An optimized approach for storing and accessing small files on cloud storage

Bo Dong¹, Qinghua Zheng¹,², Feng Tian¹,², Kuo-Ming Chao³, Rui Ma¹,², Rachid Anane ³

¹ MOE Key Lab for Intelligent Networks and Network Security, Xi’an Jiaotong University, Xi’an, China
² Department of Computer Science and Technology, Xi’an Jiaotong University, Xi’an, China
³ Faculty of Engineering and Computing, Coventry University, Coventry, UK

Abstract

Hadoop distributed file system (HDFS) is widely adopted to support Internet services. Unfortunately, native HDFS does not perform well for large numbers but small size files, which has attracted significant attention. This paper firstly analyzes and points out the reasons of small file problem of HDFS: (1) large numbers of small files impose heavy burden on NameNode of HDFS; (2) correlations between small files are not considered for data placement; and (3) no optimization mechanism, such as prefetching, is provided to improve I/O performance. Secondly, in the context of HDFS, the clear cut-off point between large and small files is determined through experimentation, which helps determine ‘how small is small’. Thirdly, according to file correlation features, files are classified into three types: structurally-related files, logically-related files, and independent files. Finally, based on the above three steps, an optimized approach is designed to improve the storage and access efficiencies of small files on HDFS. File merging and prefetching scheme is applied for structurally-related small files, while file grouping and prefetching scheme is used for managing logically-related small files. Experimental results demonstrate that the proposed schemes effectively improve the storage and access efficiencies of small files, compared with native HDFS and a Hadoop file archiving facility.

1. Introduction

Cloud computing has become increasingly popular as the next infrastructure for hosting data and deploying software and services (Vrable et al., 2009). Hadoop, an open-source software framework developed for reliable, scalable, distributed computing and storage (Hadoop, 2011), is successfully used by many companies including AOL, Amazon, Facebook, New York Times, etc. (Buyya et al., 2009).

Hadoop distributed file system (HDFS), the primary storage system of Hadoop, is composed of NameNode and DataNodes as architectural components. NameNode manages the file system namespace and regulates client accesses. DataNodes provide block storage and serve I/O requests from clients. HDFS is a typical representative for Internet service file systems running on clusters (Tantisiriroj et al., 2008), and has been widely utilized to support Internet applications.

HDFS is designed for storing large files with streaming data access patterns (White, 2010), which is suitable for the analysis of large datasets, such as machine learning, data mining, etc. In other words, it neglects the problem of storing and accessing small files. In fact, many current systems in the area of energy, climatology, astronomy, biology, e-Business, e-Library, and e-Learning contain huge amounts of small files (Altschul et al., 1990; Douceur and Bolosky, 1999; Bonfield and Staden, 2002; Chervenak et al., 2006; Gumińska and Madejski, 2007; NERSC, 2007; Neilsen, 2008; Shen et al., 2010). For instance, National energy research scientific computing center contained over 13 million files in 2007, 99% of which were under 64 MB and 43% of which were under 64 kB (NERSC, 2007). Therefore, storing and accessing a large number of small files pose a big challenge to HDFS because: (1) memory of NameNode is highly consumed by huge numbers of files; and (2) without considering file correlations for data placement and optimization mechanism, such as prefetching, high access latencies occur. So, in the paper, the efficiency problem of storing and accessing small files is called small file problem. Based on the analysis of small file problem, an optimized approach is designed for HDFS to optimize file placement, to reduce the memory consumption of NameNode, and to improve the access efficiency of small files.

Given the stated problem, criteria for quantifying storage and access efficiencies are firstly proposed for HDFS. Then, in the context of HDFS, the cut-off point between large and small files is quantified through experimentation. After that, files are classified into three types: structurally-related files, logically-related files, and independent files. Subsequently, specific schemes, including...
file merging and prefetching scheme and file grouping and prefetching scheme, are provided for structurally-related and logically-related small files respectively. Experiments based on three real application datasets demonstrate that the proposed schemes can effectively improve the storage and access efficiencies of small files, compared with native HDFS and Hadoop archives (HAR) (Hadoop Archives, 2011).

The major contributions of this paper are summarized below.

1. In the context of HDFS, the cut-off point between large and small files, i.e. ‘how small is small’, is quantified through experimentation for the first time.
2. File merging and file grouping strategies that consider file correlations, and three-level prefetching and caching strategies, are introduced to address small file problem.

The rest of this paper is organized as follows. Section 2 discusses related work. Section 3 presents the analysis of small file problem. Section 4 proposes the cut-off point between large and small files. Section 5 introduces the proposed schemes for structurally-related and logically-related small files. Section 6 evaluates the proposed schemes. Section 7 concludes the paper.

2. Related work

Hadoop is an open-source Apache project. Yahoo! has developed and contributed to 80% of the core of Hadoop (Shvachko et al., 2010). Shvachko et al. (2010) from Yahoo! described the detailed design and implementation of HDFS. They realized that their assumption that applications would mainly create large files was flawed, and new classes of applications for HDFS would need to store a large number of smaller files.

As there is only one NameNode in Hadoop and it keeps all the metadata in main memory, a large number of small files produce significant impact on the metadata performance of HDFS, and it appears to be the bottleneck for handling metadata requests of massive small files (Mackey et al., 2009; Min and Yokota, 2010).

HDFS is designed to read/write large files, and there is no optimization for small files. Mismatch of accessing patterns will emerge if HDFS is used to read/write a large amount of small files directly (Liu et al., 2009). Moreover, HDFS ignores the optimization on the native storage resource, and leads to local disk access becoming a bottleneck (Shafer et al., 2010). In addition, data prefetching is not employed to improve access performance for HDFS (Shafer et al., 2010).

Recently, research on small file problem of HDFS has attracted significant attention, but it is believed that there are three issues that need to be clarified or can be solved in a more appropriate way.

The first issue is the identification of the cut-off point between large and small files in the context of HDFS, that is, ‘how small is small’. Although Liu et al. (2009) treated the size of files smaller than 16 MB as small files, no justification or proof was provided.

The second issue is the classification of files, especially small files. White (2009) classified small files into two types: (1) files that are pieces of a large logical file; and (2) files that are inherently small.

The third issue is the solutions to small file problem. Current solutions are classified into two categories: general solutions and special solutions to meet particular scenarios.

(1) General solutions include HAR, SequenceFile (2011), and MapFile (2011): An HAR is a file archiving facility that packs files into HDFS blocks. It contains metadata files and data files. Data files contain the contents of those files which are part of the archive. In addition, Mackey et al. (2009) utilized HAR to improve the metadata management of HDFS for small files. HAR can reduce the memory consumption of NameNode, but the main problems of HAR are that (1) creating an archive generates a copy of original files, which puts extra pressure on disk space; and (2) no mechanism is provided to improve access efficiency.

A SequenceFile provides a persistent data structure for binary key–value pairs. It also works as a container for small files (White, 2010). It uses file name as key and file contents as value, and supports compression and decompression at record level or block level. The main problems of SequenceFile are that (1) it does not support update/delete method for a specified key; it only supports append method; and (2) if a particular key needs to be looked up, the entire SequenceFile has to be read; therefore the access efficiency is affected.

A MapFile is a type of sorted SequenceFiles with an index to permit lookups by key. It consists of an index file and a data file. The data file stores key–value pairs as records, which are sorted in key order. The index file stores key-location information and the location is the offset where the first record containing this key is located (Venner, 2009). Different from SequenceFile, in MapFile a particular key can be looked up without having to read the entire file. However, similarly with SequenceFile, MapFile only supports append method for a specified key, hence it cannot provide flexible APIs for applications.

