The Journal of Supercomputing (2021) 77:7149–7170
https://doi.org/10.1007/s11227-020-03526-1

Enhancing HDFS with a full-text search system for massive small files

Wentao Xu[1] · Xin Zhao[1] · Bin Lao[3] · Ge Nong[1,2]

Accepted: 17 November 2020 / Published online: 4 January 2021

Abstract

HDFS is a popular open-source system for scalable and reliable file management, which is designed as a general-purpose solution for distributed file storage. While it works well for medium or large files, it will suffer heavy performance degradations in case of lots of small files. To overcome this drawback, we propose here a system to enhance HDFS with a distributed true full-text search system SAES of 100% recall and precision ratios. By indexing the meta data of each file, e.g., name, size, date and description, files can be quickly accessed by efficient searches over metadata. Moreover, by merging many small files into a large file to be stored with better space and I/O efficiencies, the negative performance impacts caused by directly storing each small file individually are avoided. An experimental study is conducted for function and performance tests on both realistic and artificial data. The experimental results show that the system works well for file operations such as uploading, downloading and deleting. Moreover, the RAM consumption for managing massive small files is dramatically reduced, which is critical for good system performance. The proposed system could be a potential storage solution for massive small files.

Keywords Lots of small files · HDFS · Elasticsearch · Full-text search

The work of X. Zhao was done in his Master program in Sun Yat-sen University.

✉ Ge Nong
issng@mail.sysu.edu.cn

Wentao Xu
xuwt7@mail2.sysu.edu.cn

Xin Zhao
zhaox79@mail2.sysu.edu.cn

Bin Lao
laobin@gdufs.edu.cn

1 School of Computer Science and Engineering, Sun Yat-sen University, Guangzhou, China

2 Guangdong Province Key Laboratory of Information Security Technology, Guangzhou, China

3 School of Information Science and Technology, Guangdong University of Foreign Studies, Guangzhou, China

1 Introduction

Data as lots of small files (LOSF) such as pictures, logs and emails are widely observed in modern information applications. These small files are of typical sizes from several thousands to millions bytes. Efficient storage of massive small files requires the underlying file system to scale with increasing number and volume of files. However, it has been recognized that existing general-purpose file systems are commonly designed without specific concerns for small files. This inspires R&D efforts contributed to LOSF systems, the existing solutions are roughly classified into three categories [28] as follows.

1.1 Existing solutions

The first category is the proprietary distributed file systems developed by a number of institutes, e.g., Taobao File System (TFS)[1] from Taobao and Cassandra [16] from Twitter. Specifically, TFS is designed for managing small files less than 1 MiB, and Cassandra maintains the meta data of all files in RAM to speed up file seeking. Because these systems are designed with specific objectives to support applications in these institutes, efforts are required to deploy them in a third-party environment. As a result, these systems have not been widely adopted in the public domain.

The second category is to optimize existing frameworks of managing small files [13]. For example, Priyanka et al. [26] designed a mechanism called CombineFileInputFormat to improve the performance of accessing massive small files based on MapReduce framework, then Chang Choi et al. [6] integrated CombineFileInputFormat with Java Virtual Machine (JVM) to run multiple mappers on a single JVM for reducing JVM creation time and improving MapReduce's processing performance for small files. In addition, some researchers have modified the file managements of operating systems to scale with massive small files [9]. While these systems provide stronger supports for small files, their designs are tightly coupled with the underlying systems such as JVM and OS [5, 10, 15, 20]. This makes difficulties for deploying these systems in practice.

Different from the aforementioned solutions, the third category uses a general-purpose distributed file system by compacting many small files into a large file to reduce the number of physical files stored in the underlying file system. For example, HDFS provides three mechanisms for storing small files mainly for read-only accesses: (1) Hadoop Archive (HAR)[2] for read-only archive purpose, which merges multiple small files into a large file with meta and file data. Once a HAR is created, it is frozen and can not be modified any more. (2) SequenceFile[3] for storing many small files as a sequential stream of key-value records, where key and value for meta and file data, respectively. The small files in a SequenceFile can only be accessed

[1] https://github.com/alibaba/tfs.

[2] https://hadoop.apache.org/docs/current/hadoop-archives/HadoopArchives.html.

[3] https://hadoop.apache.org/docs/r2.7.5/api/org/apache/hadoop/io/SequenceFile.html.

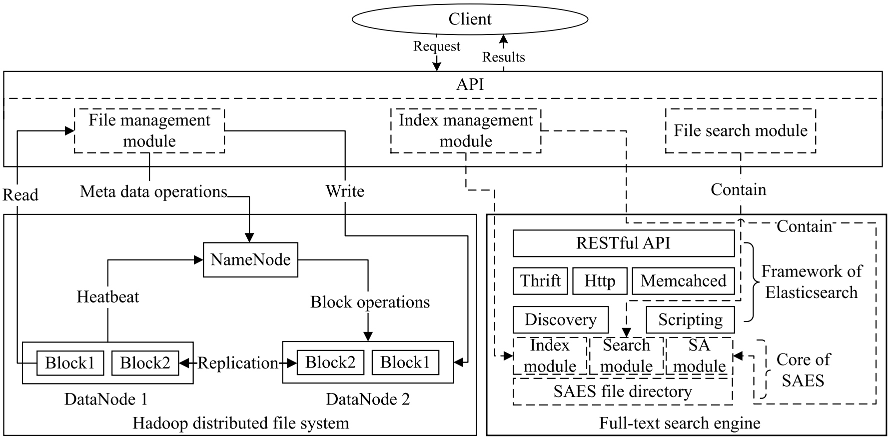

Fig. 1 The architecture of our proposed system for massive small files, where the parts in dashed boxes are developed by this work to enhance HDFS with SAES

sequentially. (3) MapFile[4] built on SequenceFile for faster accesses to small files using an index for meta data of small files. The index is loaded into RAM and searched for locating each small file to be accessed. Specifically, all keys are sorted, then 1 per 128 keys is selected to build the index. MapFile can be considered as an accelerated alternative of SequenceFile.

