SOCIETY FOR SCIENCE AND EDUCATION
UNITED KINGDOM
TMLAI TRANSACTIONS ON MACHINE LEARNING AND ARTIFICIAL INTELLIGENCE
VOLUME 5 NO 4, 2017
ISSN 2054-7390

Improving Small File Management in Hadoop

[1]O. Achandair,[2]M. Elmahouti, [3]S. Khoulji, [4]M.L. Kerkeb,
[1,2,3,4] *Information System Engineering Resarch Group*
Abdelmalek Essaadi University, National School of Applied Sciences, Tetouan, Morocco.
o.achandair@gmail.com, mouradelmahouti@gmail.com,khouljisamira@gmail.com,
kerkebml@gmail.com, acmlis.conference@gmail.com

ABSTRACT

Hadoop, considered nowadays as the de-facto platform for managing big data, has revolutionized the way customers manage their data. As an open-source implementation of map-reduce, it was designed to offer a high scalability and availability across clusters of thousands of machines. Through it two principals' components, which is HDFS for a distributed storage and MapReduce as the distributed processing engine, companies and research studies are taking a big benefit from its capabilities.

However, Hadoop was designed to handle large size files, so when it comes to a large number of small files, the performance can be heavily degraded. The small file problem has been well defined by researchers and Hadoop community, but most of the proposed approaches only deal with the pressure caused on the NameNode memory. Certainly, grouping small files in different possible formats, that are most of time supported by the actual Hadoop distribution, reduce the metadata entries and solve the memory limitation, but that remain only a part of the equation. Actually, the real impact that organizations need to solve when dealing with lot of small files, is the cluster performance when those files are processed in Hadoop clusters. In this paper, we proposed a new strategy to use efficiently some one of the common solution that group files in a MapFile format.

The core idea, is to organize small files files based on specific attributes in MapFile output files, and use prefetching and caching mechanisms during read access. This would lead to less calls of metadata from the NameNode, and better I/O performance during MapReduce jobs. The experimental results show that this approach can help to obtain better access time when the cluster contain massive number of small files.

Keywords - Cloud Hadoop, HDFS, Small Files, SequenceFile, MapFile

1 Introduction

Apache Hadoop, an open-source framework initiated by Yahoo!, developed for storing and processing data at a large scale on clusters of thousands of commodity hardware. Meanwhile, it offers a reliable, scalable, and fault tolerant platform deployed by many big companies such as Facebook, Google, IBM, Twitter and others to manage Terabytes to Petabytes of data.

Hadoop was designed to handle large files, especially when traditional systems are facing limitations to analyze this new data dimension caused by the present data explosion. However, large files are not the

DOI: 10.14738/tmlai.54.3333
Publication Date: 15th August, 2017
URL: http://dx.doi.org/10.14738/tmlai.54.3333

only kind of files that Hadoop deployments needs to manage, the diversity of all kind of data becomes a standard in most of big data platform. There are many fields that produce tremendous numbers of small files continuously such as analysis for multimedia data mining [6], astronomy [7], meteorology [8], signal recognition [9], climatology [10,11], energy, and E-learning [12] where numbers of small files are in the ranges of millions to billions. For instance, Facebook has stored more than 260billion images [13]. In biology, the human genome generates up to 30 million files averaging 190KB [14].

1.1 HDFS – Storage Layer

HDFS, The Hadoop distributed file system provides high reliability, scalability and fault tolerance. It's designed to be deployed on big clusters of commodity hardware. It's based on a master-slave architecture, the NameNode as a master and the DataNodes as slaves. The NameNode is responsible for managing the file system namespace, it keeps tracks of files during creation, deletion, replication [3] and manages all the related metadata [4] in the server memory. The NameNode splits files into blocks and sends the writes requests to be performed locally by DataNodes. To ensure a fault-tolerance system, blocks replicas are pipelined across a list of DataNodes. This architecture as shown in "Fig.1", with only one single NameNode simplifies the HDFS model, but it can cause memory overhead and reduces file access efficiency when dealing with a high rate of small files.

