A Strategy for Small Files Processing in HDFS

Zhenshan Bao(✉), Shikun Xu, Wenbo Zhang, Juncheng Chen, and Jianli Liu

College of Computer Science, Beijing University of Technology, Beijing 100124, China
baozhenshan@bjut.edu.cn

Abstract. Hadoop distributed file system (HDFS) as a popular cloud storage platform, benefiting from its scalable, reliable and low-cost storage capability. However it is mainly designed for batch processing of large files, it's mean that small files cannot be efficiently handled by HDFS. In this paper, we propose a mechanism to store small files in HDFS. In our approach, file size need to be judged before uploading to HDFS. If the file size is less than the size of the block, all correlated small files will be merged into one single file and we will build index for each small file. Furthermore, prefetching and caching mechanism are used to improve the reading efficiency of small files. Meanwhile, for the new small files, we can execute appending operation on the basis of merged file. Contrasting to original HDFS, experimental results show that the storage efficiency of small files is improved.

Keywords: Hadoop · HDFS · Small file · File merging · Prefetching and caching · Appending operation

1 Introduction

Data gradually become the most valuable information in this age, and every year to grow exponentially. Many applications in the area of education, e-business, Biology consist of mass data, and every day each industry produces large amounts of data. According to the study of relevant authorities, the amount of data in 2013 reached 4.4 zettabytes, and is forecasting a tenfold growth by 2020 to 44 zettabytes [1]. In addition, type and size of the data are also varied. Today, there are many large E- commerce companies such as TaoBao, JD and Amazon. These sites store vast amounts of small files. We need to store and manage these huge amounts of small files more effectively, and a better using experience can be provided for users. Today single machine processes massive small files seem to be difficult, the expansion of the longitudinal computer performance, not only cost too much, but also meet bottleneck sooner or later. Thus distributed processing mode has become the key to solve those problems.

Hadoop as an open source distributed framework, because of reliability, high performance, high scalability, thus it attracts more and more individuals and organizations to use [2]. HDFS is one of the core components of Hadoop. It is a storage component which can be deployed in cheap hardware. Meanwhile, as a distributed file system, it's responsible for the distributed data storage and data management. When we store the large files in HDFS, they are divided into several blocks, and the blocks are stored

W. Che et al. (Eds.): ICYCSEE 2016, Part I, CCIS 623, pp. 109–119, 2016.
DOI: 10.1007/978-981-10-2053-7_11

in the DataNode. Furthermore, blocks fit well with replication for providing fault tolerance and availability. To insure against corrupted blocks and disk and machine failure, each block is replicated to small number of physically separate machines (typically three) [3]. Each file block will generates metadata in the NameNode memory when the machines start up. Metadata records a series of information. When the client wants to read a file, first of all, it will read the metadata information in the NameNode, through metadata information to find the corresponding DataNode, and then get target files. As shown in Fig. 1 is the basic storage framework of HDFS.

