Efficient File Accessing Techniques on Hadoop Distributed File Systems

Wei Qu(✉), Siyao Cheng, and Hongzhi Wang

School of Computer Science and Technology, Harbin Institute of Technology, Harbin, China
{qwei, csy, wangzh}@hit.edu.cn

Abstract. Hadoop framework emerged at the right moment when traditional tools were powerless in terms of handling big data. Hadoop Distributed File System (HDFS) which serves as a highly fault-tolerance distributed file system in Hadoop, can improve the throughput of data access effectively. It is very suitable for the application of handling large amounts of datasets. However, Hadoop has the disadvantage that the memory usage rate in NameNode is so high when processing large amounts of small files that it has become the limit of the whole system. In this paper, we propose an approach to optimize the performance of HDFS with small files. The basic idea is to merge small files into a large one whose size is suitable for a block. Furthermore, indexes are built to meet the requirements for fast access to all files in HDFS. Preliminary experiment results show that our approach achieves better performance.

Keywords: HDFS · Hadoop · Index · Small files

1 Introduction

Hadoop [1] is an open-source distributed platform under apache software foundation in which Hadoop Distributed File System [2] and Yarn [3] act as the core. It provides users with the underlying transparent distributed infrastructure. Hadoop can be deployed on cheap hardware to build a distributed system with the advantages of high fault-tolerance, high scalability, etc. With the convenience above, Hadoop can be used to deal with many complex problems—Top K problems, K-means clustering problems, Bayes classification, for example. Moreover, from the points of e-commerce, mobile-data, image-processing and so on, it has extensive applications. Then MapReduce, a distributed computing framework, allows the users without full knowledge of fundamental details to develop parallel application in distributed system, making full use of large-scale computing resources, and solve the problems in the situations where single traditional high-performance machines have low efficiency.

In the area of social computing, one of the most popular areas at present, HDFS is also widely used. The social environment is made up of social system, social network and so on. The social network is the most important one which takes people as the nodes and relationships as the edge. All the social media are based on large-scale data center and a large number of server cluster. Here the traditional relational database is

© Springer Science+Business Media Singapore 2016
W. Che et al. (Eds.): ICYCSEE 2016, Part I, CCIS 623, pp. 350–361, 2016.
DOI: 10.1007/978-981-10-2053-7_31

hard to meet the requirements for data processing and the NoSql databases perform well such as Hadoop HBase, Google Big Table and so on.

HDFS has many advantages such as: (1) supporting big data with the size of GB or even TB; (2) hardware malfunction detection and quick response; (3) paying more attentions to the throughput of data rather than the speed of data access; (4) simplified consistency model: once a file has been created or closed, it may not be modified.

However, HDFS based on the above objectives is not suitable for some scenario, it has the following two problems:

1. Large amounts of small files. Hadoop has to face such a reality that it is short of processing massive small files. Because the HDFS assigns indexes for every file which are stored in metadata on NameNode, the memory usage rate increases as the files increase, thus, making the NameNode overwhelmed. Large amounts of small files are common in many applications. Taking TaoBao as an example, 90 % flow of the whole network is caused by image access, because image is the most convincing description of goods for both buyers and sellers. There are 28.6 billion pictures stored in the backend server system in TaoBao and the average size of them is just 17.45 KB, including 61 % less than 8 KB [4]. Another example is the web crawler technology, which is web applications or scripts that automatically collect information according to certain rules. The main goal of using web crawler is to get more web pages within a few KB which are correlative with a certain topic and prepare data for users' querying [5].
2. High latency of data access. Applications interacting with users are supposed to response within several seconds or milliseconds. But Hadoop has been optimized for high data throughput, thus, the delay is relatively high. What's more, the files on HDFS are usually accessed by path which has high expectations for users' ability. If one wants to search a file under multi-level directory, the exact path of the file is needed. So the HDFS can not process the request whether a file exists or not without its path. For example, the HDFS usually throws exception when a file is uploaded because the users can not detect the existence by name. This is more obvious with the amounts of files getting larger.

