Improving Access Efficiency of Small Files in HDFS

Monica B. Bisane, Student, Department of CSE, G.C.O.E, Amravati ,India, monica9.bisane@gmail.com

Asst.Prof. Pushpanjali M. Chouragade, Department of CSE,G.C.O.E, Amravati, India, pushpanjalic3@gmail.com

.

Abstract— *The Hadoop Distributed File System (HDFS) is designed to store very large data sets reliably, and to streamthose data sets at high bandwidth to user applications. It is designed to handle large files. Hence, it suffers performance penalty while handling a huge number of small files. Further, it does not consider the correlation between the files to provide prefetching mechanism that is useful to improve access efficiency. In this paper, we propose a novel approach to handle small files in HDFS. The proposed approach combines the correlated files into one single file to reduce the metadata storage on Namenode. We integrate the prefetching and caching mechanisms in the proposed approach to improve access efficiency of small files. Moreover, we analyze the performance of the proposed approach considering file sizes in range 32KB-4096KB. The results show that the proposed approach reduces the metadata storage compared to HDFS.*

IndexTerms—Hadoop, HDFS, small files, file correlation, prefetching

.

1.INTRODUCTION

Hadoop is a software framework developed for file storing and processing of a huge dataset with the cluster of commodity hardware [1]. Hadoop Distributed File System (HDFS) is the key storage component of the Hadoop [2][3]. It is a specially designed file system for storing large datasets with streaming access pattern and is generally used in many internet applications. HDFS that is inspired by GFS has master-slave architecture [4]. Major architectural components of HDFS are Datanodes and Namenode. In HDFS, files are divided into blocks that are placed on a set of Datanodes to ensure reliability and data availability. Namenode maintains metadata of all the files and blocks in its main memory to maximize access efficiency. It directs file access requests from client to appropriate set of Datanodes. Subsequently, the client can directly communicate with the Datanode to perform file

operations. Many applications in the area of biology, climatology, energy, e-learning, e-business and e-library consist of a huge number of small files. Though, the size of several small files is far lesser than the size of block size, HDFS stores each small file as one separate block. Therefore, with massive small files, a large number of blocks are created. Irrespective of the size of file, the metadata of each file consumes 250 bytes and its block with default three replicas consumes 368 bytes of memory of Namenode [6]. Therefore, the number of files that can be stored in HDFS is limited because to store metadata of a huge number of small files, a significant amount of memory is required. The problem of storing and accessing large number of small files is named as small file problem [5]. Though numerous approaches have been presented to address the small file problem, they have one or more of the following limitations [8-16].

• Higher time for accessing small files [8-11][14]
• Applicable towards specific applications [8][11]
• No provision for appending a new small file into existing combined file [10-12]
• Correlations between files are not considered [13][15-16]
• Prefetching is not supported [13][15-16]

Therefore, as per our observations, it is required to design an approach that not only reduces metadata storage on Namenode but also improves access efficiency of small files. To overcome the above mentioned limitations, we propose an approach for efficient handling of small files in HDFS. In the proposed approach, we combine the correlated files into one large file to decrease the memory usage of Namenode. We create a mapping file to store location information of the small files present in the combined file. The proposed approach reduces the access time of the small files via supporting prefetching and caching phenomenon as well as storing mapping file on Namenode. Further, we present an append operation to add small files into existing combined file. Thus, the proposed approach is competent to handle the small files efficiently in HDFS.

The rest of the paper is organized as follows: In section II, we discuss the existing small file handling approaches. The proposed approach for handling small files in HDFS is described in section III. Section IV provides the performance analysis of the proposed approach. Conclusion is drawn in section V and we specify some future work..

