ISSN No. 0976-5697

Volume 8, No. 5, May – June 2017

International Journal of Advanced Research in Computer Science

RESEARCH PAPER

Available Online at www.ijarcs.info

An Optimal Solution for small file problem in Hadoop

Mansoor Ahmad Mir
SEST, Jamia Hamdard
New Delhi, India

Jawed Ahmed
SEST, Jamia Hamdard
New Delhi, India

Abstract: Hadoop is an open source Apache project and software framework for distributed processing of large dataset across large clusters of computers with commodity hardware(used mainly for processing of big Data).HDFS(Hadoop distributed file system and MapReduce(a programming model) are Hadoop's main two components). Hadoop does not perform well while processing large number of small sized files(of the size of the hundreds of KB's or few MB's) posing a heavy burden on NameNode of HDFS and increases the execution time of MapReduce. Hadoop being designed to handle large sized files thus pays the penalty for handling small sized files. This research work provides an introduction to the Hadoop, Big Data and review of the existing work and proposed better efficient technique to handle small file handling problem with Hadoop based on file merging technique, hashing and caching. This technique results in saving memory at NameNode, average memory usage at DataNode and improves the access efficiency as compared to the existing techniques

Keywords: NameNode, DataNode, Amazon EC2, Merging, PreFetching

I. INTRODUCTION

All Hadoop is one of the most popular high performance distributed computing paradigm for Big data analytics. It provides reliable, scalable, distributed computing and storage with the advantage being open source. With the advent of digital technology, as data explosion took place because of the web and grew beyond the ability to be handled by traditional computing system, Apache Hadoop [1] was created by Doug Cutting. It has two main layers one is Hadoop distributed file system (HDFS) and the other is MapReduce programming model. HDFS the storage part, stores large files pretty suitable for streaming and being distributed in nature. It runs on commodity hardware. HDFS can scale to thousands of machines each running each providing same set of functionality and thus providing very high fault tolerance. HDFS has a client server type architecture with the NameNode server playing the master role and the several data nodes performing the client role (slave role)[2] . One of the main strength of the HDFS is its data protection. It protects data by replication to multiple nodes, by default replication factor is 3. MapReduce is a programming model used to process large datasets and make use of computing resources of each of the available server's CPU. It is the processing part. MapReduce as the name suggest consists of two phases one is Map phase and other is Reduce Phase. Map phase divides the files for distributed computing and generates a key value pair and then its followed by the reduction phase. MapReduce uses Jobtracker and TaskTrackers [2].

In Hadoop the storage and computing system are not separate. Hadoop divides the large files into small sized blocks generally of the size of 64MB. NameNode stores the metadata about the data blocks and Data node tores the actual blocks. These blocks are then processed by the MapReduce[3]. But since Hadoop was designed to handle large sized files it suffers when a large number of small sized files need to processed putting heavy burden on the NameNode [4]. Since the memory of the NameNode is limited and processing small files takes more time in I/O rather than processing. Hence posing a challenge. The small sized files are generally word, pdf, PowerPoint, mp3type files.

This research makes use of the caching and file merging techniques to handle this large number of small sized files problem. The technique reduces the memory at the NameNode and DataNode by merging several small sized files. Also the access time and processing is improved by using the concept of caching, which not only results in saving of time but also results in the traffic to the NameNode.

II. REVIEW OF LITERATURE

1. Hadoop Archive

Hadoop Archive (HAR) is the first technique based on archiving technique which packs number of small files into HDFS blocks more efficiently. Files in a HAR need not expanded as they can be accessed directly (HAR is different from rar, zip), as this access is done in main memory. The following diagram illustrates the data model for archiving small files [5].