Among these three categories, the third turns out to be more promising for the development of a general-purpose file system for massive small files. Currently, Hadoop and HDFS are the very popular open-source software for developing distributed data processing and storage systems [4, 27, 31], HDFS is the choice of technology for building the underlying file system. The three mechanisms currently provided by HDFS for storing small file are mainly for read-only archive purpose, random access to small files are not supported very well. Among them, MapFile provides the strongest support for random access. However, the index used by MapFile to store keys for locating small files contains only a part of all keys, and it is required to be fully loaded in RAM for searches. This index scheme might be improved by another scalable one for managing massive small files with flexible random operations such as adding, deleting and updating.

1.2 Our solution

A typical HDFS configuration is shown in the down-left box of Fig. 1, which includes: (1) a master node called NameNode to manage the namespace composed of meta data; and (2) multiple slave nodes called DataNodes to store files as blocks. To store a file, the metadata for accessing this file from DataNodes is added to NameNode, and the file is segmented into blocks to be stored in DataNodes [22, 30].

[4] https://hadoop.apache.org/docs/r2.6.2/api/org/apache/hadoop/io/MapFile.html.

While HDFS is efficient for managing medium to large files of sizes over ten MiB, it sees challenges for managing massive small files due to that both time and space requirements for file management increase with the number of files. For massive small files, a heavy burden is put on NameNode for maintaining the file management information in RAM, which brings negative performance impacts to NameNode and eventually becomes a performance bottleneck of the whole system.

With the default configuration of HDFS, the metadata maintained in RAM of NameNode consists of 250 bytes per file and 368 per block of 3 replicas [3, 33]. Each small file occupies a block, managing one million small files will consume $(250 + 368) \times 10^6/2^{20} = 589.3$ MiB RAM in NameNode, i.e., 2 million files will require about 1 GiB. Different from years ago, the cost of RAM has been reduced remarkably, and a large RAM such as 64 GiB can be reasonably assumed for NameNode. Even though the RAM's capacity can be increased, file management operations in NameNode will become slower when the number of files grows, and efficient managing massive small files on HDFS can not be done by simply using a NameNode of large RAM. To meet the demand of efficient storage of massive small files, our recent attempt for enhancing HDFS with a full-text search system called SAES is reported here.

The key idea for us to develop the enhanced system is to merge many small files into a big file, and use a full-text search system to manage the meta data such as name, date, size, description for file access. Given that the full-text search system is efficient enough to pace up with HDFS, we will have an enhanced HDFS with performance independent of file sizes, i.e., universally efficient for small, medium and big files. Currently, Elasticsearch[5] with Lucene as search engine plays a key role for text search. Because Lucene uses the inverted index of data, searches in Elasticsearch are not true full-text, in terms of that searches are performed over predefined words instead of all data. Some results may not be found even if it exists and the recall ratio is not guaranteed to be 100%. For example, a sentence "This is a book" can be divided into a word set {This, is, a, book} by the standard tokenizer of Elasticsearch, and each word is added to the inverted index. Later on, this sentence can be found by searching these words over the inverted index. However, indexing words in this way can support searches over full words only, i.e., searches over partial words are not supported. For instance, the sentence can not be found by searching the substring "ok" of "book", because "ok" has not been added as a word to the inverted index. As a resort to support full-text searches, Elasticsearch may use the N-gram tokenizer[6] instead of the standard one, in the expensive cost of dramatically increased index size. This tokenizer can extract from data all N-grams with a given size as words to be indexed. However, a size-n text with gram size k will produce $O(n)$ size-k words to be indexed, the required space $O(nk)$ is too much to be employed for supporting full-text searches over massive data.

Currently, it is challenging for Elasticsearch with inverted index to support time and space efficient true full-text searches. Such a drawback prevents Elasticsearch

[5] https://www.elastic.com/cn/blog/elastic-search-7-2-0-released

[6] https://www.elastic.co/guide/en/elasticsearch/reference/current/analysis-ngram-tokenizer.html.

from managing meta data of massive files, since both precision and recall ratios of 100% are required for purpose of file management. Recently, we built a true full-text search system called SAES by replacing the inverted index in Elasticsearch with a suffix index. Given a size-n text, the suffix index requires a space of $O(n)$ only. Using the suffix index, SAES supports efficient exact and approximate full-text searches demanded for file management. Provided with HDFS and SAES, we need to integrate them for working together to access a file in two steps: (1) locating the file by SAES to search answers for the given query; (2) access the located file by HDFS.

1.3 Contributions

This article presents our work on the R&D of a scalable distributed file system for managing massive small files. The contributions of this work mainly consist of two parts: (1) A system architecture is proposed to enhance HDFS with SAES by adding an integration layer on top of HDFS and SAES. Such an architecture allows both HDFS and SAES to evolve independently, so as to avoid possible fatal software engineering problems caused by the future developments of HDFS and SAES. (2) An experimental prototype system is built for functionality verification and performance evaluation. A set of experiments with realistic and artificial data are conducted on this prototype system to assess the feasibility of our proposed solution for efficient storage of massive small files.

2 Our system architecture

Figure 1 shows the architecture of our proposed system for managing massive small files. On top of HDFS and SAES, an integration layer of three modules for file management, index management and file search is added to provide application programming interface (API) for clients to access the system's services. A file access request from the client is processed by the file search module to produce searching tasks on SAES to find the file's metadata, then the metadata is supplied to the file management module to locate and access the file stored in DataNodes. When a file is added or deleted, the index management module produces index updating tasks to be executed in SAES. In this way, SAES serves as the middleware in between the client and HDFS, which translates a request of file access into the tasks performed on HDFS. SAES provides true full-text searching capability necessary for file management in this system. Such a searching capability is fundamental for the system to function correctly. To support full-text searches, a suffix index for the meta data of all files is maintained in real-time. When a file is added or deleted, the index is updated on-the-fly to track the file's status for management.

