International Journal of Electrical and Computer Engineering (IJECE)
Vol. 13, No. 2, April 2023, pp. 1757~1772
ISSN: 2088-8708, DOI: 10.11591/ijece.v13i2.pp1757-1772

Tiny datablock in saving Hadoop distributed file system wasted memory

Mohammad Bahjat Al-Masadeh[1], Mohd Sanusi Azmi[2], Sharifah Sakinah Syed Ahmad[2]
[1]Deanship of information technology, Umm Al-Qura University, Al-Aziziyah, Mecca, Kingdom of Saudi Arabia
[2]Fakulti Teknologi Maklumat dan Komunikasi, Universiti Teknikal Malaysia Melaka, Durian Tunggal, Malaysia

Article Info

Article history:

Received Oct 17, 2021
Revised Sep 30, 2022
Accepted Oct 21, 2022

Keywords:

Big data
Datablock
Datanode
Hadoop
Hadoop distributed file system
Wasted memory

ABSTRACT

Hadoop distributed file system (HDFS) is the file system whereby Hadoop is use it to store all the upcoming data inside it. Since it been declared, HDFS is consuming a huge memory amount in order to serve a normal dataset. Nonetheless, the current file saving mechanism in HDFS save only one file in one datablock. Thus, a file with just 5 Mb in size will take up the whole datablock capacity causing the rest of the memory unavailable for other upcoming files, and this is considered a huge waste of memory in serving a normal size dataset. This paper proposed a method called tiny datablock-HDFS (TD-HDFS) to increase the usability of HDFS memory and increase the file hosting capabilities by reducing the datablock size to the minimum capacity, and then merging all the related datablocks into one master datablock. This master datablock consists of tiny virtual datablocks that contain the related small files together; will exploit the full memory of the master datablock. The result of this study is a running HDFS with a minimum amount of wasted memory with the same read/write data performance. The results were examined through a comparison between the standard HDFS file hosting and the proposed solution of this study.

Corresponding Author:

Mohammad Bahjat Al-Masadeh
Deanship of information technology, Umm Al-Qura University
Al-Aziziyah, 24243, Mecca, KSA
Email: mbmasadeh@uqu.edu.sa

1. INTRODUCTION

In Bigdata world, many companies declared their own analysis systems as a “platform as a service” product such as HDInsight’s from Microsoft and Amazon S3 from Amazon [1], [2]. However, whether the service is based on cloud or built on premise, most of these data platforms were developed and customized based on an open sources system namely Hadoop. Hadoop is a distributed open sources system built by Apache software and it is used to host and analyze all kinds and models of bigdata. Hadoop consists of one node called namenode or master node, with thousands of nodes connected together called datanodes [3]. Hadoop is a set of sub systems called ecosystems whereby some of these systems are used for keeping and tracing purposes, while some others are for data analysis and extract, transform, load/extract, load, transform (ETL/ELT) data injection, and only one of them, which is Hadoop distributed file system (HDFS), is for data hosting [4]. HDFS is the main ecosystem that Hadoop uses to host files in a distributed manner. HDFS is used to split every new datanode into a bunch of storing units called datablocks [5]. Thus, the datablock is the smallest storing cell in HDFS that can be used to store Hadoop files in <key, value> manner [6], [7]. The default size of any datablock in all datanodes is 64 MB, but in several custom usages, it can be upgraded to be 128 Mb or even 256 MB [8]. Figure 1 shown the basic assignment in namenode to datanodes in HDFS that B1 block in all datanodes are belonging to one file. The client could be anything, a human, a sensor,

machines, and so forth. The file storing assignment begins by sending a dataset from client to the namenode. In this regard, the namenode jobs are: i) check if there is a free memory for the new dataset, ii) provide the free datablock IDs to the new dataset, and iii) splitting the large files =>64 MB to assigning to multi datablocks. After all operations above, the dataset will continue to the assigned datablock to be stored in there and send the required metadata files to the namenode to update the namenode on the new dataset files and location inside HDFS [9], [10].

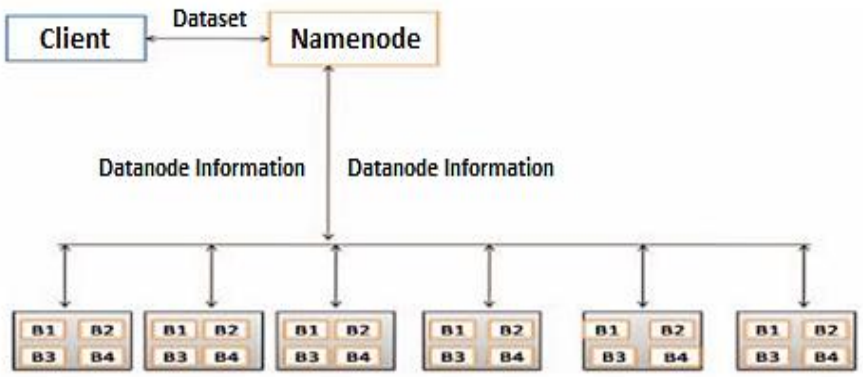


Figure 1. How HDFS stores a new dataset [11]

Each time Hadoop user injects HDFS with a bunch of new files, HDSF will go through the same scenario above to host these new data. One of Hadoop's features is the "data high availability" that's every single file is available for use even if it is corrupted or being deleted [12]. Hadoop adopts a technique called "replication manner" whereby every file must go into three copies so that if one copy is somehow un-available, the other copies will be the replacement and the processing and insight job will resume. Figure 2 demonstrates the replication manner [13], [14].

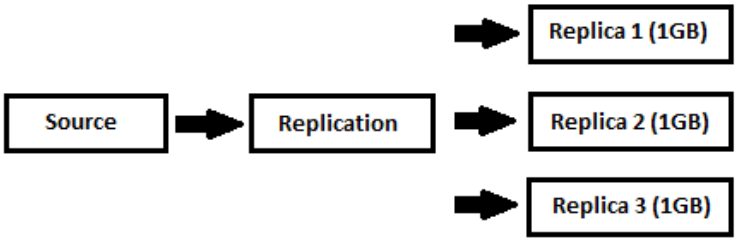


Figure 2. HDSF replication manner [15]

In Figure 2, the 1 Gb file requires approximately 16 Datablocks to store it. Then, to apply the HDFS high availability principle, all of these datablocks must be replicated 3 times in total. Lastly, all of these new datablocks and their replications must send some metadata files to the namenode to complete the files storing steps [16]. In HDFS, each datablock can host only one file, irrespective of whether or not the file fits the datablock size [17]. This technique works will with bigdata in big files that's every file in the dataset is greater than the datablock size [18]. However, things are different with big data in small files whereby every file is smaller than the default datablock size [19]. The datablock hosts only one file because the namenode can access the desired file via datablock ID only. In other words, the namenode cannot access anything inside the datablock directly. Thus, a small file will occupy the required memory and cause the rest of the datablock memory inaccessible [20]. The (1) shows the amount of wasted memory for each datablock. Wm is the wasted memory, dbs is the datablock size and fs is the file size. Figure 3 shows the standard file hosting in HDFS whereby the injected dataset to HDFS datablocks is big data in small files.

