International Journal of Database Theory and Application
Vol.8, No.5 (2015), pp.241-254
http://dx.doi.org/10.14257/ijdta.2015.8.5.21

Optimization Scheme for Small Files Storage Based on Hadoop Distributed File System

Yingchi Mao[1, 2], Bicong Jia[1], Wei Min[1] and Jiulong Wang[1]
[1]*College of Computer and Information Engineering, Hohai University, Nanjing, 211100, China*
[2]*Huaian Research Institute of Hohai University, Huaian, China*
{maoyingchi, hhuwangjl}@gmail.com, 13291275972@163.com, 1578922962@qq.com

Abstract

Hadoop Distributed File System (HDFS) becomes a representative cloud platform, benefiting from its reliable, scalable and low-cost storage capability. However, HDFS does not present good storage and access performance when processing a huge number of small files, because massive small files bring heavy burden on NameNode of HDFS. Meanwhile, HDFS does not provide any optimization solution for storing and accessing small files, as well as no prefetching mechanism to reduce I/O operations. This paper proposes an optimized scheme, Structured Index File Merging-SIFM, using two level file indexes, the structured metadata storage, and prefetching and caching strategy, to reduce the I/O operations and improve the access efficiency. Extensive experiments demonstrate that the proposed SIFM can effectively achieve better performance in the terms of the storing and accessing for a large number of small files on HDFS, compared with native HDFS and HAR.

Keywords: *Small file storage, HDFS, Structured index files storage, Chord, prefetching*

1. Introduction

With the rapid development of Internet, the amount of data growing exponentially, there have been appeared many large server architecture such as data centers and cloud computing. In the field of large data storing and processing, the Google's GFS provide an effective way to handling large files [1]. Hadoop is composed of one NameNode and some DataNodes as architecture components. NameNode stores all the metadata in main memory. A large number of small files have an important impact on the metadata performance of HDFS and become the bottleneck for handling metadata requests of massive small files.

Hadoop Distributed File System (HDFS) is designed for storing the large files, and therefore it suffers performance efficiency in dealing with small files [2]. In fact, there are many systems storing huge amounts of small files in the different application areas, such as energy, climatology, biology, social networks, e-Business, and e-Learning [3-4]. For example, over 13 million files were stored in the computing center for the energy research. 99% and 43% of files in that computer center were less than 64MB and 64KB, respectively [5-7]. Therefore, HDFS faces a great challenge when storing and accessing a large number of small files. The reason is that the huge number of files occupies the memory of NameNode, and no optimization scheme is provided to improve the access efficiency.

To improve the access performance on HDFS, the efficiency problem of reading and writing a large number of small files is analyzed. Based on the analysis of small files, an optimized scheme, Structured Index File Merging (SIFM), is proposed for HDFS to reduce the memory consumption of NameNode and to improve the reading efficiency of small files. In SIFM, the correlations between small files and directory structure of data are comprehensively considered to assist the small files to be merged into large files and generate the index files. Distributed storage architecture is used in index files management. In addition, SIFM adopts the data prefetching and caching strategies to improve the access performance.

The main contributions of this paper can be summarized as follows:

(1) The efficiency problem of storage and access a large number of small files on HDFS is analyzed;

(2) An optimized scheme is proposed for HDFS to reduce the memory consumption of NameNode and to improve the access performance of huge number of small files;

(3) Extensive experiments verify the storage and access efficiency by comparing with native HDFS and HAR.

The rest of this paper is organized as follows. Section 2 discusses the related work. Then, Section 3 addresses the analysis of small file access on HDFS. Section 4 proposes the optimized scheme for small files storage on HDFS, followed by its efficiency analysis in Section 5. The experiments evaluation is presented in Section 6. Finally, the conclusions are drawn in Section 7.

2. Related Work

HDFS is a single master and multiple slave frameworks. There is only one NameNode as master, multiple DataNode as slaves. When storing large amount of small files, NameNode will accept request for storage addresses and distributed storage block frequently. This makes the single NameNode becoming the bottleneck [7]. Because there is no optimization scheme for read/write small files in HDFS, HDFS presents poor read/write performance when accessing a large number of small files directly [1]. Furthermore, HDFS does not consider the optimization on the native storage resource, which reduce the efficiency in the local disk access [8].

In recent years, research on small file optimization for HDFS has attracted significant attention. HAR, SequenceFile, and MapFile are typical general solutions to small file optimization.

HAdoop Archive (HAR) packs a number of small files into large HDFS blocks so that the original files can be accessed in parallel transparently and efficiently without expanding the files. It contains metadata files and data files [6]. The file data is stored in multiple part files, which are indexed for keeping the original separation of data intact. The metadata files can record the original directory information and the file states. HAR can reduce the memory consumption of NameNode, but it has some problems. Firstly, it can bring extra burden on disk space when creating an archive. Secondly, there is no mechanism to improve the access efficiency.

