EasyChair Preprint

№ 773

LHF: A New Archive based Approach to Accelerate Massive Small Files Access Performance in HDFS

Wenjun Tao, Yanlong Zhai and Jude Tchaye-Kondi

EasyChair preprints are intended for rapid dissemination of research results and are integrated with the rest of EasyChair.

February 7, 2019

LHF: A New Archive based Approach to Accelerate Massive Small Files Access Performance in HDFS

Wenjun Tao
School of Computer Science and Technology
Beijing Institute of Technology
Beijing, China
2120161049@bit.edu.cn

Yanlong Zhai
School of Computer Science and Technology
Beijing Institute of Technology
Beijing, China
ylzhai@bit.edu.cn

Jude Tchaye-Kondi
School of Computer Science and Technology
Beijing Institute of Technology
Beijing, China
tchaye59@yahoo.fr

***Abstract*—As one of the most popular open source projects, Hadoop is considered nowadays as the de-facto framework for managing and analyzing huge amounts of data. HDFS (Hadoop Distributed File System) is one of the core components in Hadoop framework to store big data, especially semi-structured and unstructured data. HDFS provides high scalability and reliability when handling large files across thousands of machines. But the performance will be severely degraded while dealing with massive small files. Although some effort was spent to investigate this well-known issue, existing approaches, such as HAR, SequenceFile, and MapFile, are limited in their ability to reduce the memory consumption of the NameNode and optimize the access performance in the meantime. In this paper, we presented LHF, a solution to handle massive small files in HDFS by merging small files into big files and building a linear hashing based extendable index to speed up the process of locating a small file. The advantages of our approach are (1) it significantly reduces the size of the metadata, (2) it does not require sorting the files at the client side, (3) it supports appending more small files to the merged file afterwards and (4) it achieves good access performance. A series of experiments were performed to demonstrate the effectiveness and efficiency of LHF as well, which takes less time while accessing files compared with other methods.**

Keywords—HDFS, massive small files, linear hashing

I. INTRODUCTION

In the contemporary era of ABC (AI, Big Data, cloud computing), a huge amount of data is being generated and this data has to be stored, processed and analyzed. As a widely used cloud storage platform, Hadoop has the feature of open source, low-cost, high-fault tolerance and scalability. Hadoop distributed file system (HDFS) [1] is the storage system of Hadoop with a master-slave architecture. HDFS is used by lot of running cloud service as file systems on their clusters and has been extensively utilized to support cloud applications. With the need of computing high availability, high scalability and high-capacity storage, cloud computing [2] is the ideal solution for big data analysis and processing. HDFS is designed for managing large files such as huge machine learning data sets, big video files, etc., therefore, when accessing and storing a large number of small files, HDFS will confront several problems. But there are many big data applications generate massive small files, such as social networking, online learning, e-commerce, Internet of Things, healthcare [3], in which the amount of data generated is enormous but the file size is usually from 10KB to 10MB.

Practically, the production of small files has become increasingly common and the amount of data is too large to be stored and processed by using local file system in many applications. Moreover, small files have to be processed along with big files. It is necessary to investigate the small files problem in big data storage system. Generally, files whose size is smaller than HDFS block size can be thought as small files. There are various reasons resulting in the Hadoop small files problem [4]. As we know, HDFS consists of one NameNode as the master and a group of DataNodes as slaves. NameNode is responsible for managing the metadata of the whole file system. So the first problem of storing massive small files is the huge memory consumption of NameNode. Because every data block, file, and directory in HDFS needs some space to store the metadata in the NameNode memory. For example, each file requires 150 bytes of memory for storing its metadata. For the same physical storage to store one big file or one million small files, small files need one million times additional memory space to store the metadata. We can reduce the NameNode memory consumption by reducing the number of small files on Hadoop cluster. Secondly, the existence of a large number of small files will deteriorate the performance of MapReduce processing as a huge number of small files results in a large volume of random disk I/O. Sequential disk I/O is one of the key factors for HDFS to get high throughput. It takes less time to read the data from one big file sequentially than randomly read many small files.

Presently there is no existing standard and generic file systems that is specifically designed for storing, handling and analyzing small files. Facebook [5] stores more than 60 billion images. Facing the performance problem, Facebook designed their own distributed file system called Haystack to optimize the storing and processing of large number of small images. Taobao is one of the largest e-commerce corporations, storing more than 20 billion images, with an average size of only 15KB. They built the TFS file system, an optimized open source framework for managing massive small files. All of their work serves their specific need for images. Most of the distributed file systems are designed primarily for large files, such as Lustre, GlusterFS, GPFS, ISILON, GFS, and HDFS. The metadata management, data layout and most of the features of these file systems are designed for managing large files, which makes the performance getting worse rapidly when managing small files, sometimes the system cannot even work when the number of small files is too large. Moreover, among the existing solutions, Hadoop is prime big data analytics platform at present. It is the leading platform regarding performance, reliability, scalability. A detailed discussion and comparison of our LHF with some related works are presented in Section III.

