SHAstor: A Scalable HDFS-based Storage Framework for Small-Write Efficiency in Pervasive Computing

Lingfang Zeng[‡], Wei Shi[§], Fan Ni[♮], Jiang Song[♮], Xiaopeng Fan[†], Chengzhong Xu[†], Yang Wang[†*]
[‡] Huazhong University of Science and Technology, China
[§] School of Computer Science, Carleton University, Ottawa, Canada
[♮] University of Texas at Arlington, USA
[†] Shenzhen Institutes of Advanced Technology, Chinese Academy of Sciences, China
[*] Corresponding to: yang.wang1@siat.ac.cn

***Abstract*—It is well known that small files are often created and accessed in pervasive computing in which information is processed with limited resources via linking with objects as encountered. And the Hadoop framework, as a de facto big data processing platform though very popular in practice, cannot effectively process the small files. In this paper, we propose a scalable HDFS-based storage framework, named *SHAstor*, to improve the throughput in processing of small-writes for pervasive computing paradigm. Compared to the classic HDFS, the essence of this approach is to merge the incoming small writes into a large chunk of data, either at client side or at server side, and then store it as a big file in the framework. As a consequence, this could substantially reduce the number of small files to process the pervasively gathered information. To reach this goal, the framework takes the HDFS as the basis and adds three extra modules for merging and indexing the small files during the read/write operations in pervasive applications are performed. To further facilitate this process, a new ancillary namenode is also optionally installed to store the index table. With this optimization, *SHAstor* can not only optimize the small-writes, but also scale out with the number of datanodes to improve the performance of pervasive applications.**

***Index Terms*—pervasive computing, HDFS-based storage, small write, Hadoop framework**

I. Introduction

Hadoop framework as a core enabling technology for large-scale data processing has demonstrated its advantages over traditional approaches in terms of throughput and cost benefits for many computational tasks. As a consequence, migrating existing algorithms into their Hadoop representations is becoming a promising approach to various applications. Although this approach is attractive, the performance of the framework for small-writes is still a great challenge as many studies have demonstrated that small-writes are not amenable to Hadoop [1], [2], [3], [4], [5]. Here, a small file is the file whose size is significantly smaller than the Hadoop block size (default 64MB in HDFS).

Pervasive computing, also known as ubiquitous computing, is a computing paradigm where the information is processed by linking each object as encountered in environment. Pervasive computing typically involves a variety of connected electronic devices to achieve everyware and ambient intelligence via communicating information between them. The devices in pervasive computing usually have constant availability and are completely connected, which makes it possible for the paradigm to effectively gather the pervasive information across different electronic devices, and then efficiently process it in a centric or distributed fashion, for much more complex and intelligent applications. Cloud computing due to its abundant on-demand resources and elastic charge models provides a unique platform to reach this goal via deploying the computing utilities on it, such as the big data processing framework.

Given limited resources, accesses to small files are pervasive for the connected electronic devices, and they also are often performance critical in modern pervasive computing environments, especially with the incoming era of big data-based pervasive applications [6].

Hadoop is not geared up to efficiently accessing small files [4], [7], [8], [6]. Instead, it is originally designed for streaming access of large files, completely unaware of the nature of small files to optimize bandwidth, I/O, and CPU cycles. In particular, when dealing with a large volume of small files, more challenges are involved as summarized by Lin in [9].

1) Additional system information incurred by the managements of small files (e.g., metadata) may be larger than file data itself, and cost more memory for read and write operations as well;
2) Basic file information is typically stored in system memory, rather than hard disks, which could use more memory in both *namenode* and *datanode* in Hadoop (describe later);
3) Reading through small files normally causes lots of seeks and lots of hopping from datanode to datanode to retrieve each small file, all of which is an inefficient data access pattern.

More or less, these challenges have been tackled with different extensions to Hadoop [7], [8], [6], or alternatively, its programming paradigms [1], [2], [3], [4], [5]. However, most of the resulted frameworks or systems are not oriented to pervasive computing, which is loaded with small files from

different sources and in different formats. And also as per Lin [9], these frameworks or systems are not Hadoop [9] anymore, rendering the massive benefits of the widely deployed Hadoop-based stack (e.g., Pig, Hive, etc) hard to achieve.