2. RELATED WORK

Numerous approaches for handling small files have been presented in the literature [8-16]. In [8], tuned HDFS approach is proposed for optimizing I/O performance of small files present in the WebGIS. However, it is applicable to geographic data only. In [9], an approach is presented that facilitates in-job archival of directories and files. However, it takes high time to access the small files. In [10], authors have proposed an approach that optimizes HDFS I/O by using local cache to save some metadata of small files. However, their experimental results reveal that efficiency of accessing small files is not significant. In [11], small file tuning approach is presented that merges small files that are correlated in the PPT courseware. In addition, prefetching of index file and correlated file is introduced. However, file access efficiency can further be improved. In [12], Extended Hadoop Distributed File System (EHDFS) improves the access efficiency of small files and reduces the metadata footprint in Namenode's main memory. It does not support append operation to add small files into an existing combined file. Hadoop Archive (HAR) is mainly used to archive files in HDFS for reducing memory usage of Namenode [13]. In [14], authors have modified architecture of HAR and proposed a New Hadoop Archive (NHAR) to minimize the metadata storage requirements and to improve efficiency of accessing small files. However, correlations between files are not considered during the process of archiving. SequenceFile is a specialized key-value data structure that acts as a container for small files [15]. It takes a long time to convert small files into a SequenceFile. Moreover, to find a particular key, we have to search whole sequence file. Thus, the access efficiency of files degrades. In MapFile, sorted keyvalue pairs can be appended and the key and the offset are stored in the index file that results in a fast look up [16]. HAR, SequenceFile, and MapFile have a common disadvantage that they do not consider file correlations. As per our observations, no attempt has been made that not only reduces the metadata storage of Namenode but also improves the efficiency of accessing small files. Hence, we propose a novel approach that has the following features.

- Combining correlated small files for reducing the metadata storage of Namenode
- Prefetching and caching for improving efficiency of accessing small files.

3. SYSTEM OVERVIEW

In Table I, we describe the notations that are used throughout the paper. The proposed approach has three techniques that play crucial role in reducing storage and improving access efficiency of small files.

- File combining
- File mapping
- Prefetching and caching

3.1 File Combining

Conventionally, Namenode stores metadata of files and blocks. File metadata consists of file name, file length, replication, modification time, access time, owner, group, and file permissions. Block metadata consists of the information about the set of blocks that possesses file data and location of these blocks. Thus, in classical approaches, the Namenode consumes a large amount of memory to store metadata of massive small files. In order to reduce the metadata storage of Namenode, several small files as specified by the client are merged into one single file that is known as combined file. As a result, Namenode maintains the metadata of merely

TABLE I. LIST OF NOTATIONS

Notation	Description
F	Small file
N	Total number of correlated small files
fs={ f1, f2,...., fn }	Set of correlated small files from i=1 to n
Fm	Mapping file
Fc	Combined
NN	Namenode
DN	Datanode
CC	Client Cache
R	Mapping record (file name,block number, file offset, file length)
MD	Metadata of a file
RS	Remaining Size
BS	Block Size

combined file rather than the metadata of several small files. Moreover, file combining technique ensures that no file is splitted across two blocks, as reading of a file from two blocks reduces access efficiency.

3.2 File Mapping

In HDFS, when a request for a particular file arrives, HDFS client contacts Namenode for metadata of the file. In the proposed approach, Namenode deals with the metadata of the combined files with the help of mapping file. Mapping file contains the small file name, file offset, file length, and the logical block number of the combined file. Each entry in the mapping file is called mapping record. To obtain metadata, the requested small file first needs to be mapped to a respective combined file. If the combined file is composed of multiple blocks, mapping to the block where the requested small file resides is required to access a particular file.

3.3 Prefetching and Caching

In HDFS, when a file is to be read, a request is made to the Namenode to get the metadata of its combined file. Namenode responses with the list of blocks holding the file and the Datanodes that hold these blocks. Namenode also provides a mapping record of the small file from the mapping file. When a large number of small files are accessed, a heavy load is created on Name node. The proposed approach reduces load on Namenode by using prefetching and caching techniques that are generally used as storage optimization techniques [17]. Prefetching conceals visible I/O cost and improves access time by utilizing correlations between files and fetching data into cache prior to they are requested [18].

1) Metadata caching: When a request for a particular file arrives, metadata of the combined file from the Namenode is needed. If the client caches metadata of the combined file, the metadata can be directly acquired from the cache for subsequent file requests. Thus, it reduces communications with Namenode and improves access efficiency.