In this paper, we firstly analyze, compare and contrast various solutions available for solving Hadoop small files problem, including solution provided by Hadoop and others

which target various aspects of small files problem. We proposed a method named LHF (Linear Hashing File) which introduces the linear hashing [6] concepts to construct small files' index information and does not require small files to be sorted in advance. Our LHF not only improves the file accessing performance without altering the HDFS architecture but also allows additional files to be added to the existing archive which is the limitation of HAR [7] and MapFile's [8]. The LHF solution specifies the files into three categories according to the file size and designs two types of index structure for small files. When building the archive file, the content of the small files will be merged into some data part files, whereas the meta-information will be built into the index files following the principles of linear hashing. The index is composed of multiple hash buckets, which will dynamically increase when appending more files to the archive. With this hashing based index design, the complexity of locating any of the buckets is O(1). The offset information of each specific file inside the merged data file is automatically sorted and stored in hash buckets. Every hash bucket is stored as a separate file in HDFS and will be loaded into the DataNode's main memory to accelerate the access performance. We have implemented and evaluated LHF on Hadoop cluster and find out that the proposed solution is superior in accessing time and supports appending additional files to the archive as well.

The rest of this paper is organized as follows: The background of this research is firstly discussed in Section II. The related work is introduced in Section III. Section IV presents the detail design of our proposed method. The experimental results of evaluating the approach are summarized in Section V. Section VI concludes this paper.

II. BACKGROUND

In this section, we will discuss about the architecture of HDFS, the problems associated with processing, storing and accessing small files and the concept of the linear hashing.

A. HDFS

HDFS with a master/slave architecture is a Hadoop file system that transfers large files into smaller blocks and distributes these blocks to different nodes named DataNodes. Excessive copies of each block are maintained in HDFS for fault tolerance. NameNode maintains a database containing a mapping from logical files to physical blocks in the DataNodes. Block replication is also managed by the NameNode. The NameNode handles the management of the file system namespace, metadata, and requests from clients to access data. DataNodes provide blocking and serving IO requests for the clients. They also create, delete, and replicate data blocks upon getting command from the NameNode.

B. Problems of managing small files in HDFS

1) NameNodes memory consumption: The metadata of the file system will be loaded in the main memory of the NameNode. In general, the metadata of a file consumes approximately 150 bytes of main memory [9]. For each block that has three replicas by default, its metadata will consume approximately 368 bytes main memory. With 24 million files stored on HDFS, the NameNode will need 16GB of memory to store the metadata. So storing massive small files will consume huge memory of the NameNode and lead to performance decline for the cluster.

2) The MapReduce performance problem: Processing a large number of small files will reduce the performance of MapReduce jobs. This is mainly because accessing a large number of small files will generate numerous random disk IO, which is extremely slow compared with accessing one large file with the same size sequentially.

3) High storing time: Storing a big file to HDFS takes less time than storing many small files of the same size. This is because, for storing one file, it has to go through the process of creating files, allocate metadata and data block, writing data, making replicas and closing files. For instance, storing 550,000 small files with the size approximately from 1KB to 10KB to HDFS takes about 7.7 hours [10].

4) NameNode performance degradation: When the client needs to access the file, it first needs to obtain the metadata of the file from the NameNode, and then read the file content according to the acquired metadata to the corresponding DataNode. For small files, the time spent on access is mainly in the management of metadata and disk addressing, while data transfer takes less time. Accessing a large number of small files, HDFS clients need to interact with the NameNode frequently, so the performance of the NameNode will be seriously degraded especially facing concurrent file access requests.

C. Linear hashing

Linear hashing [6] is one type of dynamic hashing that support dynamically expand the hash table by adding one bucket at a time. This single-bucket expansion can efficiently control the collision chain. The cost of a hash table extension does not happen all at once, but throughout each hash table insert operation. It consists of a hash function *h*, data buckets, a bucket array *a*, and meta-information. The data bucket is made up of data blocks, and meta-information includes the hash level *i*, the number of data buckets *n*, the number of records *r*, and a fixed fraction *p* is the least upper bound of *r* / *n*. A data block in a data bucket stores the data. The addresses of data buckets are kept in a bucket array whose size is n. In general, the hash function h is modulo by 2^i. In the process of data access, the hash function *h* calculates the hash value of the target data *v*, and a[*v*] keeps the address of the data bucket storing the target data. When *v* is bigger than *n* - 1, a[*v*] does not exist. In this case, *v* - *i* - 1 is used for the hash value. Then search this data bucket for retrieval.

III. RELATED WORK

To solve the small files problem, we merge small files and store the larger file after merging on HDFS. Some of the solutions for solving small file problem of HDFS are discussed as below.

Hadoop Archive or HAR [7] [11] is a file archiving tool that efficiently puts small files into HDFS blocks. It can package multiple small files into one HAR file, which allows the file to be transparent while reducing the use of NameNode memory. In addition, HAR has some drawbacks: First, once created, archives cannot be changed. To add or remove files inside, you must recreate the archive. Second, when accessing the HAR file, it requires 2 index-file read

operations as well as one data-file read operation which are less efficient than reading files from HDFS.

NHAR [12] redesign the indexing structure to improve the metadata mainframe of HDFS and the accessing performance without altering the HDFS architecture. With new design, NHAR enables to allow additional files to be appended to the existing archive. In order to improve the access performance, NHAR model uses a single-level index. NHAR uses index information to create a hash table instead of master-index approach. These information is divided into multiple index files. It allows users to append additional files to the existing NHAR file. The adding process involves three steps: archiving the new small files, merging index files and moving the new part file.

