Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 154 (2019) 581–587

Procedia Computer Science

www.elsevier.com/locate/procedia

8th International Congress of Information and Communication Technology, ICICT 2019

An Optimal Solution of Storing and Processing Small Image Files on Hadoop

Lu Lu[a,b], Qiu Yan Feng[a,*]

[a] School of Computer Science & Engineering, South China University of Technology, Guangzhou, Guangdong, China
[b] Modern industrial technology research institute, South China University of Technology, Zhongshan, China

Abstract

The rapid development of the Internet, especially mobile Internet, makes it much easier for people to make social contacts online. Nowadays they tend to spend more and more time on social network service, producing a lot of image files. This brings a challenge to traditional standalone framework on handing the continued increasing image files. Therefore, it is advisable to find a new way to face the challenge. Hadoop is a notable, widely-used project for distributed storage and computations with high efficiency, data integrity, reliability and fault tolerance. Hadoop Distributed File System and MapReduce are two primary subprojects respectively for big data storage and computations. However, Hadoop do not provide any interface for image processing. Worse, both Hadoop Distributed File System and MapReduce have trouble processing large amount of small files, decreasing efficiency of files access and distributed computations. This prevents us from performing images processing actions on Hadoop. This paper proposes a method to optimize small image files storage on Hadoop and self-defines an input/output format to enable Hadoop to process image files.

Keywords: Hadoop Distributed File System (HDFS), MapReduce, small images files, selfdined storage and IO format;

1. Introduction

Hadoop is widely used in the recent years when big data and distributed computing are attracting more and more attentions. As a famous project of Apache, Hadoop is an open-source java-based software framework with the

10.1016/j.procs.2019.06.092

subprojects including Hadoop Common, Hadoop Distributed File System, Hadoop Map Reduce and Hadoop Yarn [1]. This framework shows its irreplaceable merits of high reliability, availability and data integrity while dealing with large data storage and processing massive distributed computations.

At the meantime, the rapid development of mobile technology and the wide spread of the Internet give rise to the time people spend on social networking service every day. They chat and share their life by pictures or videos with friends whenever they access to the net. Take Weibo as an example. According to The Report of Weibo Users Development 2016[2], the number of monthly active users on Weibo have reached 297 million while daily active users have got to 132 million by September, 2016. The report also revealed that these users active on Weibo mainly write tweet in the form of images with some text description, which accounts for 60%. We can simply estimate that even if we take Weibo alone into consideration, about 70 million of images will be created, stored and uploaded every day. And clearly there are many other network services similar to Weibo, popular and producing large amount of data especially images data. Thus, it is hard for traditional standalone framework to store and handle such a large number of image files. A solution to this problem is needed.

Considering that Hadoop is created to handle big data, turning to Hadoop for help can be a good choice. However, since Hadoop is designed for text file, there is no interface offered to store and process image files. Besides, big data does not mean a great amount of data. Though Hadoop is useful in storing and processing big files, it has great weakness in the face of numerous small files, while image files are always not very big. On the one hand, Hadoop Distributed File System (HDFS) stores data on the node called Data Node in the cluster, and the only one Name Node will keep the metadata of every file on the Data Node. Therefore, when it comes to a lot of small files like images, Name Node will soon run out of its memory [3]. On the other hand, the more files there are, the more MapReduce tasks are needed, and thus more interactions and communications between nodes will be necessary too, which will add to the overhead and clearly decrease the efficiency of the cluster. Hence this paper presents our solution to enable small image files storing and processing on Hadoop and optimize the performance so that it can help managing the large amount of image files flowing through the network nowadays.

2. Backgroud

2.1. Small files and image-processing problems on hadoop

A great number of image files are being produced at the time people surfing the Internet with their computers or smartphones. As netizens increase with great speed, we need to come up with new solutions to handle these image files. Hadoop is designed for big data so theoretically is a rational choice for us. But Hadoop have no interface for users to operate image files, designed to process big text file. Also, Hadoop do not work well in large number of small files. Generally, a file smaller than 64MB (the default size of a block) is thought to be a small file and will be split into an individual block in HDFS. Just like what is introduced before, Name Node will keep the metadata of each file storing in the Data Node. Therefore, if there are lots of small files in the Data Node, Name Node will have to take much storage space to hold a large amount of metadata. For example, assuming that we are going to store a 1GB big file in HDFS, it will be split into 16 blocks. The size of metadata for one block is about 150 bytes. So, the 1GB file takes up 2.4 KB storage space in the Name Node. However, if there are 2048 small files of 512KB, the size of their metadata will increase by 125 times, to 300KB. In addition, with more blocks in the HDFS, the more time will be taken to search for a required file block. Besides, speaking of the impact on MapReduce, more blocks mean that more MapReduce tasks have to be established to finish the job. Since a MapReduce job only processes a block of data (a small file), it is a waste of the potential, sharply decreasing the performance. Therefore, to apply Hadoop to store and process so many small image files, it is imperative to self-define an image file format and do some improvement on HDFS and MapReduce.