Figure 2 shows the file storage scheme for merging many small files into a large file. Each large file serves as a logical disk of many blocks. The occupation status of each block in a large file is tracked by the file management module. When a file is added, depending on the file's size, one or multiple idle blocks are allocated to store

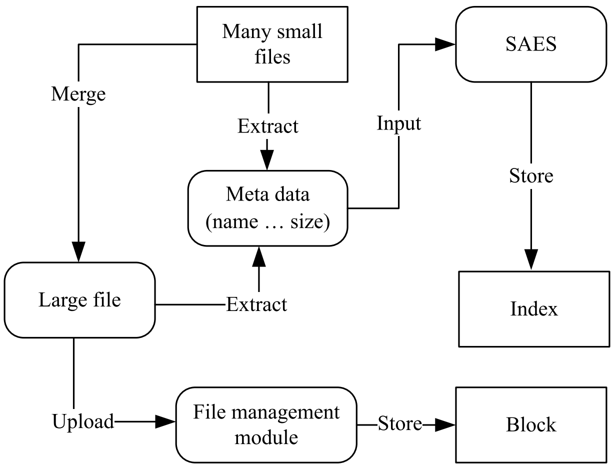

Fig. 2 The storage scheme for merging many small files into a large file

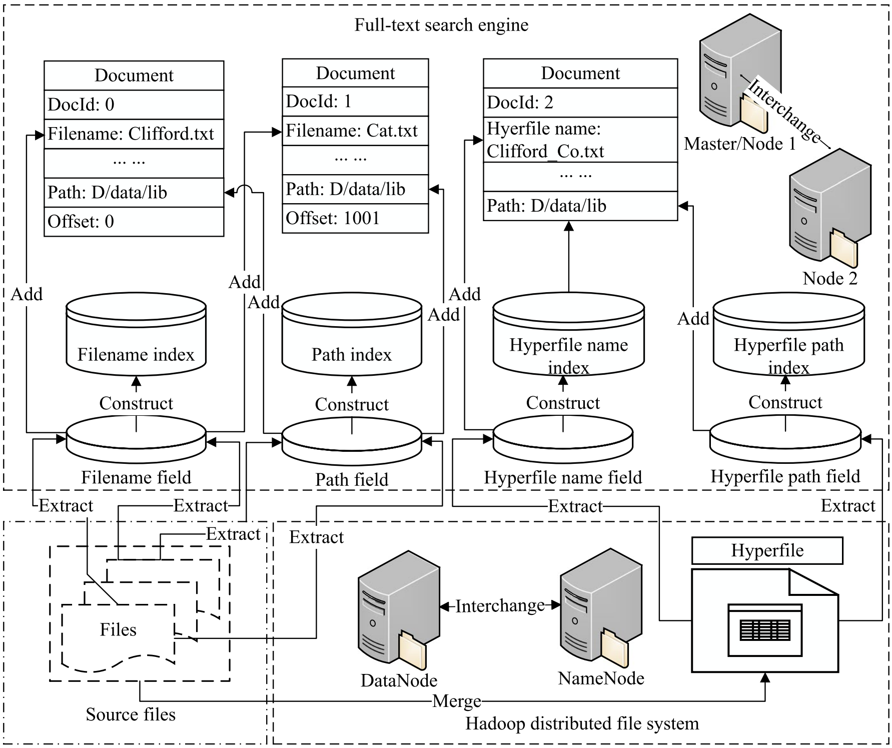

Fig. 3 The data flow for adding a file in our proposed system, where the upper part is processed by the full-text search engine while the functionalities in the lower part are provided by HDFS

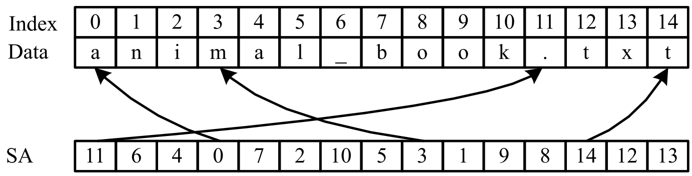

Fig. 4 An example suffix array for data "animal_book.txt"

the file; when a file is deleted, the file is marked as obsolete and its blocks are freed by periodically reorganization of large files. In order to avoid too many fragments in a large file for better space utilization, the blocks of each large file are reorganized periodically to cluster idle block together.

For more details of operations in the system, Fig. 3 shows the data flows for adding files. Basically, files can be added one by one, but adding files in this way is slow due to the communication delay for uploading each file. To speed up the process for uploading multiple files, these files can be merged as a group to be uploaded as a whole to reduce the total communication delay. In this figure, some files are organized as a group, and some files are uploaded individually. A document to be indexed is produced by extracting the meta data of each file, which consists of a set of fields such as DocID, Filename, Path, Offset, etc. To facilitate searching a field for file access, an index is dynamically built on documents for the field. SAES allows the flexible definitions of metadata for a file, saying that any field can be added, removed or indexed at anytime. With this distinct advantage, our system provides friendly user interface for accessing files by exact or approximate searches on the indexed fields of metadata. In the rest of this section, we further explain how SAES is integrated with HDFS and the processes for file operations, i.e., indexing, uploading, downloading and deleting.

2.1 SAES system

The architecture of SAES is shown in the down-right box of Fig. 1, which uses suffix index for true full-text searches instead of inverted index for keyword searches in the original Elasticsearch. The three upper layers are inherited from Elasticsearch, and the two lower layers are revised for using suffix index. We further explain how the suffix index is built for metadata of files.

