Recent Trends in Intensive Computing
M. Rajesh et al. (Eds.)

doi:10.3233/APC210205

A Novel Technique for Handling Small File Problem of HDFS: Hash Based Archive File (HBAF)

Vijay Shankar Sharma[a, 1,] and N.C Barwar[b]
[a] *Research Scholar, Dept. of CSE, M.B.M Engineering College, Jodhpur, India*
[b] *Professor & Head, Dept. of CSE, M.B.M Engineering College, Jodhpur, India*

Abstract. Now a day's, Data is exponentially increasing with the advancement in the data science. Each and every digital footprint is generating enormous amount of data, which is further used for processing various tasks to generate important information for different end user applications. To handle such enormous amount of data, there are number of technologies available, Hadoop/HDFS is one of the big data handling technology. HDFS can easily handle the large files but when there is the case to deal with massive number of small files, the performance of the HDFS degrades. In this paper we have proposed a novel technique Hash Based Archive File (HBAF) that can solve the small file problem of the HDFS. The proposed technique is capable to read the final index files partly, that will reduce the memory load on the Name Node and offer the file appending capability after creation of the archiv.

Keywords. HDFS, Small File Problem, Meta Data Management, Hash Function, HAR, Map File, SSHF, HT-MMPHF, Merging Technique.

1. Introduction

Hadoop is open-source technology to handle the vast amount of unstructured and big data, which offers the wide range functionality in comparison to the traditional relational data bases. The file system of Hadoop is known as the Hadoop Distributed File System (HDFS) that is based on the master slave architecture. In this architecture there is a Name Node that acts as a master with processing capabilities and stores the meta-data information of the files stored in the file system. There are number of Data Node's that act as the slave means these Data Node's are only used to store the data, no processing is required at the Data Node's. Once a file is stored on the HDFS it is divided in the 128 MB size blocks and then these blocks are stored on the HDFS. The size of the HDFS block is variable means client can configure the size of the HDFS block as per the requirement, by default it is 128 MB. To ensure the availability of the data, HDFS replicate the data blocks on the Data Node's and it will be decided by the replication factor that is by default 3, means each data block is written on the three Data Node's, in case if any one of the Data_Node's gets down then data block can be

[1] Vijay Shankar Sharma, Research Scholar, Dept. of CSE, M.B.M Engineering College, India
Email:vijay.mbmit09@gmail.com.

recovered from the other Data Node's that is available as a replica. Handling of large files in HDFS is done efficiently as it is designed according to the application of the large files. There are number of major platforms which generate the small files i.e., Facebook, Twitter, Instagram, LinkedIn, Amazon, Flip cart, snap deal etc. This list of platforms is very long therefore it is easy to understand small files generation platforms by the application areas i.e., social networking sites, e-commerce websites, educational websites, research and analysis websites, weather forecast websites, entertainment websites, log files generated by the severs, health care data etc. a file is termed as the small file, if it is less than the size of the default HDFS block size. The application area of the small files is very vast therefore the importance of small files in analytics and in technology is very crucial and important. Unfortunately, majority of the distributed file systems are not designed to deal with the problem of massive small files. Massive small files generate the large amount of the meta-data at the central node in the distributed system that will degrade the overall performance of the distributed file system. Unfortunately, HDFS is also not capable to deal with the massive number of small files; Name Node in the HDFS will be overloaded due to excessive meta-data generation while dealing with the massive number of the small files.

In this paper we have proposed a novel technique to handle small file problem of the HDFS called as "Hash Based Index File (HBAF Archive)". The major contribution of our technique is that one can directly access the small files meta-data without use of any caching mechanism. Now, there is no need to read the index file entirely in the memory, only required part of the index file will be read and loaded to the memory. To read the index file partly we have used the special-order preserving hash function: Hollow Trie Monotone Minimal Perfect Hash Function (HT-MMPHF) [1] [2] with index file. This function identifies the location of the searched file meta-data in the index file and calculates the limit (how much index file is to be read) and offset of the index file. With help of limit and offset our technique seeks the index file and loads the required meta-data to the memory. To access index file randomly may be an expensive operation in case of the large index files therefore to limit the size of index files another special hash function: Scalable-Spittable Hash Function (SSHF) [3] [4] [5] is used that will dynamically distribute the meta-data of the massive number of small files to the various index file in place of the single index file. The remaining section of the paper is as follows; Section 2 presents the literature review on the existing techniques to deal with small file problem. Section 3 presents the proposed technique in detail with explanation of the HBAF creation algorithm and appending files after creation of the archive. Section 4 presents details of experimental setup and analysis of the result. At last Section 5 briefs the conclusion and future work.