$$wm = dbs - fs \quad (1)$$

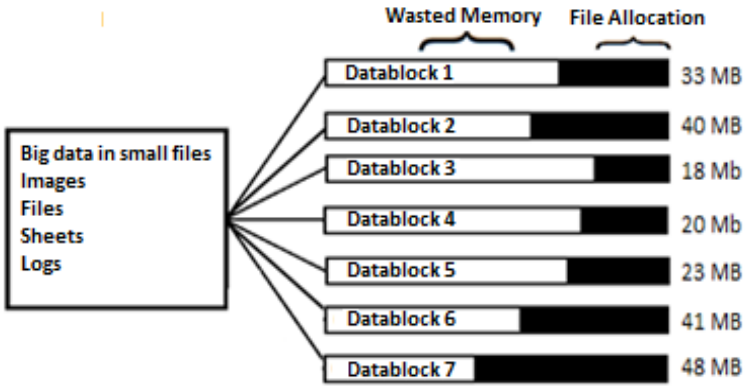


Figure 3. The traditional datablocks hosting method [17]

In order to determine the depth of the problem, (1) will be re-applied on the results in Figure 3, as (2):

$$\sum wm = dbs - fs \quad (2)$$

Thus, the summation of the wasted memory is 225 MB out of the total amount of datablocks size which is 448 MB. Lastly, the 225 MB wasted memory is one of three copies that HDFS must create in order of data high availability in case of data reading failure because of the unavailability of the desired data. Thus, a 225 MB will be turned into a 675 MB of total wasted memory [15]. These numbers shown a huge wasted memory amount just to host a small dataset [11]. The final formula for the datablock wasted memory is shown as in Table 1.

Table 1. Formula flow of the wasted memory problem

Step	Stage	Operation	Equation	Description
1	Data feeding	Feeding	*cd = cd + nd*	cd: current data, nd: new data
2	Data allocation	Appending	*db = cd.append()*	db: datablock, append (): add the file to the datablock
3	Wasted memory calculation on single file	Finding the wasted memory of a single datablock	*wm = dbs - fs*	wm: wasted memory, dbs is a collection of datablocks inside HDFS, fs: file size.
4	Total files in the datablocks	Total files size	*∑ fs.append()*	In this stage, the fs will go into cd which will be the total files size into a datablocks.
5	Total wasted memory on all of used datablocks	Total datablock size	*∑ (wm.release())-db*	wm is the wasted memory that is released from the used one
6	Results of the problem statement	Results	*wm = ∑dbs - ∑fs*	It is the final total wasted memory in Hadoop cluster.

2. RELATED STUDIES

There are some popular solutions to solve the problem, the first one is Hadoop archive (HAR). HAR compresses the small files into one HAR inside one datablock and use two index ID's are used to access the desired file inside HAR [13]. The second one is dynamic partitioning which is used to add a new node to the cluster called aggregator node to determine that if the upcoming new file if greater or less than the datablock size. The next section will discuss the most popular technique in solving the problem of this study.

2.1. HAR file

Hadoop archive (HAR) is the earliest attempt to solve the problem above which packs a number of small files into one archive file before pushing it inside HDFS. However, the small files inside this new archive file cannot be accessed directly because two index IDs have to be searched before reaching the desired file. The access is done in the main memory. Figure 4 presents HAR file access diagram [21].

Basically, HAR was designed to solve big data in small files issue. Thus, the technique that's adopted to solve this issue is also used to reduce the HDFS wasted memory [16]. So, instead of placing every small file in a standalone datablock, HAR file will do the job by archiving all of them in one file. However, creating a HAR file involves running a MapReduce job into a targeted directory "*/dir*" that consist all the desired small files to be archived.

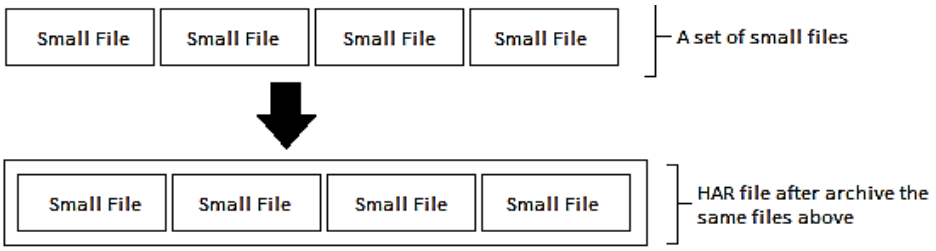


Figure 4. Hadoop archive HAR [21]

In accessing a desired file in any HDFS archive, user needs to go through two index files: the first one is master index, and the second one is index table. Figure 5 is shown the accessing method for each single HAR archive. This access mechanism slows down the HDFS data retrieval as the use of HAR technique is only appropriate for the rarely accessed data (cold data) such as logs files [22]. Another shortage for HAR is that it cannot append more files to the archive. This means that if a new file is already injected in HDFS, it will be hosted in a new datablock even if there is a HAR file that is not full yet making it unable to host more. Hence, any upcoming file cannot be appended to this HAR and it will be directed to a new one. The extra unused memory in the old HAR and the new one is considered as a wasted memory [23].

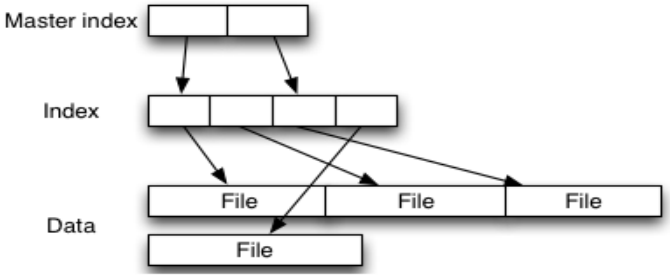


Figure 5. Accessing a file inside HAR [22]

2.2. Dynamic partitioning

Hadoop cluster principle operates on one namenode and thousands of datanodes. However, dynamic partitioning approach is used to alter is principle by appending a decision maker node namely aggregator node which is located between the namenode and the rest of the cluster [22]. Aggregator node is the decision maker of the cluster that's when a new file being injected to the namenode, namenode will split it and pass the results to the aggregator node to check if the size is 64 => dataset > 64. If the dataset's size is greater or equal than the datablock size, it will be directly assigned to the targeted datablock. On the other hand, if the dataset is less than 64 Mb [24], aggregator node will transfer it to the proposed datablock using a dynamic partitioning approach. Hence, partitioning approach will fit the size of the datablock with the upcoming dataset. The result of this approach will overcome the HDFS fragmentation that's that's causing memory wastage as demonstrated in Figure 6 [11], [25].

Dynamic partitioning is used to alter the attitude for the decision making of the namenode. The Namenode is the master of Hadoop that uses yet another resource negotiation (YARN) to ride the rest of the cluster. Thus, the decision was partially moved to be with the aggregator node which does not match the aim of this study, that is, to solve wasted memory problem without any major customization on Hadoop default structure. Thus, the reading process of dynamic partitioning will be slow because each analysis script and the attained result must go through the aggregator node, and this will be reflected negatively on the read/write performance, and the connection latency will increase.

2.3. Sequence file

This study is not targeting HDFS wasted memory. Sequence file was placed as a solution to reduce the negative impact of big data in small files. Here, a file that consists of a set of small files is created inside one file called sequence file. However, this new file will be reported to the namenode as a single file with one metadata file only, and this is a good solution for the problem of big data in small files whereby one metadata

file will be sent to the namenode directory instead of hundreds of metadata files whereby each will present one datablock to the namenode [1]. Thus, instead of hosting every small file inside a standalone datablock, Sequence file will collect all of these small files that act like a series of files with a unique ID for each one of them under the following rules: i) the sequence file cannot be greater than the datablock, and if so, the system will generate a new one in a new datablock; and ii) the reading mechanism adobes the binary search without indexing, which means that each time the search process accesses a small file in the sequence file, the binary search method must search for the desired file from the beginning of the sequence file till the end of it [13], [26].