The index management module interacts with the index and suffix array (SA) modules in the service layer to maintaining the suffix index. When a file is added, a document for the file's metadata is produced by the index management module for indexing. If this file is deleted, the document is removed from the index. The suffix index consists of the SA for each indexed field of document. The SA of each field is built by lexicographically sorting all suffixes of the field data in each document, which was initially proposed in [19] for online string searches, and has become a fundamental index data structure for full-text searches [2, 11, 29]. Figure 4 shows a suffix array example, where an arrow from the SA points to the corresponding suffix in data. Given the SA of each field, searching on the

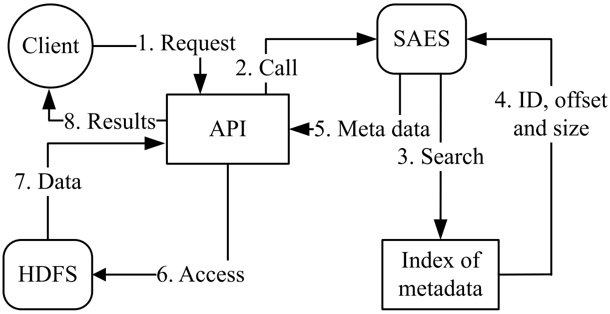

Fig. 5 The process for retrieving a small file via the system API

field is done by performing binary searches on the sorted suffixes in SA. Exactly searching a size-m substring in the size-n data is equivalent to finding all the suffixes in data starting with the substring, which takes $O(m \log n)$ time, given that a searching comparison needs to compare up to m characters.

Efficient construction of SA is critical for the usability of suffix index, many efficient suffix sorting algorithms have been proposed for different computing models by intensive researches during the past two decades, see [1, 7] for a quick survey. In particular, using the tools proposed in [12, 14, 17, 21, 23, 24, 32], SAES is capable of achieving an index building speed over millions bytes per second, which is fast enough to pace up with file accesses to HDFS in most applications. Provided with the suffix index, SAES performs exact or approximate full-text searches to locate files satisfying each client's request, then the located file is accessed by HDFS to execute the related file management operations.

2.2 File indexing

The metadata for each file is currently defined as {name, bigfile, offset, size, path, date}, and each element of metadata is a field to be indexed. Using these fields' indexes, the system can locate a file by exact or approximate searches over meta data, e.g., to access a file by its name, size, date or their combination. Given a file access request, locating the target files is done in two steps: (1) SAES conducts searches to find the IDs of files matching the request; and (2) find the locations of matching files by IDs for retrieving the files from HDFS.

In more detail, Fig. 5 shows the process for retrieving a small file via the system API. For each file, its metadata is indexed and searched by SAES for accessing files according to the client's request. For each small file x, a record {ID, offset, size, Info} is indexed with ID as key, where ID refers to the host hyperfile storing the small file, offset and size give the position and length of x in the hyperfile, respectively. The host hyperfile is located in HDFS by its ID, and x is further accessed by offset and size.

2.3 File uploading

HDFS manages its storage space in units of blocks and the typical default block size L is 128 MiB. Depending of file size, a file may occupy one or multiple blocks, and a part of tail block may not be used and wasted. One way to reduce the wasted space of a block is to merge small files into a hyperfile. Moreover, the utilization status of each hyperfile is dynamically tracked when small files are added to or deleted from the hyperfile. For hyperfiles with wasted space more than a predefined threshold, all their remaining small files are merged immediately or periodically to produce new hyperfiles with high utilizations, and the index is updated accordingly.

In practice, multiple new files likely arrive as a batch to the system, these files can be packed as a group to speed up the uploading process. Specifically, the maximum size of uploading group is set as L, i.e., the block size in HDFS, a file will be directly uploaded if it is not smaller than L, or else it will be merged with its succeeding files with the maximum merged size not more than L. Once a group has been received by the system, the files in group are unpacked and stored in one or multiple hyperfiles, and the metadata of files are indexed for searches.

A greedy merging method is employed by Algorithm 1 for adding small files to hyperfiles. The space usage of current under-utilized hyperfile H is recorded, and F is the queue of newly arriving files. Whenever F is non-empty, the hyperfile H is checked to see if its free space is enough to accommodate the head file of F. If it is, the file is stored in the hyperfile and the hyperfile's occupation status is updated, or else a new hyperfile is created to store the file. The under-utilized hyperfiles are periodically merged by system maintenance jobs for better space efficiency.

Algorithm 1 Adding queuing files

```
Require: F is not empty;
while F is not empty do
    Let H be the current under-utilized hyperfile;
    Let added=0;
    if H can accommodate the head file of F then
        Take the head file from F and store it into H;
        Update the occupation status of H;
        Let added = 1;
    end if
    if added=0 then
        Create a hyperfile to store F;
    end if
end while
```

Using this greedy merging method, a file not less than L will cause the creation of a hyperfile to store it, and a small file will likely be merged into the current under-utilized hyperfile for better space utilization. Even though this greedy method is not optimal, it works well in our experiments by significantly improving the space utilization of HDFS for massive small files.