A SequenceFile is a flat file consisting of binary key-value pairs. It uses filename as the key and file contents as the value. You can write a program to put small files into one single SequenceFile, and process the small files using MapReduce operating on the SequenceFile [2]. However, there are problems of SequenceFile. First, only APPEND operation is supported, and no update/delete operation is used for a specific key. Second, when querying a specific key in one SequenceFile, the whole file has to be viewed, which results in the poor access performance.

A MapFile is another kind of SequenceFile, which is one sorted file with an index to lookup operation by key. It includes two files, a data file and a smaller index file. All of the sorted key-value pairs are stored in the data file. The key-location information are stored in the index file [9]. The index file is read entirely into memory, so the index file should be kept itself small. Different from the SequenceFile, MapFile does not search the whole data file when looking up a specific key. Unfortunately, MapFile only provides the append operation.

On the other hand, some special solutions were proposed to deal with special type files. For example, multiple pages files are grouped into a large file, and an index file for each book is created for digital libraries [4]. Liu *et al.*. merged small files into a large one and built a hash index for each small file [1]. The large files store small data of GIS on HDFS.

Moreover, in order to handle a large number of small files in the Cloud computing platform, some task scheduling algorithms were proposed. Huang *et al.*. adopted GA model to dispatch a large number of computing tasks [17]. Lu *et al.*. presented workflow scheduling algorithm considering different QoS constraints [18]. In addition, some research works focused on the security problems for the file access in the Cloud computing platform [16].

All of the above solutions, HAR, SequenceFile, and MapFile have the same limitation is that file correlations are not considered when storing files. Moreover, there is no optimization scheme provided to improve the access efficiency.

In this paper, file correlations are considered when storing and access files, structured architecture for storing the metadata files is used to reduce the memory of NameNode, and prefetching and caching technology is provided to improve access performance on HDFS.

3. Analysis of Small File Problem on HDFS

This Section discusses the architecture and access mechanism of HDFS and the impact of small files on HDFS.

3.1. Architecture and Access Mechanism of HDFS

HDFS has one single NameNode and a number of DataNodes, and uses the master/slave architecture, as shown Figure 1. NameNode maintains the metadata of the entire file system, including the file and block namespace, the mapping between files and blocks, and the locations of each block’s replicas. All metadata is stored in the memory of the NameNode. DataNodes provide block storage, serve I/O requests from clients, and perform block operations [7].

The HDFS includes three kinds of file operations: write, read and delete. When a client needs to store data, it sends a write request to the NameNode. The NameNode will generate a block ID and find three DataNodes to store the data. Client sends the data to these DataNodes according to data flow and notify the NameNode to store the metadata if successfully write. When reading data, client sends a request to the NameNode. NameNode will find the corresponding file in the directory tree, and locate the blocks. From the Figure 1, we can get a pictorial view of HDFS architecture, as well as the read and write operations.

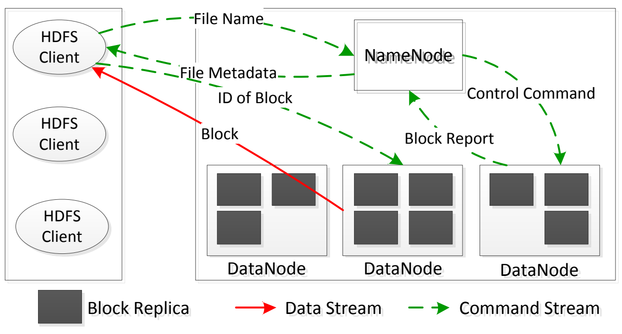


Figure 1. The Architecture of HDFS [15]

In HDFS, files are divided into several fixed-sized blocks. The default size of each block is 64MB [10]. Each block has three replicas in the separated machines for fault tolerance.

3.2. Impact on HDFS

Because HDFS block size by default is 64MB, any file smaller than this is considered as a small file. When small files are stored on HDFS, disk utilization is not a bottleneck. It is reported that a small file stored on HDFS does not occupy any more disk space than is required to store its contents [2].

There are two main reasons to result in the poor performance of small files. Firstly, every file, directory and block in HDFS is represented as an object in the memory of NameNode. The metadata of a file and a block with three replicas occupy 250 and 368 bytes of memory, respectively. Thus, 10 million files occupy about 3 GB of memory, if each file using a block. Scaling up much beyond this level is a problem with current hardware [11]. Therefore, large amount of memory of NameNode is consumed by the metadata of a large number of small files. Secondly, HDFS is not geared up to efficiently accessing small files. It is primarily designed for streaming access of large files. Obviously, reading small files normally require a large number of seek operations from DataNode to DataNode to search amd retrieve a requested file block. All of seek operations is an inefficient data access pattern. The larger the number of small files, the longer it takes. Moreover, HDFS currently does not provide prefetching and caching mechanism to reduce I/O latency.