2. Related Work: A Brief Survey

Jude Tchaye-Kondi et al. [6] proposed a archive file system, known as the Hadoop Perfect File. To access and distribute the meta-data of a particular file, special hash functions with order preserving capacity are used. Jian-feng Peng et al. [7] proposed a new variant to the HDFS with caching and merging module. The working of these modules are interrelated, to utilize the memory space efficiently the co-related files are merged and a special cache is designed for the fast access of the frequently accessed data. To solve the problem of the small files Xun Cai et al. [8] proposed the optimized

merging algorithm that is based on the correlation and distribution of the files. Hwajung Kim et al. [9] proposed a digital archive the will significantly reduce the storage I/O operation by modifying the inode structure of existing file system at both primary and secondary memory level. Yanfeng Lyu et al. [10] presented an optimized approach that will reduce the name node memory usage and access time, while handling massive number of small files. To improve the efficiency of read/write operation for the small files Xiong Fu et al. [11] proposed block replica placement algorithm. This algorithm also suitable for the cloud environment. Qi Mu et al. [12] proposed improved storage architecture for the massive small files. This architecture is based on the use of the secondary indexes. Tao Wang et al. [13] proposed a technique called as "Modified PLSA" that handles the massive small files by establishing the association among the application, access file and access tasks. The balancing of the data blocks can also be a measure of handling massive small files, Hui He et al. [14] proposed a unique algorithm that will consider the even utilization of the data blocks while merging the small files. Songling Fu et al. [15] proposed a technique that will reduce the memory required for the meta-data management while dealing with millions of small files. The proposed technique is called as the "iFlatLFS", which is based on the concept of the flat storage architecture. Yingchi Mao et al. [16] proposed SIFM technique that will use the multi level indexing for handling million of small files efficiently. Bo. Dong et al. [17] categorize the small files logically and structurally. On the basis of this division prefetching and merging of small files is applied to the structurally oriented small files and prefetching and file groping concept is used for the logically oriented small files. Ahad M. A et al. [18] proposed a dynamic merging technique. This technique identifies the small files by their size and type and Two-Fish cryptographic technique is used to secure the data in the file system.

3. Proposed Architecture

It is obvious that accessing of small files in HDFS is a complex and time-consuming task; therefore, to achieve fast meta-data access for small files we have proposed a Hash Based Archive File (HBAF) method. "Write-Once, Read-Many" is the prime property of the Hadoop Distributed File System, to keep this property in mind we designed our proposed technique in such a way that we will be able to append new files after creation of the archive. Our proposed technique will provide better processing and accessing performance in comparison to the Hadoop Archive (HAR). As the Figure.1 depicts that our HBAF Archive consist several slave indices files that will be generated from the temporary master index file using SSHF. Apart from the several index files HBAF also consists part file (file created after merging small files) and master name file which consists name of all the small files that be appended to the part file. Index files are responsible to store the meta-data of the small files; the selection of the particular index file will be done by the special hash function. In our technique two-level hashing will be used, at level-1 particular index file is identified by the SSHF and at level-2 HT-MMPHF order preserving hash function is used to locate the particular files meta-data location in the index file. The proposed approach HBAF will improve the performance in two ways, one is by the concept of merging, all the small files are merged therefore memory utilization will be improved and overall performance of Name Node will be better, due to the reduced memory load. Another way is the use of the two-level hash functions to build the index files for small files meta-data that will

provide fast access to the small files. While merging the small files, parallel multiple part files are created, this parallelism will merge the small files comparatively faster than the HAR. The level of parallelism can be increased or decreased, by default it set as 'two'. The whole process can be summarized by Algorithm 1.