In order to solve the problems mentioned above, this paper presents two approaches respectively: (1) the adaptive storage strategy and (2) name-based index.

In short, the contributions of this paper are as follows. First, an adaptive storage strategy is provided to reduce the memory usage. Second, a name-based index is proposed to accelerate file access. Third, extensive simulations and real system experiments were carried out to verify the efficiencies of all proposed methods.

The rest of this paper is organized as follows. Section 2 describes the background and preliminary knowledge of HDFS. Section 3 gives a summary of our approach. Section 4 gives the details about how an adaptive storage strategy is carried out to handle large amounts of small files. Section 5 introduces the name-based index technique. Section 6 evaluates the methods in this paper by real system experiments. Section 7 refers to some related work and Sect. 8 draws the conclusion.

2 Preliminary Knowledge

HDFS uses the master-slave structure consisting of NameNode, DataNode and Client. As the management of the whole system, NameNode manages the namespace, cluster configuration, file-based replication and so on. DataNode is the basic unit for storing files, which saves the specific data contents on HDFS and parity information in forms of block, while the client is responsible for communication with the NameNode and DataNode, data access in filesystem and document operation. In addition, there is another node named SecondaryNameNode whose function is to combine namespace image and assist daemons to edit log.

When a client has read-file operations, the DFSInputStream object will establish contact with the nearest DataNode via the stream interface. The client calls the read method repeatedly and receives the data packets. After the connection to DataNode has been closed, DFSInputStream gets information of the next block via the getBlock Locations method and receives data packets again through the interface in DataNode. Moreover, DFSInputStream may call this method several times because the client protocol will not return all the information at one turn. The client closes the stream when the read-data task is finished. The structure is shown in Fig. 1.

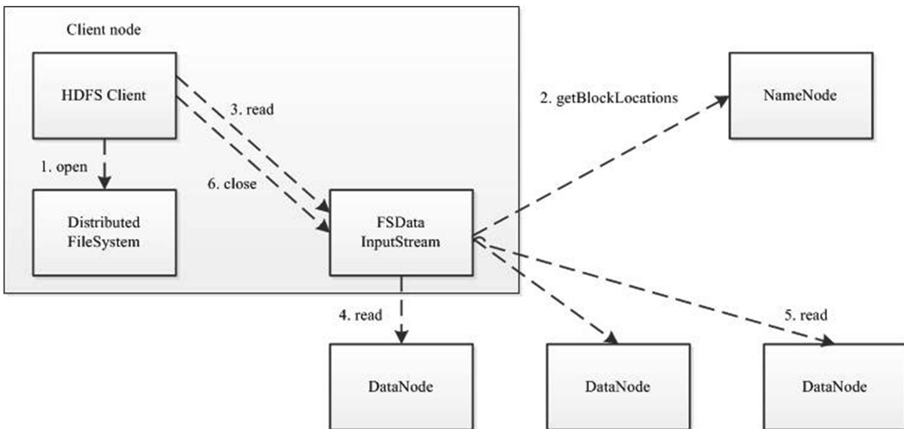


Fig. 1. Read-file process on HDFS

When the writing process begin, the client calls a method of DistributedFileSystem to create files. DistributedFileSystem creates the DFSOutputStream called from remote procedure at the same time. Then the NameNode executes the same-named method to create files in the system namespace. DistributedFileSystem packets the DFSOutputStream object into FSDataOutput instance and then sends it to client. After the empty file is created, the client applies for data blocks from NameNode. When the addBlock method is finished, it returns a LocatedBlock object which contains the identification and version number of new data blocks. Then the data in input stream is put into internal queues of DFSOutputStream object. A normal write-file process is accomplished in this way. Figure 2 shows the details [6].