In order to solve these problems, we build *SHAstor* (Scalable HDFS-based Storage Framework) based on the Hadoop framework to improve the throughput in the process of small files for pervasive computing. Compared to the classic HDFS, the essence of this framework is to merge the incoming small writes into a large chunk of data, either at client side or at server side, and then store it as a big target file in the framework. As a consequence, this could substantially reduce the number of small files. To reach this goal, the framework takes the HDFS as the basis and adds three extra modules for merging and indexing the small files during the read/write operations in pervasive applications are performed. To further facilitate this process, a new ancillary namenode is also installed to store the index table. With this optimization, *SHAstor* can not only optimize the small-writes, but also scale out with the number of datanodes to improve the performance.

We organize the paper as follows: in Section II, we survey some related work in the optimization of the Hadoop framework with respect to the small-write efficiency, and compare with our in-progress work. After that, we describe the design of *SHAstor* in Section III, including some background knowledge regarding the traditional Hadoop framework and the structure of *SHAstor*. We conduct performance studies to validate our optimizations in Section IV, and remarks the paper with some discussions on on-going work in the last section.

II. Related work

Optimizing HDFS in special and Hadoop in general for small file processing and storage is an intensively studied area [10], [7], [11], [12]. The main idea in these studies is to merge or combine the small files into fewer large files and develop different mechanisms and strategies to accommodate their metadata and index files. The efforts are roughly made from either inside the Hadoop community or from outside.

As inside the community, *HAdoop Archive* (HAR) [13] and *Sequence File* [14] are two most recent progresses in this aspect. HAR is designed to reduce the memory consumption by packing the small files into data blocks. However, it does not support the appending operation and also suffers from the low access performance due to an extra index file access. Instead, SequenceFile stores the data in a form of binary key-value pair, where the key is file name and the value is file contents. As such, it can act as a container for small files, with compression and de-compression supports inside. As with HAR, SequenceFile also bears some demerits. It only supports appending operation and lacks the mechanism to update and delete a particular key. Also, it suffers while performing random read operation due to its unsorted files in key. Our design needs to overcome the issues of both methods.

In addition to the efforts inside the community, there are also some other works for the same purpose outside the community. As we summarized, they either extended the existing Hadoop framework [7], [8] or developed new programming paradigms [1], [2], [3], [4], [5]. However, most of the resulted systems are not Hadoop [9] anymore, and also fail to suite for the pervasive computing, which is loaded with small files from different devices.

Sethia *et al.* [12] addressed this issue in their most recent publication, where an optimized mapfile-based storage for small files is proposed with an attempt to reduce internal fragmentation in data blocks, and in turn the memory consumption. In contrast, Lyu *et al.* [7] tackled the small-file problem from a different angle. They introduced an optimized algorithm by considering the sizes of small files, and generating a map record for each of them during the course of merging into large files, when prefetching and caching mechanisms are applied to enhance the access efficiency. Our work is different from theirs as we adopted different design strategies with a goal to maximize the small-write efficiency in scale-out manner for pervasive computing.

III. SHAstor Design

In this section, we first describing some preliminary regarding the Hadoop framework, and the characteristics of information processing in pervasive computing, and then introduce our proposed *SHAstor* structure, and show how its components are coordinated each other to improve the efficiency of small-writes in pervasive computing.

A. Preliminary

HDFS is a distributed file system in Hadoop framework designed for storing very large files with streaming data access patterns. It runs on clusters of commodity hardware with highly fault-tolerant performance. A HDFS cluster has two types of node operating in a master-work fashion: a *namenode* (the master) and a plurality of *datanodes* (workers). The namenode is the central control of the cluster, managing the file system namespace and maintaining the file system tree and the metadata for all the files and directories in the tree. The tree structure is built in the namenode's memory, where it is used to regulate the flow of information for all read and write operations issued by the clients. In this structure, every directory, file, and block in HDFS is represented as an object in memory in the namenode. As such, the optimization is clear that one can reduce the namenode memory footprint, start-up time and network impact by minimizing the number of small files on the cluster.