In addition, the same limitation of HAR, SequenceFile, and MapFile is that file correlations are not considered when storing files.

(2) Special solutions to meet particular scenarios: Shen et al. (2010) merged page files into a large one and built an index for each book to store book pages for digital libraries. However, there was no detailed scheme to improve access efficiency.

Liu et al. (2009) combined small files into a large one and built a hash index for each small file, to store small data of Geographic Information System on HDFS. Features such as merging neighboring files and reserving several versions of data were considered. However, there is the same problem as in the previous case, besides index technology, no mechanism was provided to improve access efficiency.

This paper tries to solve small file problem from the following perspective. Firstly, in the context of HDFS, the cut-off point between large and small files is quantified in an experimental way. Secondly, from a point of systematic view, files are classified into three types, and relevant schemes are introduced for different types of small files, which is different from previous general solutions and special solutions. Thirdly, file correlations are considered when storing files, and prefetching technology is used to improve access efficiency.

3. Small file problem of HDFS

In this section, small file problem of HDFS is analyzed. Firstly, the architecture and access mechanism of HDFS are described. Then, small file problem is illustrated by two examples. Finally, the reasons that small file problem occurred are pointed out.

3.1. Architecture and access mechanism of HDFS

HDFS has the master/slave architecture, in which a single NameNode acts as the master and a number of DataNodes act as slaves, as shown in Fig. 1. NameNode, which is the metadata server of HDFS, maintains the metadata of all directories, files, and blocks (including block size, file length, replication, modification time, ownership, permission information, etc.) (White, 2010) and regulates client accesses. DataNodes provide block storage, serve I/O requests from clients, and perform block operations upon instructions from NameNode.
In HDFS, files are broken into fixed-sized blocks (64 MB by default), which are stored as independent units (White, 2010). Each block is replicated to a small number of separate machines (typically three) for fault tolerance.

HDFS adopts direct client access mechanism to achieve high bandwidth. When HDFS clients read data from HDFS, they firstly query NameNode for file metadata, and then perform actual I/O operations with relevant DataNodes to retrieve data.

### 3.2. Small file problem of HDFS

The performance of HDFS dramatically degrades when HDFS stores large amounts of small files, which results in high memory consumption of NameNode and serious access latency.

Figure 2 shows that the memory consumptions of NameNode grow linearly with the numbers of stored files when storing multiple BlueSky primary school resource datasets (BPSR datasets, which will be described in details in Section 6.1) on HDFS. Large numbers of files consume lots of memory of NameNode. For example, when storing ten BPSR datasets (the total file number is 420,430), the consumed memory of NameNode is 189.09 MB. Hence, according to the linear relationship, it can be deduced that, if the memory allocated to NameNode is 16GB, the maximum number of files that can be stored on the HDFS is less than 38 million.

Figure 3 lists the download durations when downloading files of different sizes under the same I/O throughput (e.g., 320 MB). As shown in Fig. 3, under the same throughput, the smaller the size of files is, the more the time it takes to download. For instance, the download duration of 5120 64 kB files is four and a quarter times of that of five 64 MB files.

In conclusion, as shown in Figs. 2 and 3, a large number of small files result in high memory consumption of NameNode and serious access latency. Therefore, it presents a challenge for the storage and access efficiencies of HDFS.

### 3.3. Analysis of small file problem

Before analysis of small file problem, it should be noted that, 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 take up any more disk space than is required to store its contents (White, 2010). For example, a 2.5 MB file stored with an HDFS block size of 64 MB uses 2.5 MB of disk space, not 64 MB.

There are mainly two reasons of small file problem occurred:

1. Memory of NameNode is highly consumed by the metadata and BlockMap of huge numbers of files. NameNode stores file system metadata in main memory. In general, the metadata of a file takes about 250 bytes of memory. For each block with default three replicas, its metadata takes about 368 bytes (Shvachko, 2007).
Therefore, the metadata of huge numbers of files would consume a lot of memory of NameNode. On the other hand, DataNodes periodically send block reports to NameNode with the lists of blocks stored on them, and NameNode gathers these reports and stores them as BlockMap in memory. The memory consumed by BlockMap cannot be neglected, when massive blocks are created by a large number of small files. Note that each small file is stored as an individual block on HDFS.

Theoretical analyses are introduced as follows.

As described earlier, the metadata of a file and a block with three replicas consume 250 and 368 bytes of memory, respectively. When the file number is zero, the number of memory bytes that NameNode consumed by its own is denoted as \( \delta \). The number of memory bytes that are consumed by the BlockMap of a block is denoted as \( \beta \). The size of an HDFS block is denoted as \( B \).

Suppose that there are \( N \) files, whose lengths are denoted as \( L_1, L_2, \ldots, \text{and} L_N \), then, the computational formula of the consumed memory of NameNode is derived as

\[
M_{NN} = 250aN + (368 + \beta) \sum_{i=1}^{N} \left[ \frac{L_i}{B} \right] + \delta \tag{1}
\]

where, \( \left[ \cdot \right] \) stands for the operation of rounding a number up to the next higher integer, and \( \sum_{i=1}^{N} (L_i/B) \) stands for the number of blocks. It follows from Eq. (1) that, in order to reduce the memory consumption of NameNode, the number of files that NameNode manages and blocks should be reduced.

(2) High access latencies occur due to the access mechanism of HDFS. When reading large numbers of small files, high access latencies occur due to three reasons. Firstly, latencies of frequent per-file metadata server interactions are obvious. HDFS clients need to query NameNode for file metadata, which happens once for each file access. When reading large files, this interaction is usually a minor overhead amortized over many data accesses (Hendricks, 2006). For instance, in case of 320 MB I/O throughput, for large files (e.g., five 64 MB files), HDFS clients only need to query NameNode five times. On the contrary, for small files under the same I/O throughput (e.g. 5120 64 kB files), clients need to query NameNode 5120 times, and the latencies are obvious.

Secondly, HDFS loses inter-block locality. That is, sequential files are usually not placed sequentially in block level, or even are placed on different DataNodes although its own data placement strategy can produce balance well between reliability, read performance, and write bandwidth (White, 2010).

Thirdly, HDFS currently does not provide prefetching function to hide I/O latency. Without considering file correlations for data placement and prefetching mechanism, reading small files from HDFS normally causes a lot of seeks and hoppings from DataNode to DataNode to retrieve files (White, 2009).

Theoretical analyses are introduced as follows.

When reading a file from HDFS, the download time is composed of the following parts: (1) a client sends a read command to NameNode, and its time cost is denoted as \( \delta_{CN} \); (2) NameNode looks up the metadata of the requested file in memory, and the time cost is denoted as \( \delta_{metadata} \); (3) the metadata is returned to the client, and the time cost is denoted as \( \delta_{NC} \); (4) the client sends a read command to a relevant DataNode, and its time cost is denoted as \( \delta_{C} \); (5) the DataNode fetches the requested block from disk, and the time cost is denoted as \( \delta_{disk} \); and (6) the block is returned to the client, and its time cost is denoted as \( \delta_{network} \cdot \delta_{CN} \) and \( \delta_{CD} \) are considered as constants because they are consumed by sending commands. Similarly, \( \delta_{NC} \) is also considered as a constant because the size of metadata is very small. While, \( \delta_{network} \) is relevant with the length of files (for small files, each of which has only one block, the length of a file is equal to that of its block), \( L \), and network transmission speed, \( speed \). So, it is indicated as \( f_{network} (L/speed) \).