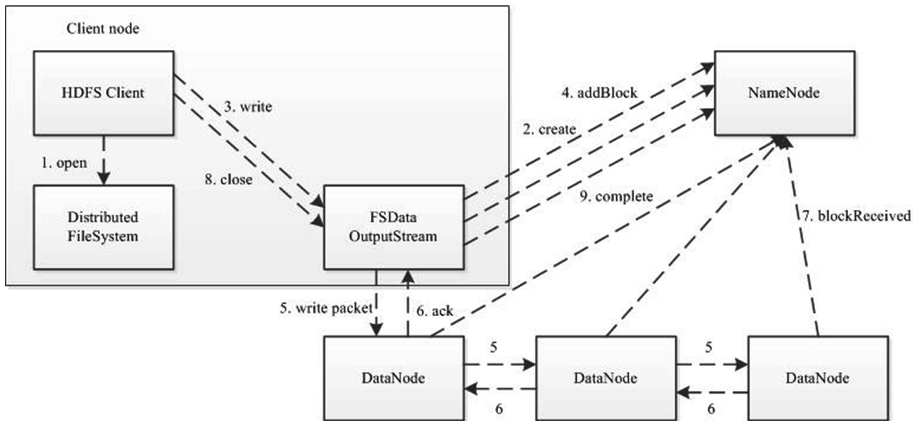


Fig. 2. Write-file process on HDFS

3 Principle of Our Approach

The basic idea of our approach is to merge small files into large ones and build indexes for them. Trie structure and hash index are well considered in the process of building index. Moreover, we follow the rule that small files are merged by type. Then the informations of the large file such as name, duplicate number and locations are send to NameNode and the data is put onto DataNode. The trie is stored in memory and the files used for map are stored on NameNode. Thus we can access a file by name rather than the path and this can be proved more effective.

The methods above can solve the problems that the memory in NameNode is not enough and can effectively manage the small files without file accessing by name.

4 Adaptive Storage Strategy

We add a preprocessing module into the traditional HDFS. The module exists in the interaction between client and NameNode. When a file is submitted, the preprocessing module first extracts the features to identify whether it is a small file or not and then renames the file for identification. Second, we merge the files into lager one by the features extracted above. The detailed processes are as follows:

4.1 Feature Extraction and Analysis

The aim of this phase is to extract features of files and make a preliminary analysis. The features including file type, file size and so forth can be stored in XML format. And thus can be transmitted between programs easily. In order to filter the big files that don't require any action, we need to determine a threshold. Here we choose the value in hdfs-site.xml as the threshold, which is 64 M by default. When the data flow through

the module, files beyond the threshold are thrown to the next step without treatment as usual, but those within need to be marked. Considering the speed of data processing, we add some tags to the filename. In consideration of the locality of reference principle that the same type of documents may be accessed continuously, we encode the tag according to the file type and then append it at the end of the filename.

For example, there is a 10 KB file with the name of book.txt. Suppose the code for txt is 0100, then the file will be rename as booktag0100.

4.2 File Merge Process

We create a queue for every file type to ensure that only files of the same type can be merged. In order to find the queue of its own according to the given type quickly, we use the hash function to map the code to the corresponding queue of which the time complexity is *O(1)*. When the file is matched and get into the queue, the system check the current size of data in this queue. Once the size of data reaches the threshold of a block, all the small files in this queue are merged into a large one and then the queue is cleared. Otherwise the file is put at the end of the queue and wait for the next one. The time complexity is *O(n)* because it scans all the files linearly. The algorithm of the file-merge process is as follows:

Algorithm 1. The file-merge algorithm

```
INPUT: small files; mergeThreshold
OUTPUT: merged files

1:   for each file in files do
2:      String name = file.getName();
3:      for (int i = 0; i < fileName.length(); i ++)      // Test whether this is a small file according to the tag;
4:         if ("tag" == fileName.substring(i, tag.length()+i))
5:            key = fileName.substring(tag.length()+i, fileName.length());
6:            Int type = Hash(decode(key));     // Get the file type by hash the decoded key
7:            Insert Queue[type] (file);
8:            if (Queue[type].size() > mergeThreshold)
9:               mergeFiles(file);
10:              Queue[type].empty();
11:           else
12:              Queue[type].size += file.size();
13:           end if
14:        end if
15:     end for
16:  end for
```

5 Name-Based Index

The index is widely used in the document retrieval. It is also perfectly suitable for files on HDFS. Here we analyze the composition of a file name. A name is a string made of many characters and usually not too long in general. In order to quickly match a name, we adopt the trie technology to build indexes for the irregular string.