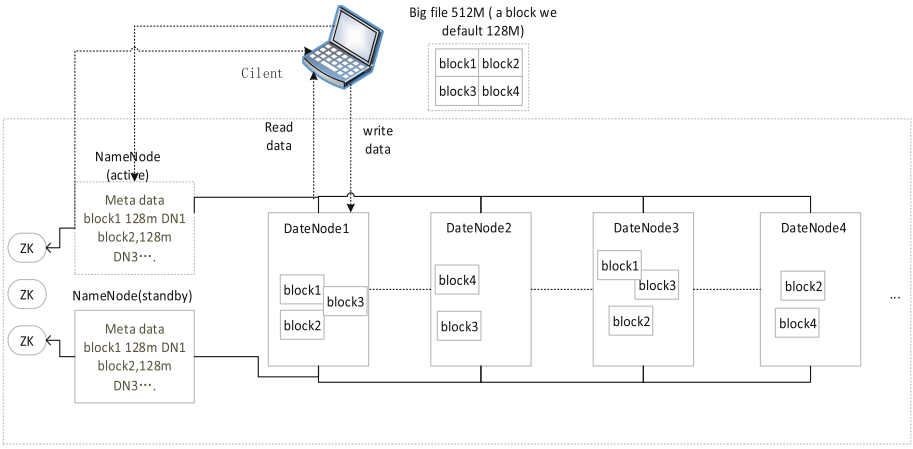

Fig. 1. Basic storage framework of HDFS

In Fig. 1, it is not hard to find that big file is divided into four blocks. Each block size is 128 Mbytes (the new version of Hadoop default size). Meanwhile in NameNode will generate corresponding metadata about blocks. Under the circumstances, HDFS for small file management will be a problem. Because namenode is responsible for storing file metadata in memory, the limit to the number of files in a filesystem is governed by a mount of memory on the NameNode. In file management strategy of HDFS, if the file size is less than a block size, this file will occupy a block alone, and in the NameNode memory it will generate metadata. If there are 10 million small files, there will be 10 million blocks. Those small files will use NameNode about 3 Gigabyte memories [4]. If we have massive small files, the memory pressure of NameNode will be huge, maybe beyond the capacity of current hardware. If you need to read a lot of small files from clusters, frequently switching between nodes also bring a huge network load.

In this paper, we mainly to solve the massive small file problem in HDFS. To reduce the NameNode memory usage, we use small files merging. On read and write we also do the corresponding processing, this way to store small files in the HDFS has certain performance improvements.

The rest of our paper is organized as follows: Sect. 2 we introduce the related works; Sect. 3 explains the proposed approach for handing small files in HDFS.

Experimental results and analysis are presented in Sect. 4; Sect. 5 presents conclusions and provides future directions.

2 Related Works

Solutions for dealing with small files can be divided into two categories: Hadoop own solutions and current academic solutions. For Hadoop Archive (HAR) is mainly used to archive small files in HDFS, the purpose is that reducing memory usage of NameNode [5]. But for HAR low speed in terms of small files retrieving. Once the package is completed, if there are some new small files, HAR need all the small files repackaged, and we can't perform appending operations on the basis of merged file. Figure 2 is a HAR file structure. SequenceFile seems to play the role of a small file container, which uses <key, value> structure. However, this structure seems not optimistic on retrieval efficiency, because it is similar to link storage structure. If you want to search a certain key and get value, you need to traverse the entire SequenceFile file, which greatly reduces the retrieval efficiency. And converted into Sequencefile it takes a very long time [6]. CombineFileInputFormat can combine multiple files into a split, but we need to design Class to achieve, and it is not easy to implement [7]. Currently there are many different ways in academic dealing with small files. In [8] presented an approach for small files in the application in WEBGIS, although processing performance for small files has improved in some extent. However, this approach applies only to geographic information data. In [9] is mainly aimed at PPT files, this article described the operation of the prefetching mechanism, index files and the correlated files, but this is a specific application mode. In [10] mainly discuss MP3 files, a new storage method by use of the rich description of MP3 files, but it is only based on MP3 files storage model. In [11] authors have proposed a new architecture of HAR and we referred to as New Hadoop Archive(NHAR). This method can improve efficiency of accessing small files, but we can't do appending operation when we have some new small files. Meanwhile, files are not considered the correlations when archiving. In [12] proposed a method for merging files, reducing memory consumption, but did not highlight the file read and write efficiency results. In [13] describes the general framework for handling small files, and introduces the several components of the framework, but for files merging algorithm and file appending operation didn't make a specific description.

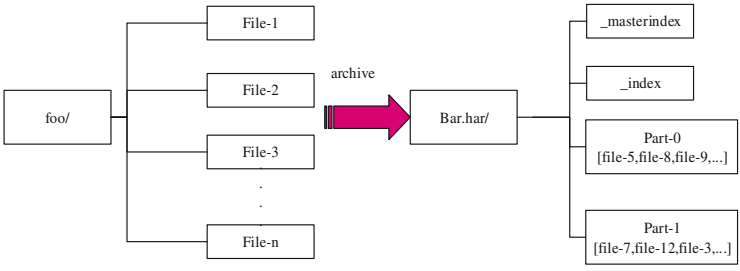

Fig. 2. Archive small files in HAR

3 Proposed Approach

In this part we will introduce our approach from 5 respects as follows:

3.1 File Merging

File Merging is the fundamental to solve the small file problems in HDFS. It can reduce the numbers of metadata files in NameNode memory. We merge massive correlated files into a single file, so the NameNode can just maintains the metadata of the single file. The main purpose of this approach is to reduce the memory load of NameNode. Neither the traditional processing way of Hadoop nor other storage strategies, the basic idea is file merging. But we must ensure that no small file is spitted across two blocks. This means that we must ensure the integrity of the file.

Figure 3 shows the basic idea of merging algorithm.

