2014 Sixth International Conference on Intelligent Human-Machine Systems and Cybernetics

A Strategy to Deal with Mass Small Files in HDFS

Shuo Zhang, Li Miao, Dafang Zhang, Yuli Wang
Department of Computer Science and Engineering
Hunan University
Changsha, China
Email: aaaim@hnu.edu.cn, miaoli2000@163.com, dfzhang@hnu.cn, 162025638@qq.com

Abstract—HDFS performs badly in storing and managing a great number of small files as a result of the great memory occupation of the single Namenode and massive seeks and hopping from datanode to datanode. Traditional solutions are only efficient for specific file size or file format. In this paper, we evaluate the performance of some different solutions such as Hbase and Avro. Then in order to compensate for the lack of their inefficiency for middle size small file, we implement a merging and prefetching mechanism. Finally for the purpose of reducing the influence of different file size distributions, we present a strategy of using different schemes for small files of different sizes. Through the experiments of performance comparison, it can be demonstrated that the strategy can improve the original HDFS's writing and reading performance by about 70%.

Keywords- HDFS; HBASE; Avro; small files; strategy

I. INTRODUCTION

HDFS is a very popular way to store TB or PB level data because of its scalable, reliable, and low-cost storage capability. However, large amounts of small files in HDFS may cause the small file problem [1], because the high memory usage and huge metadata requests will make NameNode become a bottleneck of the system.

We have learnt about some people's work on the problem. Paper [2] proposed a solution for a system called WebGIS to store small files efficiently in HDFS. Paper [3] made a solution for the PPT file's storage for Bluesky Systems. Paper [4] tried to improve the speed of massive small files storage. Paper [5] used the Sequence File for the storage of MP3 files. The TFS [6] is designed and optimized by Taobao to provide storage for small images, web pages within tens of KB. FastDFS [7] is a high performance distributed file system which can resolve the high capacity and load balancing problem. HayStack [8] is a Facebook's photo storage system which provides storage for about 260 billion pictures with less cost and higher performance.

Some solutions can only be applied to a particular file format or file size distribution. Therefore, we propose a merging and prefetching mechanism. Furthermore, we present a strategy by integrating different solutions. Finally a series of experiments were done to demonstrate its high efficiency with no limits of file size distribution and file formats.

II. BACKGROUND

A. HDFS and Small Files Problem

The HDFS [9] is designed to store and process large data sets and storing a large number of small files in HDFS is inefficient. The NameNode stores the entire metadata in the main memory for faster and efficient servicing of client requests. When files stored in HDFS are less than the block size (default 64M), they will be treated as small files and each of them will form separate blocks whose corresponding metadata occupy some NameNode's memory. When the amount of files reaches a certain number, the HDFS's performance bottlenecks will occur. For example, assume that, metadata in memory for each block of a file takes up about 150 bytes. Consequently, for sixty 1 GB files, divided into 960 64MB blocks, 144KB of metadata is stored. Whereas, for 60,000,000 files of size 1KB each (total 60 GB), about 9GB metadata is stored. Furthermore, HDFS performs inefficiently while accessing a large number of small files.

B. Some solutions to dealing with small files problem

In order to solve the problem of small files, Hadoop itself offers several solutions such as Hadoop Archive (HAR) [10] and Sequence file [11]. HAR merges several small files to a file with *.har extension and it still allows transparent access to the small file. Avro and Hbase can also be used to deal with mass small files. Avro [12] is a data serialization system which stores the user-defined patterns and specific data encoded into binary sequence into an object container file. Hbase is an open-source, distributed, versioned, non-relational database allowing you to store large amounts of data in compact files while at the same time allowing random access to them.

III. METHODOLOGY

A. File Processing Strategy

According to some relevant knowledge, we find Hbase is more suitable very small file (such as 1K) and its query efficiency is very high. However, for bigger files (such as 3M), Hbase is rather time-consuming. The Avro file can deal with these files more efficiently than very small ones but while the number of small files is large, searching for the required file is slow. Besides, they are both not very

978-1-4799-4955-7/14 $31.00 © 2014 IEEE
DOI 10.1109/IHMSC.2014.87

suitable for the middle size files (such as 300K). Therefore, we present a strategy by combining different kinds of solutions to deal with massive small files in HDFS.

Before putting some files from client to HDFS, there must be a judgment. If the file's size is more than 64M, it will be judged as a large file and directly stored in HDFS. In the case that the file size is less than 50K, it will be stored in HBase. If it is more than 1M, it will wait to be merged with another file, which is also more than 1M, to an Avro file. In the other situations, the files will be merged to bigger ones in accordance with the designed merging mechanism and small files can be rapidly retrieved by the designed index and the prefetching mechanism. While reading a single small file, it will be searched in the Hbase, Avro files and the merged files simultaneously. Once it's found, reading operation will be terminated. Fig. 1 shows the overall system architecture of the proposed approach.

