An optimization strategy of massive small files storage based on HDFS

Xun Cai[1,a,*], Cai Chen[2,b] and Yi Liang[3,c]

[1] Beijing University of Technology, Beijing, China

[2] Beijing University of Technology, Beijing, China

[3] Beijing University of Technology, Beijing, China

[a]morethancx@163.com, [b]chencai@bjut.edu.cn, [c]yliang@ bjut.edu.cn

*Corresponding author

Keywords: Storage of Small Files, Distribution of Small Files, Merge, Relationship between files.

Abstract. Nowadays, Hadoop distributed file system as a distributed storage system, has a good effect on the storage of large files. However, there is a natural flaw in the storage of small files: storing a large number of small files will produce excessive metadata, resulting in namenode memory bottlenecks; frequent RPC communications will cause time consumption due to over-provisioning. To solve these problems, this paper presents a merging algorithm based on two factors: the distribution of files and the correlation of files. The algorithm can not only reduce the HDFS blocks, but also make relevant files close. Experimental results show that the algorithm effectively improves the storage efficiency of HDFS on small files and help to optimize the access of small files.

Introduction

With the rapid development of Internet technology in recent years, the era of big data has arrived. Due to the growing demands for data storage and access, many distributed technologies and frameworks emerge as the times require. One of them is Apache Hadoop[1]. Because of its advantages on robustness, scalability and cost to other platforms, Hadoop has become the most widely used Big Data analytics platform today. Hadoop Distribute File System (HDFS)[2] has the characteristics of high throughput and high fault tolerance, and has become the most important data storage framework in the industry.

The HDFS framework is a master-slave architecture. The master node Namenode is responsible for providing the metadata service; the slave node Datanodes is responsible for storing the data by blocks. However, HDFS can not cope with the storage and access of a large number of small files: when storing small files, it generates too much metadata information and runs out of memory of the Namenode; when accessing small files, clients frequently make calls through RPC Getting metadata about the block where the small file resides adds significant time costs. In addition, HDFS allocates 128MB of data blocks for each file for storage, and the independent occupation of blocks will causes storage space fragmentation.

At present, one of the solutions to the small file problem on the HDFS is to merge the files into bigger files and then store them. Two factors need to be considered in the process of merge: 1) the relevance of small files, to ensure the efficiency of small files; 2) the distribution of small files, to ensure maximum use of storage space, optimization the processing efficiency of platform. This paper fully considers and weighs the above two factors. Based on the quantitative analysis of file relevance, this paper proposes a uniform merge algorithm based on file relevancy.

Related Work

Based on the above two factors, many efforts have been made to optimize the storage of small files on HDFS.

Article [3,4] propose two merge strategy for small files base on correlation of files and special scenarious. Article[3] is to solve BlueSky system PPT files stored problems, small files of the same PPT courseware are combined into one large file; article[4] consider the characteristics of WebGIS data system, combine the small files that has adjacent geographical location information and less than 16MB into 64MB files, and all the small files were established a hash index. Based on these two articles, there are many other mergers emerged. Literature[5] introduced the concept of priority between files. Firstly add the small file to the merge queue and select a file to be merged and sort other files in the merge queue by priority. Priority depends on two conditions: the user and the upload time of files. All of files are divided into four levels, and then according to the 0/1 knapsack problem to merge. Literature [6,7] involve the merge of educational resource files. The former combines all the small files belonging to the same large file into a file. The latter uses the VSM algorithm to judge the association between small files and calculates the cosine of the text feature vector between the two small files, and merge according to the cosine value. The considerations of these strategies are based on the relevance of the merger documents, and according to different scenarios designed different combination of strategies. Literature[8] obtained the correlation of small files by analyzing a large number of small files access logs, and described the correlation degree of files more accurately.