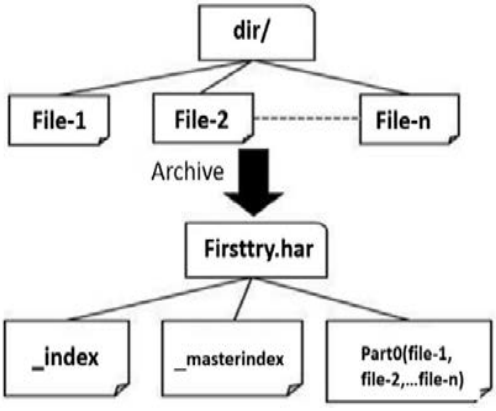

Fig 1. Data model for archiving Small Files [4] [5]

Creating HAR reduces the storage overhead of data on NameNode and reduced map operations in MapReduce program increases the performance.

Creating HAR file: A HAR file can be created using the Hadoop archive command

hadoop archive -archiveName name -p <parent> <source>* <destination> [5]

Which adds all the files being archived into a smaller number of HDFS files by running map reduce job.

Example: hadoop archive -archiveName firstfile.har -p /user/hadoop directory1 directory2 /user/Manu

HAR File layout overhead: File access in this technique requires two index-file read operations as well as one data-file read operations there is a small time overhead in accessing of files. In order to access the required client file, the request is sent to the index of metadata that the archive consists of through the metadata of the archive [5]. File reading in HAR is less efficient and slower than file reading process in HDFS. Map process fails to operate through all the files in the HAR co-resident on a HDFS block. Also upgrading HAR requires the changes to the HDFS architecture, which may become difficult [5].

2 .Improved HDFS

The improved HDFS structure comprises of two parts:

Client component which integrate small sized files into a big file and data node component which satisfies the cache resource management [6]. An improved HDFS framework is represented diagrammatically in figure 5 presented below.

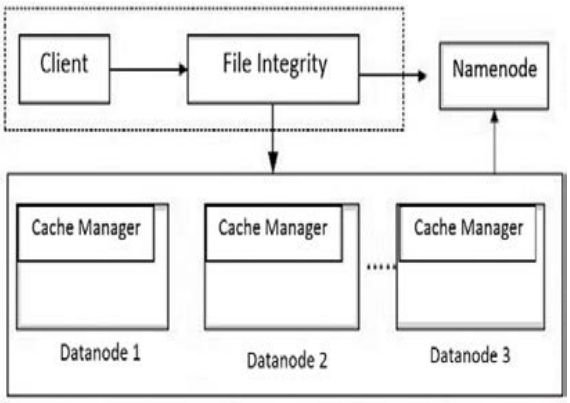

Fig 2. Improved HDFS framework [6]

Improved HDFS model is an index-based model. As per file dependency, files belonging to the same directory are merged into a big file and accordingly indexes are generated for each small file to reduce waste caused by them which reduces the burden of NameNode. Cache policy is used to increase the reading efficiency of small files. The cache manager positioned at DataNode, when reading small sized file data each time, the data in the Cache is checked first. If this resource is not present in the cache, resource can be found in the disk of DataNode. Smaller sized files are sort based on the data present. The small files are integrated into a big file and each big file has an index file which contains the offset and length of each small file. In order to store big file in a single block, the total size of the big file should be less than size of the block. If the size is greater than block size then the multiple blocks are used to store big files separately [6].

3. New HAR

Basically NHAR consist of two aspects:

- Merging small files into larger file to reduce file number and increase access performance.
- Extending the functionality of file management within HAR is similar to a typical file system [7].

NHAR creates the indexing structure to improve the metadata mainframe of HDFS and file accessing performance without changing the HDFS architecture. This design allows NHAR to add more files to be added to the existing archive which is HARs well-known limitation. While reading file from HAR, indexes need to be accessed twice which creates overhead. In order to improve the access performance NHAR model use single-level indexing. NHAR uses index information to create a hash table instead of master-index. This information is divided over multiple index files, as shown in fig.3

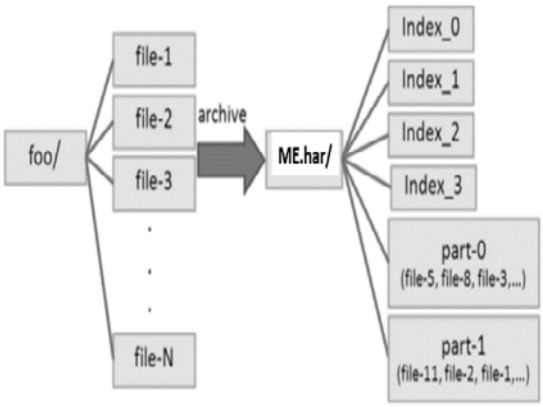

Fig 3. NHAR file structure[7]

Locating the index file containing the metadata is dependent upon the number of index files present. To generate hash code, file names are used which then mods with number of index files [7].

