Online] http://www.datacenterknowledge.com/archives/2013/01/18/facebook-builds-new-data-centers-for-cold-storage/. [Accessed 10 May 2016].
[5] “The Digital Universe.” [Online]. http://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm. [Accessed 5 May 2016].
[6] “Vertica.” [Online]. Available: http://www.vertica.com/. [Accessed 15 April 2016].
[7] “RainStor.” [Online]. Available: http://rainstor.com/products/rainstor-database/compress/. [Accessed 21 April 2016].
[8] DeWitt, D., and Gray, J. (1992). “Parallel database systems: the future of high performance database systems.” Communications of the ACM, 35(6), 85-98.
[9] Stonebraker, M., Abadi, D., DeWitt, D.J., Madden, S., Paulson, E., Pavlo, A., Rasin, A., 2010. “MapReduce and Parallel DBMSs: Friends or Foes?” Communications of the ACM, Vol. 53 No. 1, Pages 64-71.

[10] Ahmad, I., He, Y., Liou, M. L., 2002. "Video Compression with Parallel Processing." Elsevier Parallel Computing Volume 28, Issues 7-8, August 2002, Pages 1039-1078.
[11] Chen, Y., Ganapathi, A., and Katz, R. H. (2010, August). "To compress or not to compress-compute vs. I/O tradeoffs for MapReduce energy efficiency." In Proceedings of the first ACM SIGCOMM workshop on Green networking (pp. 23-28). ACM.
[12] "Data Compression in Hadoop." [Online]. Available: http://comphadoop.weebly.com/. [Accessed 21 April 2016].
[13] Pavlo, A., Paulson, E., Rasin, A., Abadi, D. J., DeWitt, D. J., Madden, S., & Stonebraker, M. (2009, June). "A comparison of approaches to large-scale data analysis." In Proceedings of the 2009 ACM SIGMOD International Conference on Management of data (pp. 165-178). ACM.
[14] DeRoos, D. "Input Splits in Hadoop's MapReduce." http://www.dummies.com/how-to/content/input-splits-in-hadoops-mapreduce.html
[15] Vavilapalli, V. K., Murthy, A. C., Douglas, C., Agarwal, S., Konar, M., Evans, R., Graves, T., Lowe, J., Shah, H., Seth, S., Saha, B., Curino, C., O'Malley, O., Radia, S., Reed, B., and Baldeschwieler, E. (2013, October). "Apache Hadoop yarn: Yet another resource negotiator." In Proceedings of the 4th annual Symposium on Cloud Computing (p. 5). ACM.
[16] Thusoo, A., Sarma, J. S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Suresh, A., Liu, H., & Murthy, R. (2010, March). "Hive–A petabyte scale data warehouse using Hadoop." In Data Engineering (ICDE), 2010 IEEE 26th International Conference on (pp. 996-1005). IEEE.
[17] "Hive Language Manual ORC." [Online]. Available: https://cwiki.apache.org/confluence/display/Hive/LanguageManual+ORC. [Accessed 21 April 2016].
[18] Ding, X., Tian, B., & Li, Y. (2015, February). "A scheme of structured data compression and query on Hadoop platform." In Digital Information, Networking, and Wireless Communications (DINWC), 2015 Third International Conference on (pp. 160-164). IEEE.
[19] Haider, A., Yang, X., Liu, N., Sun, X. H., & He, S. (2015, December). "IC-Data: Improving Compressed Data Processing in Hadoop." In 2015 IEEE 22nd International Conference on High Performance Computing (HiPC) (pp. 356-365). IEEE.
[20] "How Many Maps and Reduces." [Online]. Available: https://wiki.apache.org/hadoop/HowManyMapsAndReduces. [Accessed 21 April 2016].
[21] Borthakur, D. 2008. "HDFS Architecture Guide." The Apache Software Foundation.

Biography

Idris Hanafi is currently a graduate researcher in the field of Big Data Systems at Southern Connecticut State University (SCSU). He is also a first year Master Student at SCSU studying Computer Science. He received his Bachelor Degree in Computer Science (Honors) in the Spring of 2015. His research interest comprises of Big Data Systems, Databases, Image Extraction, and Computer Vision.

Amal Abdel-Raouf is currently a professor at the Computer Science Department, Southern Connecticut State University, United States. She is a researcher at the Information and System Department, Electronic Research Institute (ERI), and a member of the Greater Hartford ACM Chapter. Her research interests include Software Engineering, Software Quality, Real Time Systems, Parallel and Distributed Computing, Object Oriented Systems, Computer Networks and Big Data. She is the author of several papers in journals and international conferences in the field of Software Engineering. She received her PhD in Computer Science and Engineering in June 2005 from the University of Connecticut and joined Southern Connecticut State University in August 2005.