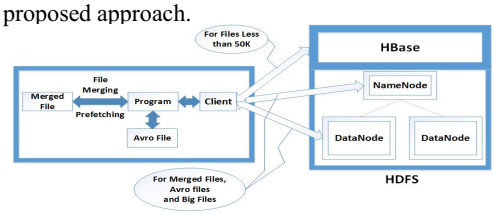

Figure 1. System Architecture

B. File Merging and Index Generated

It is an effective way to reduce the number of small files to merge them to big ones. The NameNode maintains metadata only for the combined file and not for all the small files present in it. If the size of a combined file is as the same as the chunk size, the number of files will significantly decrease. However, it's usually impossible to ensure that the merging size is equal to the file chunk size just right. Therefore, there must be a blank area which is insufficient to accommodate a new small file. If a small file is across two file chunks, probably not in the same DataNode, it will be inefficient to retrieve this file. Consequently, a judgment is required before merging a new small file. If the new small file's size exceeds the default block size (64M) by adding the current file's size, it will be merged into another file chunk. The file merging method is shown in Fig. 2.

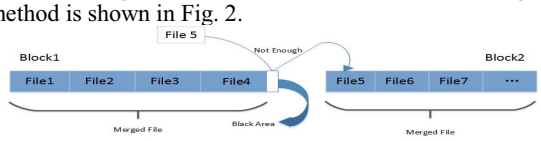

Figure 2. File Merging

It is necessary to store the index information of small files while merging them together. Here a HashTable is used so that the small file's size and length in the merged file can be recorded timely. At last the HashTable will be persisted to an index file, whose definition is shown in Fig. 3 below.

filename1	offset	length	blockId	filename2	.....

Figure 3. the Index File Information

The first small file requests to be written to HDFS, a combined file will be created and its content is copied from the first small file. Simultaneously, a HashTable is created in the client's memory and the first small file's filename, content length, offset in the combined file will be put into the HashTable. The combined file will be put into a waiting queue, then as another small file requests, its file contents will append to the end of the combined file and insert its corresponding information into the HashTable. This process cycles until the new requesting small file's size exceeds the default block size by adding the current file's size. Then the new requesting small file will be merged to another chunk. Finally the HashTable is persisted to an index file and the merged files are written into HDFS with the index file together. Until no new file requests, the file merging and writing will be finished. Fig. 4 shows the mapping structure of small files.

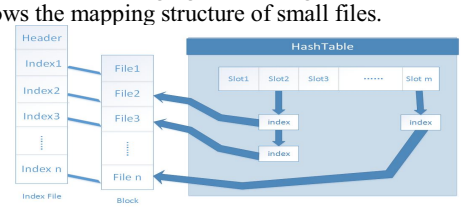

Figure 4. the Mapping Structure of Small Files

C. File Prefetching and Reading

While accessing small files, requesting the metadata from the NameNode is necessary through RPC (Remote Procedure Calls). If the number of small files is too large, massive requests with high frequency will bring a heavy burden to the NameNode.

Our prefetching method solves the problem by reducing the number of requests. While a small file reading operation is requested, the metadata of the other small files in the same block will be obtained together and cached in the client. When reading another small file whose metadata has already existed in the cache, there is no need to initiate another RPC request to the NameNode. Thus, requests are significantly reduced by caching multiple small files' metadata through a single RPC. Therefore, it becomes efficient to read small files with less requests sent to the NameNode.

When file reading operation requests, the filename must be given. Then the index file will be loaded into a HashTable, and we can know the requesting file's offset and length in the combined file which it belongs to and where the combined file is according to the file name sought in the HashTable. A Remote Procedure Calls (RPC) request will be initiated to the NameNode to obtain the metadata associated with the combined file. This metadata provides information about the list of blocks containing the file and the DataNodes that hold these blocks. Based on all the information above, we can successfully extract the small file. Fig. 5 shows the prefetching and reading mechanism.

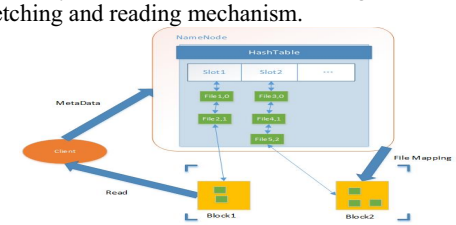

Figure 5. the Prefetching and Reading Mechanism

D. HBase and Avro

HBase is an open-source, column-oriented, distributed, non-relational database modeled after Google's Bigtable [13]. There are an arbitrary number of columns and a unique row key [14] in each row of an HBase table. And a full column name consists of a column family and a column qualifier. In this paper, the filenames are stored as row keys and the file contents as column values. The structure of the table is shown in Table 1.