Sequence file is reducing the HDFS wasted memory perfectly. But the problem of this solution is as following: i) the access method complexity is very high (n^2), and the binary search must include all files in the sequence file to find the desired file; and ii) since the datablock is not required to be changed (expanding or shrinking). Sequence file will follow the default datablock size is 64 Mb. However, in case of the sequence file is smaller than the datablock size, the rest of free memory of the datablock is unreachable and considering a wasted memory [1].

Based on the shortages of all the studied solutions (HAR, dynamic partitioning and sequence file), this study proposed a solution that will prevent the problem of slow file reading, which is the shortage of HAR and sequence file, and the problem of cluster datablock multiple capacity that could not fit all of the small files, which is the shortage of the dynamic partitioning. The next section will provide the details of the new solution proposed in this study.

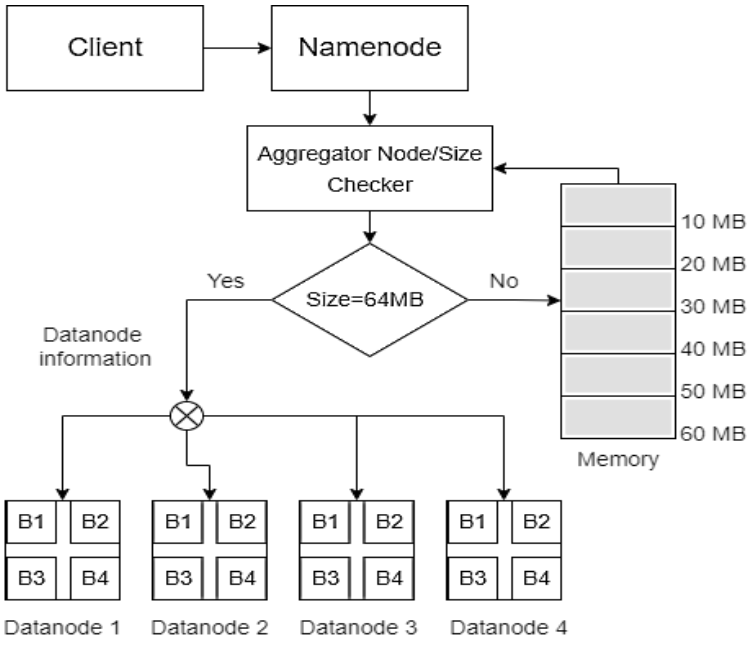


Figure 6. Dynamic partitioning [11]

3. THE PROPOSED METHOD: TINY DATABLOCK (TD)

The adopted dataset is big data in small files is a huge amount of data (more than 10 Tb) consisting of small files (each file is less than the default datablock size) [22]. This kind of data will used as a quantitative methodology to find the gap between the standard HDFS file hosting, the available methods in file hosting boost, with the proposed method as shown in detail in the next part. A solution called tiny datablock was proposed. This solution is based on the principle which requires that every datablock drops its size to be 5 Mb only by default. However, the default datablock size's changing manner will be on the top of HDFS.

3.1. The algorithm

This part is shown the proposed method to come over the HDFS wasted memory and to save them for hosting more files in the Hadoop cluster. The proposed method will adopt the shrinking mechanism to host each small file in an independent manner, then merge them all in one datablock. The proposed solution uses two methods to reduce the problem highlighted in this study:

3.1.1. Memory shrinking

This method will shrink all of the available datablocks to 5 MB. The aim of shrinking the datablock size down to be 5 MB is to match the most popular size required in hosting big data in small files in each datablock in HDFS. Figure 7 describes the shrinking action.

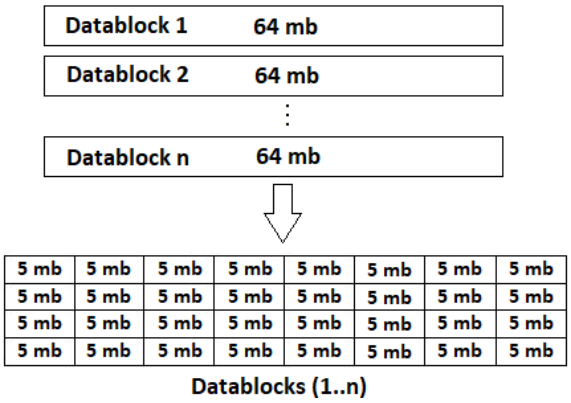


Figure 7. Memory shrinking

3.1.2. Datablock merging

This method is the next step after the execution of the memory shrinking method. Once these tiny datablocks are allocated to the new small files from the namenode, they will have an ability to merge with each other into one or more datablock(s) with only one metadata file (reference file) which is also head towards the namenode. This means that if three files with a total size of 15 Mb (small file) are injected into HDFS, then, three datablocks will be required to host all of these three files. These three datablocks can be merged together to become one datablock with the size of 15 MB and generates only one metadata file heading to the namenode, as shown in Figure 8.