2) Prefetching of mapping record: According to the metadata of the combined file, a client determines which blocks should be read to obtain the requested file. If the mapping records are prefetched from the mapping file, accessing files that are part of the same combined file needs to directly complete I/O operation. Thus, prefetching and caching prevents Namenode from becoming bottleneck in the system and improves efficiency of accessing small files.

3.4 File access operations

At this juncture, we discuss three file operations: write, read, and append.

1) *Write operation*: Once a combined file is created, it is written on HDFS by HDFS client. The write operation is described in Fig. 1. We first create empty files named fc and fm. Then for each small file in fs, check is made whether the remaining space in the block is able to accommodate the small file. If remaining space is less than or equal to the length of the small file, then small file is stored on the current block, otherwise small file is stored on the new block. Subsequently, mapping record of respective small file is inserted in the mapping file using InsertMap function that is described in Fig. 2. After all files are combined, mapping file is stored on Namenode and combined file is stored on HDFS.

2) *Read operation*: The client initiates the read operation by providing small file name and the combined file name. Algorithm for read operation is described in Fig. 3. The client requests the Namenode for the metadata of the small file. The Namenode responses by finding small file name in the mapping file for the specified combined file. Mapping record is searched in mapping file using SearchMap function. Algorithm for SearchMap function is shown in Fig. 4. The logical block number, file offset and file length are sent back to the client along with block locations. Along with the response to the current request, the Namenode also prefetch mapping records of the files that reside around the requested file. These prefetched mapping records are stored in a client cache. The cache is checked whenever a small file is requested by the client. If the metadata is present in the cache, read request is not forwarded to the Namenode. The client can then read the file by directly contacting the Datanode. Thus the response time of read operation is significantly improved. This phenomenon also decreases the Namenode interaction per file, thereby improves the efficiency of accessing consecutive small files.

3) *Append operation*: The proposed approach allows to add new small file into existing combined file. If the size of the small file is less or equal to the size of remaining space in current block, the file can be appended in the same block. However, if the file size exceeds the remaining space in current block, file is appended on a new block. Simultaneously, mapping record is created in mapping file for the new file that is appended into the combined file. Algorithm for append operation is discussed in Fig. 5.

```
Input : fs={ f1, f2,...., fn }
Output : fc, fm
// fi.name represents name of the ith file
// fi.length represents length of the ith file
// fi.offset represents current_offset of the ith file
// fi.BN represents blocknumber of the ith file
1.  Create empty files fc and fm
2.  BN = 1; current_offset = 0; limit = BS
3.  FOR i=1 to n
4.        RS = current_offset
5       . IF ((RS + fi.length) <= limit)
6.                 fi.BN = BN
7.                 IF ((RS + fi.length) = limit)
8.                         BN++
9.                         current_offset = limit + 1
10.                        limit = BN * BS
11.                    ELSE
12.                         current_offset = fi.offset +
                            fi.length
13.           ELSE
14.                    BN++
15.                    fi.BN = BN
16.                    fi.offset = limit + 1
17.                    current_offset = fi .offset + fi.length
18.                    limit = BN * BS
19.            Insert fi in fc
20.            InsertMap (fi.name, fi.BN, fi.length, fi.offset)
21. Store fm on NN
22. Write fc on HDFS
```

Fig.1.Algorithm for write operation.

```
Input: f.name: Name of f
       f.BN: Block number of f
       f.offset: Offset of f
       f.length: Length of f
Output: fm : Modified mapping file

InsertMap (f.name, f.BN, f.offset, f.length)
1. Open fm of corresponding fc
2. Create record r.
3.  r.name = f.name
4.  r.BN = f.BN
5.  r.offset = f.offset
6.  r.length = f.length
7.  Store r at the end of the fm
```

Fig.2.Algorithm to insert mapping record into mapping file

```
Input: f.name : Name of small file
fc.name : Name of combined file
Output: f
1. IF (MD of fc is present in CC)
```

```
2. IF (r of f is present in CC)
3. Obtain r from CC and contact DN to read f
4. ELSE
5. Go to step 9
6. ELSE
7. Get MD of fc
8. Store MD of fc in CC
9. Open fm of corresponding fc
10. r = SearchMap(fc.name , f.name)
11. Get mapping record of all correlated files of f
from fm and prefetch in CC
12. Contact DN to read f
```