The forms of data gathered across different devices in pervasive computing are usually multi-sources, heterogeneous and small in sizes since the types of the devices are often different and diverse, and their compute resources are fairly limited. As such, for ambient intelligence, it makes sense that deploying big-data frameworks to process the gathered massive data from the devices, and then in turn to direct the devices to react the activities with more intelligence for a certain purpose. To this end, it requires the framework to have the capability of efficiently dealing with enormous quantify of small files in different formats. As shown in

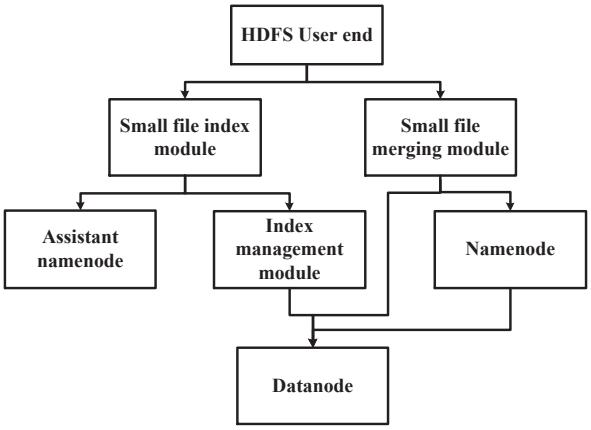


Fig. 1. SHAstor structure design

[15], this handling is dominated by the read operations to the files in the subsequent accesses, and thus, the performance of initial write operations, together with the efficiency of the components to support the subsequent read operations, are particularly important to the ambient intelligence of the pervasive computing.

B. SHAstor Structure

By following the arguments above, we design *SHAstor*, a HDFS-based storage framework, to reduce the number of files by merging small size files. The heterogeneous data across different devices could be unified either into an ASCII-hex format that can convey binary information or into an archive file format that is indexable. As such, the files with different formats can be collectively merged.

As files are merged, the index information of the merged files is stored either in the selected datanodes or in an assistant namenode for subsequent accesses. With this approach, *SHAstor* can reduce the number of files so that not only the frequency of storing small files, but also the frequency of managing blocks in namenodes are also decreased, improving, improving the speeds of read and write operations, accordingly. Moreover, when computing MapReduce tasks, the reduction of Map tasks can also decrease the memory footprint of the computation, and hence, the system performance can be significantly improved by using *SHAstor* instead of HDFS.

To facilitate the merging process, *SHAstor* adds one assistant namenode and three new modules to HDFS, that is, *Small-file Merging Module*, *Small-file Index Module* and *Index Management Module*, which are organized as shown in Fig. 1.

The Small-File Merging Module is designed to merge the small files at either the client side or or the server side, based on whether or not the data requests by users are continuous. With this module, *SHAstor* not only minimizes the memory footprint for storing small files in the namenode, but also reduces the time spent on transferring data,Therefore, this module can also support large quantity of data transfers.

The Small-File Index Module is designed to build up the index information for small files during the process of file merging at the client side, or files appending at the HDFS side. This module can improve the speed of reading and writing small files, and reduce the memory footprint in both the namenodes and datanodes, no matter where the index information is stored, either in the datanodes or in the assistant namenode.

The Index Management Module is used to select a datanode to insert the index information regarding the merged small files before merging the small files. The assistant namenode could be an idle computer, and thus it can be used to store the index information of small files if sufficient free resources are available.

C. Small File Merging Module

The Small-File Merging Module is further divided into two processes based on whether or not the data requests by users are continuous. If the requested data is continuous, the file merging process will be happened at the client side. Otherwise, the client side will merge the small files by appending them to existed files in HDFS in order to reduce the total number of files.

a) File merging at client side: This process is to merge the small files in the memory of the client devices. During the merging process, the index information of the small files is inserted into the index module, along with the small files' offsets in the target file. Additionally, the index information of the merged small files will be saved at the selected datanodes or the assistant namenode. When the total number of the small files or the size of the target file reaches a certain value, the client device will send a data input request to the namenode, so that the basic information of the target file and the index information of the merged small files will be written into HDFS.

b) File merging at server side: When a user uploads many separated small files, *SHAstor* will build an index for each client device to address the problem that more memory is needed by both the namenodes and the datanodes to handle the large number of small files. Specifically, before starting the upload of the first file, the client device checks the index module to decide whether to create a new data block for merging or looking for an existing (not full) data block for appending. The first file uploaded by the client device is processed as a regular file, and the subsequent files are appended to the first one, hence, the small files are merged at the server. As with before, during this process, both the index information of the merged small files (e.g., offsets in the target file) should be stored in the index module. Once the total number of merged files or the size of the data block reaches a certain value, a new data block will be created for further small writes.