SequenceFile [13] which stores the data in the form of binary key-value pairs is used as a storage container for small files. In this data structure, file name is stored as the key and file content is stored as the value. It supports compression and decompression both at record level and block level. SequenceFile also presents several disadvantages. For a particular key, it does not have a mechanism for update and delete operation; it only supports the append method; and secondly, this approach has low access efficiency as it takes quite a long time to make a Sequence file. If a user needs to look up for a particular key, it is required to read the Sequence File from the beginning of this SequenceFile.

A MapFile [8] which is composed of two files, an index file and a data file, is a sorted Sequence File. It maintains the index file to store keys location information to allow lookup of the data file by key. The key-value pairs are sorted by key and stored as records in the data file. MapFile facilitates to look up for the key without having to read the full file. However, it cannot provide flexible APIs for applications as only append method is supported for a particular key. In other words, not every key can be appended to a created MapFile because it must keep all the keys in order. Furthermore, while reading a file from MapFile, we need to search the index file, address the disk once and scan with 128 entries at most by default. The less key stored for the same small file set, the more entries are required to be scanned and compared with the target key.

CombineFileInputFormat [14] is also a method provided by Hadoop, however, it is MapReduce API to input small files. Chen, J., Wang, D. proposed an improved HDFS [15] whose structure consisted of two parts: client component which merges small files into a big file and data node component which satisfies the cache resource management. Gurav, Y.B. [16][17] proposed Extended Hadoop Distributed File System (EHDFS) which has been designed and implemented in such a way that a large number of small files can be merged into a single combined file and it also provides a framework for prefetching metadata for a specified number of files. Yanfeng Lyu proposed an optimized strategy for small files storing and accessing in HDFS [18]. In their work, their method considers the size of small files when merging files into combine file, and generates a map record for each small file. Meanwhile, they apply prefetching and caching mechanism to enhance the access efficiency. Z. Gao proposed an effective merge strategy based hierarchy for improving small file problem on HDFS [19], which makes a radix sort on files set and merges these files orderly. Weipeng Jing [20] and his team aim to correct the problems of IoT and CPS because of small files problem.

Most of the existing solutions do not perform well in accessing small files inside the merged files. Many methods, such as HAR [7] or MapFile [8] provided by hadoop, require small files to be ordered in advance. HAR does not support the append operation and MapFile only supports the append operation for particular keys theoretically. Although NHAR supports append operations, it is not flexible enough. The number of index files in NHAR is fixed. At the beginning, the number of small files is small. Number of index files of NHAR is set to a small digital and we create the NHAR file. However, if we constantly append additional small files to a NHAR file, each index file of NHAR will increase sharply and deteriorate the performance of reading files. Some other methods may change the architecture of HDFS, or rely on another system such as HBase [21], or propose a completely different approach [22].

In addition, most of the existing methods do not focus on the small files of relatively small sizes, such as only a few, tens or hundreds of bytes. To access these files, those methods need to read the index first and then read the data, although the size of the index information is similar to the file size, which may waste some time. As a result, we reclassified the files and changed the way we read ultra-small files for performance improvement.

The method we proposed supports the appending operation, and does not require small files to be sorted in advance and has high reading efficiency.

IV. Proposed Work

The basic idea of the proposed approach includes two steps: (1) merging small files into large files to reduce quantity of files and optimize the NameNode memory consumption, (2) storing these small files metadata in our index system based on linear hashing principle thus allow us extending the function of file management such as append additional files to the existing archive file, delete files from the archive file, etc.

A. Overview

In this paper, we proposed a new way to deal with a huge number of small files. To solve the small files problem, we merge them into large files on HDFS and use the linear hashing principle to organize these files index information in several small index files. When searching for a small file inside the archive file, instead of browsing the entire archive file from beginning to end just as the Sequence File does, we firstly retrieve the information from the index file that allows us to access the file position in the archive file directly. We improve the index information lookup by splitting total small files' index into several pieces, limiting the capacity of each piece and allowing to have a direct access to the index file piece containing the information during index record lookup.

As showed in Fig. 1, File Filtering Criteria module classifies files into three categories according to the file size: **large files, common small files, and ultra-small files**. Large files are directly uploaded to HDFS. Common small files and ultra-small files are sent to the File Merging module to build the merged file, and then the merged files and index files are uploaded to HDFS. The general process of uploading files to HDFS is presented in Algorithm 1.

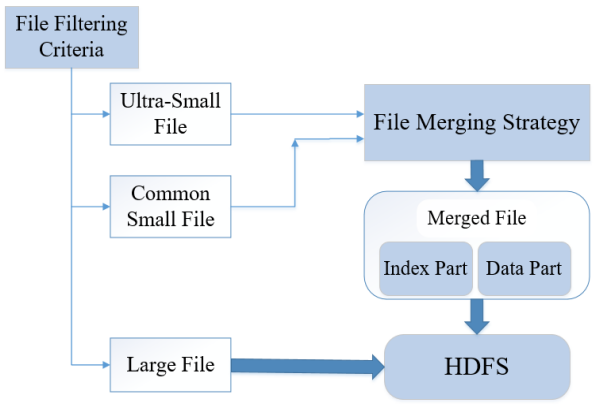


Fig. 1. Architecture of our LHF.