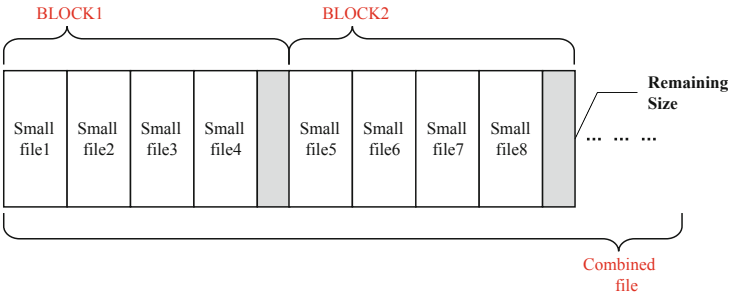

Fig. 3. Basic idea of merging algorithm

Remaining size in the Fig. 3 we can explain that we can't guarantee the merged file is just equal to the size of a block, when remaining size of current block can't suitable for the size of next small file, we should put this small file into next block.

Algorithm of small file merging

Step1: Get all correlated files in a directory

Step2: For file directory, we do traversal operation, Analyzing whether the file size is less than the block size. If the file size is larger than block size we upload this file to HDFS, otherwise we perform file merging operations. (Hadoop new version default block size is 128 M)

Step3: Initialization of variables. Create two empty file, such as "combinefile" and "mapfile", and their initial size is 0. We also set variable "current_offset" and "BlockId" is 0. And also we set variable "limitsize" equal to block size

Step4: Next, we should judge whether the sum of current file size and "current_offset" less than "limitsize", If the sum is less than or equal to "limitsize", we can set current file Block logical number equal to "BlockId", if sum is just equal to "limitsize", then we execute "BlockId++" operation, at this time

"current_offset" equal to "limitsize + 1", and "limitsize" equal to "BlockId" multiplied by block size. Otherwise, "current_offset" equal to the sum of offset and file size of this file

Step5: if the sum of current file size and "current_offset" larger than "limitsize", then we should execute "BlockId ++" operation, block logical number of current file equal to "BlockId", and "current_offset" equal to "limitsize + 1", and "current_offset" equal to the sum of offset and file size of this file, and "limitsize" equal to "BlockId" multiplied by block size. Update the current file information to the mapping file (mapping file will introduce in next part)

Step6: Continue to repeat the *Step4* and the *Step5* until no small files. Then upload merged file to HDFS, and store mapping file in NameNode

3.2 Mapping File

Build the mapping file, the purpose is to find the target file information, through relevant information, we can quickly locate to block address about small files, and then extracted from the content of the block. Mapping file structure is shown in Fig. 4:

Small file name	Block logical number	Offset	Length

Fig. 4. Mapping file structure

Small File Name: This is the small file name under the directory, that's can serve as a unique identifier for a file, regardless of the Linux system, or under the Windows system, it is not allowed to small file name repetition.

Block Logical Number: A merge file contains to a lot of blocks, each small file should be in a certain block. So if we want to get a small file, we must get the block logical number of small file.

Offset: The starting address of each small file in merged file.

Length: That's mean small file size, by the offset and length we can get the small file content from merged file.

The following Fig. 5, it is a mapping file contents demo.

```
6472 15824.txt 03 71125504 157254
6473 15825.txt 03 71282758 157254
6474 15826.txt 03 71440012 20531
6475 15827.txt 03 71460543 20531
6476 15828.txt 03 71481074 157254
6477 15829.txt 03 71638328 157254
6478 1583.txt 03 71795582 85440
```

Fig. 5. Mapping file contents demo

3.3 Prefetching and Caching

In the HDFS, when a file is read, firstly the client must request the NameNode to get metadata information of merged file, and then we will get the block information about merged file. The NameNode also provides a mapping record of small file. So the question is here. When we visit small files frequently, NameNode will get a heavy load and access speed will be slow. Thus we must reduce load on NameNode and improve access speed. So we used prefetching and caching techniques, the more details we can refer to paper [14, 15].

3.3.1 The Metadata Information Caching

When we access to a small file, Firstly we need to get the metadata information about merged file from NameNode. If the client cache has file metadata information, then we can get those metadata in client cache directly.

3.3.2 Prefetching Mapping File Information

According to the merged file metadata information, the client decided to which block we should read, if the small file mapping records are perfected from the mapping file in advance, then we can read small file directly.