In addition to considering the relevancy between small files, the distribution of small files within a block of files is also an important factor. If the size of the merged file exceeds the size of the HDFS file (64 MB), metadata will be added to the Namenode, and the blocks will be fragmented. article[9] considered the distribution of small files in the block first, and proposed the Tetris merge algorithm. The core idea is based on the balance of small files, ensure that they are evenly distributed, and then spread to the merged large files, thus ensuring the merged large files in HDFS will not be divided into extra blocks. Based on the distribution of small files, literature[10] proposed OMSS algorithm to improve MapFile's merging process and used the worst fit strategy to merge the files into MapFile. However, none of the above methods considered the impact of file relevancy on the efficiency of the merged files. Literature[11] proposed a double-merging algorithm, taking into account the relationship between the file and the balance of data block. However, this method only consider the same user's file resources as the related files, it does not fully tap the correlation between the files.

Design and implementation of small files merging based on two factors

Basic idea of algorithm

For the combination of small files, the correlation of files and the balance of merged files must be considered synthetically. For example, the following occurs when the file is merged, which is shown as Fig. 1: There are four small files, which are A: 60MB, B: 40MB, C: 20MB, D: 80MB, among them A, B and C's correlation degree is higher than the correlation degree with D. Now it is merged into two merged files. The following two situation may appear. Scenario A: *mergefileA* contains A and B with a total size of 100MB. *mergefileB* contains C and D with a total size of 100MB; scenario B: *mergefileA* contains A, B, C, size 120MB and *mergefileB* contains D, size 80MB. The two have the same result in the distribution of files, and even the distribution of files in scenario 1 is more uniformity. However, given the factor of document relevance, Option in scenario 2 is preferable. Because it merges into two file blocks, placing three highly relevant files in the same block.

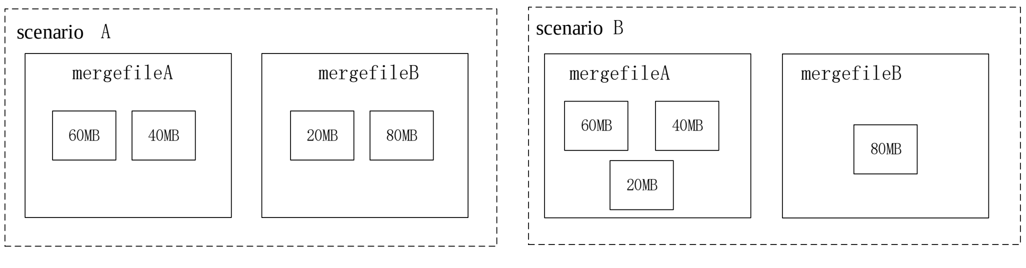

Fig.1 Two scenario of merging files

This paper presents a small file merging algorithm based on file distribution and file correlation (MBDC). The algorithm takes full account of the degree of correlation between the small files, and then make evenly distribution, finally merged into a large file. The algorithm not only reduces the number of merged files, equals to the number of blocks, to a certain extent, thus reducing the load of Namenode nodes, but also ensures that the related files are located adjacent to each other and the file access is more efficient.

Correlation of files

In order to find the association between small files more objectively and accurately, it is necessary to analyze history access logs. The analysis of the history of the file access log will produce a real and objective user access to each file, and then get the correlation between the files. Literature[8] provides a way to get a collection of related files from the logs. It draws a collection of file relations whose relevance is above some certain thresholds. Based on this, this article divides and weights the collection, different weights represent different levels of correlation and serve as the correlation of small files. As Fig. 2 shows.

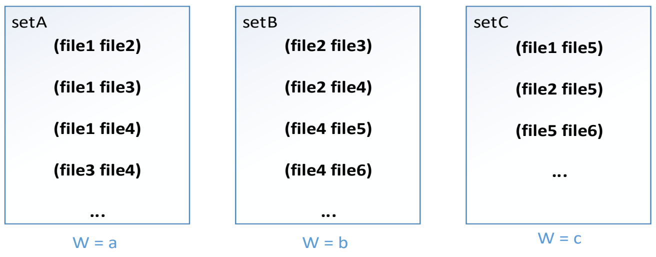

Fig.2 Sets of file relationship

Definition 1 *Correlation(file1,file2)*. It represent the correlation value between file1 and file2.

$$\text{Correlation(file1,file2)} = W(S),\ (\text{file1,file2}) \in S. \tag{1}$$

S is the set of relationship and relationship *(file1,file2)* is belong to *S*. *W* is the weight of *S*.