5.1 Trie Building Process

Trie is an ordered tree data structure used to store an associative array where the keys are usually strings. In the binomial tree, the keys are directly stored in nodes but keys in trie are determined by the location of nodes in tree. All the children of a node have the same prefix, namely the string corresponding to the node. In general, not all of the nodes have corresponding values, only the leaf node and part of the internal nodes do. The reasons why we use the trie to build index are as follows:

1. Fast and efficient. If a path is from the root to a node, then the string corresponding to this node can be achieved by connecting all the characters along the path. So the time complexity of a string with n characters is $O(n)$ at worst.
2. Less storage space. Strings with the common prefix share the ancestor nodes.
3. Keys need not to be visual when stored in nodes, so it can effectively deal with string search problems.

Based on the advantages above, we take trie as the index structure and the main idea is to trade space for time. There are three kinds of trie: standard trie, compressed trie and the suffix trie. The compressed trie is the most suitable one because it compress single child node into edges and can further reduce the query time and storage space. We can turn a trie into a compressed one by the following rules:

- The node v_i is redundant, $i = 1, 2, \ldots, k-1$
- The node v_0 and v_k are not redundant
- Then we say that the list $(v_0, v_1)(v_1, v_2) \ldots (v_{k-1}, v_k)$ for $k \geq 2$ is redundant.
- Replace the list with the single edge (v_0, v_k)

Figure 3 shows the trie of words {hadoop, have, hive, mind, mine} and its compressed type.

By this way, the $O(n)$ storage space decreases to O(s), in which n is the total length of all strings in standard trie and s is the number of strings in compressed trie.

This method for index building consumes much memory and this problem is more apparent for large amounts of small files. So we put the trie on disk in DataNode and just load it into memory whenever there is a request which is the same as traditional HDFS. There is a file on NameNode acts as a directory. It stores the file name and its physical location on DataNodes. This file is stored in disk and the trie is built according to it. When users want to access a file on HDFS, the client sends request to the NameNode for the physical location. The NameNode looks up the file by name in trie and return the block information including the IP and port of the DataNode and specific

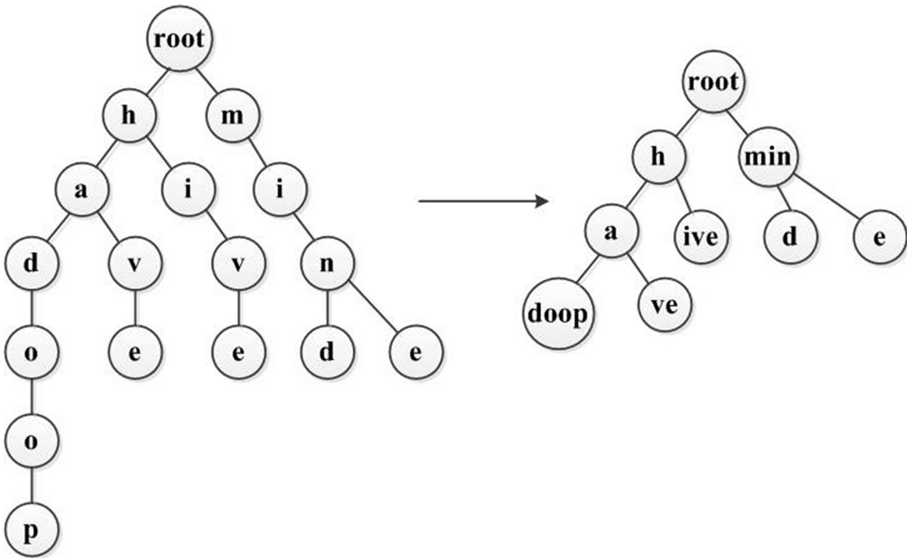


Fig. 3. Compressed trie

address. Then the client establishes connections with the clients and fetch the file according to the physical path.