Metadata information caching and prefetching mapping file information these two mechanisms can accelerate I/O access speed, thus it can be improve the efficiency of file reading.

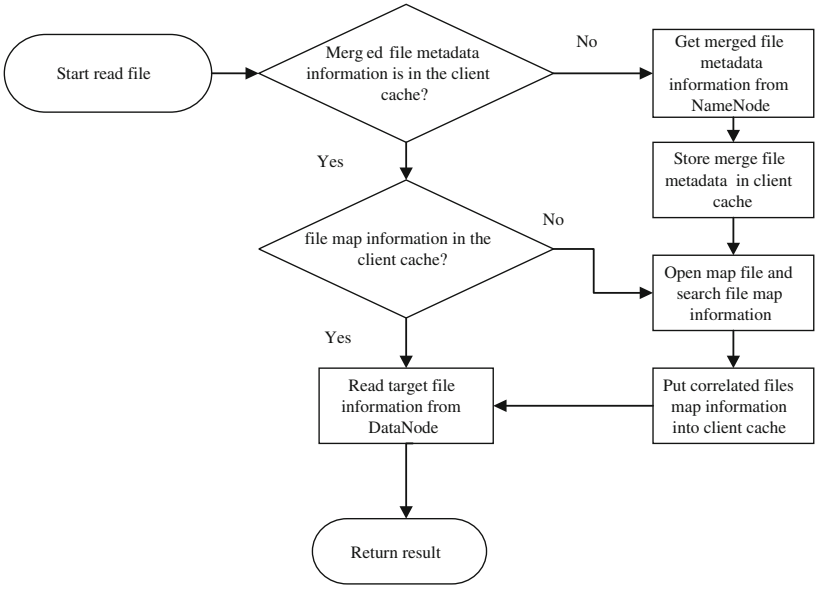

Fig. 6. File reading process

3.4 File Access Operation

When reading the files, at first we should determine whether the client cache has merged file metadata information, If the client cache has metadata information, then continue to determine whether client cache has file mapping information, if there is, we can read small file content from DataNode directly. If we can't find metadata information in client cache, we need to read it from NameNode, and store merged file metadata in client cache, then open mapping file and search file mapping information and put correlated files mapping information into client cache. At last, read target file information from DataNode. File reading process is shown in Fig. 6.

3.5 File Append Operation

In order to make full use of every piece of space, so we execute file append operations. When we merge files, we cannot guarantee the size of each block is equal to the sum of the file size. File append operation can also improve the efficiency of file merging. File appending process as shown in Fig. 7.

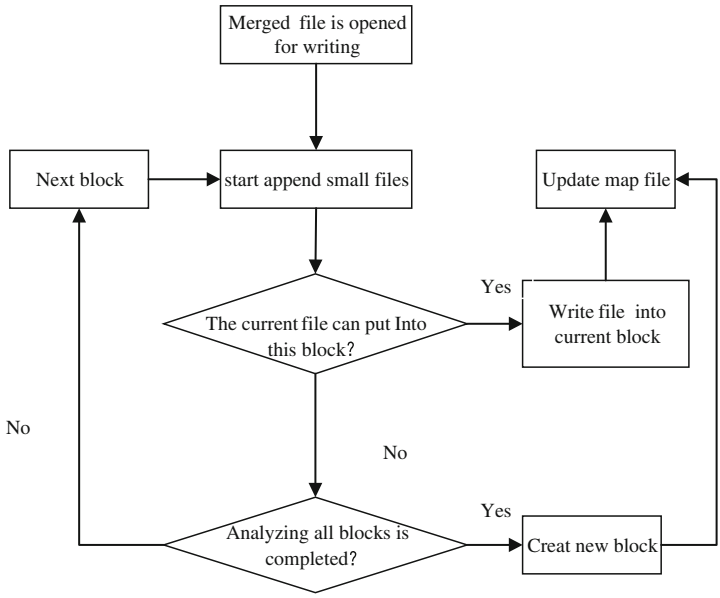

Fig. 7. File appending process

4 Evaluation and Results

4.1 Experimental Environment

Test platform includes a total of three machines. The number of copies of file blocks is set to 2 and the default block size is 64 M, The following we can see relevant configuration information.