2.2. Existing solutions to small files problem

Small files problems can be avoided by transforming small files into a part of a large file. The key is how to group the files together so as to maximize the performance of Hadoop. Here are some existing solutions about this.

Sequence File, provided by Hadoop, is a kind of flat file of binary key/value pairs. Acting like a container, Sequence File merges the small files into a big file in some way. Key/values pairs make up the SequenceFile. Key here refers to the name of the small file, and value is the file content.

Nevertheless, Sequence File does not support adding a key/value pair to the end of the file. Additionally, since Sequence File do not record the mapping relation between itself and the small files constituting it, it is timeconsuming to search a certain file in it by traversing the whole file..

Also provided by Hadoop, Map File can be treated as Sequence File with an index. Map File can be divided into two parts, index and data. Index stores the mapping relation of the Map File and its small files. Data is made up of a series of key/value pairs, in which key is the file name and value is the content, similar to Sequence File. Now that index marks down the offset of each key/value pair in the data, Map File has a better performance in looking for a certain file than Sequence File. But to pay for that Map File costs more storage space. Similarly, adding new small files to an existing MapFile is not allowed.

3. Proposed Solution

What is elaborated above indicates that MapFile is better to store image files than Sequence File with better performance. But either of them allows users to modify the file, which leads to great inconvenience since in real life people might need to do some change based on the original file instead of generating a new one. So, this paper puts forward a solution to optimize small image files storage in HDFS by self-defining an image storage file format, naming it Image Merge File. At the same time, to overcome the difficulty of lacking in APIs for image processing on Hadoop, Image Input Format and Image Output Format are defined to provide the I/O interface for MapReduce jobs.

3.1. Small files and image-processing problems on hadoop

MapFile is of high efficiency in storing and reading the file because of the index file. The design of Image Merge File learns from that advantage and meanwhile is open to being appended. See Fig 1 for the details of the structure of Image Merge File. As shown in the figure, an Image Merge File composes of an index file and a data file. Index file stores each image file's name and offset in the data file as a key/value pair. The offset part is a String containing the point where the small image's data begins and ends in the data file, which makes it possible and convenient to locate the image rapidly in the data file. The data file is simply all images' data jointing together in a certain order. What's more, it is available to append extra image at the end of the file.

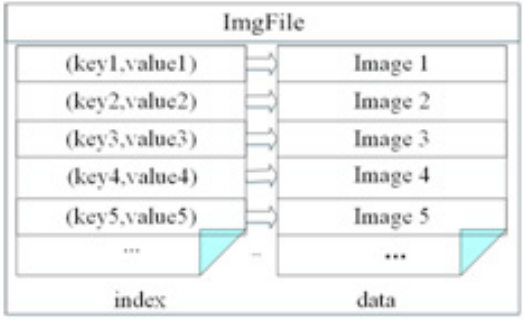

Fig.ure 1. The Structure of Image Merge File .

Image Merge File takes lots of image files as input. In its implementation, first read the configuration file written by the users to get the path of input files and the output directory. Actually, users just need to fill in the blank in the configuration file with some important arguments like file path. Next is to start a loop collecting the images from input path together. In the loop, for each small image file, designate its name as key and its offset as value. This String key/value pair will be stored in an object of a kind of data structure named Tree Map. Tree Map, based on

Red-Black Tree, is a sorted HashMap provided by Java. By default, the Tree Map sorts in lexicographical order of the key. Then save the Tree Map object into index file by serializing it. Serializing an object means encoding it into a byte stream. Java has offered object serialization mechanism to help with this. The actual image data is added to the data file. In this way the order of the keys matches the order of the image data added to the data file since files are read in lexicographical order by default from the file system. This helps a lot in speeding up searching a certain image in the Image Merge File. What is needed to extract an image is its file name. Take it as key and then we can get the value (offset) from the Tree Map object in the index file (by deserializing), and eventually locate it in the data file and read it. In order to serve the goal of appending an image at the end of the Image Merge File, the output path has to be detected first. If the path is discovered to be already existed, it means that the Image Merge File has been created before. What it is supposed to do at this moment is to open it and add content to the end of the file instead of creating a new one. This is the optimal design of storing a bunch of image files in HDFS. It is just the first step to take to process lot of image files in Hadoop since when the way of storing the images is changed, we have to redefine the Input Format and Output Format in Map Reduce jobs to adapt to the change. We will discuss it in detail in the next part.