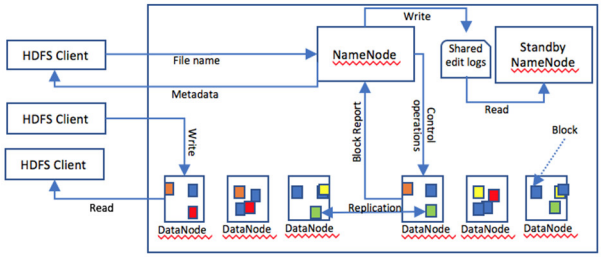

Figure. 1: HDFS Architecture

1.2 MapReduce – Processing Engine

In the current version of Hadoop, Google re-architected the processing engine to be more suitable for most of big data applications needs. The major improvement of Hadoop was the introduction of a resource management module, called YARN, independently of the processing layer. This brought significant performance improvements, offered the ability to support additional processing models, and provided a more flexible execution engine. Because of its independency architecture, existing MapReduce applications can run on YARN infrastructure without any changes.

The MapReduce program execution on YARN can be described as follows:

i. A user defines an application by submitting its configuration to the application manager
ii. The resource manager allocates a container for the application manager
iii. Resource manager submits the request to the concerned node manager
iv. The Node manager launches the application manager container
v. The application manager gets updated continuously by the node manager nodes, it monitors the progress of tasks

When all the tasks are done, the application manager unregisters from the resource manager, like so, the container can be allocated again, See "Fig. 2".

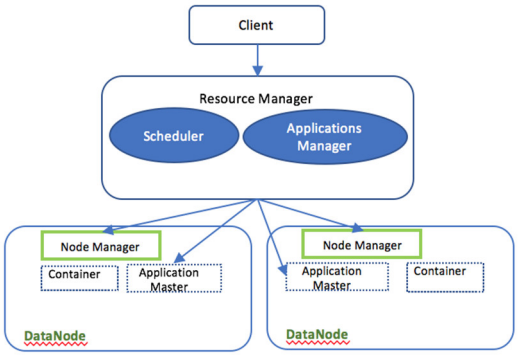

Figure. 2: YARN – Yet Another Resource Manager

The rest of this paper is divided into the following, section 2 lists the existing solutions in the related literature. Section 3 present the proposed approach. Section 4 is allocated for our experimental works and results. Finally, Section 5 for conclusion and expectation.

2 Related Work

To deal with the small file problem, numerous researchers have proposed different approaches. Some of those efforts have been adopted by Hadoop and are available for use natively, more precisely, Hadoop Archives (HAR Files) and SequenceFile.

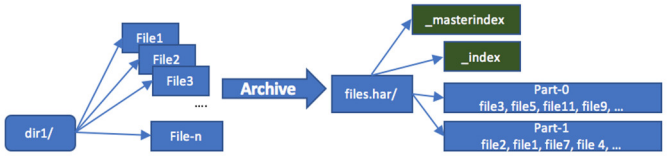

Figure. 3: HAR File Layout

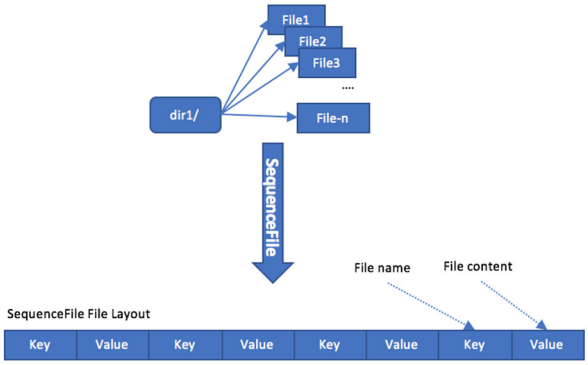

Figure 4: SequenceFile File Layout

Hadoop Archive packs small files into a large file, so that we can access original files transparently, see "Fig. 3". This technique allows more storage efficiency, as only metadata of the archive is recorded in the namespaceof the NameNode, but it doesn't resolve other constraints in terms of reading performance.Also, the archive cannot be appended while adding more small files. The SequenceFile technique is to merge a group of small files in a flat file, as key-value pairs, while key is the related file metadata and value is the related content, see "Fig. 4".

Unlike the HAR files, the SequenceFile supports compression, and they are more suitable for MapReduce tasks as they are splittable [15], so mappers can operate on chunks independently.However, converting into a SequenceFile can be a time-consuming task, and it has a poor performance during random read access.

To improve the metadata management, G. Mackey et al. [16] merged small files into a single larger file, using the HAR through a MapReduce task. The small files are referenced with an added index layer (Master index, Index) delivered with the archive, to retain the separation and to keep the original structure of files.