5.2 Several Additional Supports

The name-based index can improve the efficiency of file access. The details are as follows. First, the client parses the extension, and makes index query in NameNode. Second, the NameNode returns a list of index blocks according to which the client access to the corresponding DataNode. Then the system searches the trie from the root node. The file is located only if the key is complete matching with the characters along one top-down path. What's more, the name-based index also supports some other operations such as store, delete and update.

The process of file storage is just the process of inserting a new string into the trie. The file is searched from the root along the edge by alphabetical order until the file name has been traversed. If the file name has not been found, the current node is extended and file is added to the corresponding block. If the size of file reaches the threshold, then the current data block is closed and another block will be created. When the NameNode is closed, the trie is written into the files on disk.

When a file is deleted from HDFS, the data block should be removed from the DataNode once the delete operations on NameNode is finished in theory. But the delete function only marks the data block to be deleted rather than remove it at once. What's more, the NameNode never contacts to the DataNodes and just listen. Only in the heartbeat response from DataNodes, the NameNode is able to delete the data block through commands. The index is marked as invalid and then the files under the index

are also marked as invalid. When the files marked reach a specific threshold, the indexes and files are cleared up with the remains to be merged.

The update operation is a combination of writing and deleting files. The operations in the beginning are the same as deleting files which is finding the file locations on the basis of index. The file system returns error message. Or else it appends new data to the current file and marks the old files invalid. At the same time the index is updated and the old ones will be deleted at the right time.

6 Performance Evaluation

Our test platform is built on a cluster with three nodes. Each node has an Intel 8 CPU of 1.6 GHz, 32 GB memory and 1T SATA disk. The operating system is Ubuntu14.04. The Hadoop version is 2.6.0 with java of version 1.8.0. In the three nodes, one is master acting as the NameNode and SecondaryNameNode. The other two nodes are both slaves act as DataNode. In order to save the time of data transmission between nodes, the value of replication in hdfs-site.xml is set to 1 which means all the files need only to upload once.

6.1 Adaptive Storage Strategy Performance Evaluation

The data set is generated by the “dd” command in Ubuntu and then divided into small size. It is made up of 9536 files with the total size of 3.2 G. It has the same performance when the size is up to 6.4 G or even more. Most of the files are below 1 MB and those smaller than 100 K account for 89.25 %. We simulate the test by compressing files to nearly 64 MB manually and the performance is equal. Figure 4 shows the distribution of file size.

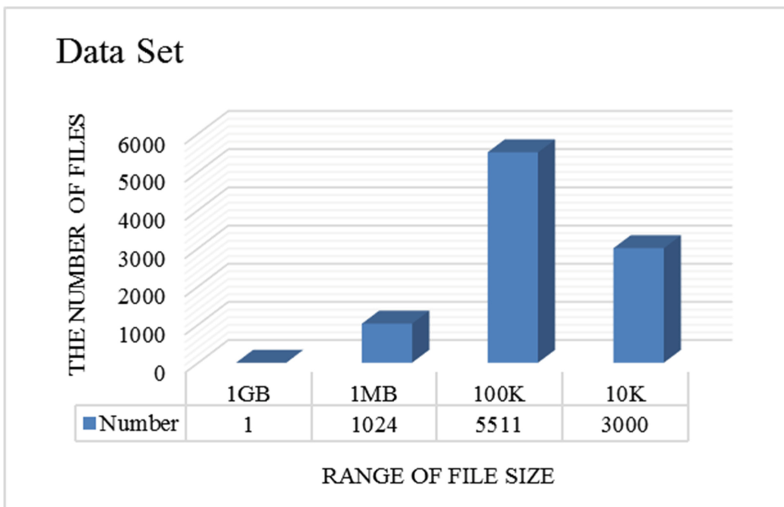


Fig. 4. Distribution of data

Figure 5 shows the time it consumes to upload the files on HDFS. It takes 1221 s to upload files on traditional HDFS and 750 s with the approach of this paper.