Suppose that there are \( N \) small files, whose lengths are denoted as \( L_1, L_2, \ldots, \text{and} L_N \). Then, the computational formula of the total time of downloading \( N \) files is derived as

\[
T_{HDFS} = N(\delta_{CN} + \delta_{NC} + \delta_{CD}) + \sum_{i=1}^{N} \delta_{metadata} + \sum_{i=1}^{M} \delta_{disk} + \sum_{i=1}^{M} f_{network} \left( \frac{L_i}{speed} \right) \tag{2}
\]

where, \( M \) stands for the number of blocks read, i.e. \( \sum_{i=1}^{N} (L_i/B) \).

For small files, because each of them has only one block, the number of blocks read, i.e. \( M \), is equal to that of files read, i.e. \( N \). Therefore, Eq. (2) can be computed as

\[
T_{HDFS} = N(\delta_{CN} + \delta_{NC} + \delta_{CD}) + \sum_{i=1}^{N} \left( \delta_{metadata} + \delta_{disk} + f_{network} \left( \frac{L_i}{speed} \right) \right) \tag{3}
\]

for small files \( M = N \)

As shown in Eq. (3), to reduce access latency needs to strive from the following aspects: (1) reducing the communications between clients and NameNode; (2) decreasing the time cost of looking up metadata; (3) decreasing the I/O cost that DataNodes fetch blocks from disk; and (4) reducing the network latency of transferring files.

(4) Cut-off point between large and small files

The first critical issue for small file problem is to identify the cut-off point between large and small files and to answer the question how small is small. In this section, the issue is studied through experimental analysis once the criteria for quantifying storage and access efficiencies are defined.

4.1. Criteria of storage and access efficiencies for HDFS

Storage capacity is normally not a bottleneck for HDFS even if it stores huge numbers of small files. On the contrary, the memory available on NameNode is the primary limitation of HDFS scalability (Hadoop, 2011). Hence, in the context of HDFS, File Number per KB of Memory of NameNode (FNPKN) is an important criterion to measure storage efficiency, which is defined as

\[
FNPKN = \frac{N}{M_{NN}} \tag{4}
\]

where, \( N \) stands for the number of files stored on HDFS, and \( M_{NN} \) stands for the consumed memory of NameNode.

For example, according to Eqs. (1) and (4), for HDFS, the store efficiency index, \( FNPKN \), is equal to:

\[
FNPKN_{HDFS} = \frac{N}{250aN + (368 + \beta) \sum_{i=1}^{N} \left[ \frac{L_i}{B} \right] + \delta} \times 1024 \tag{5}
\]

Note that the unit of \( FNPKN \) is file number per kB. Because the unit of the denominator in Eq. (5) is byte, 1024 is used as the unit conversion from byte to kB in Eq. (5).

As for access efficiency, \( MB \) of Accessed Files per Second (MFPS) is an important criterion, which is defined as:

\[
MFPS = \frac{S}{T} \tag{6}
\]

where, \( S \) stands for the total size of accessed files, and \( T \) is the access time.
4.2.2. Experimental results

MSPF of HDFS is equal to:

\[
MSPF_{HDFS} = \frac{\sum_{i=1}^{N} L_i}{N \delta_{CN} + \delta_{NC} + \delta_{CD}} + \sum_{i=1}^{N} \left( \delta_{metadata} + \delta_{disk} + f_{network} \left( \frac{L_i}{\text{speed}} \right) \right),
\]

for small files \( M = N \) \( (7) \)

When using a benchmark/dataset to evaluate, Millisecond per Accessing a File (MSPF), which stands for the average time of accessing a file in the benchmark/dataset, is also used to measure access efficiency. It is defined as

\[
MSPF = \frac{TB}{NB}
\]

where, \( TB \) stands for the total time of accessing files in the benchmark/dataset, and \( NB \) stands for the number of accessed files in the benchmark/dataset.

4.2. Storage and access efficiencies over different file sizes

The relationship among file sizes, storage and access efficiencies is analyzed through experimentation. Based on the relationship, the cut-off point between large and small files is quantified.

4.2.1. Experimental methods

Twenty three datasets are built, each of which contains 10,000 files whose sizes are 0.5 MB, 1 MB, 1.5 MB, 2 MB, 2.5 MB, 3 MB, 3.5 MB, 4 MB, 8 MB, 12 MB, 16 MB, 20 MB, 24 MB, 28 MB, 32 MB, 36 MB, 40 MB, 44 MB, 48 MB, 52 MB, 56 MB, 60 MB, and 64 MB, respectively. The following experiments are carried out on each dataset:

- Upload the dataset to an empty HDFS, and the consumed memory of NameNode is measured to calculate FNPKN;
- Randomly download 1000 files from each dataset, and the total download time is measured to calculate MFPS.

In the experiments, HDFS block size is set as 64 MB by default.

4.2.2. Experimental results

(1) Storage efficiencies over different file sizes: FNPKN of experiments on the 23 datasets are shown in Fig. 4.

As can be seen in Fig. 4, FNPKN of experiments on the 23 datasets remain essentially unchanged. For files whose sizes are not more than HDFS block size, each of them has only one block. According to Eq. (1), for the same number of files whose sizes are not more than HDFS block size, the consumed memories are roughly the same.

(2) Access efficiencies over different file sizes: MFPS of experiments on the 23 datasets are shown in Fig. 5.

In order to further analyze the variation trend and linearize file sizes which grow exponentially in Fig. 5, file size (x-axis) and MFPS (y-axis) are both taken as logarithmic values (base 2), and the result is shown in Fig. 6.

As seen in Fig. 6, it is obvious that, the latter 15 points are in a line, and the former 8 points are mainly in another line. Therefore, linear fitting method is used for the former 8 points and the latter 15 points, respectively.

After using linear fitting method, the crossing point of the solid and dotted lines in Fig. 6 is 2.12. When points falls to the range between 0 and 2.12, the growth of \( \log_{2}(MFPS) \) is sharply evident; whereas, when \( \log_{2}(file\ size) \) is larger than 2.12, the growth of \( \log_{2}(MFPS) \) is flat. Therefore, the crossing point, \( \log_{2}(file\ size) = 2.12 \), should be set as the cut-off point of access efficiencies (at this time the file size is 4.35 MB).

In addition, three experiments about cut-off values between large and small files are conducted, in which HDFS block sizes are set as 16 MB, 32 MB, and 48 MB respectively. The results are shown in Fig. 7.

As seen in Fig. 7, when HDFS block sizes are set as 16 MB, 32 MB, and 48 MB, the crossing points are 2.13, 2.08, and 2.14 respectively. Obviously, they are very close to the crossing point 2.12 that is obtained when HDFS block size is set as 64 MB. Within the limits of experimental error, the cut-off values are considered the same when HDFS block sizes are 16 MB, 32 MB, 48 MB, and 64 MB in the experiments.

In conclusion, in the context of HDFS, the cut-off point between large and small files is quantified through experimentation. To our
knowledge, this issue has not been clearly stated before. Liu et al. (2009) treated the size of files smaller than 16 MB as small files without theoretical derivation or proof.

5. Small file tuning schemes for HDFS

In this section, from the point of systematic view, files are firstly classified into three types, in order to provide specific solutions to different types of files. Then, small file tuning schemes, including file merging and prefetching scheme for structurally-related small files (FMP-SSF) and file grouping and prefetching scheme for logically-related small files (FGP-LSF), are introduced respectively. At last, FMP-SSF and FGP-LSF are compared with each other.