4. Optimization Scheme -- SIFM

In this paper, we proposed an optimization scheme, Structured Index File Merging – SIFM, to improve the storage and access efficiency for small files in HDFS. The core ideas of SIFM include: (1) File correlations are considered when merging files, which reduces the seek time and delay in reading files. (2) Structured distributed architecture for storing the metadata files is applied to reduce the seek operations of requested files. (3) Considering the access locality in the inter-block on DataNodes, prefetching and caching strategy is used to reduce the access time when reading huge numbers of small files.

According to the file merging strategy, a file is filtered to the file merged module. If the file is a small file, it is uploaded to HDFS, and is merged into a big file. Meanwhile, the metadata file is created and the structured merging index file is builit for the merged file. Then the merged file is loaded to the DataNode. For metadata, the structured P2P architecture, Chord, is adopted to store and manage the metadata in NameNode. In addition, the prefetching and caching strategy is used to cache the metadata and index file, and then obtain the small files. When the structured index file is read, based on the offset and length, the requested small file can be seek in HDFS block and return to the client. Using these strategies, SIFM can greatly reduce the communication cost and improve the I/O performance when reading files. The procedure of SIFM scheme is illustrated in Figure 2.

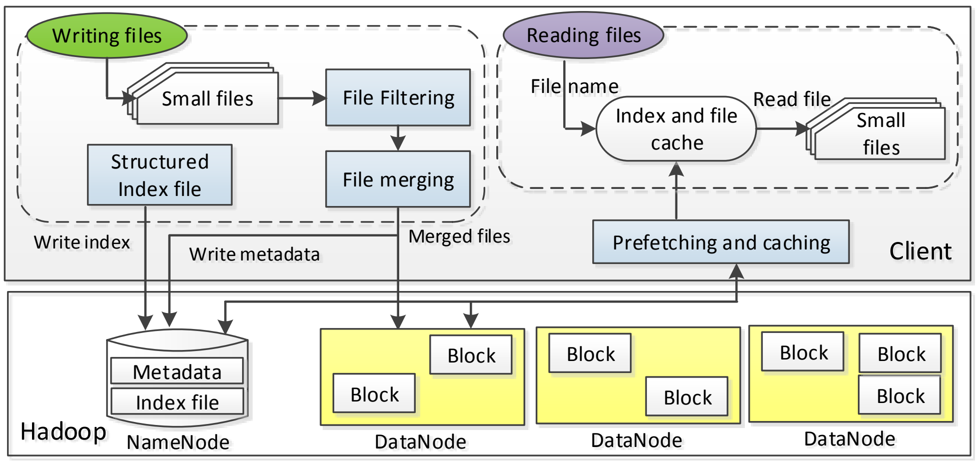


Figure 2. The Procedure of SIFM Scheme

4.1. File Merging Strategy

File merging strategy includes three parts: file filtering criteria, structured index file creation, and file merging operation.

4.1.1. File Filtering Criteria

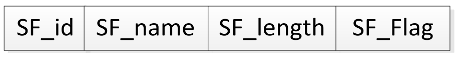
To effectively deal with the small files in HDFS, the first important issue is to identify the cut-off point between large and small files.

Many applications consist of a large number of small files. For example, in the field of climatology some applications consist of 450,000 files with an average of 61MB [12]. In biology, the human genome generates up to 30 million files averaging 190KB [13]. Sloan Digital Sky Survey hosted 20 million images with average size of less than 1MB [14]. Although Dong *et al.* discussed the cut-off point between large and small files, they just presented the experimental analytical results. According to the fact from the real applications, we treat the size of files smaller than 1 MB as small files. When storing a small file, according to the small file filtering criteria, the client checks whether the size of file is larger than 1MB. If it is a large file, it will be stored using the native HDFS method. If it is a small file, the proposed file merging operation will be executed.

4.1.2. Structured Index File Creation

NameNode only maintains the metadata of the merged files and the relevant index files are created for each original small file to indicate its offset and length in a merged file. In SIFM, a structured index file is built for each merged file and is loaded in the memory of NameNode. A structured index file is composed of two index sets, small file index and merged file index. Since the memory consumed by each index file is much smaller than that of the metadata of a file, the structured index file can still reduce the memory occupation of NameNode.

NameNode only maintains the metadata of the merged files and the relevant index files are created for each original small file to indicate its offset and length in a merged file. In SIFM, a structured index file is built for each merged file and is loaded in the memory of NameNode. A structured index file is composed of two index sets, small file index and merged file index. Since the memory consumed by each index file is much smaller than that of the metadata of a file, the structured index file can still reduce the memory occupation of NameNode.

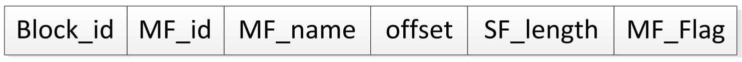


SF_id	SF_name	SF_length	SF_Flag

Figure 3. The Index Structure of a Small File

(1) A small file index is used to indicate the ID, name, and length of a small file. SF_Flag indicates its validation. Since the most frequent operations on small files index is queries by file ids, indexes are sorted by file ID. The structure of small file index is shown in Figure 3.