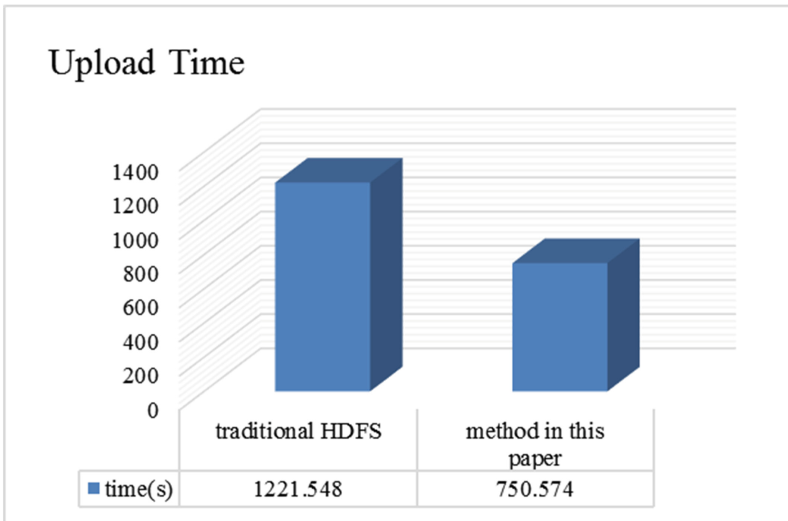


Fig. 5. The upload-time comparison

In terms of DataNode storage, the traditional HDFS applies for a data block for each small file. So the amounts of block is very large. But the number of blocks by method in this paper is much smaller. We can also infer that there is a decrease in the memory used in NameNode. The details are shown in Fig. 6.

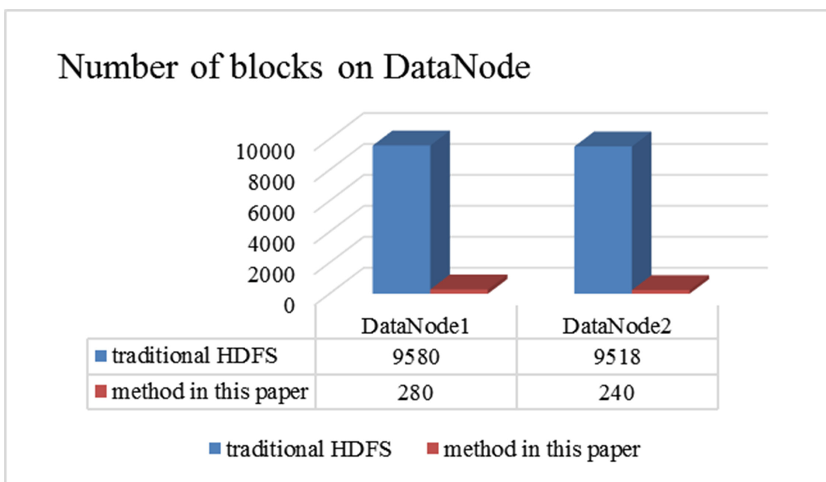
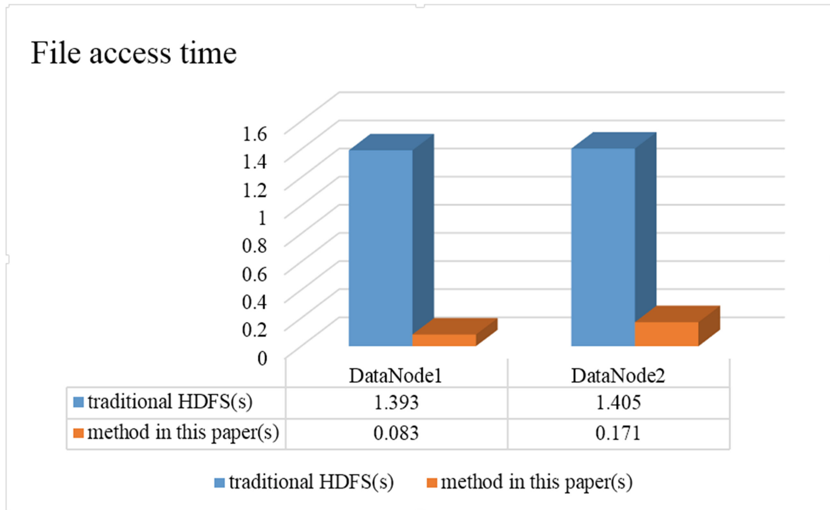