3.2. Image processing with image merge file by map reduce

MapReduce takes key/value pairs as input and generates key/value pairs as output. By consulting the source code, it is concluded that the Input Format interface is in charge of controlling the input files, defining how to split the file and read the record. And Output Format is applied to define the format of output files. The Input Format interface is defined in org.apache .hadoop. map reduce package, including two main functions: get Splits() and create Record Reader(). Inside the get Splits() function, file will be split into Input Splits in a particular way. One Input Split will be assigned to one mapper (the Task Stracker conducting a map job). The create Recoed Reader() function creates a Record Reader, which states the way to read a record from the Input Split. In other words, Record Reader transforms the Input Split into key/value pairs so that they become available for the mapper. There are plenty of built-in implementations of the Input Format such as Text Input Format and File Input Format. Text Input Format takes every line of a file as a key/value pair. The key is the offset of the line and value is the content this line [4]. The Output Format interface specifies where to store the output data and how to write them into a file. Get Record Writer() function and check Output Specs() function are two primary functions of the interface which need to be covered. Get Record Writer() creates a Record Writer, inside which there is a write() function defining the form of the key/value pairs. Except write() function, a close() function is used to close the output stream. And additionally the check Output Specs() function will check the availability of the output path before a MapReduce job takes action. Therefore, to perform distributed computations for our self-define Image Merge File in the Hadoop cluster, it is a necessity to define an Image Input Format implementing Input Format interface and an Image Output Format implementing Output Format interface. On the basis of what was discussed above, here an image processing framework is designed, shown in Fig 2 with self-defined Image Input Format and Image Output Format. During the map phase, Image Input Format will first load the index file to get the offsets and determine the number of splits, after which Input Splits are created according to the split amount and image data from data file. Next its Image Record Reader makes input splits available by separating them into key/value pairs in the forms of <Text, Bytes Writable>. That is to say that Image Record Reader will set the image's file name as Text type and convert the image into Bytes Writable type. Text class and Bytes Writable class are the data types in Hadoop implementing the Writable Interface, which is the core of Hadoop Serialization Mechanism. Transforming the image to Bytes Writable benefits a lot in transferring them between nodes, making the best use of the bandwidth and reducing the consumption of transmission time. These pairs will be pushed to the mappers. After mappers and reducers finish processing each record (each image), they output it to the HDFS as what the Image Output Format defines. Before outputting the image files, we can merge them as Image Merge File again to reduce the burden of the Name Node.

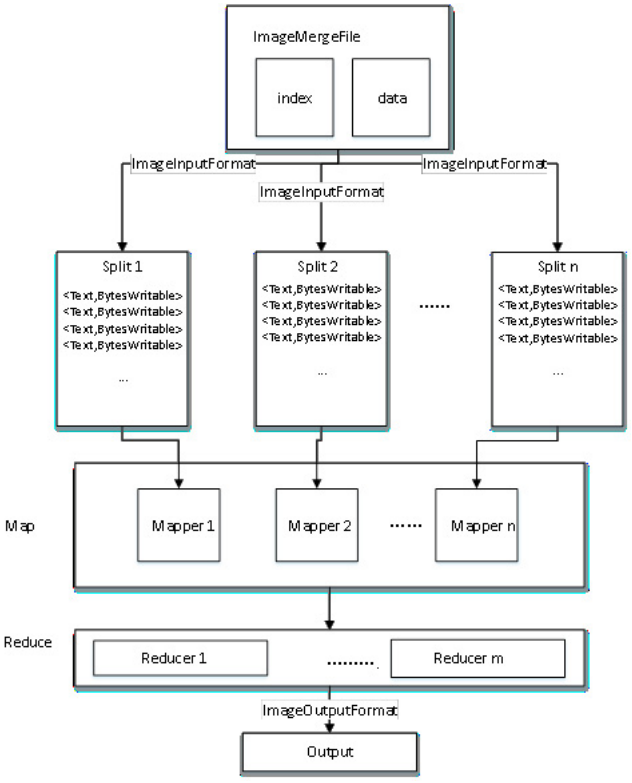

Figure. 2. Image Processing with Image Merge File.