(2) Merged file index is built for each merge file, which indicates the offset and length for each original small file in it. Beside them, because a merged file may occupy multiple blocks, the merged file index indicates the block ID where the merged file is stored, the ID of the merged file, the name of the merged file. Similarly, MF_Flag indicates the validation of the merged file. The structure of a merged file index is shown in Figure 4. Using the merged file index can be convenient to analyze the location of a small file in the read process.



Block_id	MF_id	MF_name	offset	SF_length	MF_Flag

Figure 4. The Index Structure of s Merged File

4.1.3. File Merging Operation

File merging operations are carried out in HDFS clients, which merge related small files into a large merged file. NameNode only maintains the metadata of merged files and does not store the original small files, thus file merging can reduce the number of files that need to be managed by NameNode. When writing a small file, if it is a small file, the proposed file merging operation will be carried out. Otherwise, it will use the native HDFS method. The details of file merging operations are as follows:

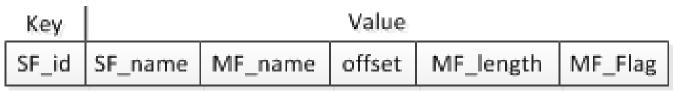
Step 1: Preparation. The number of small files is computed and the size of the small file is also calculated. The small index files will be created in Step 2.

Step 2: Creation the structured index file. If the size of current small file is less than the available size of one HDFS block, the small file index is created and the SF_Flag is set to TRUE. The offset and length of the small file are calculated, and the merged file index is updated. If current HDFS block cannot provide enough space to store the small file, the remaining small space in HDFS block will be abandoned. The small file will be written from the first place to the next new HDFS block. Meanwhile, the small file index is created, and the merged file index is created based on the offset and length of the small file.

Step 3: Small file merging. According to the offset and length of each file in the merged file, files are merged into the merged big file in turn.

4.2. Metadata Files Storage

In HDFS, metadata of small files stores the mapping information from the small file to the merged file. To reduce the memory consumption of NameNode, and improve the access performance on HDFS, it needs to optimize the metadata management. In SIFM, the structured P2P architecture, Chord, is adopted to store and manage the metadata file.



Key	Value				
SF_id	SF_name	MF_name	offset	MF_length	MF_Flag

Figure 5. The Metadata Structure of a Small File

The metadata structure of a small file is illustrated in Figure 5. Metadata of small files are stored in NameNode with the key-value pairs. For the metadata, SF_id is the only one identifier based on the filename and directory, which is as key stored in NameNode. SF_name and MF_name denote the name of original small file and the merged file, respectively. The offset represents the offset of small file in the HDFS block. MF_length indicates the length of the merged file. MF_Flag indicates the validation of the merged file. The value of SF_name, MF_name, offset, MF_length, MF_Flag are created via the SHA-1 algorithm. Based on the created hash value, the corresponding metadata can be stored in the node of cluster. Due to the small size of metadata, there is no much burden about memory consumption on NameNode. When reading one small file, utilizing the Chord routing mechanism, NameNode can quickly locate the metadata in NameNode, and seek the corresponding small file based on the mapping information in the metadata.

4.3. Prefetching and Caching Files

In general, Prefetching and caching schemes are widely adopted for improving the access efficiency [1]. Prefetching can avoid disk I/O cost and reduce the response time by considering the access locality and fetching data into cache before they are requested. In SIFM, three prefetching strategies are used to improve the access performance: metadata caching, index file fetching, and the merged data file fetching. These strategies are similar to the three-level prefetching and caching strategy in [15].

Firstly, when a client requests a small file, the small file get the metadata of the merged big file from NameNode via the metadata mapping file. If the metadata of the big file is in the local memory, the client can directly access it. Thus it can reduce the I/O cost between NameNode and the original small file. Secondly, based on the obtained metadata, the client can know the connected block and access the requested file. If the index file has been buffered from DataNode, accessing the small file belonging to the same big file can reduce the I/O operations. Thirdly, when the requested small file has been returned to the client, the related files can be cached based on their locality in a merged big file. Therefore, exploiting access locality and caching the correlated files in a merged file from DataNode can reduce the computational cost and keep high access efficiency.

5. Efficiency Analysis

To illustrate the read/write efficiency of the proposed optimization scheme, SIFM, this Section will present the efficiency analysis.

Suppose that there are N small files with lengths, denoted as L_1, L_2, …, and L_N. The N small files are merged into K big files, M_1, M_2, …, and M_K, whose lengths are denoted as L_{M1}, L_{M2}, …, and L_{MK}, respectively.