5.1. Classifications of files

According to the structural modes and file correlation features, files are classified into three types: structurally-related files, logically-related files and independent files.

(1) Structurally-related files: Structurally-related files are the dependent segments/pieces of a large/logical file, because of some reasons, the large/logical file is divided into many smaller segments/pieces. For example, in order to improve the efficiency of data transmission, map services that are common in Internet frequently split a large map (such as the map of a city) into many small pictures to be stored and accessed.

Structurally-related files have the characteristic that some of them would be merged into a fixed set to represent a large/logical file. The file number of the set is fixed (i.e. the number of files split from the large/logical file), and the set never includes other files. Each file is an indispensable and dependent part of the fixed set, and there is a clear logical order relation among these files.

(2) Logically-related files: Logically-related files are independent ones, but can be classified/grouped into a set/group according to explicit rules/relationship among these files. For instance, in e-Learning systems, educational resources of a course include courseware, exercise books, reference materials, etc., which means that these files belong to the course and certain correlations among them exist. The pictures of goods in an eBay shop and photos of a person in Facebook have similar characteristics.

One characteristic of this kind of files is that the file number in a set/group is not fixed, and also cannot be predicted since files in a set/group may be added or deleted frequently.

(3) Independent files: This kind of files is independent one, and cannot tell which set/group they belong to. There are no clear correlations among these files.

Based on the idea of file size and classification of files, in this research small file tuning schemes are proposed for structurally-related and logically-related small files, and they are described in the following sections.

5.2. File merging and prefetching scheme for structurally-related small files

Taking the characteristics of structurally-related files into consideration, FMP-SSF is proposed. The core ideas of FMP-SSF include: (1) merging structurally-related small files that belong to a large/logical file into one file (called merged file) to relieve the memory consumption of NameNode; and (2) utilizing prefetching and caching technologies to improve access efficiency. FMP-SSF is composed of local index file strategy, file merging strategy and three-level prefetching and caching strategy, as shown in Fig. 8.

According to the file merging strategy, files belonging to a large/logical file are merged in HDFS clients. A local index file is built for a merged file, and is also merged into the merged file. Then the merged file is uploaded to HDFS. The three-level prefetching and caching strategy is used to cache metadata and to prefetch local index files and correlated files. Based on the strategy, communications with HDFS are drastically reduced thus to improve access efficiency, when downloading files. However, when a requested file misses in cache, the client needs to query NameNode for file metadata. According to the metadata, the client connects with appropriate DataNodes where blocks locate. When the local index file is firstly read, based on the offset and length, the requested file is split from the block, and is returned to the client. Note that the process is not illustrated in Fig. 8.

5.2.1. Local index file strategy

Since NameNode only maintains the metadata of merged files, indexes should be built for each original file to indicate its offset
and length in a merged file. Because of the fixed feature of structurally-related files, the indexes of a merged file are also fixed. Therefore, a local index file is built for each merged file, which indicates the offset and length for each original file in it.

A merged file may have multiple blocks, and its local index file is stored in the starting location of each block of the merged file. Compared with index files loaded in the memory of NameNode in other solutions (e.g. FGP-LSF which will be introduced in Section 5.3), it does not result in additional overhead to NameNode, since local index files are stored on DataNodes.

In order to calculate the length of a local index file just according to file number, the length of an index for a file is fixed. This has two advantages: (1) in file merging process, the length of a local index file can be calculated in advance; and (2) in file reading process, it is convenient to analyze indexes.

5.2.2. File merging strategy

File merging operations are carried out in HDFS clients, which merge structurally-related small files belonging to a large/logical file into one merged file. NameNode only maintains the metadata of merged files and does not perceive the existence of original files, so file merging reduces the number of files that need to be managed by NameNode, thus to reduce the memory consumption of NameNode.

File merging operations are executed as follows:

Step1: Preparation work.

The number of original files is calculated, and the size of the local index file that will be established in Step 2 is calculated. Then, the size of all files including original files and the local index file, denoted \( L_{merge} \), is calculated.

Step2: To establish a local index file.

\( L_{merge} \) is compared with HDFS block size. According to different comparison results, a local index file will be established. If \( L_{merge} \) is less than HDFS block size, the merged file will only have one block. That is, the local index file and original files will be arranged in turn as the same as default order. Original files will be arranged by their logical order relation. Based on this order, the offset of each file is indicated. After which, the local index file is established.

If \( L_{merge} \) exceeds HDFS block size, the merged file will be broken up into multiple blocks. In which, one restriction that an original file should NOT stretch across two blocks should be confirmed with; otherwise, reading the file needs to fetch data from two blocks (what is worse, the two blocks are frequently on two DataNodes). To satisfy the restriction, a boundary filling algorithm is adopted, which is shown as Fig. 9. It fills the prior block with a blank file (called boundary fragment), and then writes the local index file and the file in the next block, when a file stretches across two blocks. Its pseudo-code is provided in Fig. 10. It is noteworthy that there are no indexes for boundary fragments in FMP-SSF. After the execution

![Fig. 8. The schematic diagram of FMP-SSF.](image)

![Fig. 9. The schematic diagram of the boundary filling algorithm. (a) File merging process without the boundary filling algorithm and (b) File merging process with the boundary filling algorithm.](image)
5.2.4. Theoretical analyses
Suppose that there are \( N \) small files, which are merged into \( K \) merged files, \( M_1, M_2, \ldots, \) and \( M_k \), whose lengths are denoted as \( L_{M_1}, L_{M_2}, \ldots, \) and \( L_{M_K} \).

(1) Storage efficiency analyses: The computational formula of the consumed memory of NameNode in FMP-SSF is derived as

\[
M_{NN} = 250K(368 + \beta) \sum_{i=1}^{K} \left( \frac{L_{M_i}}{HBS} \right) + \alpha \tag{9}
\]

where, \( \sum_{i=1}^{K} \left( \frac{L_{M_i}}{HBS} \right) \) stands for the number of blocks.

Accordingly, in FMP-SSF the store efficiency index, \( FNPKN \), is equal to:

\[
FNPKN_{\text{FMP-SSF}} = \frac{N}{250K(368 + \beta) \sum_{i=1}^{K} \left( \frac{L_{M_i}}{HBS} \right) + \alpha} \cdot 1024 \tag{10}
\]

Note that the unit of \( FNPKN \) is file number per kB. As the same as Eq. (5), because the unit of the denominator is byte, 1024 is used as the unit conversion from byte to kB in Eq. (10).

As shown in Eq. (9), in FMP-SSF, the memory consumption of NameNode does not have relations with the number of original files \( N \), but is relevant with the number of merged files \( K \), where \( K \) is much smaller than \( N \), if the stored files are mostly small files. In addition, the number of blocks, \( \sum_{i=1}^{K} \left( \frac{L_{M_i}}{HBS} \right) \), is also much smaller than that in Eq. (1), i.e. \( \sum_{i=1}^{N} \left( \frac{L_i}{HBS} \right) \).

It is concluded from Eqs. (9) and (10) that, FMP-SSF can effectively relieve the memory consumption of NameNode, and improve storage efficiency.

(2) Access efficiency analyses: In FMP-SSF, when reading a file, a local index file needs to be read from disk, and the offset and length of the requested file can then be retrieved by parsing the index file. The time consumed by this process is denoted as \( \delta_{\text{index}} \).