C. Vorapongkitipunet al. [17] proposed an improved approach of the HAR technique, by introducing a single index instead of the two-level indexes. Their new indexing mechanism aims to improve the metadata management as well as the performance during file access without changing the implemented HDFS architecture.

Patel A et al. [18] proposed to combine files using the SequenceFile method. Their approach reduces memory consumption on the NameNode, but it didn't show how much the read and write performances are impacted.

Y. Zhang et al. [19] proposed merging related small files according to WebGIS application, which improved the storage effeciency and HDFS metadata management, however the results are limited by the scene.

D. Dev et al. [20] proposed a modification of the existing HAR. They used a hashing concept based on the sha256 as a key. This can improve the reliability and the scalability of the metadata management, also the reading access time is greatly reduced, but it takes more time to create the NHAR archives compared to the HAR mechanism.

P. Gohil et al. [21] proposed a scheme for combining small file, merging, prefetchingthe related small files which improves the storage and access efficiency of small files, but does not give an appropriate solution for independent small files.

3 The Proposed Approach for Small File Management

The core idea behind our approach is to combine different clients' files that contain sets of small files, when relevant, and merge them through a merger process, to be stored in an optimal way before closing the current SFA connection. This implementation has been achieved in our previous work, See "Fig 5", as the main task of the Small File Analyzer Server. In the current research, we improved the proposed SFA server, to handle other modules that offered us the possibility to consider other parameters during the merger process.

We introduced a sorter process, that can operate as an independent module in the SFA server. The compressor module is another added module, that allowed us to compress merged files that are no longer used, or rarely called, such a way we can benefit from a considerable benefit of storage capacity. Also, we used a prefetching and caching technique to boost the performance while reading the same small files.