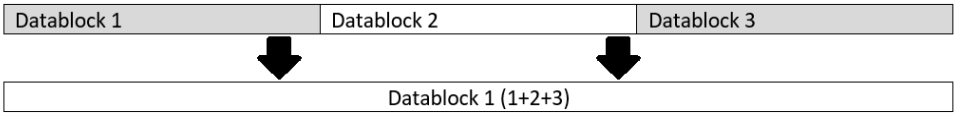


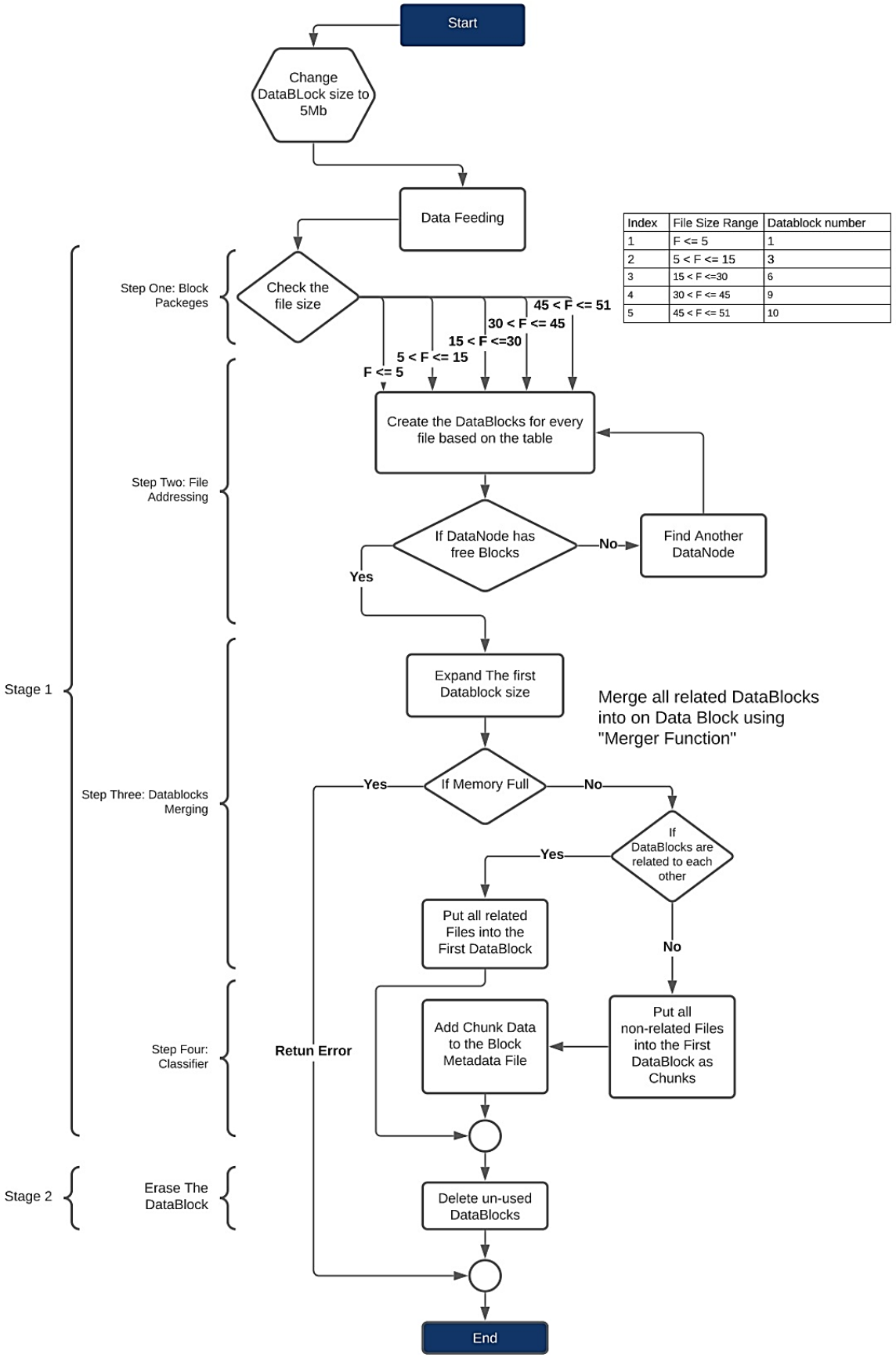
Figure 8. TD migration

3.2. Datablock reading

To read from a pre-merged datablock, this study will adopt a byte-to-byte algorithm that counts every data chunk in the merged datablock as a stand-alone file. Byte-to-byte formula is a counter that captures the end point of each chunk inside the merged file based on a pre-given address supported by the namenode. The formula of byte-to-byte will be explained in section 3.2.3. withen data reading stage. Meanwhile, the application of TD migration method in any Hadoop cluster requires that TD goes through several stages and steps to complete the migration correctly. These stages begin with reading each upcoming small file size, followed by placing the file into datablocks, and finally, migrating the datablocks into a single datablock. The second stage is related to how to read files from the new datablock. Figure 9 shows the tiny datablock-HDFS (TD-HDFS) file writing.

3.2.1. Stage one (reading each new file size)

HDFS file injection and the addressing method begin by the attainment of a positive request to the namenode about the ability to host an upcoming file(s) or dataset. In this regard, the returned response will carry out a new address somewhere in the datanodes to host the new file(s). Then, Hadoop will continue to the datablocks based on the given address (datanode then datablock inside). Finally, Hadoop will host the file. TD-HDFS will alter this technique by appending a method called size calculator that will work as follows:



Index	File Size Range	Datablock number
1	F <= 5	1
2	5 < F <= 15	3
3	15 < F <=30	6
4	30 < F <= 45	9
5	45 < F <= 51	10

Figure 9. TD-HDFS writing stage

Step 1 (blocks packages)

Namenode will be supported by a database consisting of capacity packages records. Here, the namenode will read the new files information and create a metadata file. This metadata file consists of information about the file size and the required datablocks to store it. Table 2 accordingly shows the file size range and the volume database already installed in the namenode.

Table 2. Datablock classifying packages

Index	File Size Range (Mb)	Required Datablocks
1	F <= 5	1
2	5 < F <= 15	3
3	15 < F <=30	6
4	30 < F <= 45	9
5	45 < F <= 51	10

Step 2 (file addressing)

The metadata file is now ready with all the data to be stored in the targeted datablock(s). These data consist of the datablock ID, datablock reference and the new one is the datablocks number that is required to store the new file. As a scenario: there is a new file (20 MB) that must be injected into the HDFS using one of Hadoop's ecosystems which are sqoop and/or flume. This new file is a structured database. Thus, the metadata file generated by the namenode will assign a four datablocks for this new 20 MB file. Table 2 shows the differences between the old file addressing method and the TD one to host a 20 MB file size. Based on Table 3, in the old file addressing method, HDFS will book a 64 MB datablock to host the new 20 MB file. In TD-HDFS file addressing method, HDFS will book only four tiny datablocks, and each of them is only 5MB in size.

Table 3. Old file addressing vs TD-HDFS file addressing

Old File Addressing			TD File Addressing		
New File Vol	Datablock ID	Size	New File Vol	Datablock (s) ID	Size
20 Mb	aa128885c7321s94	64 Mb	20 Mb	38821hd4E388dc21	5 Mb
				329981005t23e122	5 Mb
				218t544qw2113500	5 Mb
				6643889222221094d	5 Mb

Step 3 (datablocks merging)

Back to step two, the four datablocks must be located on the same datanode. In this regard, if the selected datanode has only 3 free tiny datablocks, the namenode must find another datanode with more side-to-side free datablocks. The datablock merging process will send the selected file to be stored in the pre-selected datablocks IDs as shown in Table 2. HDFS will split the new file of 20 to 4 MB related files that every file fits a tiny datablock. These datablocks are on the same datanode and each one of them hosts one file. TD-HDFS will alter HDFS behavior with an equation called "merger." However, "merger" equation will use the metadata file generated from the namenode back in step one to merge the four datablocks above into one datablock with only one reference ID similar to the earliest datablock ID. The goal of "merger" equation is to minimize the metadata files that are returning to the namenode. Figure 10 shows the mechanism of merger equation.

The merger mechanism equation will merge all the pre-booked datablock1 to datablock4 into one datablock referred as datablock1. The merging procedure will cover all the following: i) merger will start reading all the datablocks that hold a part of the same file. In Figure 10 there will be four datablocks; ii) merger already knows that there are four datablocks with a total size of 20 MB; iii) merger will expand the first datablock (datablock1) size into 20 MB; iv) if the datanode is full, merger will stop with an error message asking for free datablocks; v) if all of datablock1, datablock2, datablock3 and datablock4 are related to each other (each one is a part of the original file), the next step (step four) is not applicable; vi) datablock1 is ready to be read without a need for the reading stage (later on this paper); vii) if each of datablock1, datablock2, datablock3 and datablock4 has a non-related file, step four is required; viii) all of datablock2, datablock3 and datablock4 will be migrated to the new datablock1 as a new chunk; ix) the new datablock will be full of non-related chunks that each chunk presents only one file; and x) by now, all of datablock2, datablock3 and datablock4 are useless and need to be assigned as an empty block by the namenode.

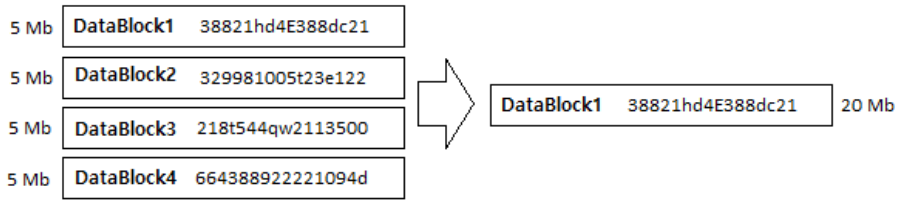


Figure 10. Merger equation mechanism

Step 4 (classifier)

Before setting up the final phase of datablock1, the chunks inside it are separated, and at this point, there is no relation between these chucks but they reside as one piece that is not readable yet. Thus, the equation (classifier) will add an extra metadata to the original datablock metadata to tell the next equation (stage 3 in this paper) how to read these non-related and non-addressed chunks. This extra metadata is presented in Table 4. Byte capacity is the capacity of each chunk inside the datablock, and this information will be used by the namenode in stage three to read the chunks.