Design of algorithm

Before introducing the design of this algorithm, we first explain the concepts and data structure involved in the algorithm. *f* is small file to be merged; q is the queue for storing files temporarily; *qList* is the list that include all of *q*; *inuseList* is a subset of *qList* and it include all of *q* that is not empty and dose not reach the merge threshold. *fitList* is a subset of *inuseList* and it include all of *q* whose remaining space is more than size of *f* ; *candidateList* is a subset of *fitList* and it include all of *q* whose correlation value exceeds threshold.

Definition 2 *Correlation(file,q)*. It represent the average correlation value between file and every file in q and *n* is the number of files in *q*.

$$\text{Correlation(file,q)} = \sum_{i=1}^{n} \text{Correlation(f,fi)}/n\ ,\ \text{fi} \in \text{q}. \tag{2}$$

The execution process of the algorithm is as below:

1. Initialize Merge Queue Complete set *qList* and size *n* and by analyzing a large number of log files to get the file relationship mapping set *{Sn}*.
2. For the current small file *f*, check *inuseList* first. If *inuseList* non-empty, then select all eligible *q* (ie, the *q* whose remaining space can accommodate the current *f*), join into *fitList*. If *inuseList* is empty, then put *f* into any empty *q* and put *q* into *inuseList*.
3. If *fitList* non-empty, traverse *fitList*. For each queue q_i in *fitList*, find the average correlation *Correlation (f, qi)* according to *{Sn}*. If *Correlation (f, qi)*>firstThreshold, put *f* into q_i; else then q_i is removed from the *fitList* and enters the *candidateList*.
4. *candidateList* is set to judge and weigh the correlation of files and the distribution of files. If we only consider the degree of correlation, then when the overall related level of files is not so closely, it will result in space waste. In this case, we need a compromise: in the *candidateList* select the highest average correlation queue q_{max} for *f*, determine the value of *Correlation (f,* q_{max}*)*, if the

value exceeds the secondThreshold, then let *f* into q_{max}; otherwise they will not consider correlation, make *f* into the new queue.

5. If *fitList* is empty, it means that there is not a merge queue at this time can accommodate the current small file. Check the minimum remaining space in the *inuseList* queue q_m, if the size of the q_m reaches the merge threshold (HDFS block size of 97%), then the files in q_m are converted into MapFile[12].

Design of algorithm

Pseudocode of core implementation of algorithm is described as below:

```
Input: merge small file set named FileSet, file relationship mapping set named CorrelationMap,
       number of queue named n, threshold of correlation named firstThreshold and
       secondThreshold
Output: After the merger of large files named MergedFile
 1. Initialization: qList, inuseList, fitList, candidateList
 2. Foreach file f in FileSet, then goto step 17
 3. If inuseList.isEmpty(),goto step 4,else goto step 5
 4. Select q form qList, put f into q, put q into inuseList
 5. Foreach q in inuseList. if q.emptySize() > f.size, then put q into fitList
 6. If fitList.isEmpty(), goto step 7,else goto step 9
 7. Foreach q in inuseList, calc max q.size queue $q_m$
 8. If $q_m$.size>MergeLimit*0.97, then goto step 17
 9. Foreach queue $q_i$ in fitList, do calc correlation(f,qi) = $\frac{\sum_{i=1}^{n} R(f,fi)}{n}$, $fi \in qi$
10. If correlation(f,$q_i$)> FirTh, goto step 16,else goto step 11
11. Move $q_i$ from fitList then put $q_i$ into candidateList
12. If fitList.isEmpty(),goto step 13.else goto step 9
13. Calc max correlation(f,$q_i$) from candidateList queue $q_{max}$
14. If correlation(f,$q_{max}$)<SecTh, goto step 4, else goto step 15
15. Put f into $q_{max}$, then move $q_{max}$ from candidateList into inuseList
16. Put f into q, then move q from fitList into inuseList
17. Merge files in q.
```

Experiments

Experimental environment

Experiments in this paper uses a three-node Hadoop cluster. It contains a Namenode and two Datenodes. All three nodes are Intel CPU (3.30 GHz), 16GB memory and 240 GB disk, and act as Datanodes and Namenode.