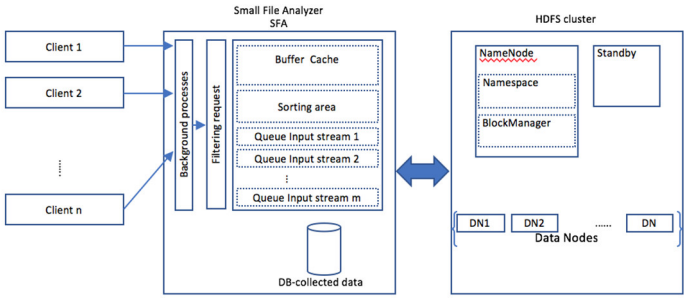

Figure. 5: SFA Architecture

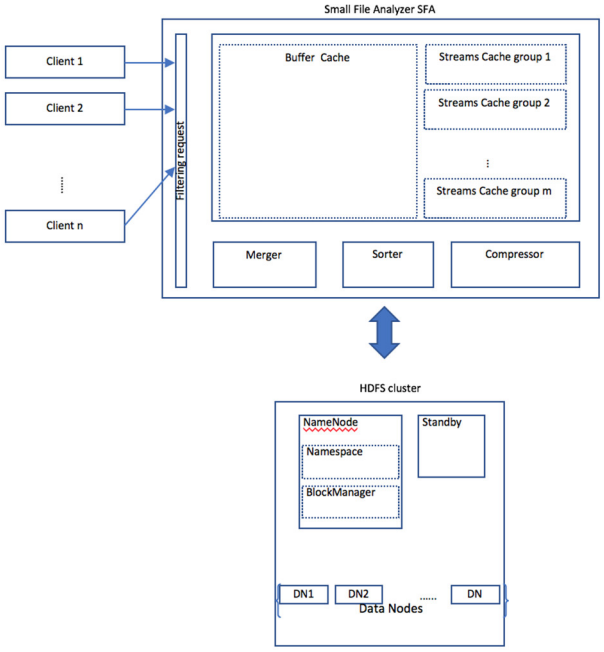

Figure. 6: SFA improved Architecture

For simplicity, the clients' files are stored in the archaically files system of the SFA server, connecting clients' streams directly to the merger is not covered in the scope of the current study. However, its considered to maintain the same concept when extending that implementation. The clients' files are scanned through the java "BasicFileAttributes" interface.

```
import java.sfa.file.Files;
import java.sfa.file.Path;
import java.sfa.file.Paths;
import java.sfa.file.attribute.BasicFileAttributes;
public class FileAttributesSFA{
     public static void main(String[] args) throws Exception {
          String path = "/sfa/input/client1_f1.txt";
          Path file = Paths.get(path);
          BasicFileAttributes attr =
               Files.readAttributes(file, BasicFileAttributes.class);
               System.out.println("creationTime = " + attr.creationTime());
               System.out.println("lastAccessTime = " + attr.lastAccessTime());
               System.out.println("lastModifiedTime = " + attr.lastModifiedTime());
               System.out.println("isDirectory = " + attr.isDirectory()); System.out.println("isOther
               = " + attr.isOther()); System.out.println("isRegularFile = " + attr.isRegularFile());
               System.out.println("isSymbolicLink = " + attr.isSymbolicLink());
               System.out.println("size = " + attr.size());
          }
     }
```

3.1 Merger

This process group sets of files together based on the collected file system attributes, as for many application logic, this can make sense when files are consumedfor processing. For example, in some weather application, it may be interesting to combine files of the same period of each year. In E-learning, files are grouped by disciplines, and so on. The SFA can also combine files based on substring of the file name, which is in many production cases, a hidden attribute of other properties like location, gender, or simply a type of source (station, sensor, satellite…). “See. fig 7”

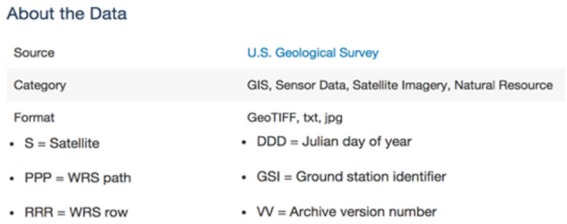

Figure 7: AWS dataset example - continuous record of Earth’s land surface as seen from space.

The predefined strategy of the merger is defined in the master configuration file. Common attributessuch as date, size, owner, type and name are assigned coefficients up to the administrator strategy.The coefficients are set by a group of combined files “See. Fig 8"

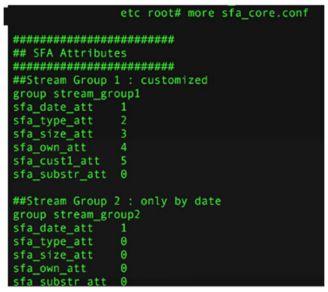

Figure 8: SFAcoefficient of attributes per combined queue

To maintain a flexible implementation, one can define a new attribute in a specified section of the master configuration file (exp : sfa_cust1_att in "Fig 8"), then it’s possible to use it similarly as the common predefined attributes. The assigned priority is from 0 to 9 while 0 refer to an ignorable attribute. Finally, the list of files is identified in the corresponding queue of the SFA memory. Each file is assigned a value based on the priority of attributes strategy.

3.2 Sorter

This process is tightly coupled to the merger, as the list of files is ready to be consumed. the sorter use the assigned values by the merger to each file, and use a sorting algorithm based on quicksort to generate the ordered final list.

Quicksortis a well-known sorting algorithm[19], that performs the best among its competitors. It’s based on divide-and-conquer logic, through the three principals’ phases described below, Quicksort is used to sort a random array of numbers, that refer in our case to the generated values for each file by the merger.

Consider an array S[1 .. k]

Phase I: Divide

The input array S[l .. k] is divided in two empty subarrays:

S[l .. m-1] and S[m+1 .. k]. The subarrays are constructed in a way that every left element of S[1 .. m-1] should be smaller or equal to S[m], and every right element of S[m+1 .. k] should be bigger than S[m]. This phase is about choosing the index m, which is called the pivot element.

Phase II: Conquer

Its about sorting each subarray, S[l .. m-1] and S[m+1 .. k] trough recursive calls of quicksort.

Phase III: Combine

One the subarrays are sorted, we combine them together to form the sorted array S[l .. k].

Sorter Algorithm:

Consider a list of numbers S [1 .. k]

quicksort(S)

if |S| = 0 then S

else let

p = pick a pivot from S

$S1 = s \in S \mid s < p$

$S2 = s \in S \mid s = p$

$S3 = s \in S \mid s > p$

(S1 ,S3) = quicksort(S1) || quicksort(S3)

```
in

append(S1 ,append(S2 ,S3 ))

end
```

For our implementation, the values assigned by the merger, permit us to use a priority-based selection technique. Where the pivot is not selected randomly in the beginning, then its selected with the highest value. This can help reducing time to choose better pivot.We should emphasize that the pivot elements selected in each subarray created are used only for dividing the elements in the two subarrays, their position is not part of a comparison, but it is a result of the last move of this comparison.