TABLE I. THE STRUCTURE OF HBASE TABLE

Row Key	Timestamp	Column Family	
		Info	*Value*
Filename1	T1		

Avro is a data serialization system which relies on schemas. Because Avro stores the user-defined patterns and specific data encoded into binary sequence into a large containing file, you can save many small files in a single Avro file in HDFS to reduce the NameNode memory usage. Besides, you can retrieve the original files in MapReduce. Fig. 6 shows how to store small files in Avro.

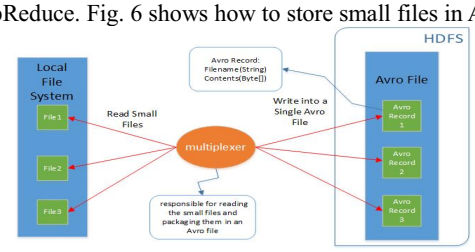

Figure 6. Store Small Files into Avro File

IV. EXPERIMENT AND EVALUATION

The experimental test platform is built on a HDFS cluster of 4 machines. The server configuration is two Inter Core i3-3240 3.4GHz CPU, 4GB of memory and 500GB hard drive. Each node has installed Ubuntu12.04 operating system. The Hadoop version is 1.2.1, and Java version is 1.6. The HDFS minimum block size is set to 64MB. Each experiment was repeated for three times and the average values were calculated and used for analysis.

A. Memory Usage Test

The size of these files range from 1KB to 10MB including various file formats. Files less than 1M account for 90%. The distribution of file sizes is shown in Fig. 7.

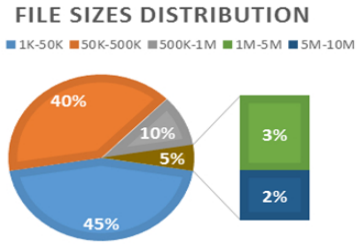

Figure 7. File Sizes Range

For original HDFS, Hadoop archive (HAR) and the proposed approach, the main memory usage of NameNode is monitored when the system stores 20000, 40000, 60000, 80000 and 100000 files respectively. The experiment results are shown in Fig. 8.

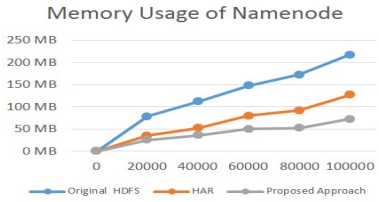

Figure 8. Memory Usage of NameNode

As expected, HAR and the proposed approach achieve much better efficiency of storing small files than original HDFS, due to their file archiving facilities. Particularly, the proposed approach further obtains less memory than HAR because of its local index file design and combining multiple solutions' advantages.

B. Writing and Reading Test for Different Numbers of Files

The workload for memory usage experiment is also used to measure the time taken for writing operation and reading operation. The performance of original HDFS, Hbase, Avro and the proposed approach were evaluated by comparing the consumed time depicted in the graph shown in Fig. 9.

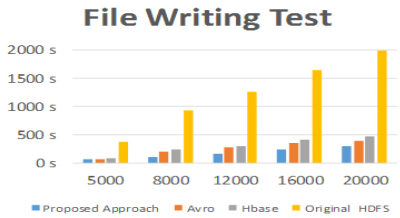

Figure 9. Time Taken for Writing Operation

The writing operation with Hbase, Avro and the proposed approach is considerably faster than the writing operation in original HDFS, which can be indicated by the results of experimental analysis above. It can be seen from the Figure 10 that time significantly increases with the growth of the number of files in original HDFS. Hbase is a little slower than Avro and the proposed approach is almost as the same as Avro.

The experiment of reading operation is done in the manner of extracting 40% of all the files in each group. That is to say, 2000, 3200, 4800, 6400 and 8000 files are respectively retrieved. Moreover, these retrieved files maintain the distribution in Figure 8. Experimental results were shown in Fig. 10.

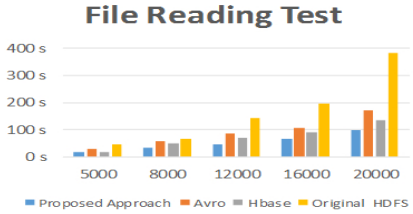

Figure 10. Time Taken for Reading Operation

According to the experiment, reading small files from Hbase is faster than Avro file. The proposed approach has

much better performance than other solutions and with the growth of file amount, more time can be reduced. In general, the proposed approach only takes about 30% of the original HDFS to complete the reading operation.

C. *Writing and Reading Test for Different Distributions of Files*

In addition to the influence of file number, the efficiency of file writing and reading differs from the changes of file sizes distribution. Hbase performs especially badly while the small file is large enough. It is not always efficient to apply a single method to all the small files. Before processing a file, there must be a judgment for file size and the corresponding scheme. In this experiment, four groups of files were tested. All groups have 5000 files, the distributions of which are different from each other. Four kinds of file distributions can be seen in Fig. 11.