D. Small File Index Module

There are two key issues in design of the index module for efficient small writes. First, we should determine the index

TABLE I
DATA STRUCTURE OF SMALL FILE INDEX MODULE

Name	Type	Instruction
Filename	String	The name of small file
Filesize	int	The size of small file
targetfile_name	String	The name of target file in HDFS
targetfile_size	int	The size of target file in HDFS
HDFS_path	String	The path of target file in HDFS
target_file_offset	int	the small file's offset in target file
can_use	int	Whether the small file is available
time	long	File uploaded time

information for the merged small files, and design the data structures used by the module to manage the information. Second, depending on the workload distribution, we also need to select a vantage location to store the index table.

a) Module Container: SHAstor saves the data structure of the index module as key/value pairs in a container, where the key is the name of the small file, and the value includes the size of the small file, the name, size, the path of the target file in HDFS, the small file's offset in the target file, and as well as whether or not the small file is available. In the design, *SHAstor* chooses *HashMap* to manage the key/value pairs, because in *SHAstor*, there is large quantity of client devices who may merge the small files with enough memory, and HashMap is good at inserting and searching operations in high frequency. The hash function is shown as below, and we use this function to store the key/value pair as shown in Fig. 2.

An array with a large range of index is used to store the key/value pairs, and each string-type hashcode of a key is reflected by the value of the following function, and this value is also the index of the array.

$$
\begin{aligned}
&\text{private static int Hash(int h) } \{ \\
&\quad h+= (h << 9); \\
&\quad h^{=} (h >>> 14); \\
&\quad h+= (h << 4); \\
&\quad h^{=} (h >>> 10); \\
&\quad \text{return } h; \}
\end{aligned}
$$

b) Physical Location: This module can be installed at either the assistant namenode (if available) or a datanode. In order to select an appropriate datanode, *SHAstor* first gets a string of the client device's IP address followed by the file name, and then calculates the hashcode of the string, which is further modulo-ed by the number of datanodes. Finally, *SHAstor* selects the datanode whose number is equal to the remainder.

E. Read, Search, Insertion and Deletion

There are four major operations on small files: *Insert*, *Search*, *Read* and *Delete*, which are implemented via the accesses to the index table. With these operations, one can retrieve the original merged small files from the target file.

a) Insert:: There are several step for insertion 1) Use the name of the small file as the key value to calculate the string type hashcode; 2) Then, use the hash function and the hashcode to calculate the hash value as the array index; 3) Finally, store the key/value pair of the small file into the array indexed by the hash value.

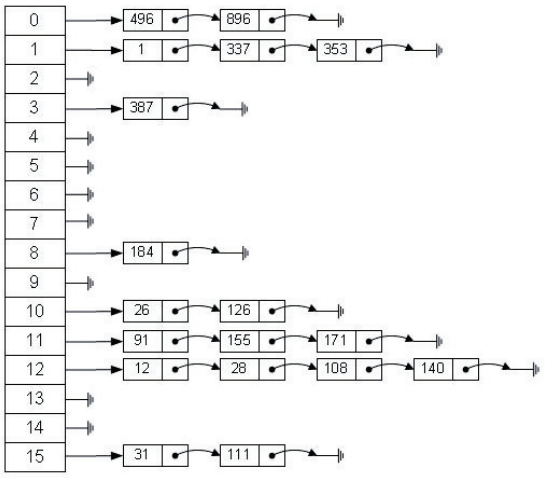


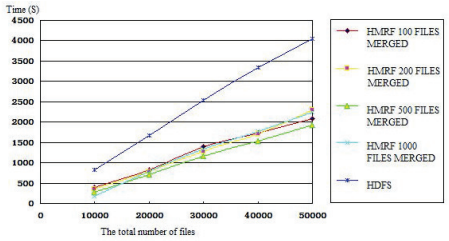
Fig. 2. Structure of HashMap

b) Search:: The first three steps are the same as those in the insert process. After that, compare the Key value from the array with the small file name. If they are equal, the value has been found, and use this value to locate the small file in the target file. Otherwise, there is no such small file in the target file.