Table 4. Chunks inside datablock

Datablock 1	Byte capacity
Chunk 1	5,000,000
Chunk 2	5,000,000
Chunk 3	4,000,000
Chunk 4	5,000,000

3.2.2. Stage two (erase the datablock)

Since Merger does not have a master permission to delete the useless datablocks, there is another equation that will assign this job to the namenode-YARN ecosystem. Thus, the new equation here is called a "deleter." This equation will deceive the namenode by sending it the empty datablocks ID. Thus, the namenode will automatically update the HDFS by the following new situation: i) there is no change on datablock1 but the size is expanded from 5 to 20 MB; ii) all of datablock2, datablock3 and datablock4 are assigned as free datablocks. Thus, the namenode-YARN will assign all of them as empty datablocks that are ready to use for another upcoming data; and iii) the final results are displayed in Figure 11.

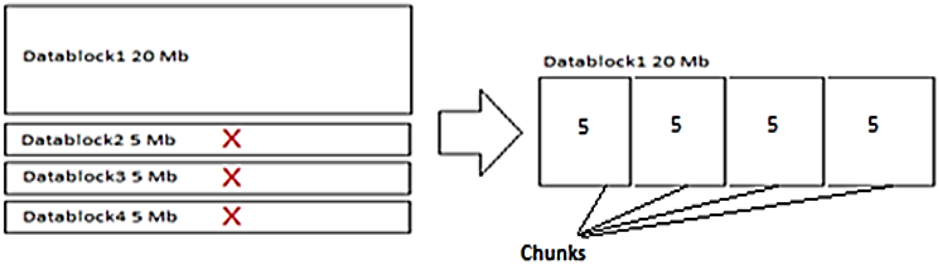


Figure 11. Chunks inside each datablock as a final step

3.2.3. Stage three (data reading)

There is no way to read each chunk inside a single datablock because the chunks are already separated but acting like one file to the namenode. HDFS files hosting is only 80% of the total datablock size, while the rest of it is used to host the metadata on the datablock, providing the namenode more information about it. To make the chunks inside each datablock readable and discernible to the namenode, the metadata in each datablock has to be altered as appeared in Table 3. before providing more information about the desired chunk in the datablock. TD-HDFS added an equation called "founder" and this equation is controlled by the namenode-YARN to read out the metadata about each datablock that is already updated by the "classifier" equation. However, founder will retrieve everything in the datablock as a metadata to the namenode memory in order to read them correctly. Moreover, "classifier" equation will tell the namenode how to separate the

chunks inside each datablock to fetch the desired chunk data. Founder will use the byte-to-byte formula to separate the chunks from each other based on the given metadata. Byte by byte formula will read the whole data as a counting bytes of array. The array is the data inside the desired datablock. Thus, byte by byte will obtain the information about each chunk volume, and in which byte it will start and end. Table 5. shows an example about the chunks in a single datablock, with byte information about each chunk. Byte information is retrieved from datablock metadata.

Table 5. An example of each chunk inside each datablock

Datablock 1	Byte Capacity (Byte)	File Info
Chunk 1	5000000	Image
Chunk 2	5000000	Xls file
Chunk 3	4000000	Pdf file
Chunk 4	5000000	Image
Total	19000000	-

Founder via byte-to-byte formula will start separating the chunks away from each other, and the beginning of the datablock array is the beginning of the first chunk. The byte counter will count 5,000,000 bytes before it stops and checks out this chunk as a separated file. The second 5,000,000 bytes are counted in similar manner. The byte counting will end when the counting reaches the total datablock byte capacity. The outcome encompasses four separated files that are ready to show up to the end user. Byte-to-byte formula will not run if the datanode holds only one file as mentioned in step three of the first stage. Figure 12 is describing the steps to read the desired file in each datablock.

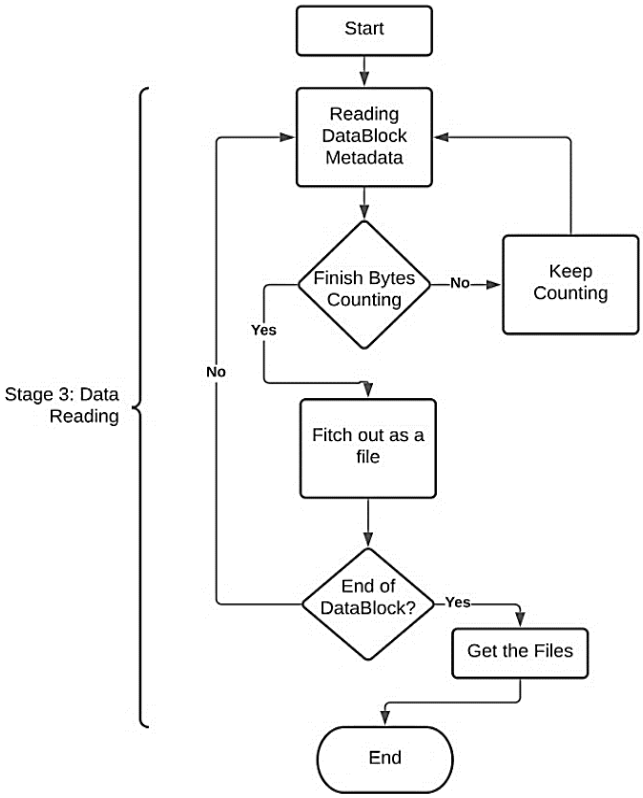


Figure 12. TD-HDFS data reading

4. TESTING AND COMPARING

This paper attempts to fit the current HDFS files with the available in-charge datablocks. However, the TD technique utilizes a comprehensive model to absorb a lot of small files on the same datablocks total capacity without the traditional memory wasting like in the standard HDFS technique. Figure 13 shows a

parallel test of pushing only 1 GB dataset of small files into a Hadoop cluster. However, the first cluster runs a standard HDFS while the second one runs a customized HDFS with TD technique (TD-HDFS).

Both of the clusters are running a native copy of Hadoop 3.1.2 version. Thus, since Hadoop is an open-sources that's written in Java, TD-HDFS cluster will got the updates equation on the default size of each datablock. The new added equations are:

- *MergingEquation()*. Included in Hadoop datanodes.
- *FounderEquation()*. Included in Hadoop namenode and running byte-to-byte formula
- *ClassifierEquation()*. Used by the namenode metadata.

Both of the clusters will running the same hardware. Normally, Hadoop used the commodity hardware to run the analysis jobs. Typically, each machine in the cluster must not less than 16 GB RAM, 500 GB SHD with most recent version of core i5 processor. However, each node of the cluster will run an OS (Ubuntu 20) that's the best practice for the Hadoop performance and reliability. Figure 13 presents a full series of traditional HDFS filing as compared to the TD technique. However, the TD technique shows that there is more added time to the file injection steps to be completed in terms of the extra steps (step one, two, three and four) and two more stages (stage one and stage two) that are already appended to the whole system.