The actual file will be stored in part file, similar to HAR. Fig. 3 presents the hashing mechanism of NHAR.

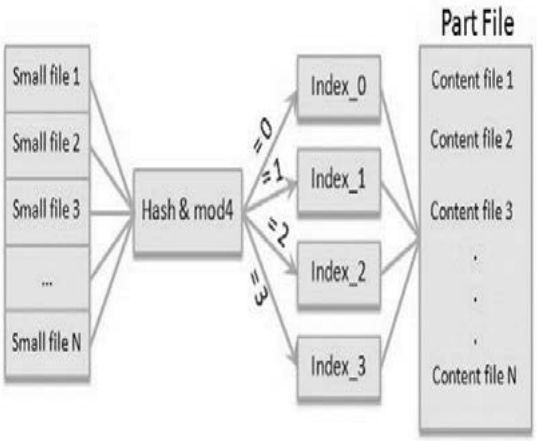

Fig 4. New archive technique [7]

To add a new file to the HAR file, a new HAR file should be created, which is inefficient. In terms of NHAR there is no need to reconstruct a new NHAR file. It allows to add additional files to the existing NHAR file. The inserting process involves three steps: Archiving the new files, merging index files and moving the new file part [7].

Table I. comparison Analysis [10]

Parameter used	Reduction of data at Namenode in HDFS using Harballing Technique [8]	An improved small file processing method for HDFS[9]	Efficient Way for Handling Small Files using Extended HDFS[10]	Improving Performance of small-file Accessing in Hadoop[11]
Method Archive-	Archive-Based	Index-Based	Index-Based	Archive-Based
Positioning	Namenode	Datanode	Namenode	Namenode
Memory Usage	Very Low	Low	Moderate	Slightly High
Reading Efficiency / Addressing Time	Moderate	Moderate	High	High
Performance	Moderate	Moderate	High	High
Scalability	High	High	High	High
Overhead	Slight	High	Low	Low

III. PROPOSED APPROACH

While handling small files in Hadoop, the first problem encountered is the overhead of the NameNode. This involves keeping separate metadata for all the small sized files, which in turn uses more mappers.

The proposed approach resolves this by the process called "merging". In this process the small files will be merged into one file upto the size of the data block [8]. By doing so Namenode will not be required to treat all the small files individually rather it will treat whole block as one. The larger files will be allowed to pass through directly to the NameNode.

Along with every block we will put index table which is hash indexing based. Which will remove the drawback of using two index mechanism as already used in advanced archived Hadoop based solution.

A caching mechanism will be used along with the file, for fast retrieval/processing of the information [9]. We will be moving metadata and corresponding hash tables into the cache so that whenever the user tries to access the information, its tried to be sorted from cache first if not present in the cache then only it will be moved to NameNode level . This process we call "Fetching". This will remove the performance issue of the NameNode while Dealing with the small sized files. As the access time of data from Cache is very low as compared to access time from the memory.

1. Algorithm for Merging
   (1) Read file name and file size.
   (2) If file size is greater than threshold, ignore to add in list (Default Threshold value=80% of HDFS block size, default block size is 64MB)
   (3) Consider the file name and apply hashing.
   (4) Maintain a hash value for all the files stored in a data block.
   (5) Append the hash index with the corresponding block.
   (6) Pass the block (entire list) to the reducer.
   (7) After this flush the list and take the new input.

2. Algorithm for fetching
   (1) Take the file name to be accessed and determine hash.
   (2) Check the cache for the corresponding block.
   (3) If not present in cache then access the file information from NameNode.
   (4) Go to the corresponding block based on block value and the hash value.
   (5) Access the element
   (6) Copy the metadata of corresponding block to the cache along with the hash index file.