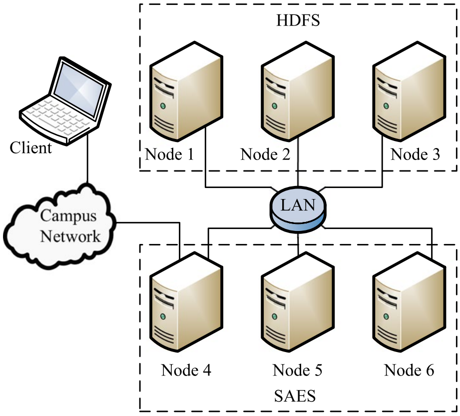

Fig. 6 The topology of our experimental platform, where the LOSF system is installed in a LAN, and the client communicates with the LOSF via campus network

Table 1 Configurations of server and client nodes on our experimental platform

Node	Node type	RAM (GiB)	Disk (TiB)	CPU	Operating system	JVM	
						-Xmx (GiB)	-Xms (GiB)
1	Storage	16	1.8	1 of 4 cores	CentOS 6.6	4	0.2
2	Storage	8	1.8	1 of 4 cores	CentOS 6.10	2	0.1
3	Storage	32	1.8	2 of 12 cores	CentOS 6.7	8	0.5
4	Index	64	0.5	2 of 12 cores	CentOS 6.8	16	1.0
5	Index	16	1.8	1 of 4 cores	CentOS 6.7	4	0.2
6	Index	16	2.0	1 of 4 cores	CentOS 6.8	4	0.2
7	Client	8	0.6	1 of 4 cores	Windows 10	2	0.1

2.4 File downloading and deleting

The client can issue a file downloading request by exact or approximate queries on the indexed meta data of files, i.e., name, size, date, etc. The query is executed by SAES to find the access information of each file meeting the query, and the access information of found files is returned to the client. Given the access information of a file, HDFS can quickly locate and retrieve the file stored in a hyperfile.

The process for deleting a file consists of two steps: (1) logically deleting the file by marking it as obsolete in the file's management record; and (2) physically deleting the obsolete file by releasing its occupied space. Specifically, given the name of a file to be deleted, the file's ID is found by SAES, then the file's management record is found by the ID and updated to mark the file as obsolete. Moreover, the system periodically executes maintenance jobs to clean up obsolete files in hyperfiles and merge hyperfiles as needed for better space utilization.

3 Experiments

Both HDFS and SAES are developed in Java, and our software system is also developed in Java as an application running on SAES and HDFS. The software is deployed on our experimental platform with network topology shown in Fig. 6, and the configurations of servers and client are given in Table 1. Three servers are used to build the HDFS of a NameNode and three DataNodes, i.e., Node 1 is used as NameNode and DataNode, Node 2 and 3 are used as DataNodes, and the small files are stored in DataNodes. Moreover, Nodes 4, 5 and 6 are used to build the SAES for indexing the meta data of small files. The system services are accessed by the client via Node 4. In particular, Node 4 accepts job requests from the client, produces the tasks for each request and assigns the tasks to their destined nodes for processing, then collects the processing results to respond to the client.

A series of experiments were conducted for function verification and performance evaluation of our experimental system on both realistic and artificial datasets. The functions to be verified are file uploading, downloading, deleting and updating, the performance to be evaluated are time for retrieving files and memory consumption for NameNode of HDFS. Table 2 shows the realistic data composed of texts and pictures. For evaluating memory consumption of NameNode, we adopt the far more larger artificial data of text, picture and XML files generated by a model built on statistics collected from realistic data. Specifically, the text, XML and log files are produced with file size distribution as datasets novel, annotations and logs, respectively, and the picture files are produced with file size distribution as datasets coco, voc2009 and pet. The artificial dataset comprises one million files with size distribution shown in Fig. 7, where the volume percentages for picture, text, XML and log files are 89.41%, 9.78%, 0.07% and 0.75%, respectively.

3.1 File Storing and Searching

Table 3 shows the experimental results for verifying whether the system can properly store and search small files. All files in dataset "pet" are uploaded to the system, then some exact or approximate queries are issued to retrieve files by name, data or size for verifying with the original ones. For each query, the search time is for the system to search all matching files, and the download time is for the client to download all found files one by one.

After uploading the dataset, 6 hyperfiles are generated and stored in HDFS, which is consist with the merging scheme. Moreover, the number and size of files are checked to be correct. In order to test the correctness of file indexes, the queries listed in Table 3 are submitted to search files. For each query, the found files are checked to be identical with our statistics on the dataset. All the files for each query are found correctly, both recall and precision ratios for each search are 100%, saying that a file is in the searching results if and only if it meets the query. For example, the first approximate query is to find each file of name with "beagle" as prefix and ".jpg" as suffix, and a total of 200 files are found, which consists of all the matching

Table 2 Realistic datasets used in the experiments

Dataset	Type	Files	Total size (MiB)	Mean size (KiB)	Description
coco [18]	Picture	532856	83865.6	161.2	Coco is a large-scale object detection and segmentation dataset, at http://cocodataset.org/#download
novel	Text	1052	699.0	680.4	English novel, at http://novel.tingroom.com
voc2009 [8]	Picture	11321	857.0	77.5	Visual image data, at http://host.robots.ox.ac.uk/pascal/VOC/voc2009
annotations [8]	XML	4809	10.8	2.3	Annotation for the voc2009 database, each xml file gives the details of each image
pet [25]	Picture	7393	759.6	105.2	Oxford image data, at http://www.robots.ox.ac.uk/~vgg/data/pets
logs	Log	90	53.7	611.0	Log files, at https://codeload.github.com/kevinaangstadt/quadcopter-logs/zip/master

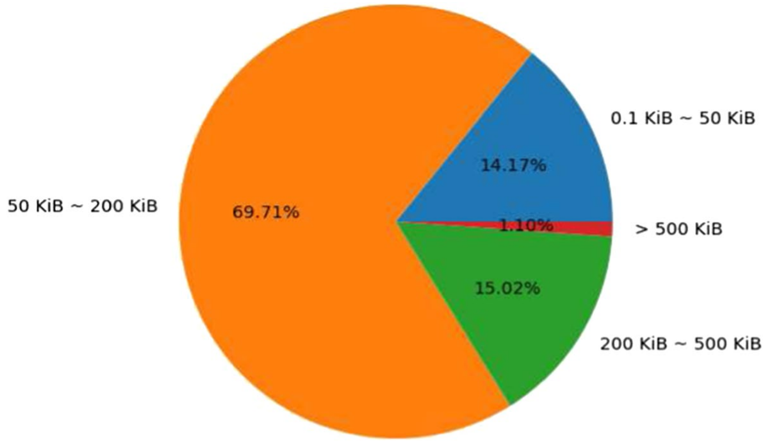

Fig. 7 The size distribution of one million files in the artificial data

files in dataset. For all queries, the searching time do not vary much, but the download time are dependent of volumes of found files. The 1st and 3rd queries have much faster mean download speeds than the other two, because their mean sizes of found files are much larger. Given that the round-trip communication delay for file download control is almost fixed, a larger file will see a faster mean download speed.