"See. Fig 9"

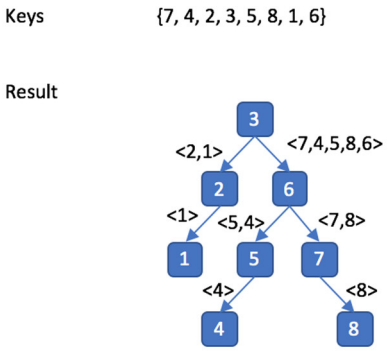

Figure 9: Run example of quicksort

The sorted list of files generated in this phase is stored on the HDFS in MapFile format, which is one of the HDFS supported format, that was introduced to reduce entries in the NameNode memory while dealing with a lot of small files. Moreover, it consists of an indexed SequenceFile to reduce time access compare to sequential read in SequenceFiles. See “Fig 10”.

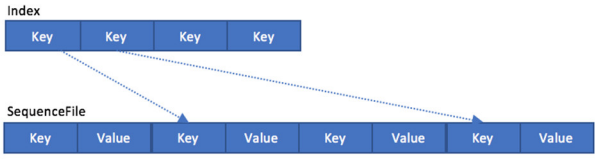

Figure. 10: MapFile File Layout

3.3 Compressor

This process, is not by default activated. In fact, to analyze how the small files are affecting the Hadoop cluster, we import records from the FSimage of the NameNode, which contain a complete state of the HDFS state, then we aggregate data to store it in a reference handled by the database of the SFA, see “Fig. 11”.

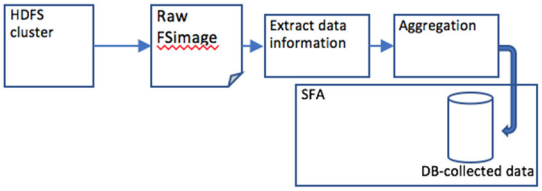

Figure. 11: SFA reference steps

This reference is completely independent of the functionality of our SFA sever, and can be built on a new server if the performance is impacted. Though, it allow us to get some very useful information and reports about the small files in our cluster, such as :

- Files < 512 Kb)
- Files < 512 Kb (Grouped by Path)
- Files < 512 Kb (Grouped by Owner)
- Files < 512 Kb (Grouped by Date)
- Number of blocks with a used space < 60%
- Most called files per MapFile (hot small files)
- Files being called in more than one month (per user, per path, per MapFile)
- …

The compressor process, is used to compress MapFiles values when relevant, scheduling this feature is not recommended but it remain possible in this implementation based on MapFiles age if needed.

3.4 File Prefetching and Caching :

Prefetching and caching technique has been implemented in this study, to improve MapReduce jobs performance when small files are processed. In fact, prefetching can reduce the disk I/O overhead, and improve the access time as data can be brought to the cache before being requested. When a small file is requested, the metadata for the MapFile are pulled from the NameNode and the DataNodes that holds the related blocks is identified. Though, if the metadata of that MapFile exist in the local memory of the SFA server, no request is sent to the NameNode. Therefore, it becomes efficient to read small files with less requests sent to the NameNode. Also, on the DataNode level, a small file request, is usually combined with the MapFile that contain it, that specific format provide an index to determine the location of each small file through its key and then its offset and length. Once the small file is located, the MapFile index is cached in the SFA memory, and the specifics requested small files are cached in the DataNode memory. This can help accessing contained files directly in the next subsequent calls. To avoid memory overhead on the DataNodes, LRU algorithm was used to keep most called objects and manage future allocations efficiently [20]. Caching the specified objects in the DataNodes are achieved through mlock() and mmap(), because storing data off-heap will not affect garbage collection when the amount of cached data is large.

4 Experimental Evaluation

The experimental test platform is in this paper is achieved on an implemented cluster of one NameNode, 4 Datanodes, and the SFA server. The NameNode is a server of 3.10 GHz clock speed, 16GB of RAM and a gigabit Ethernet NIC. Each DataNode offer a 500GB Hard Disk, and they are deployed on Ubuntu 14.04. The SFA server is built on 3.10 GHz clock speed, 16GB of RAM, and deployed on Ubuntu 14.04.The replication factor is kept as the default value 3, and the block size of HDFS is chosen as 64Mb. The experimental datasets are basically standard auto-generated.We use Hadoop version 2.6.2 and java version 1.8.0_65.