Moreover, in FMP-SSF, the number of blocks read is equal to that of files read because: (1) the files are small files, whose sizes are smaller than HDFS block size; and (2) the boundary filling algorithm avoids the condition that a small file stretches across two blocks. Then, the computational formula of the total time of downloading \( N \) files is derived as

\[
T_{\text{FMP-SSF}} = N\delta_{\text{CN}} + \delta_{\text{NC}} + \delta_{\text{CD}} + \sum_{i=1}^{N} \left( \delta_{\text{metadata}} + \delta_{\text{index}} + \delta_{\text{disk}} + f_{\text{network}} \left( \frac{L_i}{\text{speed}} \right) \right) \tag{11}
\]

As shown in Eq. (11), in FMP-SSF, the download time is more than that of native HDFS, in the same condition. This is because local index files need to be read from the disks of DataNodes and parsed. Therefore, the three-level prefetching and caching strategy is adopted to improve access efficiency which is introduced as follows. Note that prefetching commands are asynchronous.

1. Only metadata caching works: Suppose that \( N \) small files which belong to \( K \) merged files are requested in turn. Only when the first requested file in each merged file is read, the metadata of the merged file needs to be looked up. In other conditions, the metadata can be directly obtained from cache thus to reduce the communications with NameNode. The eviction of prefetch metadata is not considered since the size of metadata is very small. Then, the total download time is denoted as

\[
T_{\text{FMP-SSF}} = K(\delta_{\text{CN}} + \delta_{\text{NC}}) + N\delta_{\text{CD}} + \sum_{i=1}^{K} \delta_{\text{metadata}} + \sum_{i=1}^{N} \left( \delta_{\text{disk}} + \delta_{\text{index}} + f_{\text{network}} \left( \frac{L_i}{\text{speed}} \right) \right) \tag{12}
\]

2. Metadata caching and index prefetching work: Similarly, only the index of the first requested file in each merged file needs
3. Metadata caching, index prefetching, and correlated file prefetching work.
Suppose that \( p \) original files are directly obtained from cache, then, the total download time is denoted as

\[
T_{\text{FMP-SSF}} = K(\delta_{\text{CN}} + \delta_{\text{NC}}) + N_0\delta_{\text{CD}} + \sum_{i=1}^{K} (\delta_{\text{metadata}} + \delta_{\text{index}})
+ \sum_{i=1}^{N} \left( \delta_{\text{disk}} + f_{\text{network}} \left( \frac{L_{\text{disk}}}{\text{speed}_{\text{disk}}} \right) \right)
\]

(13)

5.3. File grouping and prefetching scheme for logically-related small files
The core ideas of FGP-LSF include: (1) grouping logically-related small files that belong to a set/group into one file (called logic unit) to relieve the memory consumption of NameNode; and (2) utilizing prefetching and caching technologies to improve access efficiency. A logic unit is a concept higher than file level, which generally refers to a set/group, in the logical viewpoint of conceptual hierarchy. For example, in the e-Learning system, a course is a logic unit, and all small files belonging to a course will be grouped together. A logic unit may have multiple blocks. FGP-LSF is composed of global index file strategy, file grouping strategy, and three-level prefetching and caching strategy, as shown in Fig. 11.

According to the file grouping strategy, a file is uploaded to HDFS, and is grouped into the logic unit that it belongs to. The index of the file is added to the global index file of the logic unit. Note that the global index file is loaded in the memory of NameNode. Similarly, the three-level prefetching and caching strategy is used to cache metadata and to prefetch global index files and correlated files. Based on the strategy, metadata, indexes, and correlated files can be read directly from cache. However, when requested files miss in cache, clients need to query NameNode for the metadata of logic units and the indexes of requested files. According to the metadata and indexes, clients perform I/O operations with relevant DataNodes to retrieve data. Note that the process is not illustrated in Fig. 11.

5.3.1. Global index file strategy
Similarly, NameNode only maintains the metadata of logic units, and indexes should be built for each original file to indicate its offset and length in a logic unit. In addition, the storage of logically-related files is a dynamic process. That is, files may be added or deleted frequently. Therefore, the previous local index file strategy of FMP-SSF is not suitable for logically-related small files. Consequently, a global index file is built for each logic unit, and is loaded in the memory of NameNode. A global index file is composed of two index sets: small file index set and fragment index set. The former one is designed for small files and the latter one is for fragments. Since the memory consumed by each index is much smaller than that of the metadata of a file, the global index file strategy still relieves the memory consumption of NameNode.

- A small file index set is used to indicate the offsets and lengths of small files. Since the most frequent operation on small file index sets is queries by file names, indexes are sorted by file names.
- A fragment index set is used to locate fragments. For logic units, there exist two types of fragments: (1) fragments issued by deletion operations (when deleting a file, its content is not deleted immediately but only to mark it deleted in index level); and (2) boundary fragments due to the boundary filling algorithm, when file grouping. Since the most frequent operation on fragment index sets is queries by lengths, indexes are sorted by fragment lengths. It is noteworthy that fragments are not named.

5.3.2. File grouping strategy
File grouping is a process to group logically-related small files into logic units that they belong to. When uploading a logically-related small file, according to the cut-off point between large and small files, the client checks whether the file is a large file. If it is a
large file, it will be stored using the native method of HDFS. If it is a small file, the proposed file grouping will be carried out. According to the write request, NameNode determines which logic unit the file belongs to, and then groups the file into the logic unit. However, in some scenarios, not all of small files can be classified into a logic unit. In this case, they will be arranged into pending units. According to file scale, several pending units exist. Strategies, such as polling or assigning by Hash value, are adopted to arrange files into specific pending units. At particular times, based on access locality (i.e., through the analysis of user access logs, Lou et al., 2010), files in pending units are classified into appropriate logic units. This strategy has two advantages: (1) overcoming the difficulty of grouping files without exploiting access locality, especially when uploading them; and (2) sharing the computing load through a batch process since the cost of exploiting access locality for all files once is high.

When grouping files, two strategies are adopted. The first one is the boundary filling algorithm, which is similar to that of FMP-SSF and is used to avoid the condition that a small file stretches across two blocks. The second one is fragment redistribution algorithm to improve the utilization ratio of disk space. In which, when a small file (denoted as \( F_{wr} \)) is grouped into a logic unit, NameNode first queries the fragment index set of the logic unit to check whether there are appropriate fragments for \( F_{wr} \) to be filled into. If so, \( F_{wr} \) is filled into an appropriate fragment, which results in the split of the fragment; otherwise, the unassigned space is allocated to \( F_{wr} \). The algorithm is introduced as Fig. 12.

5.3.3. Three-level prefetching and caching strategy

Just like FMP-SSF, FGP-LSF also has a three-level prefetching and caching strategy, which is composed of metadata caching, index prefetching, and correlated file prefetching. Since metadata caching and index prefetching are the same with those of FMP-SSF, the descriptions of them are omitted. Correlated file prefetching of FGP-LSF is introduced as follows.

For a merged file, there is a clear logical order relation among files. Based on the logical order relation, correlated files are determined and are then to be prefetched. While, although files in a logic unit have certain correlations and they are often accessed continuously, there is no clear relations among them. Therefore, it is necessary to exploit access locality among files in a logic unit to determine correlated files.

After a requested file is read, correlated files are to be prefetched in the term of access locality among files in the same logic unit. Although there may be correlations among logic units, exploiting access locality and prefetching correlated files within a logic unit reduce computational complexity, meet the scenario of logically-related files and hold high efficiency. Obviously, this is similar to the notion of context-aware prefetching (Soundararajan et al., 2008).

5.3.4. Theoretical analyses

Suppose that there are \( N \) small files, which are grouped into \( K \) logic units, \( LU_1, LU_2, \ldots, \) and \( LU_K \), whose lengths are denoted as \( L_{d1}, L_{d2}, \ldots, \) and \( L_{dK} \), respectively.