Experimental Environment	Related parameters
JDK	1.7.0_79
OS	Ubuntu 12.04 64-bit
Hadoop	2.6.0
Memory	2G
Hard disk	100G
CPU	Intel core 2 /2.4GHz
NameNode	1
DateNode	2

4.2 Workload Overview

The workload for main memory usage measurement contains a total of about 100,000 files. The size of the small files range from 3 KB to 16000 KB, The size of all files is approximately 7.5 GB. The distribution of the file sizes is shown in Fig. 8.

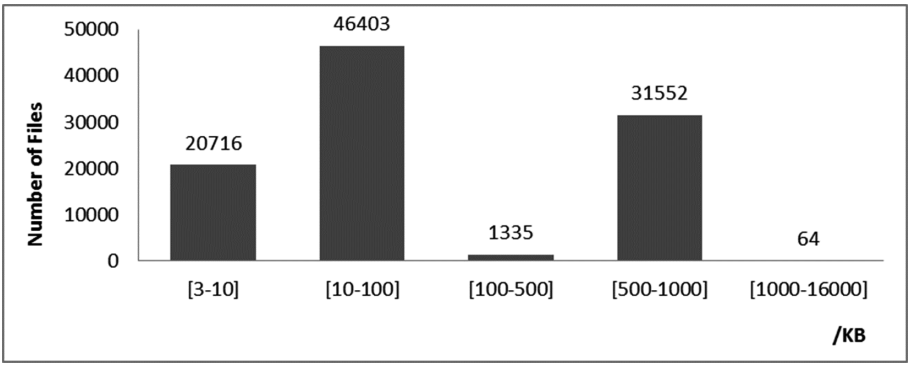

Fig. 8. Distribution of file sizes in workload

4.3 Memory Usage Measurement

The different number of files stored in HDFS. Random read files after each startup NameNode, analyzing the NameNode Memory. The memory usage of NameNode for original HDFS and the proposed approach is shown in Fig. 9.

We can find that in Fig. 9 the memory used by proposed Approach is less than Original HDFS. However, when the number of small files is 10000, in this paper, the

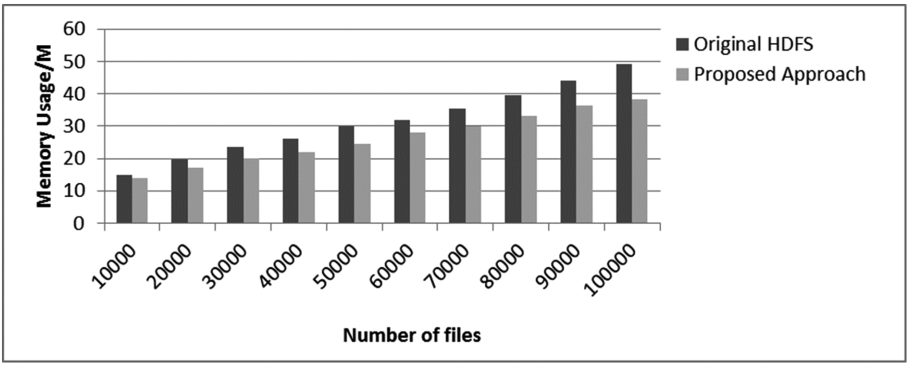

Fig. 9. Memory usages of HDFS and proposed approach

optimization of design effect is not obvious, with the increase of the number of files advantage gradually. The main reason proposed approach can save memory, because we use File merging strategy. The block metadata is stored by NameNode for single combined file and not for every single small file. So this can reduce memory usage.

4.4 Time Taken for File Access

File access time, group by the number of the small files in 10000, 20000, 30000, 40000, 50000, 60000, 70000, 80000, 90000 and 100000. The access time of every group is recorded. Each group is test 3 times. Ignoring the impact of network latency and the average of the residual values is obtains as the access time of the each group. The time taken for read operation in HDFS and proposed approach is depicted in the graph shown in Fig. 10.