Algorithm 1: File Processing Algorithm

```
1  begin
2    Initialize File set (FS) and create a new combined file or
     open an existing combined file.
3    for each file Fi in FS do
4      if(Fi ∈ FS and Fi is a big file)  then
5        Upload the file to HDFS directly.
6      else
7        Merge the file using the File Merging Strategy module.
8      end
9    end
10   Close the combined file.
11 end
```

B. *Design*

In our proposed LHF, we firstly classify files based on the file size. The File Filtering Criteria module will classify files into three categories:

- **Large files**: whose size are larger than the HDFS default block size (128 MB). They will be uploaded to HDFS directly.
- **Ultra-small files**: whose size are smaller than predefined value, in our case 1 KB. They will be processed by the File Merging module and stored in the index file directly.
- **Common small files**: whose size are between the predefined value and the default HDFS block size. They will be processed by the File Merging module and merged into big data part files.

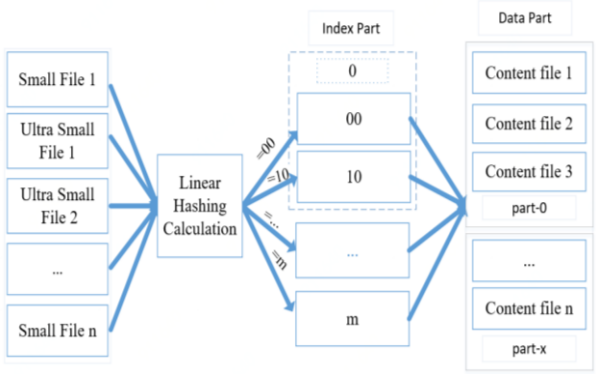


Fig. 2. Our proposed LHF technique.

Fig. 2 shows the proposed structure of the archive file. The archive file consists of two parts, the Index Part and the Data Part. The Data Part stores the content of common small files, the Index Part stores the ultra-small files' content and the index information of all files in this archive.

1) Merging of small files

Common small files and ultra-small files are handled differently in the File Merging module. In case of common small file, the file content is appended to the data part and information like their size, the location of the small file in the data part file and so on are used to build an index record that is inserted to the index part. In case of ultra-small file, we directly write the contents of the file to the appropriate bucket of the index part using linear hashing algorithm. The merging process of our LHF is described in Fig. 3.

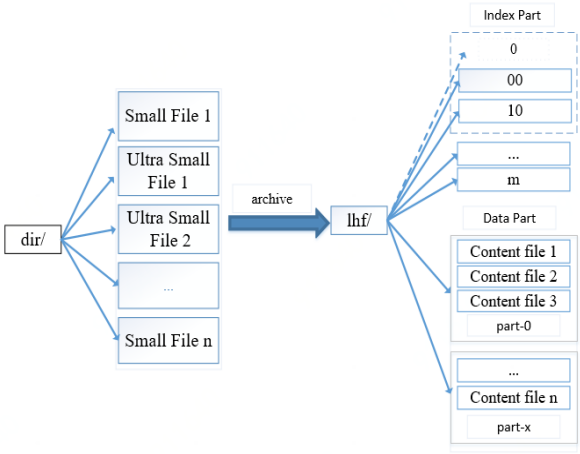


Fig. 3. The merging process of LHF.

The Data Part consists of a sequence of part-* files. During the merging process if the remaining space of the part file is smaller than the small file's size, we create a new part file to continue writing new small files' content. As a result, a small file's content is in only one data part file. Algorithm 2 describes the process of file merging. Since common small files and ultra-small files are handled differently, their information stored in linear hashing also different. The index information whose format is described in Fig. 4 which includes the full file name, the file identifier which indicates that the file is a common small file, the part file where this small file is located, the starting offset of this small file's content in the part file and the small file's length in the merged file is temporarily stored in the client's linear hashing, and the ultra-small files' content is also stored in the client's same linear hashing in the format described in Fig. 5 which includes the full file name, the file identifier which indicates that the file is an ultra-small file or common small file.

File Full Name	File Identifier	Part File	Starting Offset	File Length	File Ending

Fig. 4. Common small file's index information.

File Full Name	File Identifier	Data Content	File Ending

Fig. 5. Ultra small file's index information.

To increase the access performance, the LHF use single index level instead of using master-index approach as in HAR. LHF organizes Index Part as linear hashing where the index records are split into several index files that represent the hashing buckets. Each bucket is stored as a HDFS file and the name of the bucket file is specified as the bucket label for easy locating. As the number of records in the index files increases, the buckets are split according to the linear hashing split policy. After all the files are merged, the contents of each bucket in the linear hashing are sorted by file name, and then written to HDFS.

Algorithm 2: File Merging Algorithm

```
1 begin
2   initiate the linear hashing0 with two buckets whose
    labels are "0" and "1"
3   create Data Part file part-0
4   for each small file do
4    if it is a common small file then
5     if small file's size+part-n file's size>fixed size then
6       n←n+1
7       create file part-n
8     end
9       write the small file's content to part-n and record
        the starting offset, small file's length and Data
        Part file's name in the given format to the linear
        hashing0
10    else if it is ultra-small file then
11     write the content of the ultra-small in the given
       format to linear hashing0
12    end
13   end
14   sort each bucket and store the linear hashing0 to
       HDFS
15 end
```

The linear hashing we used to store index records is described as follows, and we define some parameters like:

- **n**: the number of buckets, the buckets are numbered from **0** to **n-1.**
- **level**: the number of bits used in the file name hash value to decide in which bucket to store it information's. It is also the number of bits necessary to represent n-1.
- **ɣ**: defined as the total number of records in all buckets
- **α:** the average capacity defined for each bucket
- **Load Factor (LF)**: the fill rate of all index files and is calculated using (1).

$$LF = \frac{\gamma}{n * \alpha} \quad (1)$$

- **The split policy**: It is the policy that we define that will allow us to create a new bucket by calling the split operation when the available buckets are getting filled. In our experiment, the split policy is verified when the **Load Factor** is greater than **75%** (LF>75%).