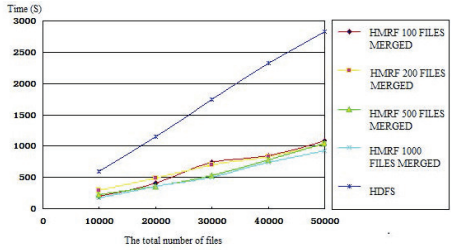
c) Single file reading: The client device first sends a search request to the namenode to obtain the basic information of the small files as shown in Table I. Then, the client device sends a read request to the namenode to locate the target file. If the target file is found, then the namenode returns the block number and its hosting datanode; otherwise, it returns an error code. After obtaining the merged file's location, the client device connects the hosting datanode via socket to read the requested file block by moving the offset to the corresponding location in the target file.

d) Batch files reading: The batch files is referred to a sequence of files with continuous file numbers that are saved together in the target file. As with the single file reading, for the batch file reading the client device first sends a search request to the namenode for the basic information of the small file, then, uses this information to locate the small files in the target file. Once the client device obtains the first and last files' names and offsets, it can read all the files in between. After reading these small files, the client device uses the basic information of each small file to separately identify its block contents.

e) Delete:: The client device first sends a search request to the index module. If the file does not exist, the module returns a *no-file* code; otherwise, the *can_use* value in the key/value pair will be set to 0, which indicates the file has been deleted. Then in the next time when this file is requested again, a *no-file* code could be sent to the client device.



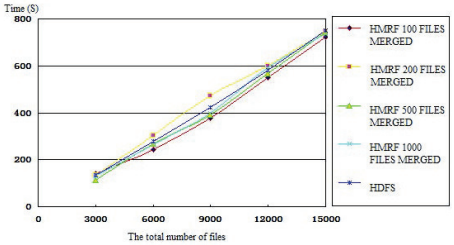
(a) $127KB$ files



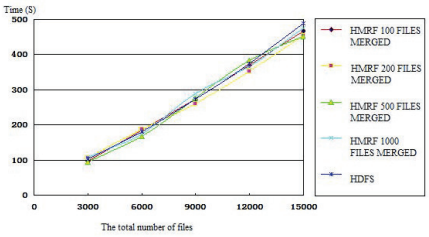
(b) $63KB$ files

	10000	20000	30000	40000	50000
Group1(GB)	1.2	2.4	3.6	4.8	6.0
Group2(GB)	0.6	1.2	1.8	2.4	3.0

Fig. 3. Write performance comparisons between different input file sizes: 127KB vs. 63KB



(a) $127KB$ files



(b) $63KB$ files

	3000	6000	9000	12000	15000
Group1(GB)	0.4	0.8	1.2	1.6	2.0
Group2(GB)	0.2	0.4	0.6	0.8	1.0

Fig. 4. Read performance comparisons between different input file sizes: 127KB vs. 63KB

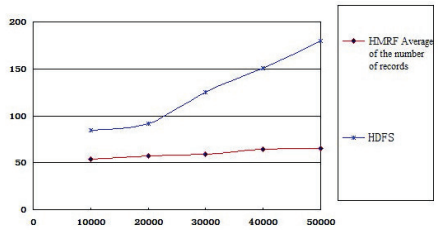
IV. Performance Evaluation

We set up a testbed to evaluate the performance of *SHAstor*. The testbed consists of two physical machines that are connected via a network with 100MB bandwidth, each hosting two virtual machines (VM) (CPU 2.66GHz/mem 1GB/Disk 80GB) created by Windows VMWare. Moreover, we also install Ubuntu 9.04 on each node in the cluster, and whereby the Hadoop (0.20.2) is deployed with the support of Java environment (Java-6-openjdk).