IV. EXPERIMENTAL SET UP

The experiment was done on Amazon elastic compute cloud (EC2). EC2 is a web service that provides elastic compute capacity in cloud for making web scale computing easier for developers. For our experiment we have two types of instances . One is "m1.medium" and another is "m1.small". M1.medium is based on Intel® Xenon® CPU E5-2650 with 2.00GHZ processor with 3.7GB ram and operating system is 64 bit Ubuntu server 12.04.3LTS. M1.small has Intel® with 1.7GB RAM ,500GB hard disk and Ubuntu server 12.04.3LTS. Hadoop 2.5.2 and 64 bit java JDK is installed in both the instances. "m1.medium" was used as the NameNode and "m1.small" as the data node by us.

V. METHODOLOGY

In order to handle the size problem at NameNode the proposed solution uses the merging process. In merging process the small files are merged into a single block until the block size becomes less to handle anymore file. NameNode maintains only the metadata of the merged files rather than the individual files and hence result in saving a lot of memory of the memory at NameNode [10].

In merging process we are not concerned with the files whose file size is greater than the block size. We will let them go to the mapper directly. When file of size less than our threshold size arrives its hash is calculated using its Name and the file is added to block. The files are added continuously until the block gets filled. In order to fill the vacant space which can't be filled by any small sized file is filled using boundary filling fragment.

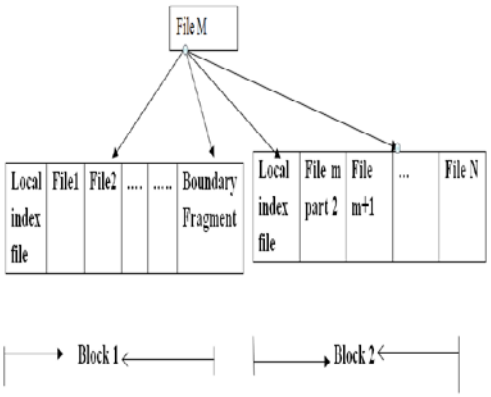

Figure 5. The schematic representation of merging process

A. Prefetching and caching Strategy

Prefetching and caching technologies have been used widely for storage optimization. Prefetching although adds a little bit of complexity but it does wonder to the response time and data fetching. Prefetching uses principle of locality of reference and using cache to keep the fetched data. In this work we have assumed the files are in logical order relation in a merged file [11].

In the proposed method, a three-level prefetching and caching strategy is used, and is composed of index prefetching, metadata caching, and prefetching of correlated files.

(1) Metadata caching: When a file is requested by the client, mapping needs to be done to a merged file to get the metadata of the merged file from NameNode. If the client has cached the metadata of the merged file already, then the metadata can be directly obtained from cache i.e. our first location of search is cache wherein we will start the search, and thus minimize the communication overhead with NameNode, then original files of the merged file are then requested and data is read. However if it's not present in the cache we need to fetch its metadata from the NameNode and this can be done when the client passes the name of the small file to be accessed and the merged file to the name node. Then NameNode will forward the metadata of the file which includes the block number, DataNode on which the fragment is actually present.

(2) Index prefetching: The metadata of a merged file, helps client in identification of block and DataNode which should be connected with to get the user requested file. The actual location can be determined after calculating the hash of the file name and compared with the metadata table which will guide to the local index file. If the above said index file has been already retrieved from DataNode, then we can calculate the value and fetch the file, else what we are going to do is we will fetch whole index file and cache that index file i.e. preftech rest of the index which are most likely to be requested next by the client. Thus improving the access performance to a great extent.