To insert a small file index record, we need to calculate in which bucket the insertion must be done. To do so, from the file name we calculate a unique number using a hash function, and we get the bucket position by considering the last level bits of the binary representation of this unique number.

The split operations are done in round-robin. They serve to extend the number of index files when existing index files are reaching their maximum capacity. At each record insertion in a bucket, we check if split policy is verified (LF> 75%) if yes, we call the split operation. During the split operation, a new bucket is created and data from the corresponding bucket in **level-1** are redistributed into the new bucket. Because HDFS is used to store files, the split operation is performed by using Algorithm 3.

In Fig. 6, we have an example where we have 2 buckets and a new bucket is created during the split operation. In this example, we have in total 3 buckets (0, 1, 2) then **n** = 3; **level** = 2(because last bucket label is **2 = 10 in binary**, we need two bits to represent 2). The actual **level** is 2 to find the corresponding bucket at **level-1** position, we just take the corresponding binary representation of the new bucket position without the first bit. The new bucket position is 2 (10 in binary) of the corresponding bucket at level-1 is at position 0 (2=10 in binary without the first bit).

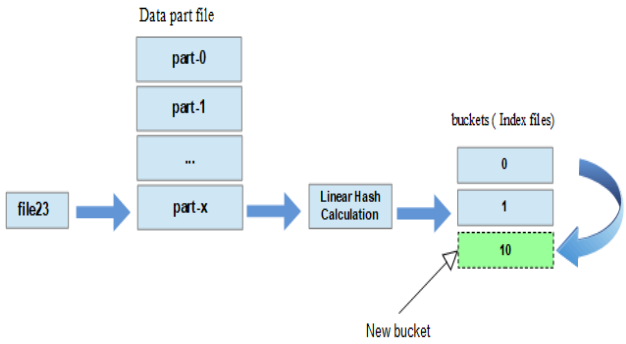


Fig. 6. Example of split operation

Algorithm 3: Index File Splitting Algorithm

```
/*This is executed at client side.*/
1  begin
4   Calculate Load Factor(LF) using (1)
5   if (LF > 75%) then
6      label1 ←Calculate the next label in the index files
         sequence (last index file label+1)
7      label2 ← Get the label given by label1 without the first bit
8         Create two files (file1 & file2)
9         Initialize Index records(R) from index with label2
10        for each record r in R do
11            α←Calculate r label using linear hashing
              calculation
12            if(α == label1) then
13               Add r to file1
14            else if(α == label2)then
15               Add r to file2
16            end
17        end
18        Rename file2 to label1 and upload it to HDFS
```

19 Overwrite label2 index file with file1 content
20 **end**
21 end

2) Accessing small files

To access a file in the merged file, which is illustrated in Fig. 7 and algorithm 4, our archiving program use a hash function to calculate the linear hash code from this file's full name. With Linear Hashing Calculation module, we locate the index file containing the information of this file by considering the bucket with a label equal to the last level bits of the binary representation of the calculated code. Then we lookup the file information in the index file using a binary search method. With the acquired information, our program will extract the file identifier to determine if this file is a common small file or an ultra-small file. If it is a common small file, locate the data part file and seek the data part file to the starting position and then read file's data. If it is an ultra-small we get the file content within the information obtained directly.

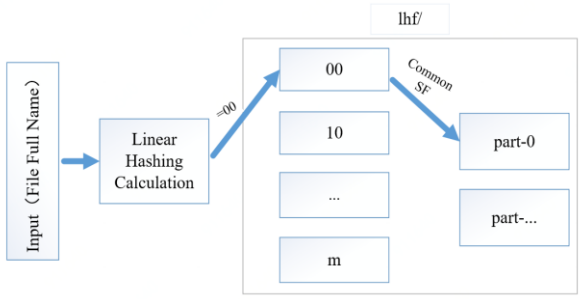


Fig. 7. Accessing a file in our LHF.

Algorithm 4: File Accessing Algorithm

1 **begin**
2 **input** small file name name0
3 hashCode0 ← linearHashingCaculation(name0)
4 indexFile0 ← getIndexFile(hashCode0)
5 indexInformation←binarysearch(indexFile0,name0)
6 fileIdentifier←extract(indexInformation)
7 **if**(fileIdentifier==”1”) **then**
8 locate the data part file and read the small file’s content with starting offset and this small file’s length
9 **else if**(fileIdentifier==”0”) **then**
10 get the ultra-small file’s content from indexInformation
11 **end**
12 **end**

To further improve the performance of accessing files in the merged file, we can prefetch the Index Part files to the DataNode memory.

3) Appending small files

The process of appending small files is similar to merging small files. Certainly, there are some differences. We ought to read all the Index Part files on HDFS to build a linear hashing in the client's memory. During the appending process, common small files' content has to be written into the last created part file that is smaller than the defined maximum size value, otherwise, written into a new part file. Fig. 8 describes the splitting process of the bucket while appending files.