(1) Storage efficiency analyses: Since in FGP-LSF global index files are loaded in the memory of NameNode, the length of an index for an original file is denoted as \( \gamma \). Then, the computational formula of the consumed memory of NameNode is derived as

\[
M_{NN} = 250\times k + (368 + \beta)\times N = 1, L_{LU1} + N\times \gamma + \alpha
\]

where, \( \sum_{i=1}^{K} (L_{LUi} / HBS) \) stands for the number of blocks.

As shown in Eq. (15), in FGP-LSF, the memory consumption of NameNode has relations with the number of original files \( N \), since indexes for each original file are loaded in the memory of NameNode. Fortunately, \( \gamma \) is smaller than the length of the metadata of a file. In addition, the number of blocks, \( \sum_{i=1}^{K} (L_{LUi} / HBS) \), is also much smaller than that in Eq. (1), i.e., \( \sum_{i=1}^{N} (L_{Ui} / HBS) \). Therefore, FGP-LSF can relieve the memory consumption of NameNode.

(2) Access efficiency analyses: In FGP-LSF, when reading a file, its index needs to be read from the memory of NameNode, and the time consumed by this process is denoted as \( \delta_{index} \). Similarly, the number of blocks read is equal to that of files read. Then, the computational formula of the total download time is derived as

\[
T_{FGP-LSF} = N(\delta_{CN} + \delta_{NC} + \delta_{CD}) + \sum_{i=1}^{N} (\delta_{metadata} + \delta_{index} + \delta_{disk})
\]

\[+ f_{network} \left( \frac{L_{M}}{speed} \right) \]

\[ (16) \]

Fig. 12. The pseudo-code of the fragment redistribution algorithm.
As shown in Eq. (16), in FGP-LSF the download time is more than that of native HDFS. Therefore, the three-level prefetching and caching strategy is adopted to improve access efficiency, and is introduced as follows. Note that prefetching commands are asynchronous.

1. Metadata caching and index prefetching work: Because the metadata and indexes of files are both loaded in the memory of NameNode, metadata caching and index prefetching are described together. Only when the first requested file in each logic unit is read, the metadata and global index file of the logic unit need to be looked up from the memory of NameNode. In other conditions, metadata and indexes can be directly obtained from cache. The eviction of prefetched metadata and indexes is not considered since their sizes are very small. Then, the total download time is denoted as

\[
T_{\text{FGP-LSF}} = K(\delta_{\text{CN}} + \delta_{\text{HC}}) + N\delta_{\text{CD}} + \sum_{i=1}^{K}(\delta_{\text{metadata}} + \delta_{\text{index}})
\]

\[+ \sum_{i=1}^{N} \left( \delta_{\text{disk}} + f_{\text{network}} \left( \frac{L_i}{\text{speed}} \right) \right) \]  

(17)

2. Metadata caching, index prefetching, and correlated file prefetching work: Suppose that \( p \) original files are directly obtained from cache, then, the total download time is denoted as

\[
T_{\text{FGP-LSF}} = K(\delta_{\text{CN}} + \delta_{\text{HC}}) + (N-p)\delta_{\text{CD}} + \sum_{i=1}^{K}(\delta_{\text{metadata}} + \delta_{\text{index}})
\]

\[+ \sum_{i=1}^{N-p} \left( \delta_{\text{disk}} + f_{\text{network}} \left( \frac{L_i}{\text{speed}} \right) \right)
\]

(18)

While, the performance of prefetching correlated files has relations with the accuracy of file correlation predictions. When in high concurrency and low accuracy situation, correlated file prefetching triggers a lot of useless prefetching activities. These activities lead to a great many useless I/O operations and network transmissions. Serious network contention and congestion would be caused by these network transmissions, which deteriorate speed. According to Eq. (18), the download time is influenced by speed. Therefore, useless prefetching activities would result in the deterioration of speed and access efficiency. In this condition, metadata caching and index prefetching (without the use of correlated file prefetching) can still reduce access latency.

5.4. Summary and comparison

In FMP-SSF, the file merging strategy is used to reduce the number of files that NameNode handles. When merging file, file correlations are considered, that is, merging all files which belong to a large/logical file into a merged file. On the other hand, since indexes of a merged file are fixed, the local index file strategy is adopted, and a local index file is built for each merged file. Local index files are stored on DataNodes, which does not result in additional overhead to NameNode.

In FGP-LSF, the file grouping strategy is used to reduce the number of files that NameNode deals with. When grouping files, file correlations are also considered, i.e. grouping small files which belong to a set/group into a logic unit. Files in a logic unit may be added or deleted frequently. A global index file is built for each logic unit, which is loaded in the memory of NameNode. The global index file strategy puts a certain burden on NameNode.

Since file correlations are taken into consideration when file merging and grouping, three-level prefetching and caching strategies are adopted in both FMP-SSF and FGP-LSF, but correlated file prefetchings are different. In FMP-SSF, there is a clear logical order relation among files in a merged file, and correlated file prefetching can work best based on the relation. But in FGP-LSF, correlated files need to be firstly exploited within a logic unit, and access efficiency is influenced by the accuracy of file correlation predictions.

6. Experimental evaluation

In this section, in order to demonstrate that the proposed schemes can improve the storage and access efficiencies of small files, extensive experiments on three real application datasets are conducted, and their results are compared with the ones of native HDFS and HAR.

6.1. Experimental environment and workload overview

The experimental platform is built on a cluster with nine nodes. One node, which is IBM x3650 server, acts as NameNode. It has 8 Intel Xeon CPU (2.00 GHz), 16 GB memory and 1TB disk. The other eight nodes, which are IBM X3610 servers, act as DataNodes. Each of them has 8 Intel Xeon CPU (2.00 GHz), 8 GB memory and 3TB disk. All nodes are interconnected with 1.0 Gbps Ethernet network.

In each node, Ubuntu server 9.04 with the kernel of version 2.6.28-11-server is installed. Hadoop version is 0.20.1 and Java version is 1.6.0. The number of replicas is set to 3 and HDFS block size is 64 MB by default.

For structurally-related small files, Yotta PowerPoint courseware resource dataset (YPCCR) is used as the benchmark. A PowerPoint courseware consists of a PowerPoint file and a certain amount of preview pictures which snapshot each single page of the PowerPoint file. YPCCR dataset contains 2000 PowerPoint courseware, and the total number of files is 119,600. The maximum number of files in a PowerPoint courseware is 69, the minimum is 49, and the average value is 59.8. In the experiments, each PowerPoint courseware is treated as a merged file. The distribution of file sizes in YPCCR dataset is shown in Fig. 13. The file sizes in YPCCR dataset range from 28 kB to 3340 kB, and files whose sizes between 32 kB and 256 kB account for 96% of the total files.

For logically-related small files, BlueSky primary school resource dataset (BPSR) dataset and Yotta computer science resource dataset (YCSR) dataset are used as benchmarks. There are 42,043 files in BPSR dataset, which belong to 794 courses. The maximum number of files in a course is 651, the minimum is 1, and the average value is 52.95. YCSR dataset contains 3428 files,
belonging to seven courses. The maximum number of files in a course is 1140, the minimum is 25, and the average is 489.71. In the experiments, each course is treated as a logic unit. The distributions of file sizes in BPSR and YCSR dataset are shown in Table 1.

According to Section 4.2, in the context of HDFS, 4.35 MB is taken as the cut-off point between large and small files. All files of YPCR dataset are small files. While for BPSR dataset, 41,756 files are small files, accounting for 99% of the total files, and 3224 files of YCSR dataset are small files, accounting for 94%.