5.1. Writing Efficiency Analysis

As described in Section 3.2, the metadata of a file and a block with three replicas occupy 250 and 368 bytes of memory, respectively. If there is no data file, NameNode consumes the number of memory bytes denotes as θ. The block mapping of a block consumes the number of memory bytes denotes as η. The size of a block in HDFS is denoted as B_{HDFS}.

Because the index file is stored in the memory of NameNode, the length of the index file is denoted as δ. Then, the number of memory bytes consumed by NameNode is derived as

$$M_{namenode} = 250 \times K + (368 + \eta) \times \sum_{i=1}^{K} \left\lceil \frac{L_{Mi}}{B_{HDFS}} \right\rceil + N \times \delta + \theta \qquad (1)$$

where $\sum_{i=1}^{K}\left\lceil \frac{L_{Mi}}{B_{HDFS}} \right\rceil$ denotes the number of blocks in HDFS.

According to Eq. (1), in SIFM, the memory consumption of NameNode is related to the number of small files. The smaller file, the more occupied memory of NameNode. Moreover, the number of blocks, $\sum_{i=1}^{K}\left\lceil \frac{L_{Mi}}{B_{HDFS}} \right\rceil$ is much smaller than $\sum_{i=1}^{N}\left\lceil \frac{L_{i}}{B_{HDFS}} \right\rceil$, therefore, SIFM can relieve the memory consumption of NameNode and reduce the number of files and blocks.

5.2. Reading Efficiency Analysis

When reading a file from HDFS, the accessing time includes the following parts. (1) A client sends command to NameNode. The time cost is denoted as T_{send}; (2) NameNode seeks the metadata of the requested file. The time cost is denoted as $T_{metadata}$; (3) The time cost for the metadata returned to the client is denoted as T_{return}; (4) The client sends a read command to the corresponding DataNode. The time is denoted as T_{read}; (5) The DataNode obtains the requested file from disk. The time is denoted as T_{data}; (6) The file is returned to the client via the network. The time is denoted as $T_{network}$; (7) The index file is read from NameNode. The time is denoted as T_{index}. For all of the consumed time, T_{send}, T_{return}, and $T_{metadata}$ are considered as constants. $T_{network}$ is relevant with the size of file.

In HDFS, the total accessing time for the requested file is derived as:

$$T_{total\_DHFS} = N(T_{send} + T_{return} + T_{read}) + \sum_{i=1}^{N}((T_{metadata\_i} + T_{index\_i} + T_{data\_i} + T_{network\_i}) \quad (2)$$

In SIFM, utilizing the metadata and index caching approach, the metadata and index file of the merged big file need to be located from the memory of NameNode only when the first requested small file is read. Thus, the metadata and index file can be directly read from cache without considering the searching time for metadata and the index file. In addition, the data file prefetching method is also applied in SIFM. Suppose that parts of the original small files p are directly get from cache, therefore, the total accessing time for the requested file in SIFM is derived as:

$$T_{total\_SIFM} = K(T_{send} + T_{return}) + (N-p)T_{read} + \sum_{j=1}^{K}(T_{metadata\_j} + T_{index\_j}) + \sum_{j=1}^{N-p}(T_{data\_j} + T_{network\_j}) \quad (3)$$

Obviously, the accessing performance is related with the accuracy of file locality predictions. High concurrency and low accuracy will deteriorate the access efficiency. High accuracy can greatly improve the access performance.

6. Experiment Evaluation

In this Section, we evaluate the reading and writing efficiencies for the small files with the proposed optimization scheme in this paper. We also compare it with the native

6.1. Experimental Settings

The experimental platform is established on a cluster with five nodes. One node, which is HP server with 4 Intel Xeon CPU (2.6 GHz), 8GB memory and 800 GB memory, acts as NameNode. The other four nodes, which are DELL PC with 2 Intel CPU (2.6 GHz), 2GB memory and 500 GB disk, act as DataNodes. All of the nodes are interconnected with 100Mbps Ethernet network.

For each node, the operation system, Fedora 10, Hadoop 0.20.2 and Java 1.7.0 are installed. The number of replicas is set to 3 and HDFS block size is 64MB by default, respectively. The configurations of experiment are shown in Table 1.

Table 1. The Configurations of Experiment on HDFS

NodeType	CPU	RAM	Disk
NameNode	4 Intel Xeon CPU (2.6 GHz)	8GB	800 GB
DataNode01	2 Intel CPU (2.6 GHz)	2GB	500 GB
DataNode02	2 Intel CPU (2.6 GHz)	2GB	500 GB
DataNode03	2 Intel CPU (2.6 GHz)	2GB	500 GB
DataNode04	2 Intel CPU (2.6 GHz)	2GB	500 GB

6.2. Experimental Methodology

Experiments include two aspects: writing efficiency and reading efficiency for a large number of small files.

For HAR, all files of a merged file are stored as an HAR file. Because creating a merged HAR file can result in a copy of each original HAR file, the original files are deleted after storing. To compare the writing and reading efficiency of the optimized scheme, native HDFS and HAR, we evaluate them in terms of the memory overhead and time-cost for writing and reading small files.