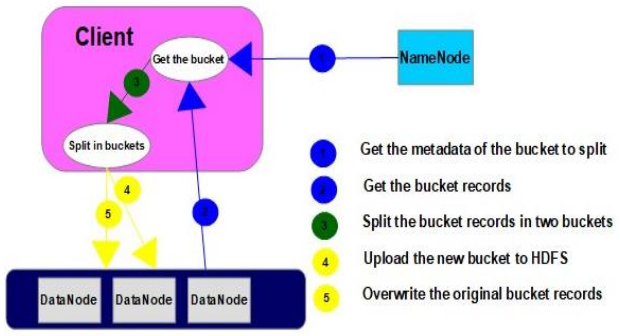


Fig. 8. The splitting process of the bucket while appending files.

V. Experimental Evaluation

In this section, we present experimental results of our LHF comparing to native HDFS, HAR and MapFile.

A. Experimental environment

The experiment environment is built on a cluster of 4 nodes. One node serves as NameNode and the other 3 nodes act as DataNodes. Each of them has Intel(R) Core(TM) i5-2400 CPU @ 3.10GHz processors and 4 GB memory. The operating system is Centos 6.6, the jdk version is 1.8.0 and the Hadoop version is 2.6.5. The number of replicas is set to 3 and HDFS block size is 128MB. The files we use in the experiments include text files, web page files and images. To test the performance of LHF comprehensively, we utilized different kinds of file sets to do the experiments. First kind of file set is text file sets, which contain 20,000, 40,000，60,000 and 80,000 files respectively. The total size of file set is up to 288MB and file sizes range from 250B to 160KB which means many ultra-small files in this data set. The second type of file set is about 4.5GB and contains about 20,000 files whose size range from 20KB to 30MB, which means all files are common small files in this data set. All the small files are collected from the internet. It is mentioned that our method concentrates on how to store and access small files on HDFS and certain features of the data for evaluation will not make a difference to the evaluation results, unlike Machine Learning Algorithms.

B. Experiments and Analysis

Our experiments evaluate the LHF performance by five parameters: memory usage of NameNode, I/O throughput, creation time of the merged file, reading performance and time to append files to an existing archive file. We set the size of the bucket to 10 thousand.

To evaluate the performance of accessing small files, we performed read operations directly from HDFS, HAR, MapFile and our LHF on first file sets and MapFile and our LHF on the second file sets which are repeated 5 times.

For the first file sets, we read part of the file sets and the whole file sets. When reading part of the file sets, each time is tested with the same 2000, 4000, 6000, 8000, 10000 random files with different method respectively. The result is shown in the following figures.

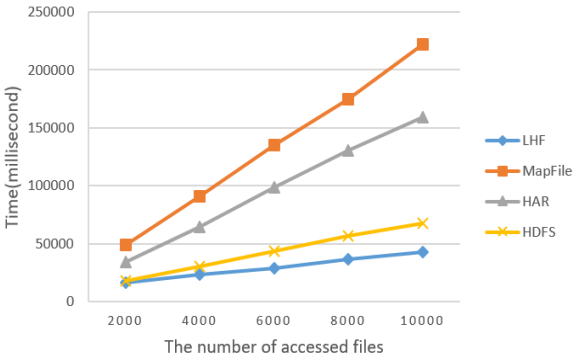


Fig. 9. Performance of accessing files from 20,000 files.

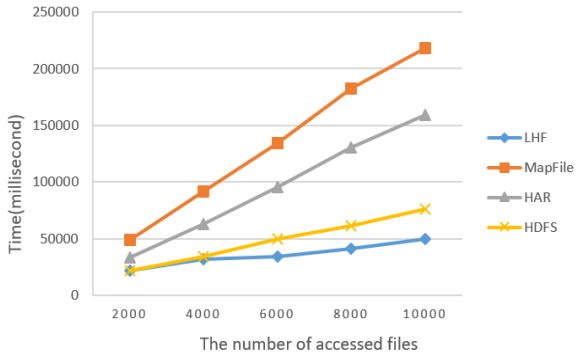


Fig. 10. Performance of accessing files from 40,000 files.

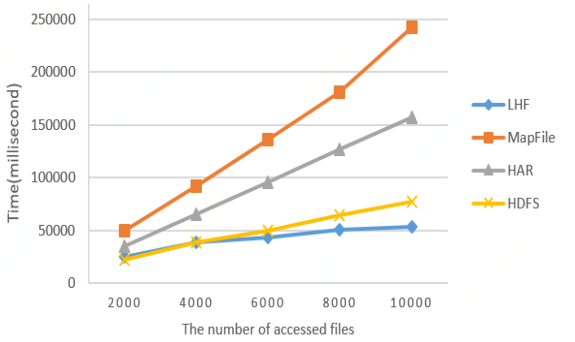


Fig. 11. Performance of accessing files from 60,000 files.

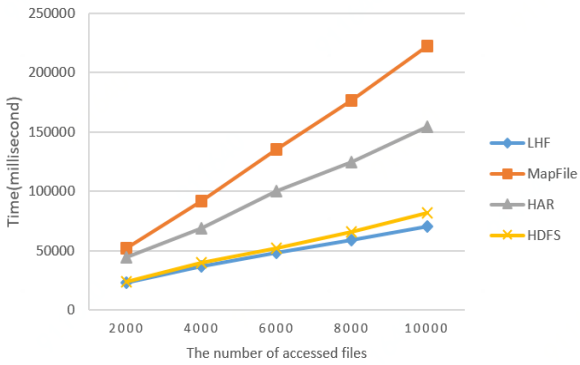


Fig. 12. Performance of accessing files from 80,000 files.