### 6.2. Experimental methodology

Experiment items include two categories: storage efficiency experiments and access efficiency experiments. In the experiments, the proposed schemes, including FMP-SSF and FGP-LSF, are compared with native HDFS and HAR. For HAR, all files of a merged file or a logic unit are stored as an HAR file. Considering that creating an HAR generates a copy for each original file, original files are deleted after archiving.

In storage efficiency experiments, the consumed memories of NameNode are measured by JConsole provided by Java 2 Platform Standard Edition (J2SE), and FNPKN criterion is used to compare the results. In access efficiency experiments, MSPF criterion is used to compare the results. The experiments of concurrent requests are simulated through 1, 2, 4, and 8 HDFS clients respectively. For each client, 100 files are downloaded in order.

Unless noted otherwise, each access efficiency experiment is repeated seven times, and for each result the average value is obtained with eliminating two maximums and two minimums. By doing so, outliers are filtered out which may be caused by network congestion and other uncertain factors (Zhang et al., 2009).

### 6.3. Experiments of structurally-related small files

#### 6.3.1. Storage efficiency experiments of structurally-related small files

For native HDFS, HAR, and FMP-SSF, the consumed memories of NameNode are measured to calculate FNPKN when storing YPCR dataset. The results are shown in Fig. 14.

For native HDFS, HAR, and FMP-SSF, FNPKN are 2.47, 15.78, and 26.32 respectively. As expected, due to their file archiving and merging facilities, HAR and FMP-SSF achieve much better storage efficiencies than native HDFS, increasing by up to 540% and 967%, respectively. As for HAR the additions of MSPF are 69%, 92%, and 77% respectively.

In conclusion, for YPCR dataset, FMP-SSF effectively improves storage efficiency by up to 967%, which is superior to HAR. As for access efficiency, FMP-SSF reduces MSPF by 70–80%. The improvement on access efficiency benefits from two aspects: (1) file correlations are considered when merging files, which reduces seek time and rotational delay of disks in file reading process; and (2) taking into account the logical order relation among files in a merged file, the three-level prefetching and caching strategy can work at its best. In contrast, HAR always deteriorates access efficiency in these cases.

#### 6.3.2. Access efficiency experiments of structurally-related small files

Twenty-five merged files are selected randomly, and four files of each selected merged file are downloaded based on the logical order relation. The results are shown in Fig. 15.

When client number is 1, MSPF of native HDFS is 40.73. FMP-SSF reduces this by up to 72%, which is 11.47 and is mainly profited by its three-level prefetching and caching strategy. On the contrary, an MSPF of HAR is 49.34, which is about 121% compared with that of native HDFS. Similarly, when concurrent client numbers are 2, 4, and 8, for FMP-SSF the reductions of MSPF are 82%, 82%, and 80%, and for HAR the additions of MSPF are 69%, 92%, and 77% respectively.

In conclusion, for YPCR dataset, FMP-SSF effectively improves access efficiency by up to 967%, which is superior to HAR. As for access efficiency, FMP-SSF reduces MSPF by 70–80%. The improvement on access efficiency benefits from two aspects: (1) file correlations are considered when merging files, which reduces seek time and rotational delay of disks in file reading process; and (2) taking into account the logical order relation among files in a merged file, the three-level prefetching and caching strategy can work at its best. In contrast, HAR always deteriorates access efficiency in these cases.

### 6.4. Experiments of logically-related small files

#### 6.4.1. Storage efficiency experiments of logically-related small files

For native HDFS, HAR, and FGP-LSF, the consumed memories of NameNode are measured to calculate FNPKN when storing BPSR and YCSR dataset respectively. The results are shown in Fig. 16.

When storing BPSR dataset, for native HDFS, HAR, and FGP-LSF, FNPKN are 2.17, 15.92, and 7.08 respectively. As expected, HAR and FGP-LSF achieve much better storage efficiencies than native HDFS, increasing by up to 633% and 226%, due to their file archiving and grouping facilities. However, different from FMP-SSF, FGP-LSF consumes more memory than HAR, since its global

### Table 1

The distributions of file sizes in BPSR and YCSR dataset.

<table>
<thead>
<tr>
<th>File size MB</th>
<th>(0, 0.5]</th>
<th>(0.5, 1]</th>
<th>(1, 2]</th>
<th>(2, 4]</th>
<th>(4, 8]</th>
<th>(8, 16]</th>
<th>(16, 32]</th>
<th>(32, 64]</th>
<th>&gt; 64</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPSR dataset</td>
<td>35,816</td>
<td>1221</td>
<td>2112</td>
<td>2165</td>
<td>729</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>YCSR dataset</td>
<td>1812</td>
<td>571</td>
<td>518</td>
<td>283</td>
<td>116</td>
<td>102</td>
<td>24</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

![Fig. 14. FNPKN of native HDFS, HAR, and FMP-SSF when storing YPCR dataset.](image)

![Fig. 15. MSPF of native HDFS, HAR, and FMP-SSF when downloading 100 files of YPCR dataset.](image)
index file strategy, which loads index files in the memory of NameNode, results in additional overhead to NameNode.

The results of storage efficiency experiments when storing YCSR dataset are similar, which are shown in Table 2.

In addition, for HAR, the storage efficiency of BPSR dataset is better than that of YCSR dataset. This is because the most files in BPSR dataset are very small (as shown in Table 1, 35,816 files, accounting for 85% of BPSR dataset, are under 0.5 MB). Therefore, each HAR file has more original files for BPSR dataset, and FNPKN of BPSR dataset is greater than that of YCSR dataset.

6.4.2. Access efficiency experiments of logically-related small files

Since different file correlation prediction algorithms have different accuracy rates, for FGP-LSF, access efficiency experiments do not focus on specific file correlation prediction algorithms, but evaluate the performances in two cases: the best prefetching situation (i.e., for correlated file prefetching, the next requested files have always been prefetched) and the worst prefetching situation (i.e., the next requested files always miss in cache).

(1) BPSR dataset: Fig. 17 shows MSPF of native HDFS, HAR, and FGP-LSF in the best prefetching situation of BPSR dataset.

When client number is 1, MSPF of native HDFS is 54.81. FGP-LSF reduces this by around 83%, which is 11.43 and is mainly profited by its three-level prefetching and caching strategy. MSPF of HAR is 66.74, which is about 122% compared with that of native HDFS. Similarly, when concurrent client numbers are 2, 4, and 8, for FGP-LSF the reductions of MSPF are 79%, 74%, and 70%, and for HAR the additions of MSPF are 18%, 7%, and 11% respectively.

Figure 18 shows MSPF of native HDFS, HAR, and FGP-LSF in the worst prefetching situation of BPSR dataset.

When client number is 1, MSPF of native HDFS is 71.26. Although correlated file prefetchings always miss, FGP-LSF still reduces MSPF by around 10%, which is 64.48, mainly due to the metadata caching and index prefetching. MSPF of HAR is 85.91, which is about 121% compared with that of HDFS. Similarly, when concurrent client numbers are 2 and 4, for FGP-LSF the reductions of MSPF are 12% and 8%, and for HAR the additions of MSPF are 10% and 3% respectively. However, when concurrent client number is 8, MSPF of FGP-LSF is about 182% compared with that of native HDFS. The performance deteriorations are caused by high concurrency of requests and useless prefetching activities.