Fig. 6. The comparison of number of blocks. (Color figure online)

6.2 Name-Based Index Performance Evaluation

Figure 7 shows the time of reading file consumption by using two methods. Accessing times of the two DataNodes are 1.393 s and 1.405 s for traditional HDFS. While using the method in this paper, the times are 0.083 s and 0.171 s respectively.



	DataNode1	DataNode2
traditional HDFS(s)	1.393	1.405
method in this paper(s)	0.083	0.171

Fig. 7. The file access time comparison. (Color figure online)

7 Related Work

In the area of social computing, with the development of models and countermeasures for rumor spreading in Online Social Networks [7], and approximation algorithms for the routing problem [8–10], it is highly required that the distributed file system in cloud computing has high file-access efficiency.

HDFS can not well satisfies the current situation where large amounts of small files need to be managed. There are several methods to solve these problems at present that can be classified into three groups.

The first one is Hadoop Archives (HAR files) [11] which is file archiving tools that comes with Hadoop itself. Its emergence is to solve the memory consumption problem. HAR plays a role by building a hierarchical file system on HDFS. A HAR file can be created by the Hadoop command which runs a MapReduce task that packs small files into HAR format. But the efficiency of reading files can not be as high as reading files directly from HDFS. Moreover, the efficiency may be slightly lower to a certain extent, because each file access process contains reading operations of two layers of index and data. Although HAR files can be used as the input of MapReduce job, there is no special way for map to regard the file package as a HDFS file to handle. When users want to delete or update files, a new archive file should be created or else it will throw exceptions.

The second one is Sequence Files [12]. It is the common response to “the small file problem”. This method sets filename as key and file contents as value so that many small files are organized and stored into sequence files. MapReduce can break them into chunks because the sequence files are divisible. It provides a good support for local data operation. However, the lack of maps between files makes the retrieval process of small files inefficient.

The third group contains many other research focusing on reducing the latency of small-file access on HDFS. The WebGIS system [13] is the most classic one which merges the small files on the basis of time and place. It also builds indexes for access and achieves good results. Shaikh [14] proposed a novel adaptive migration strategy that is incorporated into metadata-based optimization to alleviate these side effects by migrating file dynamically. These file migration can reduce the server load. Schemes of latency hiding and migration consistency are also introduced to reduce overhead induced by small file optimization by 20 %. Carns et al. [15] proposed five techniques which are all implemented in a single parallel file system and then systematically assessed on two test platforms. It achieves an improvement of 700 %. Hendricks et al. [16] proposed the protocol of the Chirp file system for grid computing to improve small file performance.

All the methods above have similar thoughts can be summarized into two points. The first is that small files are merged to reduce the memory usage in NameNode and the second is Performance improved by building index for file access.