In all the above figures, our LHF almost provided the best performance and HAR performed the worst. The only set of data in Fig. 12 shows that reading 2000 files from HDFS directly performed better than our LHF. This may be because reading index part files ahead needs some time.

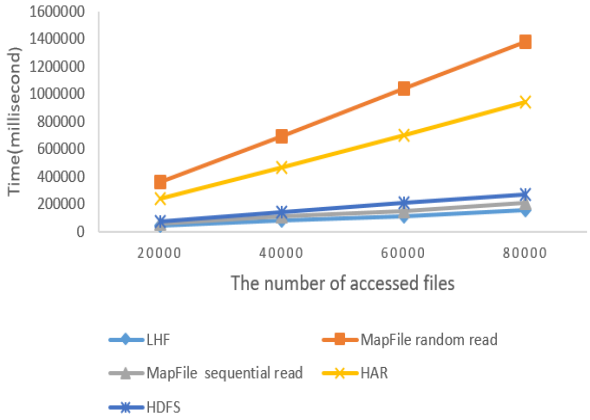


Fig. 13. Performance of accessing all files in file sets.

When reading all the files in the merged file, MapFile has two ways to read the file sets--random reading and sequential reading. From Fig. 13, our LHF still performed best and sequential reading of MapFile also provides good performance.

We also calculated the throughput rate of reading entire file sets depicted in Fig. 14 from the reading time and the merged file's size.

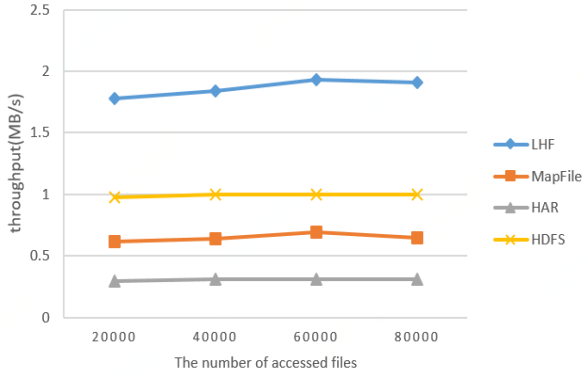


Fig. 14. Throughput of accessing files.

In Fig. 15, we evaluated the time spent on creating an archive file to store the file sets. Our LHF costs more time slightly. And appending 20 thousand, 40 thousand, 60 thousand to the archive file which contains 20,000 files cost 89096, 144761, 203941 milliseconds respectively.

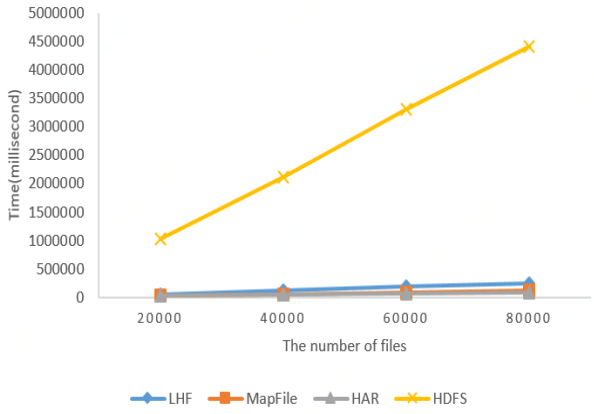
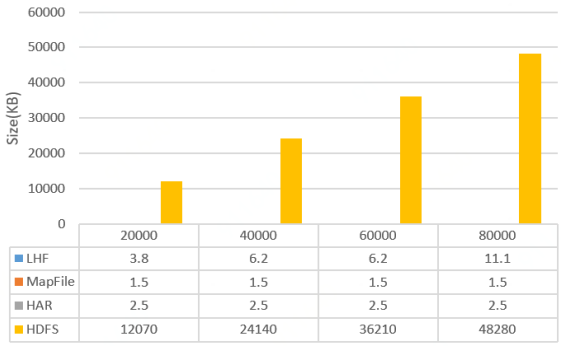


Fig. 15. The performance of creating merged or uploading files.

We also evaluated the memory usage in NameNode of storing small file. We measured memory usage as showed Fig. 16.



	20000	40000	60000	80000
LHF	3.8	6.2	6.2	11.1
MapFile	1.5	1.5	1.5	1.5
HAR	2.5	2.5	2.5	2.5
HDFS	12070	24140	36210	48280

Fig. 16. Memory usage of NameNode.

As expected, LHF, MapFile and HAR provide much better efficiency of storing small files than the original HDFS. Comparing to MapFile and HAR, our LHF consumes more memory slightly. This is owing to each bucket in linear hashing needs a file to store.

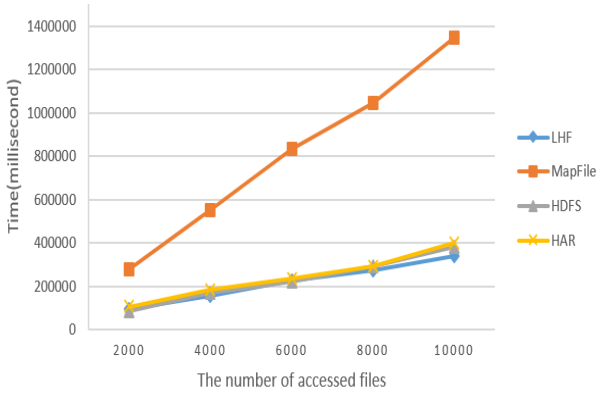


Fig. 17. The performance of accessing files in second file set.