(2) YCSR dataset: Figs. 19 and 20 show MSPF of native HDFS, HAR, and FGP-LSF in the best and worst prefetching situations of YCSR dataset.
correlated file prefetching (i.e., only metadata caching and index prefetching can still reduce requests, performance deteriorations are caused. In order to verify

In the case of the worst prefetching and high concurrency of situations of

Table 4

<table>
<thead>
<tr>
<th>Concurrent clients</th>
<th>MSPF of HDFS</th>
<th>MSPF of HAR</th>
<th>MSPF of FGP-LSF</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>66.25</td>
<td>81.86 (127% compared with MSPF of HDFS when client number is 1)</td>
<td>20.06 (24% compared with MSPF of HDFS when client number is 1)</td>
</tr>
<tr>
<td>2</td>
<td>67.98</td>
<td>86.49 (127% compared with MSPF of HDFS when client number is 2)</td>
<td>22.67 (26% compared with MSPF of HDFS when client number is 2)</td>
</tr>
<tr>
<td>4</td>
<td>76.92</td>
<td>93.46 (121% compared with MSPF of HDFS when client number is 4)</td>
<td>28.34 (30% compared with MSPF of HDFS when client number is 4)</td>
</tr>
<tr>
<td>8</td>
<td>100.21</td>
<td>116.54 (116% compared with MSPF of HDFS when client number is 8)</td>
<td>41.59 (36% compared with MSPF of HDFS when client number is 8)</td>
</tr>
</tbody>
</table>

7. Conclusion

In this paper, small file problem of HDFS is studied in details. Some contributions are outlined as follows:

Firstly, criteria for quantifying storage and access efficiencies for HDFS are defined. Particularly, since the memory available on NameNode is the primary limitation of HDFS scalability, File Number per kB of Memory of NameNode (FNPKN), is used to quantify the storage efficiency of HDFS in the first instance.

Next, in the context of HDFS, it can be concluded that 4.35 MB is the size set as the cut-off point between large and small files. This is proved by experimental results. It solves a critical issue of small file problem, i.e. ‘how small is small’, which to our best knowledge has not been done previously.

Thirdly, in contrast with previous research which provide either general solutions or special solutions to meet particular scenarios, this paper classifies files into three types. In addition, FMP-SSF and FGP-LSF are introduced for structurally-related and logically-related small files respectively. The experiments demonstrate that FMP-SSF and FGP-LSF adopted in our system can improve storage efficiency by factors of 9 and 2 respectively, and reduce access latency by 8–83%, compared with native HDFS. The reasons for that could be:

1) In order to improve the storage efficiency of HDFS, previous research mainly adopted file archiving (Hadoop Archives, 2011; Mackey et al., 2009; Shen et al., 2010) or key-value pairs (SequenceFile, 2011; MapFile, 2011) to reduce the number of files NameNode handles. Whereas, in this research, file merging strategy and file grouping strategy are adopted in FMP-SSF and FGP-LSF. Particularly, it is worthwhile noting that file correlations are considered for merging or grouping files. This has two advantages: (1) disk arm movements are reduced when files are requested; and (2) three-level prefetching and caching strategies can work at their best. FMP-SSF is similar to file archiving, but there are two main differences: (1) file correlations are considered when merging files in FMP-SSF; and (2) the local index file strategy is used in FMP-SSF. FGP-LSF is similar to HDWebGIS (Liu et al., 2009), but

The results of experiments in the best and worst prefetching situations of YCSR dataset are similar to those of BPSR dataset, which are shown in Tables 3 and 4.

(3) Access efficiency experiments without correlated file prefetching:
In the case of the worst prefetching and high concurrency of requests, performance deteriorations are caused. In order to verify that the metadata caching and index prefetching can still reduce access latency, access efficiency experiments without the use of correlated file prefetching (i.e., only metadata caching and index prefetching work) are carried out when concurrent HDFS client number is eight. The results are shown in Fig. 21.

For BPSR dataset, an MSPF of FGP-LSF is about 90% compared with that of native HDFS, due to the metadata caching and index prefetching. Similarly, for YCSR dataset, MSPF of FGP-LSF is also about 90% compared with that of native HDFS. Therefore, the results demonstrate that FGP-LSF without correlated file prefetching still improves access efficiency even in the situation of high concurrency and low accuracy of file correlation predictions.

In conclusion, for logically-related small files, FGP-LSF improves storage efficiency by about 200%, compared with native HDFS. HAR works better than FGP-LSF. This is because although the file grouping strategy relieves the memory consumptions of NameNode, its global index file strategy results in additional overhead to NameNode. As for access efficiency, the performances strongly depend on the accuracy of file correlation predictions. In the best prefetching situation, FGP-LSF reduces download times by 64–83%. In the worst prefetching and low concurrency situation, FGP-LSF reduces download times by 8–26%. However, when concurrent HDFS client numbers are eight, the download times increase to 140% (for YCSR dataset) and 182% (for BPSR dataset) in the worst prefetching situation. Even so, in this condition, FGP-LSF without the use of correlated file prefetching still reduces download times by about 10%. In contrast, HAR always deteriorates access efficiency in these cases.
which greatly improved the quality of the paper. Laboratory for Manufacturing Systems Engineering. The authors acknowledge the Fundamental Research Funds for the Central Universities Information Technology under Grant No. MOE-INTEL-2012-04, the MOE-Intel Special Research Foundation of China. The National Key Technologies R&D Program, the National Key Technologies R&D Program of China under Grant No. 2009BAH16F02, the MOE-Intel Special Research Foundation of China under Grant No. MOE-INTEL-2012-04, the Fundamental Research Funds for the Central Universities under Grant No. xj20100057, the Open Funding of State Key Laboratory for Manufacturing Systems Engineering. The authors are grateful to the anonymous reviewers for their comments which greatly improved the quality of the paper.

Acknowledgments

This research was partially supported by the National Science Foundation of China under Grant No. 60825202, 60921003, 91118005, 61070072, and 61103239, Cheung Kong Scholar’s Program, the National Key Technologies R&D Program of China under Grant No.2009BAH51B00, 2011BAK08B02 and 2012BAH16F02, the MOE-Intel Special Research Foundation of Information Technology under Grant No. MOE-INTEL-2012-04, the Fundamental Research Funds for the Central Universities under Grant No. xj20100057, the Open Funding of State Key Laboratory for Manufacturing Systems Engineering. The authors are grateful to the anonymous reviewers for their comments which greatly improved the quality of the paper.

References


HDSWebGIS is limited to geographic correlations, and files are grouped consecutively in terms of their geographic locations, while FMP-SSF is suitable for logically-related small files.

(2) As for access efficiency, previous research mainly focused on the use of index technology for HDFS (Liu et al., 2009; Shen et al., 2010; MapFile, 2011). In addition, predictive data grouping (Essary and Amer, 2008) was provided to reduce disk arm movement, thereby simultaneously reducing energy consumption and data access latency. Server-driven metadata prefetching (Hendricks, 2006) was introduced to improve the efficiency of small files in object-based storage systems. In this paper, three-level prefetching and caching strategies are introduced to improve the access efficiency of small files. The strategies can reduce the per-file metadata server interactions, decrease the I/O cost that DataNodes fetch files from disk, and reduce the latency of transferring files.

In future work, the reason and theoretical formula of cut-off points between large and small files will be further studied. In addition, the relationship among storage and access efficiencies and the size of a merged file/logic unit will be investigated, with the aim of optimizing the file merging/grouping strategy and the peak size of a merged file/logic unit.

Fig. 21. MSPF when concurrent client number is eight and without correlated file prefetching. (a) BPSR dataset and (b) YCSR dataset.