Algorithm 1: HBAF Creation & Updation

Step-1: **Initial Variable Declaration and their Initialization**
1.1 small files // a set of small files;
1.2 slave index file // temporary index files generated by SSHF;
1.3-part file; // creation of initial part file
1.4 temporary master index file; // creation of the initial temp master index file
1.5 master name file; // creation of the master's name file
1.6 meta-data; // creation of string variable for storing meta-data of small file
1.7 small file name; // creation of string variable for storing name of small file
1.8 final index files; // creation of initial empty final index file
Step-2: Process of Merging Small Files and Building Client-Side slave_index_files
2.1 start of loop-1; // for each small file from small files
2.2 merge each small file to the part file;
2.3 copy the small file meta-data to the meta-data variable;
2.4 copy the name of small file to the small file name variable;
2.5 append the meta-data of the small file to the temporary master index file;
2.6 append the name of the small file to the master name file;
2.7 provide unique id to the slave index file and final index file by using SSHF;
2.8 append the value of meta-data to the unique slave index file created in previous step;
2.9 check the threshold limit of the slave_index_file, if limit reaches its maximum, then create another unique slave index -file and final index file using SSHF and then continue with the append operation of meta-data
2.10 end of loop-1;
Step-3: Sorting slave_index_files and building final_index_files
3.1 start of loop-2 // for each slave_index_files with unique id
3.2 sort the slave index file's meta-data
3.3 implement the HT-MMPHF for all the all-slave index files
3.4 associate HT-MMPHF to the respective final index files according to their mirror slave index files
3.5 copy all the meta-data entries from slave index files to the corresponding final index files along with order preserving mechanism of HT-MMPHF;
3.6 end of loop-2;

Initially temporary master index file and master name file are created, temporary master index file is used for the purpose of backup once the slave index files are created finally this temporary master index file will be deleted, master name file is a file which reside permanently with the HBAF archive and hold the names of all the small files to process. Before appending to the part files, small files can be compressed at client side and can take advantage of fast processing at client side in comparison to the HDFS. A threshold limit on the capacity of part files is fixed and checked regularly while appending/adding the files to the part file. Once the threshold limit reached to its

maximum, the new part file will be created and rests of the small files are appended to the newly created part file. There is also requirement for imposing the limit on the size of index file because when each time a random seek operations is performed a new connection is established to read a file from various data node blocks therefore it is desirable that index file should be less than the size of the HDFS block. One of the important concerns in our approach is the dynamic distribution of the small files metadata to the various slave index files, to implement this dynamic distribution we are using SSHF, later on these slave index files will be converted to the final index files. The process of building final index files is accomplished in two phases, the first phase of building final index files starts along with the merging process of small files, when a small file is added/appended to the part file, simultaneously its meta-data and name of the file will be added to the temporary master index file and master name file respectively after that with the help of SSHF its meta-data will be added to the corresponding slave index file.

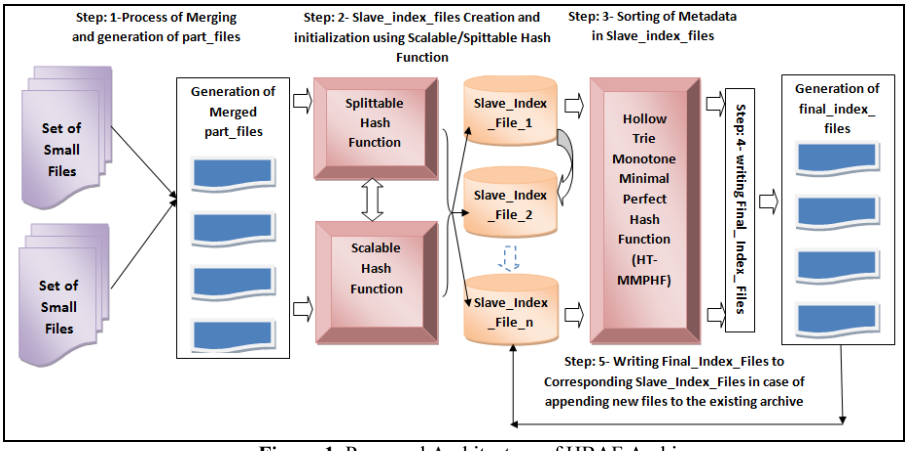

Figure 1. Proposed Architecture of HBAF Archive

SSHF belongs to the class of extendible hashing Zhang D. et al. [5] that uses dynamic hashing technique to allocate meta-data of the small files to the slave index files. In this technique hash is considered as the bit string and uses an ordered tree data structure Tarjan R.E. et al. [19] for lookup purpose. Figure 1 the decision of choosing particular slave index file for the entry of file's meta-data will be done by the hash function that will determined by the last two bits of the bit string of a file name hash value. The entries that have same pattern at last bits will belong to the same slave index file. While addition and deletion operations are performed on slave index files they grow and shrink dynamically with help of Scalable-Spittable Hashing. If a slave index file reaches its threshold limit it spilt and generate a new slave index file. The spilt hash operation is responsible for the dynamic creation of the slave index files at the same time slave index files can be directly accessed during look-up. Creation of slave index files and corresponding final index files is a parallel process, both files are created simultaneously. Re-arrangement of the meta-data entries is done in newly created slave index file during the spilt hash operation as it is highly essential for the synchronization of old and newly spitted slave index file.