Fig. 3. Algorithm for read operation.

```
Input: fc.name: Name of combined file
f.name: Name of f
Output: r
SearchMap (fc.name, f.name)
1. Open fm of corresponding fc
2. FOR each record r in fm do
3. IF (r.name = f.name)
4. RETURN (r.name, r.BN, r.offset,
r.length)
5. ELSE
6. Display there is no such file exists
```

Fig. 4. Algorithm to search mapping record from mapping file.

```
Input: f
Output: fc : Modified combined file
1. Open fc in write mode
2. IF (f can be accomodated in the current block)
3. Write f in current block of fc
4. ELSE
5. Request for new block
6. Write f in new block of fc
7. InsertMap (f.name, f.BN, f.offset, f.length)
```

Fig. 5. Algorithm for append operation.

4. PERFORMANCE ANALYSIS

The The performance of the proposed approach with respect to memory usage of Namenode is compared with original HDFS. The experiment is performed in a cluster having single node. Single node acts as Namenode, Datanode as well as client. The node has following configuration:

1) Intel(R) Core(M) i3-2348M @2.30 GHz
2) 4GB RAM
3) 500GB SATA HDD

The operating system of node is fedora 20 with kernel version 3.17.7-200. We use Hadoop version 2.2.0 and java version 1.7.0_71. The number of replicas for data blocks is set to 1 and default block size is 64MB.

In Fig. 6, we present the distribution of the file sizes in workload. Total 10,000 files have been created. The size of these files range from 32KB to 4096KB. Files in the range between 32KB to 256KB account for 98% of the total file [11]. The performance has been measured based on the amount of main memory used by Namenode to store metadata. The memory used by Namenode is measured using JConsole provided by Java 2 Platform Standard Edition (J2SE). For original HDFS and the proposed approach, Namenode memory usage is evaluated after placing sets of 1000 files into HDFS. A total of 10,000 files have been placed and 10 readings have been taken for every 1000 files placed into HDFS. The memory usage of Namenode for original HDFS and the proposed approach is shown in Fig. 7.

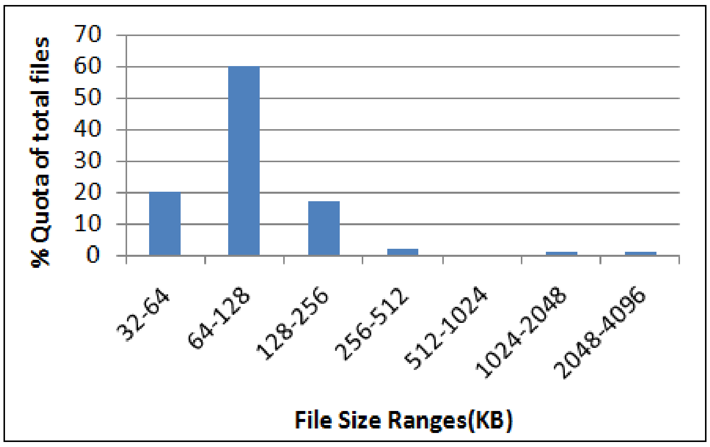

Fig. 6. Distribution of file sizes in workload.

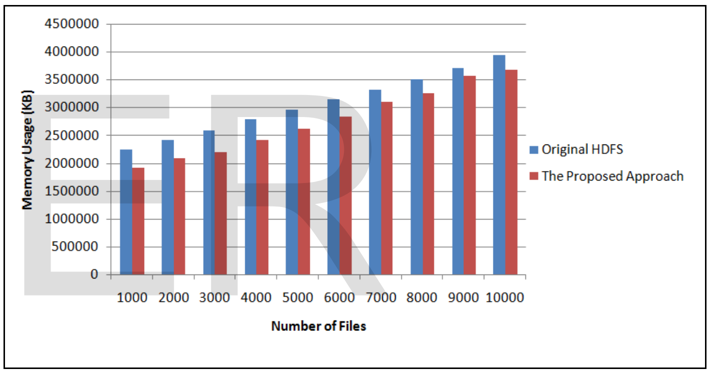

Fig. 7. Memory usage of HDFS and the proposed approach.