Define class Image Input Format extending File Input Format and rewrite the get Splits() function. Inside the class, at first read the configuration file and get the location of input Image MergeFile and the number of the splits users want to divide into (Users should always set this number based on the status of the cluster). Then read the index file into a Tree Map object. The split strategy used here in the get Splits() function is to assign the same amount of image files to each split, since every small images being processed should have similar small size. That's the reason why the splits amount should be set in advance by users. The number of the images can be gained from the index file by the Tree Map object and then how many image files each split holds can be determined. If the total number of images is m with n splits, there will be m/n images in every split. The exception will be added to the former splits. After confirming the number of images each split, copy the actual image from the data file to the split according to the offset in index file. Also, it is needed to define an Image Record Reader to read a single image (one image as a record) from our self defined splits into a key/value pair. The most important functions to override inside the class are initialize() and next Key Value(). Function initialize(), as its name implies, do some initialization before beginning to read. This function accepts the input split and open it, marking down its start offset and length. In order to extract an image from the split, it is necessary to load the index file here to get the size of each image in the splits, which helps to read the exact and complete image in the next Key Value() function. Function next Key Value() is invoked every time when the mappers are in need of a key/value pair as input. It sets the image name in Text type as key and the image content in Byte Writable type as value, copying the image date from the split. The last step to take is to locate the inputstream forward to the next image, preparing the next copying operation until the end of the split. Define Image Output Format extending File Output Format and override the get Record Writer() returning a self-defined Image Record Writer. Image Record Writer writes the image to HDFS by write() function and close the outputstream by close() function. After defining all the stuff adapted to the Image Merge File, it is doable to process image files with MapReduce now. First start a MapReduce job and set the input path (the path of the Image Merge File) and output path. Then define our own mapper and reducer. Mapper performs some image-processing operation like binaryzation on each image and reducer outputs the images. Off course the input format (Image Input Format)

and output format (Image Output Format) should be specified when configuring the MapReduce job. Then we can handle lots of small image files with better efficiency by mean of Hadoop overcoming its bottleneck. The whole process is shown as follows:

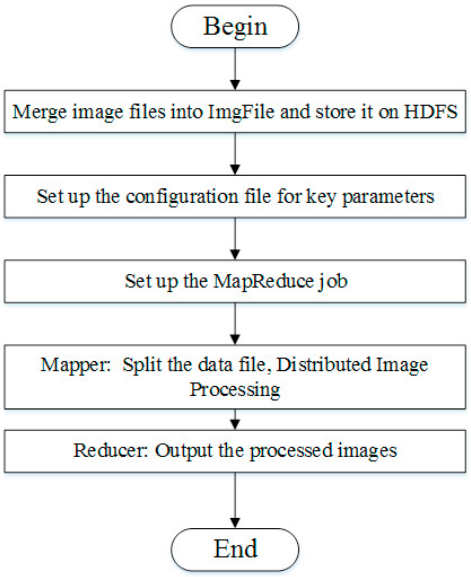

Figure. 3. The flow chart of image processing..

4. Conclusion

This paper presents a complete process of handling small image files on Hadoop overcoming its shortcomings of lacking interface for image processing as well as low efficiency in dealing with the storage and management of small files. In the respect of storage, a method is put forward to group all the small image files into a big file named Image Merge File before uploading them to HDFS. Image Merge File is divided into two parts, index and data, supporting addition of new file at the end of it. Then to make it possible for MapReduce to do with the Image Merge File, the redefinition of input format and output format was presented.

Acknowledgement

This research was supported by the National Nature Science Foundation of China (No. 61370103), Guangdong Province Science & Technology Fund (2015B010131013), Guangzhou Province Science & Technology Fund (201802020006) and Zhongshan Produce & Research Fund (2017A1014).

References

1. Mohandas .N and Thampi. S. M, “Improving Hadoop performance in handling small files,” Advances in Computing and Communications, 2011, p.187-194.
2. Weibo Data Center, The Report of Weibo Users Development 2016, http://data.weibo.com/report/reportDetail?id=346
3. Bende.S and Shedge .R, “Dealing with Small Files Problem in Hadoop Distributed File System,” Procedia Computer Science, Vol. 79, 2016, pp.1001-1012.
4. White. T, “Hadoop: The definitive guide,” O'Reilly Media, Inc., 2012.
5.Apache Hadoop, “HDFS Architecture Guide,” <http://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/HdfsDesig n.html>, 2016 (accessed 17.04.15).
6. Ghazi.M.R and Gangodkar. D, “Hadoop, MapReduce and HDFS: A Developers Perspective,” Procedia Computer Science, Vol. 48, 2015, pp.45-50.
7. Dean.J and Ghemawat.S, “MapReduce: simplified data processing on large clusters,”Communications of the ACM, Vol. 51(1), 2008, pp.107-113.

8. He.H, Du.Z, and Zhang.W, et al, “Optimization strategy of Hadoop small file storage for big data in healthcare,” The Journal of Supercomputing, Vol. 72(10), 2016, pp. 3696-3707.