HT-MMPHF is a perfect hash function, in this function a set of 'p' key that are static type are mapped with the 'q' numbers of integer type without any collision and there should be the value of integer number is always greater than or equal to the value of static key (q>=P). When value of 'q' and 'p' is equal, the hash function satisfies the 'minimal' property and hash function is called as the minimal perfect hash function. To preserve the order of keys we are supposed to use order preserving minimal perfect hash function, this function return the integer values strictly in the order of the static key, hence using this function the lexicographic order of meta-data entries in final index files are kept in order. Meta-data entries in slave index file are sorted in lexicographic order on the basis of file name hash values, these hash values act as the keys of the hash function and finally the minimal perfect hash function is created and added in the beginning of the final index files. At last, after writing the entire slave index files to the corresponding final index files, temporary master index file will be deleted. The main advantage of this minimal function is that its time and space complexity (logarithmic) is much lesser in comparison to the other comparative hash functions. As the meta-data entries in final_index_files are sorted and can be accessed directly therefore access time of particular record is minimum (Big O (1)) [2].

4. Experimental Setup & Result Analysis

To test the proposed HBAF Archive and other competitive archives, a cluster of 5 nodes is being setup. The configuration of the Name Node and Data Node are same, that is Intel® core™ i5-7500 CPU@3.40GHz, 64-bit Operating System with 4 GB Installed RAM. Ubuntu 18.04.1 LTS is used as the operating system with open JDK-11.0.4 in the system in the cluster. The latest version of Hadoop (3.1.3) is used in all the machines over the 1 GBPS (Backbone) / 100 MBPS Network. The replication factor and block size of the HDFS are set to its default values that is 3 and 128 MB respectively. For the purpose of the testing, we created five data sets with different number of files i.e., 10000, 20000, 30000, 60000, 120000. The size of files in theses data sets will ranges from 1 KB to 1 MB. To evaluate the performance of the archives with proposed technique we have analyzed the few parameters while creation of archives and few parameters after creation of the archives i.e., Time To Create Archive (Milli-Seconds), Meta Data Usage (Bytes) are the parameters that will be analyzed while creating the archives and Time Required to Randomly Accessing 10, 50 and 100 Files from different archives with caching (Milli-Seconds), Time required to randomly accessing 10, 50 and 100 files from different archives without caching (Milli-Seconds) are the parameters that will be analyzed after creating the archives. The concept of caching is all about the using client's memory while accessing the files from the archives therefore resultant access time can be reduced. HAR Archive and Map File Archive supports the caching means corresponding index files are loaded into the client's memory when accessed first time after that corresponding files meta-data will be perfetched on the basis of LRU Bok K. et al. [20] Dong Bo. Et al. [21]. This caching mechanism put the burden on the client's memory and will be problematic where memory is limited at the client side therefore in our proposed design HBAF Archive; we implemented the concept of the centralized caching of HDFS [22] and use the memory of the Data Node's for managing the caching operation

4.1. Time to Create Archive (Milli-Seconds)

Map File Archive will take the minimum time for creation and HAR Archive will take the highest time for creation. Our proposed approach HBAF Archive lies in between Map File Archive and HAR Archive. Experimental result shows that proposed HBAF Archive is 28% to 34% (result varies for different data sets) faster than the HAR Archive. The Archive creation time of Map File Archive is 56% to 58% faster than the Hadoop Archive creation time and 24% to 40% faster than proposed HBAF Archive. Figure.2 depicts that although Map File Archive is taking minimum time for creation as it is based on the sequential file approach but this cannot be a performance measuring parameter of the archive, we have considered this parameter only to show that our proposed approach 'HBAF Archive' is taking moderate time for creation of the archive that is acceptable.