3.2 File deleting and updating

Table 4 shows the results for deleting 3 files. First, the file of name "2007_000676.xml" is deleted from dataset "annotations". Before the deletion, all files in dataset "annotations" were uploaded on December 31, 2019, and there is only one file named "2007_000676.xml" in all datasets. The searches in top two rows are performed to verify this deletion: (1) the exact searches at row 1 for file name "2007_000676.xml" find 1 and 0 matching file before and after deletion, respectively; (2) the exact searches at row 2 for "20191231" find 7818 and 7817 files before and after deletion, respectively. This confirms that one file has been deleted as expected. Next, the files of name prefix "newfound" and suffix "jpg" are deleted from dataset "pet". Before the deletion, all 200 files of name prefix "newfound" and suffix "jpg" are from dataset "pet" uploaded on December 30, 2019. The searches at rows 3 and 4 verify these deleting operations, i.e. 200 out of 7393 files are removed. Similarly, the tests in last 3 rows are observed to work correctly.

Updating a file can be done by deleting the file and then uploading a new file of the same name, our system also provides the updating function. Table 5 shows the experimental results for testing file updates. First, rows 1 and 2 delete files of names "The Story-book of Science.txt" and "Flowers of the Sky.txt", respectively. The index information for each file before and after updating show that the file has been updated as expected, i.e., both file size and date are modified accordingly. Next, 17 files in dataset "novel" are updated. When the files in "novel" were uploaded to the system, they were merged to be stored in a hyperfile "MyLife.

Table 3 Retrieving files in dataset "pet" by exact or approximate queries on metadata

Query	Search field	Found files	Files in dataset	Search time (s)	Download time (s)	Mean size of found files (MiB)	Mean download speed (MiB/s)
beagle*.jpg	Name	200	200	0.61	15.17	0.11	1.51
Abyssinian_66.jpg	Name	1	1	0.28	1.90	0.02	0.01
20191230	Date	7393	7393	0.51	1304.51	0.10	0.58
66109	Size	1	1	0.30	1.92	0.06	0.03

Table 4 Verification experiment for file deleting operations

Query	Dataset	Search field	Deleted files	Deleting criterion	Matching files before deleting	Matching files after deleting
2007_000676.xml	Annotations	Name	1	2007_000676.xml	1	0
20191231	Annotations	Date	1	2007_000676.xml	7818	7817
newfound*jpg	Pet	Name	200	newfound*jpg	200	0
20191230	Pet	Date	200	newfound*jpg	7393	7193
Flowers of the Sky.txt	Novel	Name	1	Flowers of the Sky.txt	1	0
2009_005299.jpg	voc2009	Name	1	2009_005299.jpg	1	0
test1.log	Logs	Name	1	test1.log	1	0

Table 5 Verification experiment for file updating operations

Query	Dataset	Searching field	Matching files before deleting	Matching files after deleting	Index information	
					Before updating	After updating
The Story-book of Science.txt	Novel	Name	1	1	size: 517827B; date: 2020-01-02 11:26:49	size: 90227B; date: 2020-01-02 17:52:26
Flowers of the Sky.txt	Novel	Name	1	1	size: 338628B; date: 2020-01-02 11: 21:52	size: 136672B; date: 2020-01-02 17:52:26
MyLife.txt20200 104154634986	Novel	Hyperfile name	167	150	–	–
NewGrubStreet.txt 20200104155545693	Novel	Hyperfile name	0	17	–	–

txt20200104154634986". By updating, these 17 files are removed from this hyperfile shown at row 3, then their modified copies are uploaded and stored into a new hyperfile called "NewGrubStreet.txt20200104155545693" shown at row 4. The index information for hyperfile is invisible for client and shown here only for verification. The experiment results in this table confirm the correctness of updating operations.

3.3 File reorganizing

In our system, a hyperfile with space usage less than 50% is marked as underutilized. A file reorganization is triggered when the number of underutilized hyperfiles exceeds a given threshold or a periodical system maintenance schedule happens. For better space utilization, the threshold should not be too large and some values around ten are reasonable. Table 6 shows the experiments for file reorganization, where the thresholds for underutilized hyperfiles are ranging from 12 to 6. At row 1, the small files in 12 underutilized hyperfiles are merged to 2 new hyperfiles with the average space usage improved from 14.8 to 88.2%. For the other rows, significant space usages are also observed to be improved from around 35% to beyond 75%. The improvement for threshold 6 is substantially less than that for larger thresholds. The reason should be that more small files are stored in more underutilized files, and merging more small files will likely increase the space usage of a hyperfile.

Given the merging mechanism in our system, the number of hyperfiles after reorganization can be estimated as $\lceil (S_t - S_f)/L \rceil$, where S_t and S_f are the total and free sizes of hyperfiles for reorganization, respectively, and L is set as the default block size of 128 MiB. By this formula, the numbers of hyperfiles after reorganization are estimated as $\{2, 4, 4, 3\}$, which are consistent with the experiment results shown in the table and can be considered as a verification for the reorganizing operations.