To avoid the coincidence that our LHF provides good performance tested by only the first kind of file sets, we further use the second kind of file set in which files size range from 20KB to 30MB so there are no ultra-small files. The reading performance is shown in Fig. 17.

From the figure, we can see that our LHF still works better than all the other solutions.

VI. Conclusion

In our method, small files problem of HDFS is solved and the files to be merged do not need to be sorted, which makes appending additional files to existing merged file possible. For random file access, you can use the file name to locate the bucket file in the index, which only need one time hash computation; when reading the file content, read the corresponding content according to the index information to the specified location of the specified file, without having to find the corresponding content in the data file like MapFile. Compared to HAR and MapFile, both appending function and the performance of random access are advantages of our approach.

Compared with multiple merged files, appending additional files to the existing merged file can make full use of the data block, avoiding the case that a small number of files also occupying a merged file and corresponding index file, to some extent alleviate the memory pressure of the NameNode. At the same time, there is no need to care about which merged file the small file is in, which is easy to access. In addition, due to the number of records in the index file is always within a certain range, there is no need to worry about the performance degradation of the index information as the number of additional files increases. Certainly, when to start a new file is up to the user and a little overhead such as a little longer merging time and the use of memory while appending files to existing merged file is needed.

In the near future, we will study the maximum number of elements store in each bucket of the linear hashing to get the best reading performance. And we will also improve prefetching and caching mechanisms.

Acknowledgment

This work is supported by the National Nature Science Foundation of China (Grant No. 61602037).

References

[1] Hdfs architecture guide. [Online]. Available:https://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html.

[2] P. Mell and T. Grance, “A nist definition of cloud computing, national institute of standards and technology. nist sp 800-145,” 2009.

[3] B. Dong, Q. Zheng, M. Qiao, J. Shu, and J. Yang, “Bluesky cloud framework: an e-learning framework embracing cloud computing,” in IEEE International Conference on Cloud Computing. Springer, 2009, pp. 577–582.

[4] The small files problem. [Online]. Available:http://blog.cloudera.com/blog/2009/02/the-small-files-problem/.

[5] Tim Kraska, “Finding the Needle in the Big Data Systems Haystack,” IEEE Internet Computing 17(1), pp.84-86, 2013.

[6] W. Litwin, "Linear hashing: new tool for file and table addressing," Proc. Intle. Conf. on Very Large Databases, pp.212-233.

[7] HAR.[Online].Available:https://hadoop.apache.org/docs/current/hadoop-archives/HadoopArchives.html.

[8] MapFile.[Online].Available:http://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/io/MapFile.html.

[9] Divyashikha Sethia, Shalini Sheoran and Huzur Saran, “Optimized MapFile based Storage of Small files in Hadoop,” CCGrid, 2017, pp.906-912.

[10] 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,” in 2009 IEEE International Conference on Cluster Computing and Workshops. IEEE, 2009, pp. 1–8.

[11] Sachin Bende and Rajashree Shedge, “Dealing with Small Files Problem in Hadoop Distributed File System,” International Conference on Communication, Computing and Virtualization (ICCCV), 2016, pp.1001-1012.

[12] C. Vorapongkitipun, N. Nupairoj. “Improving performance of small-file accessing in Hadoop,” IEEE International Conference on Computer Science and Software Engineering (JCSSE), 2014, pp.200-205.

[13] SequenceFile.[Online].Available:https://wiki.apache.org/hadoop/SequenceFile.

[14] CombineFileInputFormat.[Online].Available:http://hadoop.apache.org/docs/r2.7.1/api/org/apache/hadoop/mapreduce/lib/input/CombineFileInputFormat.html

[15] Chen, J., Wang, D., Fu, L., and Zhao, W., “An Improved Small File Processing Method for HDFS,” in International Journal of Digital

Content Technology and its Applications (JDCTA), Vol.6, pp.296-304, Nov 2012.

[16] Gurav, Y.B. and Jayakar, K.P., "Efficient Way for Handling Small Files using Extended HDFS," International Journal of Computer Science and Mobile Computing, Vol.3, pp.785-789, June 2014.

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

[18] Yanfeng Lyu, Xunli Fan and Kun Liu, "An Optimized Strategy for Small Files Storing and Accessing in HDFS," IEEE International Conference on Computational Science and Engineering (CSE), Volume: 1, pp.611 - 614, 2017.

[19] Z. Gao, Y. Qin, and K. Niu, "An effective merge strategy based hierarchy for improving small file problem on HDFS," in 2016 4th International Conference on Cloud Computing and Intelligence Systems (CCIS), 2016, pp.327–331.

[20] Weipeng Jing, Danyu Tong, Guangsheng Chen, Chuanyu Zhao, and Liangkuan Zhu, "An optimized method of HDFS for massive small files storage," Comput. Sci. Inf. Syst. 15(3), pp.533-548, 2018.

[21] Tong Ouyang, Yizhen Cao, "Research and Optimization of Massive Music Data Access Based on HDFS," ICIS, 2018, pp.697-700.

[22] Pierre Matri, María S. Pérez, Alexandru Costan, Gabriel Antoniu, "TýrFS: Increasing Small Files Access Performance with Dynamic Metadata Replication," CCGrid, 2018, pp.452-461.