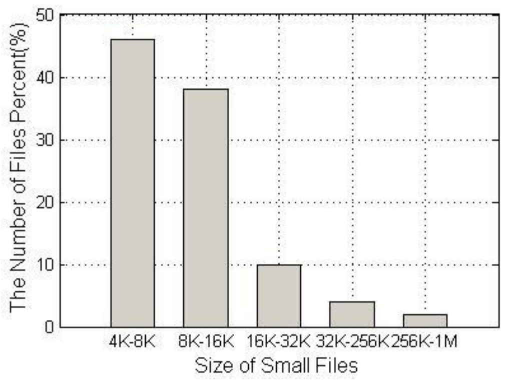


Figure 6. The Distribution of File Size

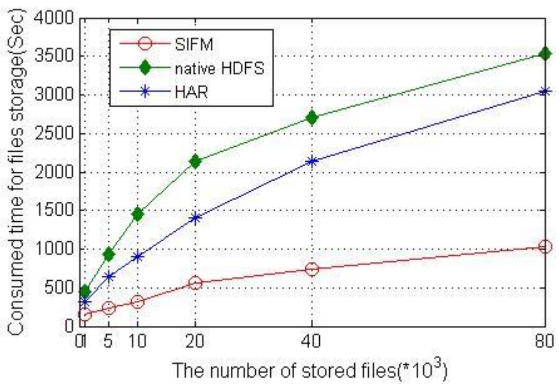


Figure 7. Comparison under Time for File Storage

To evaluate the writing and reading efficiency of the proposed scheme, 80,000 files are selected as the dataset. The distribution of file sizes is shown in Figure 6. The file sizes in the dataset range from 4kB to 1MB, and files whose size is less than 32kB account for 95% of the total files. All files of dataset are small files. In order to accurately evaluate the writing and reading performance for a large number of small files, all of the statistics are averaged over 20 runs for high confidence.

6.3. Experimental Results

(1) Writing efficiency

In the storage/writing operation, we evaluate the storage time and memories usage while uploading 1,000, 5,000, 10,000, 20,000, 40,000, and 80,000 small files, respectively, to an empty HDFS.

Figure 7 shows that the time consumption for storage increases as the number of files increase. For native HDFS, HAR, and SIFM, when writing 1,000 small files, the storage time are 450s, 313s, and 151s, respectively. For 80,000 files, the time is 3530s, 3051s, and 1034s, respectively. SIFM greatly outperforms native HDFS and HAR. The reason is that SIFM adopts the merging scheme to reduce the I/O operation between NameNode and DataNodes when writing a large number of small files.

(2) Memory usage

For native HDFS, HAR and SIFM, the occupied memories of NameNode are measured when storing a lot of small files. The results are shown in Figure 8. As expected, due to their file archiving and merging facilities, SIFM can consume less memories of NameNode than native HDFS and HAR. When storing 80,000 small files, the storage efficiency increases up to 42% and 26%, respectively. Because SIFM can effectively reduce the number of stored files by merging scheme as well as optimizing the structured index file, SIFM can consume the less memories usage.

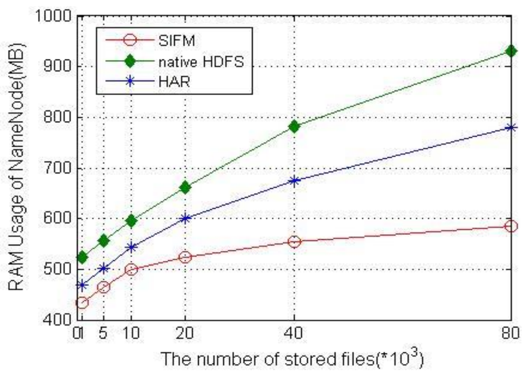


Figure 8. Memory Usage of NameNode

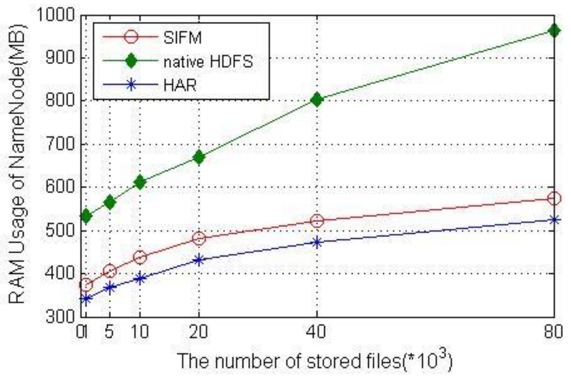


Figure 9. Memory Usage of DataNode

In addition, SIFM also has advantage on the memories usage of DataNode, as illustrated in Figure 9. Because of its structured index file strategy, SIFM consumes little more memory compared with HAR, and much less memory than native HDFS. Although SIFM generates the extra overhead on the DataNode, the whole performance of SIFM is better than that of HAR.