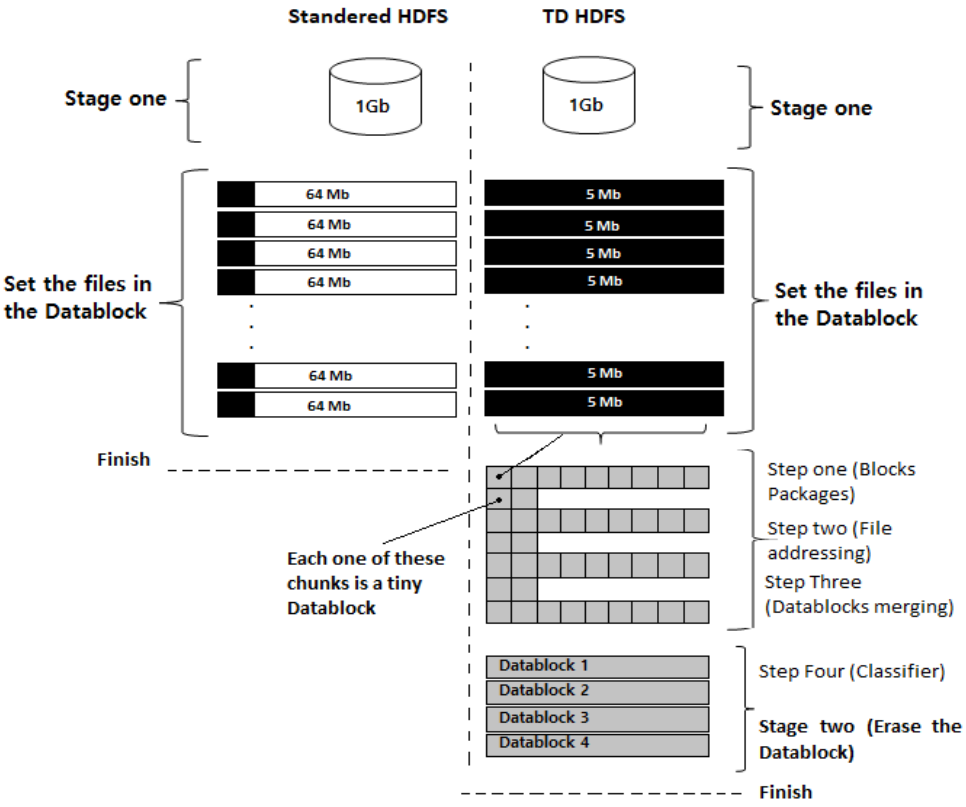


Figure 13. A parallel test between HDFS and TD-HDFS

4.1. Data reading

In data reading (stage 3), there are two scenarios for TD technique. The first one involves a datablock with related chunks, which is easy to read. The second one involves a datablock with non-related chunks. However, reading from no-related chunks inside the datablock requires metadata generated by the previous classifier. These metadata are: i) metadata about every chunk inside the datablock (in the namenode metadata file) and ii) metadata about the byte capacity for every chunk (in the datablock metadata part).

However, the first metadata is resident in the namenode memory, and the metadata make every chunk well known to the namenode in order to easily find it in the desired datablock. Suppose the namenode is willing to read chunk number 3 in datablock 1. Thus, in Hadoop, there is no actual reading mechanism as data processing is happening in the datablock, and the namenode job is to send the scripts to the data place to be processed and a readable insight is obtained. Thus, the namenode will send the "founder" script to the datablock, and the byte-to-byte counting is applied to reach the desired chunk. Figure 14 shows the reading

series. Regarding Table 5, byte to byte counting already knows that the desired chunk 3 length is 4 MB. Thus, byte-to-byte will start from the end of chunk 2 and count 4 MB of bytes before stopping the counting and fetching the results.

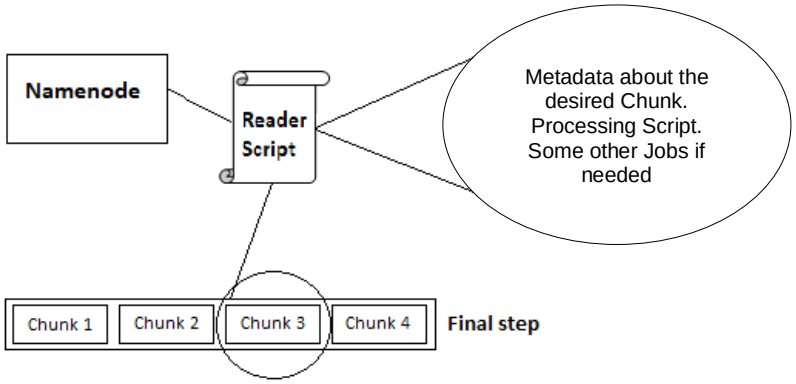


Figure 14. Check reading series

5. RESULTS AND DISCUSSION

The result of this study is standing on a comparison between the previous solutions to short coming HDFS memory wasting and the new proposed solution TD-HDFS that's presenting in Table 6. Figure 15 is transforming the results n Table 6 into a meaningful way in order to clarify the positive memory saving result of TD-HDFS against the other primary studies in this field.

Table 6. A results table for all previous studies and TD-HDSF

	Reading Complexity	Write Complexity	Reduce Wasted Memory
HDFS	Medium	Medium	Low
HAR	High	High	Medium
Dynamic Partitioning	High	Low	Medium
TD-HDFS	Low	High	High

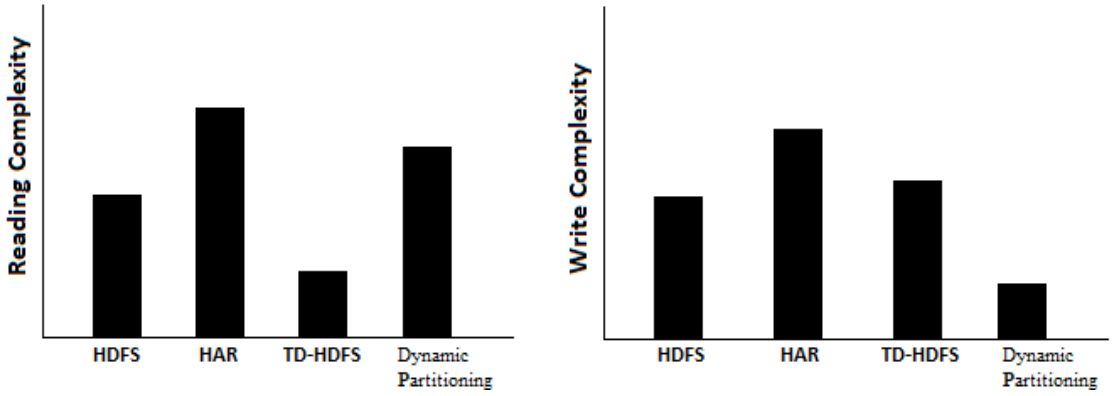


Figure 15. Reading/writing complexity results for all related studies

In Figure 14 TD-HDFS has the best results in reading complexity. The steps were taken during the writing step to generate more metadata on the desired chunk ID, and the byte-by-byte counter makes it easy to use these metadata to make a quick access to that file as a chunk in combined datablock. However, these extra steps increase the writing complexity and take more time to complete a single file write. However, the most important question in data hosting and analytics is "How fast can the system retrieve the desired file?" In this regard, TD-HDFS based on the results above provide the best performance in data retrieving. Figure 16 presents memory saving between HDFS and TD-HDFS. The comparison is based on the injection of

20 files towards Hadoop cluster, followed by the examination of the results of the standard HDFS vs. TD-HDFS. Each file of the testing sample is of 5 MB in size.