(3) Correlated file prefetching: As we have already said files are assumed to be logically ordered in a merged file and intuitional correlations. Once the file requested by the system end user is returned to him, we will prefetch the related files (files in same block) which shall be auto trigger based. The logical order among the files will determine the file prefetch preference, related files in the same block will be prefetched into the cache for further likely access. This will reduce i/o access time and thus improve access efficiency

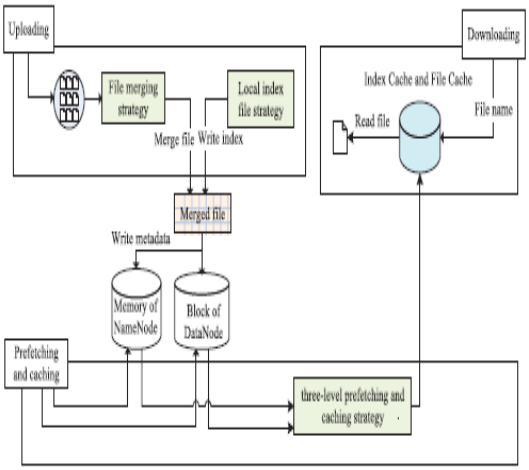

Figure 6. The schematic diagram of handling small sized files in hadoop [11]

VI. RESULTS AND OBSERVATIONS

Suppose that there are N number of small files, which we have merged into M big files, K_1, K_2, Ky, and K_M, with lengths denoted as LK_1, LK_2,……, LK_M.

(1)Storage Efficiency: The consumed memory at NameNode can be calculated as under :

$M_{NN} = 250*N+(368+\beta)*B$ (1)

Where B= number of blocks.

N= number of merged files

250 bytes = Size metadata file takes.

386 bytes= default three replica value for each block.

As can be seen from the above formula, the memory consumption of the NameNode is independent of the number of number of original files and depends only on the number of merged files N, also the number of blocks is very less. Thus saving the memory overhead of the NameNode.

(2)Access efficiency: In this proposed work, in order to read a file the hash of the requested file is checked with the hash table at NameNode which gives the metadata to access the file from the disk. When the access request comes the DataNode will look into the cache for the information required, saving access time, And making NameNode performance better as there will be low load on the NameNode. If the data is not present in the cache we will be accessing the NameNode for required information that's the data file

we are looking for. Once we get the metadata and address of that location we will be moving whole block into the cache and thus making use of the locality of Reference principle, Which makes the access performance better and hence help in removing the performance bottleneck of the small size file handling problem of Hadoop. The processing time of small sized files has reduced up to 85.67% as compared to the existing HAR based technique

Table II- Memory usage at NameNode

Approach	Memory Usage(MB)
Existing HAR Approach	598
Proposed Approach	310

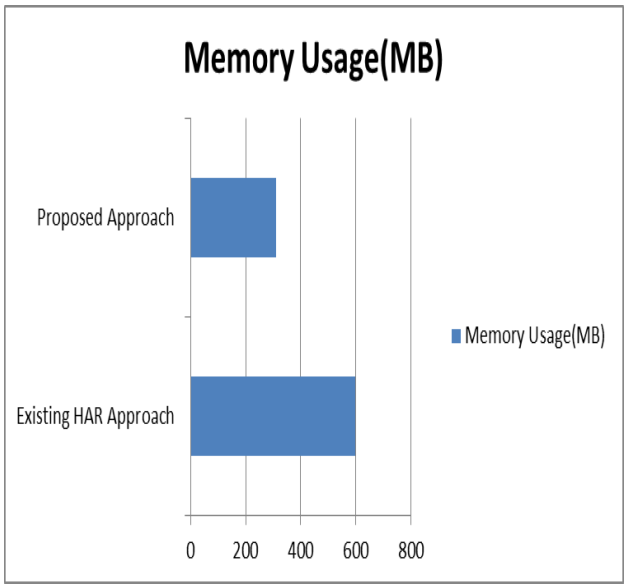

Figure 7 : Memory Usage at Name Node comparison

TableIII- Map, Reduce and Total CPU Time

Approach	Map time (sec)	Reduce time (sec)	Total CPU time (sec)
Existing HAR Approach	61	10	65
Proposed Approach	38	9	15