Each node of the cluster has Ubuntu 14.04 installed on it, an the Hadoop version is 2.7.1 for this experiment, JDK version is 1.7, HDFS data block size is 128MB. The number of copies of the data block is 3.

In order to verify the algorithm for small file storage optimization better, this paper collected 3132 files, the total size is 7.93GB. The file size distribution varies from below 100KB to 64MB. Among them, files under 5MB in size account for 93% of the total files. According to literature [8]we get all the files relationship mapping, divided them into three mapping sets, weights of sets are:

$$W(Sn)=\begin{cases}10,n=1\\5\ \ ,n=2\\1\ \ ,n=3\end{cases} \tag{3}$$

Experiment on time consumption of importing files into HDFS

In order to verify the merging effect of MBDC on small files, this paper compares the time cost of importing the merged files to HDFS. The object of comparison is HDFS normal import, using PS algorithm[11] and using MBDC. The total number of normal imported files is 3132, and the number of imported files processed by the PS algorithm and MBDC are 65 and 69. The experimental results is shown as Fig. 3. As can be seen from the comparative test, MBDC has a 33.8% improvement over the normal import, slightly inferior to the PS algorithm. This is mainly due to the fact that the algorithm in this paper takes the correlation of files into full consideration and the extra cost involved in calculating the correlation between files leads to a slight disadvantage in the comparison of import times.

Experiment on memory consumption of Namenode

The histogram in Fig. 4 visually shows the consumption of NameNode memory by importing directly, using the PS algorithm and using MBDC. It can be seen from the figure that compared with the normal import, the PS algorithm and MBDC that take full account of the file block balance have a great decrease on the memory consumption of the Namenode. However, due to the trade-off of this method of files' correlation, the number of final merged files is slightly larger, resulting in a slightly lower memory overhead in the Namenode than the PS algorithm.

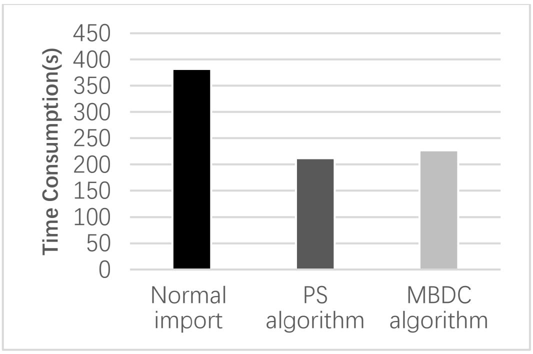

Fig.3 Comparison of time consumption of importing small files

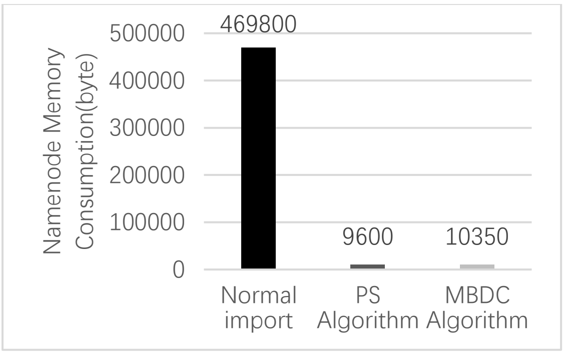

Fig.4 Comparison of memory consumptions at Namenode

Experiment on time consumption of file access

In order to verify the optimization effect of MBDC on file access, this paper uses cache to deal with the access procedures of small files. We caches the block contents of the current access and prioritizes the data in the cache when accessing the file. The cache size is 1GB, about 8 times the block size. Cache replacement strategy is FIFO. According to the user's habit of visiting, this paper tests the access time of 500,1000,1500,2000 small files respectively.