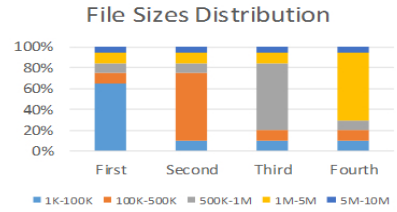

Figure 11. File Sizes Distribution

The time consumption of writing and reading operations is recorded for the original HDFS, Hbase, Avro and the proposed approach. The results are shown in Fig. 12.

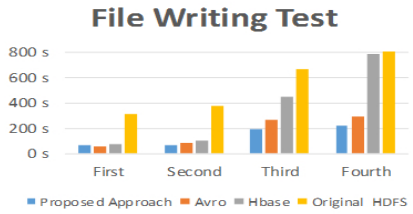

Figure 12. Time Consuming for Writing Operation

It can be seen that while bigger files' proportion rises, Hbase performs worse, but Avro and the proposed approach are still much more efficient than the original HDFS.

As for the reading operation, 2000 files were randomly extracted from all 5000 files for each group. Furthermore, the extracted files remain the distribution of file sizes in each group. In other words, there are 40% files retrieved in all file size segments. Time consuming of each group is compared in in Fig. 13.

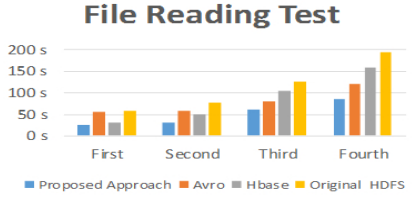

Figure 13. Time Consuming for Reading Operation

The query performance ranges by different file distributions. Different solutions can be applied to different file sizes. The proposed approach can deal with small files of multiple distributions efficiently and save about 80% time compared to the original HDFS.

V. CONCLUSION

When massive small files were stored on HDFS, high memory usage was caused by huge numbers of files and high read cost was caused by the existing file access mechanism. Some traditional solutions are only efficient for specific file size or file format. In order to deal with the inefficiency for middle size small file, we propose a merging and prefetching mechanism. For the sake of the efficiency of different file size distributions and file formats, a strategy is presented by integrating different solutions for different file sizes. At last, a series of experiments demonstrate that the proposed approach reduces the load of NameNode significantly and improves the performance in storing and accessing small files by about 70%. Utilizing our proposed strategy, small files can be handled effectively in HDFS. In the future, a faster merging mechanism maybe designed and multiple nodes can be designed to store the metadata of small files.

ACKNOWLEDGMENT

The research work is supported by the National Natural Science Foundation of China (61272546).

REFERENCES

[1] Tom White, The Small Files Problem, http://www.cloudera.com/blog/2009/02/02/the-small-files-problem/.

[2] Xuhui Liu, Jizhong Han, Yunqin Zhong, Chengde Han, and Xubin He, Implementing Web GIS on Hadoop: A Case Study of Improving Small File I/O Performance on HDFS, Proceedings of the 2009 IEEE Conference on Cluster Computing, DOI:10.1109/CLUSTR.2009.5289 196.

[3] B. Dong, J. Qiu, Q. Zheng, X. Zhong, J. Li and Y. Li. A novel approach to improving the efficiency of storing and accessing small files on hadoop: a case study by powerpoint files. 2010. IEEE International Conference on Services Computing, SCC, IEEE (2010), pp. 65-72.

[4] Liu Jiang, Bing Li, Meina Song. The optimization of HDFS based on small files' Proceedings of IC-BNMT, 2010:912-915.

[5] Xiaoyong Zhao, Yang Yang, and Lili Sun. Hadoop-based storage architecture for mass MP3 files. [J]. Journal of Computer Applications, 2012, 32(6):1724-1726.

[6] Taobao File System official site, http://tfs.taobao.org/.

[7] FastDFS, http://code.google.com/p/fastdfs/.

[8] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, Peter Vajgel. Finding a needle in Haystack: Facebook's photo storage.

[9] K. Schvachko, H. Kuang, S. Radia, R. Chansler. "The Hadoop Distributed File System". In Proceedings of IEEE 26th symposium on Mass Storage Systems and Technologies (MSST), Incline Village, Nevada, USA, May 2010.

[10] Hadoop archive official javadoc, http://hadoop.apache.org/common/docs/currentlhadoop_archives.html.

[11] Sequence File official wiki, http://wiki.apache.org/hadoop/SequenceFile.

[12] Avro official site,http://avro.apache.org/

[13] BigTable wiki, http://en.wikipedia.org/wiki/BigTable.

[14] O'Reilly Media, Inc. HBase: The Definitive Guide.