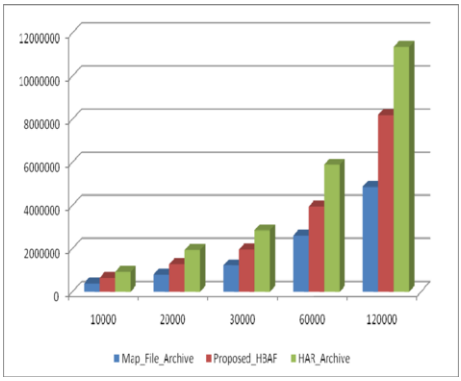

Figure 2. Time to Create Archive (Milli-Seconds)

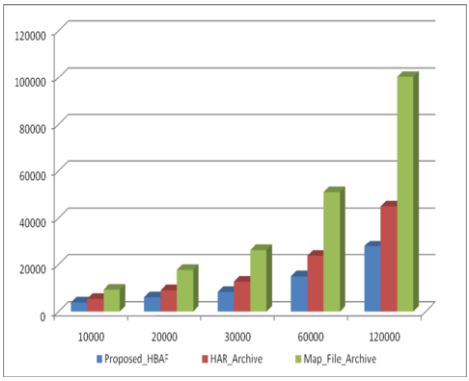

Figure 3. Meta-data Usage (Bytes)

4.2. Meta Data Usage (Bytes)

Meta Data Usage of the proposed HBAF Archive is minimum in comparison with the HAR Archive and Map File Archive. There is one important point to note that when we increase the size of our datasets (i.e., 10000, 20000, 30000, 60000, 120000), we are continuously getting the better results means there is need of lesser space to manage the meta-data for larger datasets. This phenomenon proves our theory that proposed HBAF Archive will be able to handle millions of small files while consuming the minimum space for storing the meta-data of small files. Experimental result shows that in terms of Meta Data Usage proposed HBAF Archive perform 28% to 38% (result varies for different data sets) better than the HAR Archive and 58% to 72% better than the Map File Archive. These results can be clearly visualized in the Figure.3, which will show the strength of our proposed approach.

4.3. Time Required to Randomly Accessing 10, 50 and 100 Files from different Archives with Caching (Milli-Second)

When randomly accessing 10 files, the access time of proposed HBAF Archive is 23% to 73% (result varies for different data sets) faster than the HAR Archive. Experimental result shows that Map File Archive will take the highest access time, HBAF Archive is 14 to 18 time faster than the Map File Archive. The minimum access time will be taken

by the native HDFS but it can clearly see that proposed HBAF Archive is very close to the native HDFS. Native HDFS will perform 8% to 21% faster than HBAF Archive. As the Figure.4 depicts that HBAF Archive will continuously improve the access time with the increase of the number of files in the data set. To ensure the correctness and preciseness of the results we also accessed 50 and 100 files from the HBAF Archive and found the approximately same pattern results. In case of accessing 50 files, HBAF Archive will be 4 to 34% faster than the HAR Archive. In this case it has been noted that for lower data sets there is much lesser difference in the performance of the HAR Archive and HBAF Archive. Experimental result shows that Map File Archive will take the highest access time; HBAF Archive is 18 to 20 times faster than the Map File Archive. As the Figure.5 depicts that Native HDFS will perform 10% to 21% faster than HBAF Archive and HBAF Archive results improve for the larger data sets. If we calculate the average performance of the native HDFS in comparison with our proposed HBAF then it is overall 15% faster. In future the work can be carried out to improve our proposed technique in this direction. In case of accessing 100 files from the archives, HBAF Archive is 1% to 18% faster than the HAR Archive.

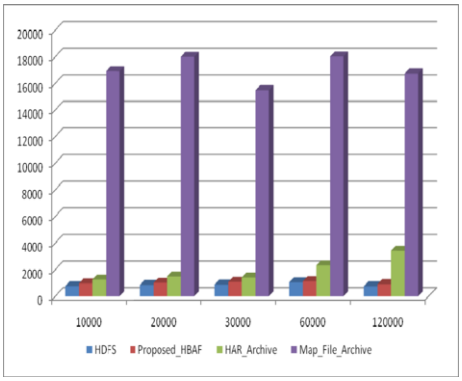

Figure 4. Access Time for 10 Files with Caching

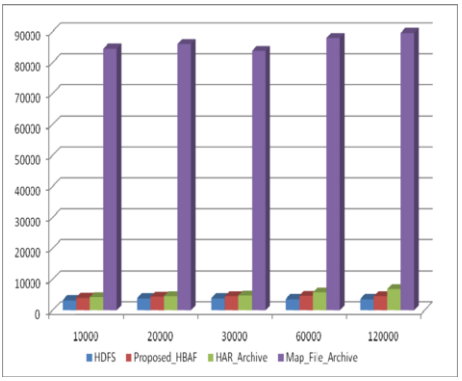

Figure 5. Access Time for 50 Files with Caching

Experimental result shows that Map File Archive will take the maximum access time and HBAF Archive is 18 times to 22 times faster than the Map File Archive. Figure.6 depicts that there is no impact on the performance of the Map File Archive by varying the number of files in the data sets. Native HDFS will perform 12% to 22% faster than HBAF Archive, in terms of average results Native HDFS is 15 to 16% faster than our HBAF Archive.