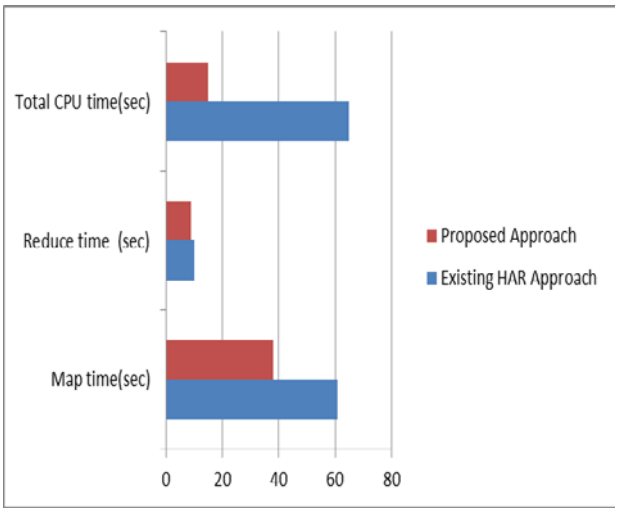

Figure 8: Memory map reduce and total CPU time

VII.CONCLUSION AND FUTURE WORK

Hadoop being wide area of research and one of the topics of research is handling of small files in HDFS, so this research focuses on a MapReduce approach to handle small files, considering mainly two parameters. Firstly, execution time to run file on Hadoop Cluster and secondly the memory utilization by NameNode. By considering these parameters, proposed algorithm improves the result compared to existing approaches. It can handle both sequence file and text file efficiently and also avoid files whose size is greater than threshold.

This work presents a solution to handle small files problem in Hadoop based on file merging and caching techniques. The possibility of making the searching faster with the help of prefetching and hashed index file has been exploited in this work. The experimental evaluation has demonstrated that the proposed technique has reduced the NameNode memory consumption and has improved the efficacy of storage and access of small files in HDFS

In future, the reason and theoretical formula to determine edge points between the hefty and small files will be further analyzed. We will scrutinize the relationship among storage, access efficiencies and the size of a merged file, with the objective to obtain an optimized file merging strategy and to get the peak size of a merged file.

VIII. REFERENCES

[1] "Welcome to Apache™ Hadoop®!", Hadoop.apache.org, 2017. [Online]. Available: http://hadoop.apache.org/. [Accessed: 10- March- 2017].

[2] Shvachko, Konstantin, et al. "The hadoop distributed file system." Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on. IEEE, 2010.

[3] White, T. 2010. Hadoop: The Definitive Guide. 2nd ed. O'Reilly Media, Sebastopol, CA. 41-45

[4] White, Tom. "The small files problem." Cloudera Blog, blog. cloudera. com/blog/2009/02/the-small-filesproblem (2009).

[5] "Hadoop Archives Guide", Hadoop.apache.org, 2017. [Online].Available:https://hadoop.apache.org/docs/r1.2.1/hadoop_archives.html. [Accessed: 16- May- 2017].

[6] Chen, Jilan, et al. "An improved small file processing method for hdfs." International Journal of Digital Content Technology and its Applications 6.20 (2012): 296.

[7] Vorapongkitipun, Chatuporn, and Natawut Nupairoj. "Improving performance of small-file accessing in Hadoop." Computer Science and Software Engineering (JCSSE), 2014 11th International Joint Conference on. IEEE, 2014.

[8] Korat, Vaibhav Gopal, and Kumar Swamy Pamu. "Reduction of Data at Namenode in HDFS using harballing Technique." International Journal of Advanced Research in Computer Engineering & Technology 1.4 (2012): 2278-1323.

[9] Chen, Jilan, et al. "An improved small file processing method for hdfs." International Journal of Digital Content Technology and its Applications 6.20 (2012): 296.

[10] Jayakar, Kashmira P., and Y. B. Gurav. "Efficient Way for Handling Small Files using Extended HDFS." (2014).

[11] Vorapongkitipun, Chatuporn, and Natawut Nupairoj. "Improving performance of small-file accessing in Hadoop." Computer Science and Software Engineering (JCSSE), 2014 11th International Joint Conference on. IEEE, 2014.