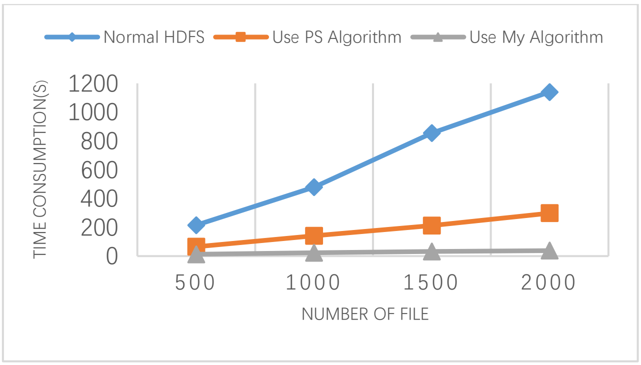

Fig.5 Comparison of time consumption of data access

Test results as Fig.5 shown. As can be seen from the figure, compared with the default HDFS access method and the access method using PS algorithm storage, the algorithm greatly reduces the access time and improves the access efficiency. This is mainly due to MBDC is more fully than the PS algorithm to the relevant files in the same block, making the cache hit rate extremely high, effectively reducing the time consumption of I/O between client and Namenode.

Conclusions

This paper, aiming at the defects of Hadoop distributed file system for small file storage, taking into account the correlation between the files and the size of the file after the merger of these two factors, puts forward an optimized merge algorithm, MBDC. It is based on the substantial optimization for HDFS storage efficiency of small files, to ensure efficient file access. The experimental results show that MBDC can effectively improve the import efficiency of small files and solve the bottleneck of Namenode memory in the aspect of storage optimization of small files, and greatly improve the efficiency of reading related files. The disadvantages of this approach are a) a certain degree of time consumption associated with increasing file size as computing correlation of files; b) there is no obvious advantage in random access to small files. Storage optimization of small files on HDFS still needs further study.

References

[1] T. White and D. Cutting, Hadoop : the definitive guide, O'reilly Media Inc Gravenstein Highway North, vol. 215, 2009, pp. 1–4.

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, The Hadoop Distributed File System, MASS Storage Systems and Technologies, 2010, pp. 1–10.

[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, IEEE International Conference on Services Computing, SCC 2010, Miami, Florida, Usa, July, 2010, pp. 65–72.

[4] X. Liu, J. Han, Y. Zhong, C. Han, and X. He, Implementing WebGIS on Hadoop: A case study of improving small file I/O performance on HDFS, IEEE International Conference on CLUSTER Computing and Workshops, 2009, pp. 1–8.

[5] ZOUZhenyu, ZHENGQuan, WANGSong, and YANGJian, Optimization Scheme of Small File in Cloud Storage System Based on HDFS, Computer Engineering, vol. 42, 2016, pp. 34－40.

[6] LIMeng, CAOSheng, and QINZhi-guang, Storage Optimization Method of Small Files Based on Hadoop, Journal of University of Electronic Science and Technology of China, 2016, pp. 141－145.

[7] YOU Xiao-rong,CAO Sheng, Storage Research of Small Files in Massive Education Resource, Computer Science, vol. 42, 2015, pp. 76–80.

[8] Gu Yuwan， Wang Wenwen， Sun Yuqiang, Optimization of massive small files storage and accessing on HDFS, Application Research of Computers, vol. 34, 2017, pp. 2319–2323.

[9] H. He, Z. Du, A. Chen, and A. Chen, Optimization strategy of Hadoop small file storage for big data in healthcare, Journal of Supercomputing, vol. 72, 2016, pp. 3696–3707.

[10] D. Sethia, S. Sheoran, and H. Saran, Optimized MapFile based Storage of Small files in Hadoop, Ieee/acm International Symposium on Cluster, Cloud and Grid Computing, 2017, pp. 906–912.

[11] WANG Quan-min, ZHANG Cheng, ZHAO Xiao-tong, LEI Jia-wei, A Small Hadoop File Storage Optimization Scheme, Computer Technology And Development, vol. 26, 2016, pp. 41－44.

[12] Information onhttps://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/io/MapFile.html