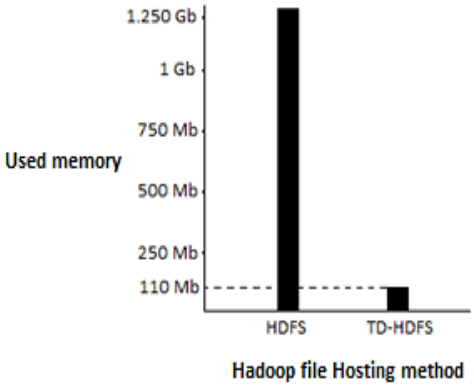


Figure 16. Memory saving in HDFS and TD-HDFS

The test sample requires almost 1.250 Gb of datablocks capacity (20 datablocks) to host these 20 files, but when it comes to TD-HDFS, the required resources will reduce to only 2 datablocks with a total capacity 110 MB. This result can be change up and/or down depending on the following parameters: i) files number, ii) files size for each file, iii) default datablock capacity either for the standard HDFS or for TD-HDFS, and iv) total files capacity (for big and small files).

In Figure 17, HAR will working well in a comparison with the standard HDFS due to HAR will archive all of the small files in one file. But on the other hand, HAR is treating the files as a file, not like TD-HDFS that's treating the file as a datablock. Thus, in HAR case, the archive file could consist of files only and set the rest of the datablock free but un-reachable. TD-HDFS is treating the new files as a datablock and combine the datablocks depending one the new file size. The final results of comparing TD-HDFS with HAR shown that's TD-HDFS doing better than HAR regarding HAR file is just an archive inside a datablock with a chance to not fill it completely depends on the fils capacity.

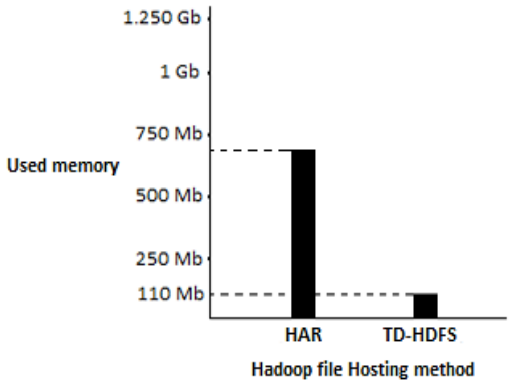


Figure 17. Memory saving in HAR and TD-HDFS

One last study, is to make the comparison results with TD-HDFS and dynamic partitioning. However, dynamic partitioning used to classify the upcoming data files based on its size to be stored in a suitable datablock. The results shown a light different between dynamic partitioning and TD-HDFS in memory saving due to dynamic partitioning does not has the combine step in combing the related datablocks together. Moreover, if dynamic partitioning has been injected with many of small files, are smaller than the smallest datablock in the cluster, the datablocks cannot combine to each other to release more memory. In other words, the small file will take place in the datablock and leave the rest free but unreachable, which

returns to the first square of the problem. Figure 18 shown the final result for dynamic partitioning in a comparison with TD-HDFS.

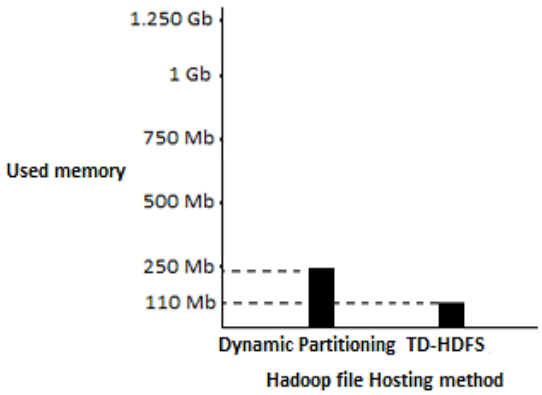


Figure 18. Memory saving in dynamic partitioning and TD-HDFS

This study adopted three parameters to compare between TD-HDFS and other solutions in the same area. These parameters are reading, writing and wasted memory reduction. In data reading, all of other studies are slower than TD-HDFS. Achieving the final chunk in TD-HDFS and reading it are easy because all the required IDs and addresses are supported in the metadata file already stored in the namenode, and thus, it is easy to find the desired datablock and fetch the pointed chunk to read it. Meanwhile, the writing step is complex because it involves setting up the merging steps and generating the metadata file with all IDs and addresses which will consume more time compared to other studies. In reducing HDFS wasted memory which is the core of this study, TD-HDFS with all equations presented in section 6 were used to fit the required memory with the new files. Meanwhile, HAR cannot fit the required memory with the new files volume because HAR could archive 2 files only in one datablock and keep the rest of it free and usable.

5.1. Data integrity

Moving the data from one datablock to another datablock will not cause any data corruption because the moving mechanism is not of cut-paste, but of copy-paste followed by the deletion of the original one. Thus, the copy-paste mechanism is the same mechanism that's used in HDFS file replication manner already adopted in all Hadoop files transactions. This means that the chunks are holding a file with a start offset and end offset, and away from any customization.

5.2. Cold and hot data

TD is a customized HDFS behavior used to add extra steps to the existing system of writing purposes in order to reduce the HDFS memory wasting. Thus, these extra steps are adding more complexity to the write performance as the steps will slow down the daily files feeding performance. On the other hand, TD-HDFS will work much better in data reading due to there is no complexity in data reading. Thus, due to one of Hadoop principles is "write one-read many", the focusing on reading data performance is matter rather than the data writing, this will make TD-HDFS is reliable in data reading with both of hot and cold data.

6. CONCLUSION

There are several popular solutions to HDFS wasted memory problem. The first one is HAR file which is basically directed to solve big data in small files issue, but it is only able to save some wasted memory. Accordingly, the aim of this study is to solve the HDFS wasted memory without placing a major upgrade on the HDFS structure. Hence, both HAR and dynamic partitioning were used to move with the HDFS wasted memory problem to a good solution and save some memory. However, the complexity in reading from HAR makes this solution appropriate only with cold data. The other solution namely dynamic partitioning was used to alter the HDFS communication structure by adding a new node called aggregate node between the namenode and the datanodes, but this could expand the connection latency between the original nodes of HDFS which will slow down the reading process.

Comparatively, TD-HDFS was able to resolve the problem of HDFS wasted memory without majorly altering the HDFS behavior. However, TD-HDFS has a complex writing mechanism during new data injection to HDFS because of the new metadata creation and appendage to the original datablock metadata. Thus, this new metadata will simplify the analysis and reading stage which will make the reading mechanism quick in fetching the required data from each combined datablock.

ACKNOWLEDGEMENTS

The authors would like to thank the Deanship of Scientific Research at Umm Al-Qura University for supporting this work by Grant Code: (22UQU4331450DSR03).

REFERENCES

[1] M. B. Masadeh, M. S. Azmi, and S. S. S. Ahmad, "Available techniques in Hadoop small file issue," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 10, no. 2, pp. 2097–2101, Apr. 2020, doi: 10.11591/ijece.v10i2.pp2097-2101.

[2] K. Salah, K. Elbadawi, and R. Boutaba, "An analytical model for estimating cloud resources of elastic services," *Journal of Network and Systems Management*, vol. 24, no. 2, pp. 285–308, Apr. 2016, doi: 10.1007/s10922-015-9352-x.

[3] Ishwarappa and J. Anuradha, "A brief introduction on big data 5Vs characteristics and Hadoop technology," *Procedia Computer Science*, vol. 48, no. C, pp. 319–324, 2015, doi: 10.1016/j.procs.2015.04.188.