(3) File reading efficiency

To evaluate the small file reading performance, different number of small files is randomly selected. The file types contain random files and sequence files. So, randomly download 100, 500. 1000, 2000, 5000, and 8000 files from all of the small files, respectively, we evaluate the total download time for these selected small files.

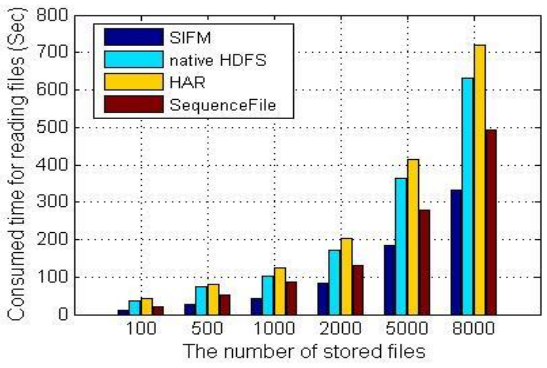


Figure 10. Reading Time for Random Files

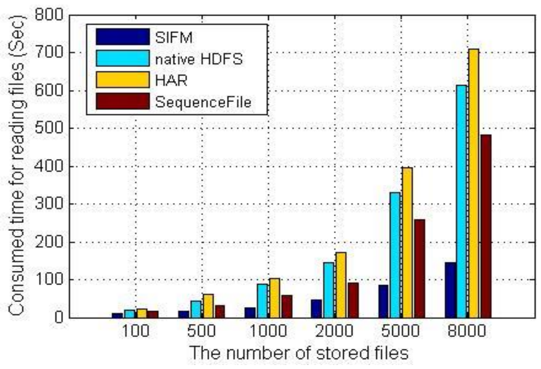


Figure 11. Access Time for Sequence Files

Figure 10 and Figure 11 illustrate the reading time by applying SIFM, native HDFS, HAR and SequenceFile for a large number of random files and sequence files, respectively. From the results in Figure 10 and 11, when reading 8000 random files, the SIFM scheme can reduce the reading time by 47%, 52%, and 33%, compared with native HDFS, HAR, and SequenceFile, respectively. Moreover, with the increase in the number of files, varying from 100 to 8000, the reading time of all of the schemes also increases. The reason is that the increase of the number of files can increase the number of seek operations on DataNodes, which can result in the higher reading latency. In addition, reading random files and sequence files, the access time of the proposed SIFM scheme has some differences. For SIFM, reading random files has better access efficiency than reading sequence files. Reading sequence files can reduce the number of I/O operations between NameNode and DataNodes, which leads to the improvement of reading performance.

(4) Concurrent reading efficiency

In order to evaluate the concurrent efficiency, we adopt the multithreads to simulate the concurrent clients in the experiments. During the experiments, we simulate 1, 2, 4 and 8 clients to send requests to access 8000 sequence small files with different schemes. The experiment results are shown in Figure 12. When client's number is 1, the reading time of native HDFS is 146s, SIFM can reduces the reading time by up to 76%, which is benefited from the prefetching and caching strategy. On the contrary, the reading time of HAR is 710s, which is about 116% compared with that of native HDFS. Similarity, when the number of concurrent client is 2, 4, and 8, respectively, the reductions of the reading time with SIFM are 80%, 78%, and 79%. On the other hand, for HAR, the reading time increase 18%, 20%, and 19%, respectively.

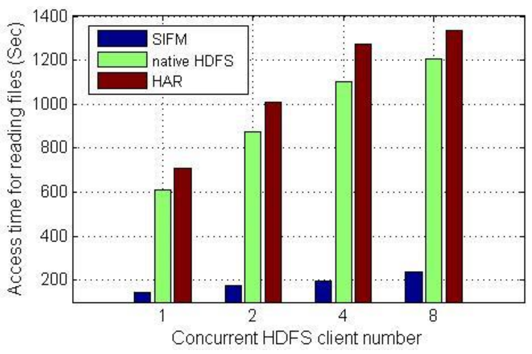


Figure 12. Access Time of different Schemes when Reading 8000 Sequence Small Files

7. Conclusion

HDFS is designed to store large files and suffers performance penalty while writing and reading large amount of small files. In this paper, the optimized scheme, SIFM is proposed to effectively improve the storage performance, which outperforms HAR and SequenceFile. As for the reading efficiency, SIFM can reduce the access time by 50-80%. The improvement on access efficiency benefits from three aspects: (1) File correlations are considered when merging files, which reduce the seek time and delay in reading files. (2) Structured distributed architecture for storing the metadata files is applied to reduce the seek operations of requested files. (3) Considering the access locality in inter-block on DataNodes, prefetching and caching strategy is used to reduce the access time when reading huge number of small files.