To evaluate our approach, the same datasets are stored in three different ways. Initially, the datasets are uploaded directly on HDFS without grouping in any format. Then the datasets are uploaded into HDFS using Hadoop Archive (HAR format).

Finally, we stored files through the SFA server to compare the result of the previous strategies with our approach. The evaluation was based on memory usage and time-cost for writing and reading small files.In the HAR case, original files are deleted at the end of HAR merging.

4.1 Comparison of the NameNode Memory Usage

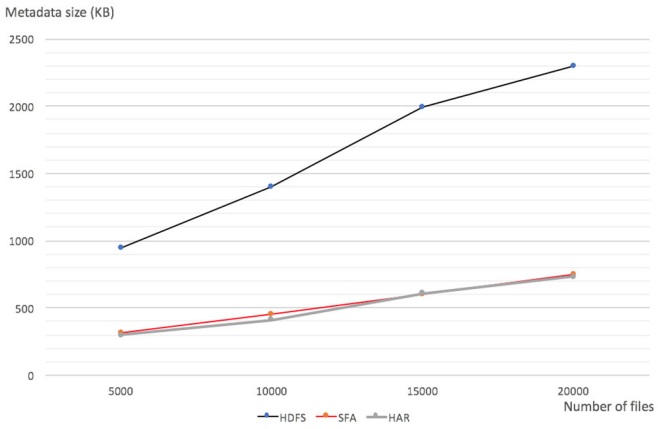

Figure. 12: NameNode memory usage

According to “Fig. 12”, when we store small files through SFA, or in HAR format, The NameNode memory consumption is too low due to the reduced entries of metadata, as each list of files in MapFile or HAR file require only the related metadata for the whole merged file.

4.2 Comparison ofthe required time for storage

To compare the required time for storage, we uploaded a set of datasets, from 1000to 20 000 small files, in 3 ways as described for memory usage evaluation.

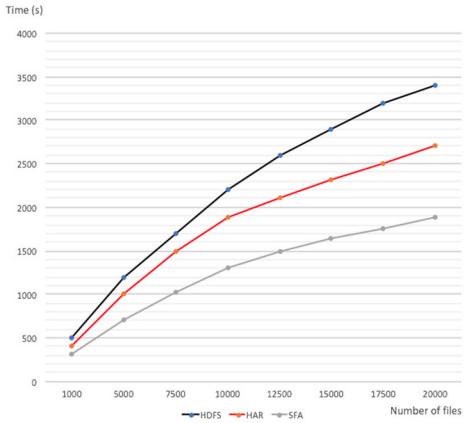

Figure. 13 Comparison of the required time for storage

According to“Fig. 13”, the SFA greatly outperforms native HDFS and the HAR. This is because combing clients’ files and attributes aggregation allow better block allocations, and reduces the NameNode requests for each client’s writing request.

4.3 Comparison of the random and sequentialreading time of small files

To evaluate the reading time, we selected randomly different number of small files from each previous upload case (HDFS, HAR and SFA). We downloaded randomly 200, 500, 1000, 1500, 2000, 4000, 6000 and 8000 files from the total amount (20 000 files) in each case. Respectively we evaluate the required time for each download operation.