3.4 Performance of NameNode

The NameNode can become a performance bottleneck of HDFS. When more files are stored in HDFS, more RAM are required in the NameNode to maintain file management information and the response for a file access will become slower. By merging multiple small files into a hyperfile, the number of files managed by HDFS is dramatically reduced. Consequently, the burden on HDFS is decreased and the NameNode can keep working efficiently. Figure 8 shows the RAM consumptions of managing metadata in NameNode with or without merging small files in the artificial dataset with size distribution given in Fig. 7, where the RAM consumptions are given in logarithmic scale. The gap between two lines is very large, i.e., merging small files can reduce the RAM consumption to be less than 1% of that in the original HDFS. Since the RAM consumption of metadata is reversely proportional to the performance of NameNode, HDFS in our system will see far more less negative performance impacts caused by small files, i.e., the system's performance can remain stable for files of various size distributions.

Table 6 Verification experiment for file reorganization

Hyperfiles for reorganization	Space usage (%)	Total size (byte)	Free size (byte)	Hyperfiles after reorganization	Space usage after reorganization (%)	Total size after reorganization (byte)
12	14.8	1603478582	1366743153	2	88.2	236735429
10	36.5	1325386336	834994735	4	91.3	490391601
9	33.6	1022208246	616521034	4	75.6	405687212
6	35.8	733230185	444818433	3	71.6	288411752

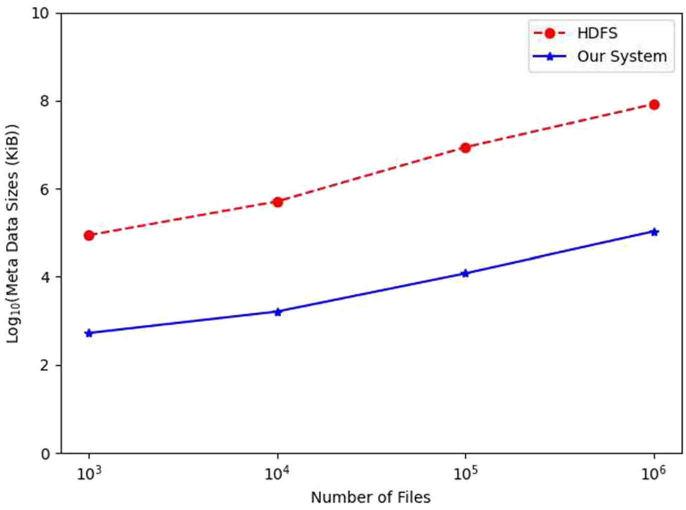

Fig. 8 The memory consumptions of metadata for NameNodes of HDFS with or without merging files

Table 7 Stress testing for concurrent searching and downloading files in dataset "pet"

Query	Operation	Threads	Throughput	Response time
Abyssinian_66.jpg	Search	20	115.0	0.1
		60	160.4	0.3
		100	169.5	0.5
		200	172.9	1.0
	Download	20	7.2	2.8
		60	8.0	6.9
		100	6.8	14.3
		200	4.0	48.1
Abyssi*66.jpg	Search	20	170.9	0.1
		60	187.5	0.3
		100	177.0	0.5
		200	162.6	1.1
	Download	20	3.7	5.4
		60	3.0	20.1
		100	4.1	24.4
		200	3.2	62.5

3.5 Stress testing

The Apache JMeter[7] is a widely used tool for stress testing, which is employed to generate concurrent tasks for searching and downloading files in dataset "pet".

[7] https://jmeter.apache.org/.

Table 7 shows the experimental results for retrieving files by names given as exact or approximate queries. The number of concurrent threads for searching a query and downloading the matching files varies from 20 to 200. The throughput is the average number of finished tasks per second, and the response time is the average execution time for each task. For each query, all the concurrent tasks were done successfully, i.e., no failed task was observed. The throughputs and response time are reasonable for our experiment platform, which are constrained by the network bandwidth between the client and system.

4 Conclusion

HDFS has seen rich successes as a scalable solution for distributed file storage. File access jobs in HDFS are typically streaming, saying that a batch of files instead of a single file are uploaded or downloaded. In the original HDFS, a heavy burden is put on the NameNode for managing lots of small files. In order to avoid the NameNode to be overloaded, we adapt a distributed full-text search system SAES recently developed in our laboratory for merging multiple small files into a large hyperfile to be managed as ordinary files by the NameNode. SAES is built for true full-text searches by replacing the inverted index in Elasticsearch with the suffix index and hence inherits the good scalability of Elasticsearch. The design of our experimental system for enhancing HDFS by SAES is presented in this article, and a series of experiments have been conducted for function verification and performance evaluation of this system. The experiment results show that the capability of SAES for true full-text searches is efficient enough to enhance HDFS for massive small files. Given the popularity of HDFS, instead of revising HDFS, an integration layer with three modules is developed to enhance HDFS by SAES, and our system is built as an application on HDFS and SAES. Such a system design allows HDFS and SAES to evolve independently, which helps reduce the burden for software engineering of our system. As a next step to apply our solution in practice, we are currently improving the system for higher performance, e.g., designing better algorithms for merging small files and refining the code. We hope that this work suggests a potential solution for enhancing HDFS to provide efficient storage for massive small files.

Acknowledgements This work was funded by the National Natural Science Foundation of China (Grant number 61872391), the Guangzhou Science and Technology Program (Grant No. 201802010011), and the Foundation for Young Talents in Higher Education of Guangdong, China (Grant number 2019KQNCX031).