4.4. Time Required to Randomly Accessing 10, 50 and 100 Files from different Archives without Caching (Milli-Second)

As the Figure.7 depicts that the minimum access time will be taken by the native HDFS and there is very minor and negligible difference between the performance of the HBAF_Archive and Native HDFS. Native HDFS will perform 5% to 18% faster than HBAF Archive. The pattern of performance improvement of HBAF Archive is same as with caching enabled, HBAF Archive will perform better for the larger data sets. To analyze the pattern, correctness and preciseness of the results, we also accessed 50 and 100 random files from the archives and found the approximately same pattern results while caching is disabled. Experimental result shows that in case of accessing 50 files,

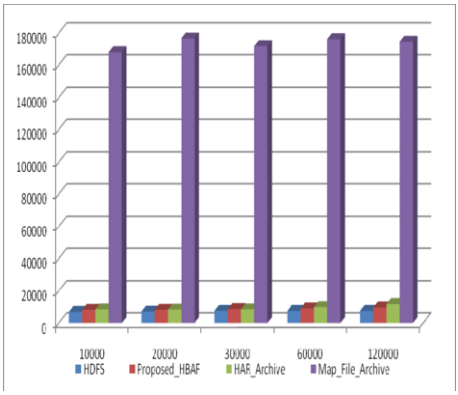

Figure 6. Access Time for 100 Files with Caching

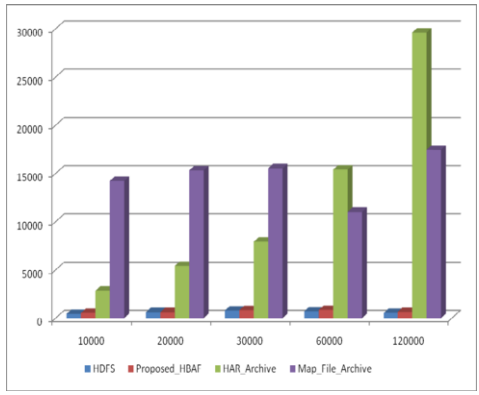

Figure 7. Access Time for 10 Files without Caching

HBAF_Archive will be 5 to 45% faster than the HAR_Archive that will be same as the accessing 10 files, means the impact of increasing number of files for accessing is negligible or is very less. As the Figure. 8 depicts that Map_File_Archive will take the highest access time; HBAF_Archive is 26 to 28 times faster than the Map_File_Archive, this will prove that when cache is disabled our HBAF_Archive will perform better than cache enabled environment.

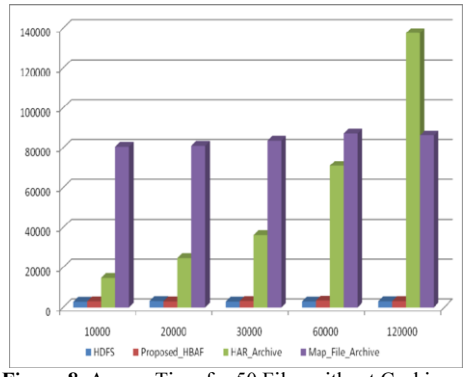

Figure 8. Access Time for 50 Files without Caching

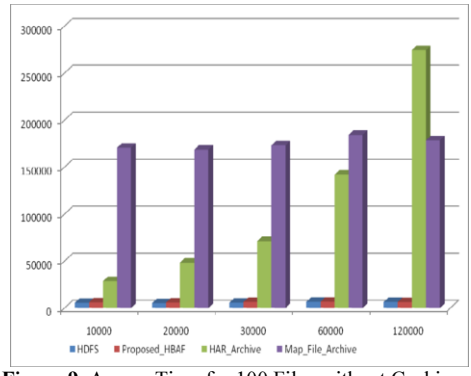

Figure 9. Access Time for 100 Files without Caching

Native HDFS will perform 8% to 14% faster than HBAF_Archive but the performance of Native HDFS degrades in comparison to the cache enabled environment. As the Figure.9 depicts that in case of accessing 100 files from the archives, HBAF_Archive is 5% to 43% faster than the HAR_Archive that is more or less equal to the previous reading while accessing 10 and 50 files. Experimental result shows that as usual Map_File_Archive will take the maximum access time and HBAF_Archive is 26 times to 28 times faster than the Map_File_Archive, these results are same as it was with accessing 50 files. Native HDFS will perform 8% to 11% faster than HBAF_Archive. There are also few cases where proposed HBAF_Archive will perform better than the Native HDFS but the difference in the performance is minor therefore it can be neglected.