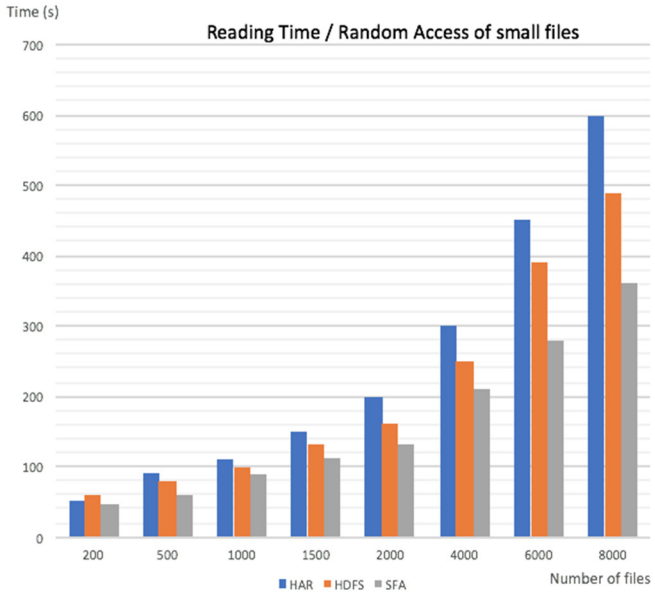

Figure. 14 Comparison of the reading time of random small files

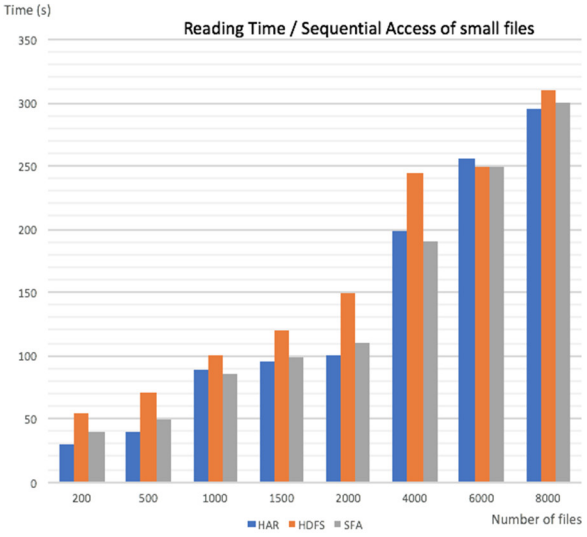

Figure. 15 Comparison of the reading time of sequential small files

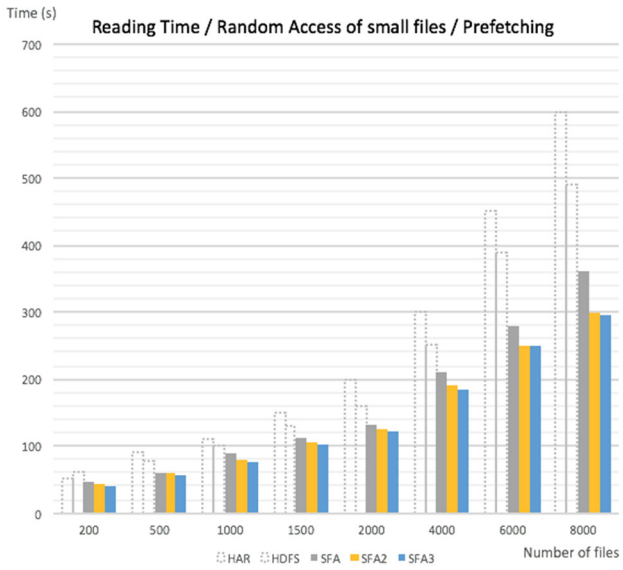

Figure. 16 Comparison of the reading time of random small files / 3 times

According to"Fig. 14" and "Fig. 15",when reading random selected files, SFA can reduce the access time by 24% and 13% compared to HDFS and HAR respectively. When reading sequential lists of files, SFA outperforms HDFS but it remains a bit slower than HAR. The reason is that the increase of the number of files can increase the number of seek operations on DataNodes through the index structure of MapFiles, which can result in the higher reading latency. For SFA, reading random files results in better access efficiency than reading sequential lists of files. Moreover, according to "Fig. 16",the random-access time can be improved after the second and third call of the same sets of small files, which proves clearly the efficiency of prefetching and caching mechanism on the DataNodes and the SFA sever.

5 Conclsion and Future Works

In this paper, we focus on improving the cluster performance while processing small files. The SFA server has been used to combine clients' small files into MapFiles. This technique has been improved in the global approach, which consist of grouping elements in specific order. Sorting small files per categories, up to the application logic, is a promising concept that can improve the reading time access. In fact, the memory usage has been addressed as the main problem in many researches, while in this study, we consider it as only one part of the small files problem equation. Our approach addresses the aspect of how the distribution of small files in a specific format can influence the cluster performance. Different strategies are now adopted in Hadoop to solve the small files problem, but there is a lack of standardization, as most of the solutions remain useful in specific environmentsbut not in others. Offering a system to analyze different aspects of the small files problem can help organizations to understand better the real factors that control the impact of their datasets.