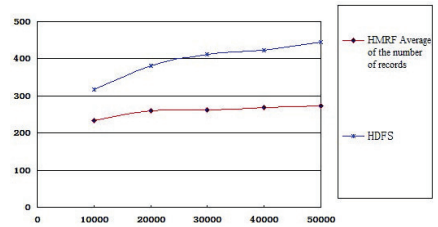
a) File Writing: The files used in the test are simulated records of telephone GPS information, a typical kind of pervasive information. There are two sizes of files, one is $63KB$, having 720 records, and the other is $127KB$, having 1440 records. We compare the read performance between HDFS and *SHAstor* by setting the total number of files between 10000 and 50000 as shown in the table of Fig. 3.

The number of files merged at the client side is set to 100, 200, 500, and 1000, respectively. The performance comparison of uploading $127KB$ files (Fig. 3(a)), and $63KB$ files (Fig. 3(b)). According to this figure, compare to HDFS, the performance in *SHAstor* has been improved 100% when file size is $127KB$ and reaches the best when merging 500 small files, and 230% when file size is $63KB$ and reaches the best when merging 1000 small files.

b) File Reading: From the files written in the previous test, we randomly read 33.3% of them, whose total sizes are shown in the table of Fig. 4. The performance comparison of reading $127KB$ files and $63KB$ files is also shown in Fig. 4. From this figure, one can observe that the file reading performance is kept the same in both HDFS and *SHAstor*, demonstrating the efficiency of *SHAstor* in handling the small-writes.



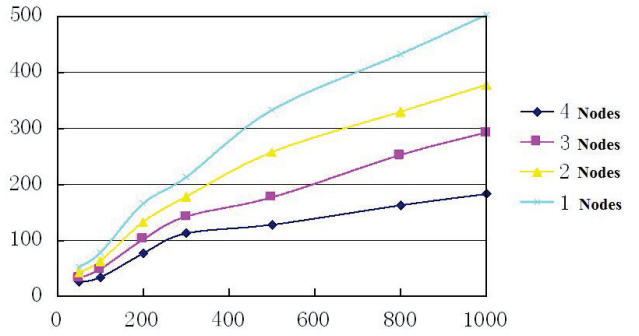
(a) Namenode



(b) Datanode

Fig. 5. Memory footprints in namenode and datanode.

c) Memory footprints in namenode and datanode: We test the memory footprints in both namenodes (Fig. 5(a)) and datanodes (Fig. 5(b)). From these figures, the memory footprints have been reduced 63.2% for the namenodes and 38.7% for the datanodes, respectively.



#. of rec.(M)	50	100	200	300	500	800	1000
Size(MB)	46	92	184	276	460	736	920

Fig. 6. SHAstor scalability test

d) SHAstor Scalability: Given the number of records in a file is increased from $50M$ to $1000M$ as shown in Fig. 6, we also compare the time overhead between HDFS and *SHAstor* as the the number of compute nodes is varied from 1 to 4 (Fig. 6). Our results demonstrate that *SHAstor* can exhibit approximately linear scalability as the number of nodes and the size of file are increased, a good property to facilitate a large amount of small-writes.

V. Remarks and On-going Work

In this paper, we described a scalable storage framework based on HDFS, called *SHAstor*, for small-write efficiency in pervasive computing. *SHAstor* achieves this efficiency by merging small files into a large target file and facilitating the insertion, search, read and deletion operations with the aid of designed components. To evaluate *SHAstor*, we have prototyped it and made some experiments by using the typical GPS datasets in pervasive computing. Our preliminary results demonstrated that *SHAstor* is not only efficient but also scalable for small writes, having potentials to realize the ambient intelligence.

Although *SHAstor* has exhibited some promising potentials, it is still in its early stage and has much room left to be further improved. Currently, we are working on the algorithms and mechanisms for data heterogeneity that would allow the data files from different devices and in different formats to be processed in a unified way. The challenge of this work is to strike a good balance between the benefits of the small writes and the costs in handling the different formats. Another effort we are making in progress is to implement the update operation with respect to the merged files. Although the update operation is not that often, compared to its read counterpart, in pervasive computing, it is still useful in certain cases as shown in [15]. To support the update operation, the current components of *SHAstor* need to be substantially improved to record and index the update operations. In the long run, we plan to combine *SHAstor* with our *hitchhike* technique, which is an I/O scheduler optimization enabling writeback for small synchronous writes [16], and then integrate them into a real pervasive computing platform to provide diverse electronic devices with an efficient remote control for ambient intelligence in pervasive computing.