5. Conclusion and Future Work

The purpose behind the design of the HDFS is to manage the daily growing big data/large files efficiently and ensure the data availability at all time along with fast access of the data. The design of HDFS is not compatible with the small files means handling small files is quite complex in the HDFS in terms of accessing of small files and their meta-data management. There is requirement of the mechanism that will handle the small files efficiently as well as reduce the Name_Node memory usage and access time for the small files. Number of researchers worked in this field and proposed various solutions to efficiently handle the small files. Most of the solutions provided reduce the meta-data usage of the Name_Node by shifting the process of indexing at the client side but these approaches are lagging behind while analyzed in terms access time. There is a requirement of such a method that will reduce the Name_Node memory usage as well as provide fast access to the small files. This paper presents Hash Based Archive File (HBAF) that will provide reasonable fast meta-data access for the small files along with the appending facility after creation of the archive. Data Node's are used for the purpose of the caching; this concept will reduce the memory pressure from the client side that results in the reduced access time for the small files. Small file's meta-data will be placed to the particular slave index file with help of the special hash function (SSHF). The use of this hash function for placement of meta-data will lead to the efficient seek operation for accessing the content of the small files. To preserve the order of the meta-data stored in the final index files a order preserving hash function (HT-MMPHF) is used. With help of this hash function, we will be able to read the final index files partially means when there is a access request for a particular small file's meta-data, final index files are read partly (only the part which contain the accessed file's meta-data), there is no need to read the entire index file hence this will result in the faster meta-data access form the final index files.

Experimental result shows that Proposed HBAF Archive performing better than the HAR Archive and Map File Archive. It is clear that when caching is disabled the access time will be very higher in case of the HAR Archive, its due to the multi level index files in the HAR Archive, but our approach is not affected form the impact of caching enable or disabled as our approach is not dependent on the client's memory. There is little limitation in our approach that cannot be addressed in this paper; these limitations will be resolved in future. The following are the key points for the future work on our approach.

- Experimental result shows that in terms *of* access time our approach performing better than the HAR Archive and Map File Archive but when it compared to the native HDFS, the results are not satisfactory; the further work can be carried out to make our approach better than the native HDFS.
- A number of other hash function combination can be used to further improve the performance of the proposed HBAF Archive.
- In this paper text files are considered as data sets, proposed HBAF Archive can also be modified for other file formats and results can be compared with original one.
- The minimization of client memory usage can be carried out, there is need to identify the various factors in our approach that are still using the client memory i.e., client memory used by the hash functions.

- We have proposed the appending facility to proposed HBAF Archive; deletion facility is still a future work in our approach.
- While merging small files, we have not applied any compression technique, implementing a compression technique at HDFS block level will be a future work for our approach.

References

[1] Belazzougui, Djamal & Boldi, Paolo & Pagh, Rasmus & Vigna, Sebastiano. (2009), Theory and Practice of Monotone Minimal Perfect Hashing. 2009 Proceedings of the 11th Workshop on Algorithm Engineering and Experiments, *ALENEX 2009. 132-144. 10.1137/1.9781611972894.13.*

[2] Belazzougui, D., Boldi, P., Pagh, R., & Vigna, S. (2009). Monotone minimal perfect hashing: searching a sorted table with O(1) accesses. *SODA*.

[3] Mendelson, G., Vargaftik, S., Barabash, K., Lorenz, D.H., Keslassy, I., & Orda, A. (2018). AnchorHash: A Scalable Consistent Hash. *ArXiv, abs/1812.09674*.

[4] Claessen, K., & Palka, M. (2013). Splittable pseudorandom number generators using cryptographic hashing. *Haskell '13*.

[5] Zhang D., Manolopoulos Y., Theodoridis Y., Tsotras V.J. (2018) Extendible Hashing. In: Liu L., Özsu M.T. (eds) Encyclopedia of Database Systems. Springer, New York, NY. https://doi.org/10.1007/ 978-1- 4614-8265-9_741

[6] Tchaye-Kondi, et al. (2019) Hadoop Perfect File: A Fast Access Container for Small Files with Direct in Disc Metadata Access. ArXiv.org, 26 Apr. 2019, arxiv.org/abs/1903.05838.