8 Conclusion

Hadoop is a cluster made up of HDFS, MapReduce and Yarn which is now widely used in handling big data. But its short slab is low efficiency on dealing with large amounts of small files and high latency of data access. The NameNode is responsible for managing metadata for every block including the property and information about storage which will consume much memory. Large amounts of map tasks jam together. This paper studies the following contents to solve these problems. First we create new processing module to extract features and then merge those small files into large ones. Then we use file name to build index. The novelty of this approach is to build index by trie structure. It has a faster construction speed and files can be accessed by name easily. We achieve good result in file access operations.

For the future work, feature space are expected to build so that the compression process can be achieved according to the distance. Then the prefetch of files can be considered in which a buffer is used to further improve the access efficiency.

Acknowledgement. This paper was partially supported by National Sci-Tech Support Plan 2015BAH10F01 and NSFC grant U1509216,61472099,61133002 and the Scientific Research Foundation for the Returned Overseas Chinese Scholars of Heilongjiang Provience LC2016026.

References

1. Dong, X.: Hadoop Internals: In-depth Study of MapReduce. China Machine Press, China (2013)
2. Cai, B., Chen, X.: Hadoop Internals: In-depth Study of Common and HDFS, pp. 216–217. China Machine Press, Beijing (2013)
3. Murthy, A.C., Vavilapalli, V.K., Eadline, D., Niemiec, J., Markham, J.: Apache Hadoop YARN: Moving Beyond MapReduce and Batch Processing with Apache Hadoop 2. Pearson Education (2013)
4. Massive image storage and processing architecture in TaoBao. http://www.lxway.com/655880062.htm
5. http://baike.baidu.com/link?url=Mp9WPuBy-D0tKQeAIG-mJn9hk3UwS0fhkaQbm3xlcwxitswRHqVDjV5AfqmmJDK7NbvCzb3klb8ybFH9vkiGx_
6. Hadoop Distributed File System. http://hadoop.apache.org/docs/current/hadoop-project-dist/hadoop-hdfs/HdfsUserGuide.html
7. He, Z., Cai, Z., Wang, X.: Modeling propagation dynamics and developing optimized countermeasures for rumor spreading in online social networks. In: 2015 IEEE 35th International Conference on Distributed Computing Systems (ICDCS), pp. 205–214. IEEE (2015)
8. He, Z., Cai, Z., Cheng, S., et al.: Approximate aggregation for tracking quantiles and range countings in wireless sensor networks. Theor. Comput. Sci. **607**, 381–390 (2015)
9. Cai, Z., Lin, G., Xue, G.: Improved approximation algorithms for the capacitated multicast routing problem. In: Wang, L. (ed.) COCOON 2005. LNCS, vol. 3595, pp. 136–145. Springer, Heidelberg (2005)
10. Cai, Z., Goebel, R., Lin, G.: Size-constrained tree partitioning: a story on approximation algorithm design for the multicast *k*-Tree routing problem. In: Du, D.-Z., Hu, X., Pardalos, P.M. (eds.) COCOA 2009. LNCS, vol. 5573, pp. 363–374. Springer, Heidelberg (2009)
11. Hadoop archives. http://hadoop.apache.org/docs/current/hadoop-archives/HadoopArchives.html
12. Sequence File. http://hadoop.apache.org/docs/current/api/org/apache/hadoop/io/SequenceFile.html
13. Liu, X., Han, J., Zhong, Y., et al.: 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, CLUSTER 2009, pp. 1–8. IEEE (2009)
14. Shaikh, F., Chainani, M.: A case for small file packing in parallel virtual file system. http://www.andrew.emu.edu/user/mchainan/FinalPaper.pdf. Accessed 07 July 2007
15. Carns, P., Lang, S., Ross, R., et al.: Small-file access in parallel file systems. In: IEEE International Symposium on Parallel & Distributed Processing, IPDPS 2009, pp. 1–11. IEEE (2009)
16. Hendricks, J., Sambasivan, R.R., Sinnamohideen, S., et al.: Improving small file performance in object-based storage. Carnegie-Mellon Univ. Parallel Data Laboratory, Pittsburgh (2006)