Acknowledgment

This work was supported in part by National Key R&D Program of China (No. 2018YFB1004804), Shenzhen Oversea High-Caliber Personnel Innovation Funds (KQCX20170331161854), Shenzhen Strategic Emerging Industry Development Funds (JCYJ20170818163026031), NSFC (61672513, 61572487), the CAS Exchange & Cooperation (SQ2016YFHZ020520), Shenzhen International S&T Cooperation (GJHZ20160229194322570), the CAS Light of West China Program (2016-QNXZ-A-5).

References

[1] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A system for large-scale graph processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '10, 2010, pp. 135–146.

[2] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst, "Haloop: Efficient iterative data processing on large clusters," *Proc. VLDB Endow.*, vol. 3, no. 1-2, pp. 285–296, Sep. 2010.

[3] E. Elnikety, T. Elsayed, and H. E. Ramadan, "ihadoop: Asynchronous iterations for mapreduce," in *Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science*, ser. CLOUDCOM '11, 2011, pp. 81–90.

[4] Y. Zhang, Q. Gao, L. Gao, and C. Wang, "Priter: A distributed framework for prioritizing iterative computations," *Parallel and Distributed Systems, IEEE Transactions on*, vol. 24, no. 9, pp. 1884–1893, 2013.

[5] J. Lin and M. Schatz, "Design patterns for efficient graph algorithms in mapreduce," in *Proceedings of the Eighth Workshop on Mining and Learning with Graphs*, ser. MLG '10. New York, NY, USA: ACM, 2010, pp. 78–85.

[6] S. Bende and R. Shedge, "Dealing with small files problem in hadoop distributed file system," *Procedia Computer Science*, vol. 79, pp. 1001 – 1012, 2016, proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016.

[7] Y. Lyu, X. Fan, and K. Liu, "An optimized strategy for small files storing and accessing in hdfs," in *2017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*, vol. 1, July 2017, pp. 611–614.

[8] N. Mohandas and S. M. Thampi, "Improving hadoop performance in handling small files," in *Advances in Computing and Communications*, A. Abraham, J. L. Mauri, J. F. Buford, J. Suzuki, and S. M. Thampi, Eds., 2011, pp. 187–194.

[9] J. Lin, "Mapreduce is good enough? if all you have is a hammer, throw away everything that's not a nail!" http://arxiv.org/abs/1209.2191.

[10] X. Zhao, Y. Yang, L.-l. Sun, and H. Huang, "Metadata-aware small files storage architecture on hadoop," in *Web Information Systems and Mining*, F. L. Wang, J. Lei, Z. Gong, and X. Luo, Eds. Springer Berlin Heidelberg, 2012, pp. 136–143.

[11] T. Renner, J. Müller, L. Thamsen, and O. Kao, "Addressing hadoop's small file problem with an appendable archive file format," in *Proceedings of the Computing Frontiers Conference*, 2017, pp. 367–372.

[12] S. Sheoran, D. Sethia, and H. Saran, "Optimized mapfile based storage of small files in hadoop," in *17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, May 2017, pp. 906–912.

[13] Hadoop archive guide, https://hadoop.apache.org/docs/current/hadoop-archives/HadoopArchives.html [Online; accessed Jan-11-2018].

[14] SequenceFile, https://wiki.apache.org/hadoop/SequenceFile [Online; accessed Jan-11-2018].

[15] U. Hengartner and P. Steenkiste, "Access control to information in pervasive computing environments," in *Proceedings of the 9th Conference on Hot Topics in Operating Systems - Volume 9*, ser. HOTOS'03, 2003, pp. 27–27.

[16] X. Liu, S. Jiang, Y. Wang, and C. Xu, "Hitchhike: An i/o scheduler enabling writeback for small synchronous writes," in *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*, Nov 2016, pp. 64–68.