Acknowledgments

This research is partially supported by the National Key Technology Research and Development Program of the Ministry of Science and Technology of China under Grant No. 2013BAB06B04; Key Technology Project of China Huaneng Group under Grant No.HNKJ13-H17-04; National Science Foundation of China under Grant No. 60903018, 61272543; Nature Science Fund of Jiangsu Province under Grant No. BK2012584, Key Laboratory of Geo-informatics of State Bureau of Surveying and Mapping 201005.

References

[1] 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", Proceedings of IEEE International Conference on Cluster Computing, New Orleans, USA, August 31 - September 4, **(2009)**.
[2] T. White," Hadoop: the definitive guide", Yahoo Press, **(2010)**.

[3] E. Neilsen, "The Sloan digital sky survey data archive server", Computing in Science & Engineering, vol. 10, no. 1, **(2008)**, pp.13-17.
[4] C. Shen, W. Lu, J. Wu and B. Wei, "A digital library architecture supporting massive small files and efficient replica maintenance", Proceedings of the 10th annual joint conference on digital libraries, ACM Press, QLD, Australia, June 21-25, **(2010)**, pp.391-394
[5] Petascale Data Storage Institue, "NERSC file system statistics," World Wide Web electronic publication, Available: http://pdsi.nersc.gov/filesystem.htm, **(2007)**.
[6] G. Mackey, S. Sehrish and J. Wang, "Improving metadata management for small files in HDFS", Proceedings of IEEE International Conference on Cluster computing, New Orleans, USA, August 31 - September 4, **(2009)**.
[7] Shafer J., Rixner S. and Cox A., "The Hadoop Distributed File System: Balancing Portability and Performance", Proceedings of 2010 IEEE International Symposium on Performance Analysis of Systems & Software, White Plains, NY, USA, March 28-30, **(2010)**, pp. 122-133
[8] J. Venner, "Pro Hadoop", Springer Press, **(2009)**.
[9] K. Shvachko, H. Kuang, S. Radia and R. Chansler, "The Hadoop Distributed File System", Proceedings of the 26th IEEE Symposium on Massive Storage Systems and Technologies, Incline Village, NV, USA, May 3-7, **(2010)**.
[10] "The Hadoop Distributed File System: Architecture and Design", available: http://hadoop.apache.org/common/docs/r0.20.1/hdfsdesign.html, **(2010)**.
[11] "The major issues identified: The small files problem", available: http://www.cloudera.com/blog/2009/02/02/the-small-files-problem, **(2010)**.
[12] A. Chervenak, J. M. Schopf, L. Pearlman, M.-H. Su, S. Bharathi, L. Cinquini, M. D'Arcy, N. Miller and D. Bernholdt, "Monitoring the Earth System Grid with MDS4", Proceedings of the Second IEEE International Conference on e-Science and Grid Computing. Washington: IEEE Computer Society, **(2006)**.
[13] J. K. Bonfield and R. Staden, "ZTR: A new format for DNA sequence trace data", Bioinformatics, vol. 18, no. 1, **(2002)**, pp. 3–10.
[14] E. H. Neilsen Jr., "The Sloan Digital Sky Survey data archive server", Computing in Science and Engineering, vol. 10, no. 1, **(2008)**, pp.13–17.
[15] B. Dong, "An Optimized Approach for Storing and Accessing Small Files on Cloud Storage", Journal of Network and Computer Applications, vol. 35, **(2012)**, pp. 1847-1862.
[16] A. Mohammed, "A survey on Data Security Issues in Cloud Computing: From Single to Multi-Clouds", Journal of Software, vol. 8, no. 5, **(2013)**, pp.1068-1078.
[17] J. Huang, "The Workflow Task Scheduling Algorithm Based on the GA Model in the Cloud Computing Environment", Journal of Software, vol. 9, no. 4, **(2014)**, pp.873-880.
[18] G. Lu, "QoS Constraint Based Workflow Scheduling for Cloud Computing Services", Journal of Software, vol. 9, no. 4, **(2014)**, pp. 926-930.

Authors

Yingchi Mao, is an associate professor at College of Computer and Information Engineering of Hohai University, China. She received her B.Sc. and M.Sc. degrees in computer science and technology from Hohai University in 1999 and 2003, respectively. She received her Ph.D. degree in computer science and technology from Nanjing University, China in 2007. Her research areas include distributed data management, distributed computing system.

Bicong Jia, is a master candidate at College of Computer and Information, Hohai University, China. He was received his B. Sc degree in Information Science from Yancheng Normal Institute in 2014. His research interests include big data analysis and management.

Wei Min, is a master candidate at College of Computer and Information, Hohai University, China. He was received his B.Sc. degree in computer science and technology from Southeast University Chengxian College in 2011. His research interests include data storage, cloud computing and big data.

Jiulong Wang, is a master candidate at College of Computer and Information, Hohai University, China. He was received his B. Sc degree in Computer Science from Hohai University in 2013. His research interests include distributed data management, big data index scheme.