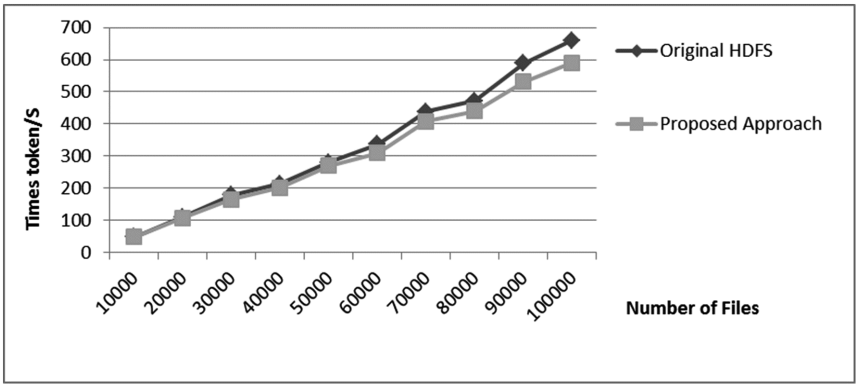

Fig. 10. Time taken for read operation

In this scenario, for small file reading efficiency in performance is not very obvious, that's maybe caused by our caching strategies or some other factors such as Time-consuming when searching for small files. But after the 50000 files, this scheme is superior to the original HDFS in reading time. Because we use combined file to reduce the number of metadata in the NameNode, it also can reduce the NameNode access frequency, in some extend that's reduce the network load. Meanwhile Using the caching and prefetching strategy can make efficiency improve.

5 Conclusions

In this paper we focused on small file problems in HDFS. As the growth of the number of small files, it will gradually bring the load to NameNode. In this paper, starting from this issue, we merged a large number of small file into single file to reduce the memory usage of NameNode. When reading the file we use caching and prefetching mechanism. We confirmed our method by experiments, this scheme can reduce the NameNode memory consumption, and improve the reading efficiency, but the effect is not too prominent. In future work, we will do more research in file reading, such as caching and prefetching strategy.

Acknowledgement. This research supported by Beijing Key Laboratory on Integration and Analysis of Large Scale Stream Data (ID: PXM2015_014204_500221) and the significant special project for Core electronic devices, high-end general chips and basic software products. (2012ZX01039-004).

References

1. http://www.emc.com/leadership/digital-universe/2014iview/index.html
2. Apache Hadoop. http://hadoop.apache.org/
3. White, T.: Hadoop: The Definitive Guide, 4E. O'Reilly Media (2015)
4. Liu, X., Peng, C., Yu, Z.: Research on the small files problem of Hadoop. In: International Conference on Education, Management, Commerce and Society (EMCS 2015). Atlantis Press (2015)
5. HadoopArchivesGuide. http://hadoop.apache.org/docs/stable/hadoop-archives/HadoopArchives.html
6. SequenceFile. http://wiki.apache.org/hadoop/SequenceFile
7. CombineFileInputFormat. http://hadoop.apache.org/docs/stable/api/org/apache/hadoop/mapred/lib/CombineFileInputFormat.html
8. Liu, X., Han, J., Zhong, Y., Han, C., He, X.: Implementing WebGIS on Hadoop: a case study of improving small file I/O performance on HDFS. In: IEEE International Conference on Cluster Computing and Workshops, pp. 1–8 (2009)
9. Dong, B., Qiu, J., Zheng, Q., Zhong, X., Li, J., Li, Y.: A novel approach to improving the efficiency of storing and accessing small files on Hadoop: a case study by PowerPoint files. In: IEEE International Conference on Services Computing (SCC), pp. 65–72 (2010)

10. Zhao, X., Yang, Y., Sun, L.-L., et al.: Based on the Hadoop mass MP3 file storage structure. J. Comput. Appl. **32**(6), 1724–1726 (2012)
11. Vorapongkitipun, C., Nupairoj, N.: Improving performance of small-file accessing in Hadoop. In: 11th International Joint Conference on Computer Science and Software Engineering (JCSSE), pp. 200–205 (2014)
12. Patel, A., Mehta, M.A.: A novel approach for efficient handling of small files in HDFS. In: 2015 IEEE International Advance Computing Conference (IACC), pp. 1258–1262 (2015)
13. Changtong, L.: An improved HDFS for small file. In: 2016 18th International Conference on Advanced Communication Technology (ICACT) (2016). doi:10.1109/ICACT.2016.7423438
14. Peng, X., Feng, D., Jiang, H., Wang, F.: FARMER: a novel approach to file access correlation mining and evaluation reference model for optimizing peta-scale filesystem performance. In: Proceedings of the 17th International Symposium on High Performance Distributed Computing, pp. 185–196 (2008)
15. Dong, B., Zhong, X., Zheng, Q., Jian, L., Liu, J., Qiu, J., Li, Y.: Correlation based file prefetching approach for Hadoop. In: IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), pp. 41–48 (2010). [14]