REFERENCES

[1] B. White, T. Yeh, J. Lin, and L. Davis, "Web-scale computer vision using mapreduce for multimedia data mining," in Proceedings of the Tenth International Workshop on Multimedia Data Mining. ACM, 2010, p. 9.

[2] K. Wiley, A. Connolly, J. Gardner, S. Krughoff, M. Balazinska, B. Howe, Y. Kwon, and Y. Bu, “Astronomy in the cloud: using mapreduce for image co-addition,” Astronomy, vol. 123, no. 901, pp. 366–380, 2011.

[3] W. Fang, V. Sheng, X. Wen, and W. Pan, “Meteorological data analysis using mapreduce,” The Scientific World Journal, vol. 2014, 2014.

[4] F. Wang and M. Liao, “A map-reduce based fast speaker recognition,” in Information, Communications and Signal Processing (ICICS) 2013 9th International Conference on. IEEE, 2013, pp. 1–5.

[5] K. P. Ajay, K. C. Gouda, H. R. Nagesh, “A Study for Handelling of High-Performance Climate Data using Hadoop, Proceedings of the International Conference, pp: 197-202, April 2015.

[6] D. Q. Duffy, J. L. Schnase, J. H. Thompson, S. M. Freeman, and T. L. Clune, “Preliminary evaluation of mapreduce for high-performance climate data analysis,” 2012.

[7] C. Shen, W. Lu, J. Wu, and B. Wei, “A digital library architecture supporting massive small files and efficient replica maintenance,” in Proceedings of the 10th Annual Joint Conference on Digital Libraries (JCDL '10), pp. 391–392, June 2010

[8] A. Thusoo, Z. Shao, S. Anthony, D. Borthakur, N. Jain, J. Sen Sarma, R. Murthy, and H. Liu, “Data warehousing and analytics infrastructure at facebook,” in Proceedings of the 2010 ACM SIGMOD International Conference on Management of data. ACM, 2010, pp. 1013–1020.

[9] J. K. Bonfield and R. Staden, “ZTR: A new format for DNA sequence trace data”, Bioinformatics, vol. 18, no. 1, (2002), pp. 3–10.

[10] T. Nukarapu, B. Tang, L. Wang, and S. Lu, “Data replication in data intensive scientific applications with performance guarantee,” IEEE Transactions on Parallel and Distributed Systems, vol. 22, no. 8, pp. 1299–1306, 2011.

[11] Y. Gao and S. Zheng, "A Metadata Access Strategy of Learning Resources Based on HDFS," in proceeding International Conference on Image Analysis and Signal Processing (IASP), pp. 620—622, 2011.

[12] J. Xie, S. Yin, et al. “Improving MapReduce performance through data placement in heterogeneous Hadoop clusters”, In 2010 IEEE International Symposium on Parallel & Distributed

[13] G. Mackey; S. Sehrish; J. Wang. Improving metadata management for small files in HDFS. IEEE International Conference on Cluster Computing and Workshops (CLUSTR). 2009. pp.1-4.

[14] C. Vorapongkitipun; N. Nupairoj. Improving performance of small-file accessing in Hadoop. IEEE International Conference on Computer Science and Software Engineering (JCSSE). 2014. pp.200-205.

[15] Patel A, Mehta M A. A novel approach for efficient handling of small files in HDFS, 2015 IEEE International Advance Computing Conference (IACC), pp. 1258-1262.

[16] Y. Zhang; D. Liu. Improving the Efficiency of Storing for Small Files in HDFS. International Conference on Computer Science & Service System (CSSS). 2012. pp.2239-2242

[17] D. Dev; R. Patgiri. HAR+: Archive and metadata distribution! Why not both?. IEEE International Conference on Computer Communication and Informatics (ICCCI). 2015. pp.1-6.

[18] P. Gohil; B. Panchal; J. S. Dhobi. A novel approach to improve the performance of Hadoop in handling of small files. International Conference on Electrical, Computer and Communication Technologies (ICECCT). 2015. pp.1-5.

[19] Hoare CAR. Quicksort[J]. The Computer Journal, 1962, 5(1): 10-16

[20] E.J. O'Neil, P.E. O'Neil, and G. Weikum, "An Opti- mality Proof of the LRU-K Page Replacement Algorithm," J. ACM, vol. 46, no. 1, 1999, pp. 92-112