References

1. Apostolico A, Crochemore M, Farach-Colton M, Galil Z, Muthukrishnan S (2016) 40 years of suffix trees. Commun ACM 59(4):66–73
2. Arroyuelo D, Bonacic C, Gil-Costa V, Marin M, Navarro G (2014) Distributed text search using suffix arrays. Parallel Comput 40(9):471–495

3. Chandrasekar A, Chandrasekar K, Ramasatagopan H, Rafica AR, Balasubramaniyan J (2012) Classification based metadata management for HDFS. In: HPCC 2012 and ICESS 2012
4. Chen G, Hu T, Jiang D, Lu P, Tan KL, Vo HT, Wu S (2014) BestPeer++: a peer-to-peer based large-scale data processing platform. IEEE Trans Knowl Data Eng 26(6):1316–1331
5. Chen Y, Zhou Y, Taneja S, Qin X, Huang J (2017) aHDFS: an erasure-coded data archival system for Hadoop clusters. IEEE Trans Parallel Distrib Syst 28(11):3060–3073
6. Choi C, Choi C, Choi J, Kim P (2016) Improved performance optimization for massive small files in cloud computing environment. Ann Oper Res 265(2):305–317
7. Dhaliwal J, Puglisi SJ, Turpin A (2012) Trends in suffix sorting: a survey of low memory algorithms. In: Proceedings of the Thirty-Fifth Australasian Computer Science Conference-Volume, vol 122, pp 91–98
8. Everingham M, Gool LV, Williams CKI, Winn J, Zisserman A (2009) The pascal visual object classes (VOC) challenge. Int J Comput Vis 88(2):303–338
9. Fu S, He L, Huang C, Liao X, Li K (2015) Performance optimization for managing massive numbers of small files in distributed file systems. IEEE Trans Parallel Distrib Syst 26(12):3433–3448
10. Gao Z, Qin Y, Niu K (2016) An effective merge strategy based hierarchy for improving small file problem on HDFS. In: 2016 4th International Conference on Cloud Computing and Intelligence Systems
11. Gupta S, Yadav S, Prasad R (2018) Document retrieval using efficient indexing techniques. In: Information retrieval and management, pp 1745–1764
12. Han LB, Wu Y, Nong G (2020) Succinct suffix sorting in external memory. Inf Process Manag. https://doi.org/10.1016/j.ipm.2020.102378
13. He H, Du Z, Zhang W, Chen A (2015) Optimization strategy of Hadoop small file storage for big data in healthcare. J Supercomput 72(10):3696–3707
14. Kärkkäinen J, Kempa D, Puglisi SJ (2015) Parallel external memory suffix sorting. In: Proceedings of the 26th Annual Symposium on Combinatorial Pattern Matching, pp 329–342
15. Kim H, Yeom H (2017) Improving small file I/O performance for massive digital archives. In: 2017 IEEE 13th International Conference on E-Science
16. Lakshman A, Malik P (2010) Cassandra: a decentralized structured storage system. ACM SIGOPS Oper Syst Rev 44(2):35–40
17. Lao B, Nong G, Chan WH, Xie JY (2018) Fast in-place suffix sorting on a multicore computer. IEEE Trans Comput 67(12):1737–1749
18. Lin TY, Maire M, Belongie S, Hays J, Perona P, Ramanan D, Dollár P, Zitnick CL (2014) Microsoft COCO: common objects in context. In: Computer Vision—ECCV, pp 740–755
19. Manber U, Myers G (1993) Suffix arrays: a new method for on-line string searches. SIAM J Comput 22(5):935–948
20. Meng B, Bin Guo W, Sheng Fan G, Wu Qian N (2016) A novel approach for efficient accessing of small files in HDFS: TLB-MapFile. In: 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing
21. Mori Y Libdivsufsort, a software library that implements a lightweight suffix array construction algorithm. Available: https://github.com/y-256/libdivsufsort
22. Nguyen MC, Won H, Son S, Gil MS, Moon YS (2017) Prefetching-based metadata management in advanced multitenant Hadoop. J Supercomput 2:1–21
23. Nong G (2013) Practical linear-time O(1)-workspace suffix sorting for constant alphabets. ACM Trans Inf Syst 31(3):1–15
24. Nong G, Zhang S, Chan WH (2011) Two efficient algorithms for linear time suffix array construction. IEEE Trans Comput 60(10):1471–1484
25. Parkhi O.M, Vedaldi A, Zisserman A, Jawahar CV (2012) Cats and dogs. In: 2012 IEEE Conference on Computer Vision and Pattern Recognition, pp 3498–3505
26. Phakade P, Raut S (2014) An innovative strategy for improved processing of small files in Hadoop. Int J Appl Innov Eng Manag 3(7):278–280
27. Song J, He H, Thomas R, Bao Y, Yu G (2019) Haery: a Hadoop based query system on accumulative and high-dimensional data model for big data. IEEE Trans Knowl Data Eng 32(7):1362–1377
28. Tchaye-Kondi J, Zhai Y, Lin KJ, Tao W, Yang K (2019) Hadoop perfect file: a fast access container for small files with direct in disc metadata access. arXiv preprint arXiv:1903.05838
29. Transier F, Sanders P (2010) Engineering basic algorithms of an in-memory text search engine. ACM Trans Inf Syst 29(1):1–37

30. Wang Y, Ma C, Wang W, Meng D (2014) An approach of fast data manipulation in HDFS with supplementary mechanisms. J Supercomput 71(5):1736–1753
31. Wu S, Chen G, Chen K, Li F, Shou L (2015) HM: a column-oriented MapReduce system on hybrid storage. IEEE Trans Knowl Data Eng 27(12):3304–3317
32. Xie JY, Nong G, Lao B, Xu W (2020) Scalable suffix sorting on a multicore machine. IEEE Trans Comput 69(9):1364–1375
33. Zhang Y, Liu D (2012) Improving the efficiency of storing for small files in HDFS. In: 2012 International Conference on Computer Science and Service System