[7] Peng, J., Wei, W., Zhao, H., Dai, Q., Xie, G., Cai, J., & He, K. (2018). Hadoop Massive Small File Merging Technology Based on Visiting Hot-Spot and Associated File Optimization: 9th International Conference, BICS 2018, Xi'an, China, July 7-8, 2018, Proceedings. 10.1007/978-3-030-00563-4_50.

[8] Cai, Xun, et al. (2018) An Optimization Strategy of Massive Small Files Storage Based on HDFS. Proceedings of the 2018 Joint International Advanced Engineering and Technology Research Conference (JIAET 2018), 2018, doi:10.2991/jiaet-18.2018.40.

[9] Kim, H., & Yeom, H. (2017). Improving Small File I/O Performance for Massive Digital Archives. 2017 IEEE 13th International Conference on e-Science (e-Science). doi:10.1109/escience.2017.39.

[10] Lyu, Y., Fan, X., & Liu, K. (2017). An Optimized Strategy for Small Files Storing and Accessing in HDFS. 22017 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC). doi:10.1109/cse-euc.2017.112.

[11] Fu, X., Liu, W., Cang, Y., Gong, X., & Deng, S. (2016). Optimized Data Replication for Small Files in Cloud Storage Systems. Mathematical Problems in Engineering, 2016, 1–8. doi:10.1155/2016 /4837894.

[12] Mu, Q., Jia, Y., & Luo, B. (2015). The Optimization Scheme Research of Small Files Storage Based on HDFS. 2015 8th International Symposium on Computational Intelligence and Design (ISCID). doi:10.1109/iscid.2015.285.

[13] Wang, T., Yao, S., Xu, Z., Xiong, L., Gu, X., & Yang, X. (2015). An Effective Strategy for Improving Small File Problem in Distributed File System. 2015 2nd International Conference on Information Science and Control Engineering. doi:10.1109/icisce.2015.35.

[14] He, H., Du, Z., Zhang, W., & Chen, A. (2015). Optimization strategy of Hadoop small file storage for big data in healthcare. The Journal of Supercomputing, 72(10), 3696–3707. doi:10.1007/ s11227-015-1462-4.

[15] 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 Transactions on Parallel and Distributed Systems, 26(12), 3433– 3448. doi: 10.1109/tpds. 2014.2377720.

[16] Mao, Yingchi, et al. (2015), Optimization Scheme for Small Files Storage Based on Hadoop Distributed File System. International Journal of Database Theory and Application, vol. 8, no. 5, 2015, pp. 241–254., doi:10.14257/ijdta.2015.8.5.21.

[17] Dong, B., Zheng, Q., Tian, F., Chao, K.-M., Ma, R., & Anane, R. (2012). An optimized approach for storing and accessing small files on cloud storage. Journal of Network and Computer Applications, 35(6), 1847–1862. doi:10.1016/j. jnca.2012. 07.009.

[18] Ahad, M. A., & Biswas, R. (2018). Dynamic Merging based Small File Storage (DM-SFS) Architecture for Efficiently Storing Small Size Files in Hadoop. Procedia Computer Science, 132,

[19] Tarjan R.E., Werneck R.F. (2007) Dynamic Trees in Practice. In: Demetrescu C. (eds) Experimental Algorithms. WEA 2007. Lecture Notes in Computer Science, vol 4525. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-72845-0_7

[20] Bok, K., Oh, H., Lim, J. et al.(2017) An efficient distributed caching for accessing small files in HDFS. Cluster Compute 20, 3579–3592 (2017). https: //doi.org/10.1007/s10586-017-1147-2

[21] Dong, Bo & Zhong, Xiao & Zheng, Qinghua & Jian, Lirong & Liu, Jian & Qiu, Jie & Li, Ying. (2010). Correlation Based File Prefetching Approach for Hadoop. Proceedings - 2nd IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2010. 41-48. 10.1109/CloudCom.2010.60.

[22] Zhang, Jing & Wu, Gongqing & Xuegang, Hu & Wu, Xindong. (2012). A Distributed Cache for Hadoop Distributed File System in Real-Time Cloud Services. Proceedings - IEEE/ACM International Workshop on Grid Computing. 12-21. 10.1109/Grid.2012.17.