The results show that the memory used by the proposed approach is 10% less than the original HDFS. The memory usage is more in original HDFS because Namenode stores file as well as block metadata of each file. Thus, as the number of files increases, the memory usage also increases. The memory used by the proposed approach is less as Namenode stores merely file metadata of each small file. The block metadata is stored by the Namenode for single combined file and not for every single small file. As a consequence, the memory usage of Namenode is reduced.

5. CONCLUSION

In this paper, we have focused on the small file problem of HDFS. Storing a huge number of small files results in high memory usage of Namenode and increased access cost. We have proposed a novel approach for efficient handling of small files. The proposed approach combines a large number of small files into single combined file to reduce the memory usage of Namenode. We have evaluated the performance of the proposed approach in Hadoop considering the file sizes in range 32KB-4096KB. The experimental results show that the storage required for metadata in main memory of Namenode is reduced by 10%. Further, prefetching and caching mechanisms are integrated in the proposed approach to improve access efficiency of small files. In future, we aim to evaluate the performance of the proposed approach with various cluster settings and different ranges of file sizes.

REFERENCES

[1]Apache Hadoop [online]." Available: http://hadoop.apache.org/

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), 2010.

[3] D. Borthakur, "The Hadoop Distributed File System: Architecture andDesign[online]."Available:https://hadoop.apache.org/docs/r0.18.0/ hdfs_design.pdf

[4] S. Ghemawat, H. Gobioff, and S.Leung, "The Google file system," in Proceedings Of ACM Symposium on Operating System Principles, pp. 29-43, Octomber 2003.

[5] T. White, "The small files problem [online]." Available: http://blog.cloudera.com/blog/2009/02/the-small-filesproblem/

[6] "Name-node memory size estimates and optimization proposal[online]."Availablehttps://issues.apache.org/jira/browse/HAD OOP-1687

[7] J. Dean, and S. Ghemavat, "MapReduce: simplified data processing on large clusters," in Proceedings of the 6th Symposium on Operating Systems Design and Implementation, pp. 137-150, USENIX Association , 2004.

[8] 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, pp. 1-8, 2009.

[9] G. Mackey, S. Sehrish, and J. Wang, "Improving metadata management for small files in HDFS," IEEE International Conference on Cluster Computing and Workshops, pp. 1-4, 2009.

[10] L. Jiang, B. Li, and M. Song, "The optimization of HDFS based on small files," 3rd IEEE International Conference on Broadband Network and Multimedia Technology (ICBNMT), pp. 912-915, 2010.

[11] 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), pp. 65-72, 2010.

[12] Chandrasekar S, Dakshinamurthy R, Seshakumar P G, Prabavathy B, and Chitra Babu, "A novel indexing scheme for efficient handling of small files in Hadoop distributed file system," IEEE International Conference on Computer Communication and Informatics (ICCCI), pp. 1-8, 2013.

[13] "Hadoop Archives Guide [online]." Available: http://hadoop.apache.org/docs/r1.2.1/hadoop_archives.html

[14] C. Vorapongkitipun, and N. Nupairoj, "Improving performance of small-file accessing in Hadoop," 11th International Joint Conference on Computer Science and Software Engineering (JCSSE), pp. 200-205, 2014.

[15]"SequenceFile [online]." Available: http://wiki.apache.org/hadoop/SequenceFile

[16]"Class MapFile [online]." Available: http://hadoop.apache.org/docs/current/api/org/apache/hadoop/io/MapFile.html

[17] X. Peng, D. Feng, H. Jiang, and F. Wang, "FARMER: a novel approach to file access correlation mining and evaluation reference model for optimizing peta-scale file system performance," in Proceedings of the 17th international symposium on High performance distributed computing, pp. 185-196, 2008.

[18] B. Dong, X. Zhong, Q. Zheng, L. Jian, J. Liu, J. Qiu, and Y. Li, "Correlation Based File Prefetching Approach for Hadoop," IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), pp. 41- 48, 2010.