[4] A. A. Yulianto, "Extract transform load (ETL) process in distributed database academic data warehouse," *APTIKOM Journal on Computer Science and Information Technologies*, vol. 4, no. 2, pp. 61–68, Jul. 2019, doi: 10.11591/APTIKOM.J.CSIT.36.

[5] Y. Guan, Z. Ma, and L. Li, "HDFS optimization strategy based on hierarchical storage of hot and cold data," *Procedia CIRP*, vol. 83, pp. 415–418, 2019, doi: 10.1016/j.procir.2019.04.086.

[6] A. Bhaskar and R. Ranjan, "Optimized memory model for Hadoop map reduce framework," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 9, no. 5, pp. 4396–4407, Oct. 2019, doi: 10.11591/ijece.v9i5.pp4396-4407.

[7] P. Haripriya, "An efficient storage and retrieval of DICOM objects using big data technologies," *International Journal of Advanced Research in Computer Science*, vol. 8, no. 3, pp. 271–275, 2017, doi: 10.26483/ijarcs.v8i3.2993.

[8] C. Choi, C. Choi, J. Choi, and P. Kim, "Improved performance optimization for massive small files in cloud computing environment," *Annals of Operations Research*, vol. 265, no. 2, pp. 305–317, Jun. 2018, doi: 10.1007/s10479-016-2376-0.

[9] P. P. Deshpande, "Hadoop distributed FileSystem: Metadata management," *International Research Journal of Engineering and Technology*, vol. 4, no. 10, pp. 1830–1833, 2017.

[10] M. Chi, J. Liu, and J. Yang, "ColdStore: A storage system for archival data," *Wireless Personal Communications*, vol. 111, no. 4, pp. 2325–2351, Apr. 2020, doi: 10.1007/s11277-019-06989-5.

[11] B. Jena, P. K. Kanaujia, S. Rautaray, and M. Pandey, "Improvising block placement policy in Hadoop framework," in *2017 International Conference on Computer Communication and Informatics (ICCCI)*, Jan. 2017, pp. 1–3, doi: 10.1109/ICCCI.2017.8117743.

[12] M. M. Alshammari, A. A. Alwan, A. Nordin, and I. F. Al-Shaikhli, "Disaster recovery in single-cloud and multi-cloud environments: Issues and challenges," in *2017 4th IEEE International Conference on Engineering Technologies and Applied Sciences (ICETAS)*, Nov. 2017, pp. 1–7, doi: 10.1109/ICETAS.2017.8277868.

[13] A. Mohanty, P. Ranjana, and D. V. Subramanian, "Small files consolidation technique in Hadoop cluster," *International journal of simulation: systems, science & technology*, vol. 19, no. 6, pp. 31.1--31.5, Feb. 2019, doi: 10.5013/IJSSST.a.19.06.31.

[14] C. B. V. Vardhan and P. K. Baruah, "Improving the performance of heterogeneous Hadoop cluster," in *2016 Fourth International Conference on Parallel, Distributed and Grid Computing (PDGC)*, 2016, pp. 225–230, doi: 10.1109/PDGC.2016.7913150.

[15] A. Chiniah and A. Mungur, "Dynamic erasure coding policy allocation (DECPA) in Hadoop 3.0," in *2019 6th IEEE International Conference on Cyber Security and Cloud Computing (CSCloud)/ 2019 5th IEEE International Conference on Edge Computing and Scalable Cloud (EdgeCom)*, Jun. 2019, pp. 29–33, doi: 10.1109/CSCloud/EdgeCom.2019.00015.

[16] P. Matri and S. P, "TýrFS: Increasing small files access performance with dynamic metadata replication," in *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2018, pp. 452–461, doi: 10.1109/CCGRID.2018.00072.

[17] İ. Demir and A. Sayar, "Procedia technology Hadoop optimization for massive image processing: case study face detection," *International Journal of Computers Communications & Control*, vol. 9, no. 6, pp. 664–671, 2012.

[18] G. Attigeri, M. Pai. M. M, and R. M. Pai, "Framework to predict NPA/willful defaults in corporate loans: a big data approach," *International Journal of Electrical and Computer Engineering (IJECE)*, vol. 9, no. 5, pp. 3786–3797, Oct. 2019, doi: 10.11591/ijece.v9i5.pp3786-3797.

[19] R. Rathidevi and S. Srinivasan, "Small files problem in Hadoop-a survey," *International Journal of Pure and Applied Mathematics*, vol. 119, no. 15 Special Issue B, pp. 2833–2841, 2018.

[20] L. Tu, S. Liu, Y. Wang, C. Zhang, and P. Li, "An optimized cluster storage method for real-time big data in internet of things," *The Journal of Supercomputing*, vol. 76, no. 7, pp. 5175–5191, Jul. 2020, doi: 10.1007/s11227-019-02773-1.

[21] 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*, May 2017, pp. 367–372, doi: 10.1145/3075564.3078888.

[22] M. A. Ahad and R. Biswas, "Dynamic merging based small file storage (DM-SFS) architecture for efficiently storing small size files in Hadoop," *Procedia Computer Science*, vol. 132, pp. 1626–1635, 2018, doi: 10.1016/j.procs.2018.05.128.

[23] A. Khare and B. Indira, "A review on small files in Hadoop," *International Journal of Innovative Technology and Research (IJITR)*, vol. 5, no. 4, pp. 6585–6588, 2017.

[24] H. Brabra, A. Mtibaa, W. Gaaloul, and B. Benatallah, "Toward higher-level abstractions based on state machine for cloud resources elasticity," *Information Systems*, vol. 90, May 2020, doi: 10.1016/j.is.2019.101450.

[25] J. Geetha, D. S. Jayalakshmi, and N. G. Harshit, "Implementation and performance analysis of dynamic partitioning of graphs in Apache Spark," *International Journal of Advanced Computer Research*, vol. 10, no. 48, pp. 116–127, May 2020, doi: 10.19101/IJACR.2020.1048023.

[26] B. Gupta, R. Nath, G. Gopal, and K. Kartik, "An efficient approach for storing and accessing small files with big data technology," *International Journal of Computer Applications*, vol. 146, no. 1, pp. 36–39, Jul. 2016, doi: 10.5120/ijca2016910611.

BIOGRAPHIES OF AUTHORS

Mohammad Bahjat Al-Masadeh PhD candidate data mining/big data Universiti Teknikal Malaysia Melaka (UTeM), Master in information technology Universiti Utara Malaysia (UUM) in 2008, 2009. Currently a data engineer/big data researcher at Umm alQurah University, Mecca. He is a Sr .Net developer. Sr Sharepoint developer in info tech deanship Umm alQurah university, Mecca, Wordpress blogger about Hadoop administration, big data geek, Co-founder of wrshaa.com. He can be contacted at email: mbmasadeh@uqu.edu.sa.

Mohd Sanusi Azmi BSc., Msc and Ph.D from Universiti Kebangsaan Malaysia (UKM) in 2000, 2003 and 2013. He joined Department of Software Engineering, Universiti Teknikal Malaysia Melaka (UTeM) in 2003. Now, he is currently an Associate Professor at UTeM. He is the Malaysian pioneer researcher in identification and verification of digital images of Al-Quran Mushaf. He is also involved in Digital Jawi Paleography. He actively contributes in the feature extraction domain. He has proposed a novel technique based on geometry feature used in Digit and Arabic based handwritten documents. He can be contacted at email: sanusi@utem.edu.my.

Sharifah Sakinah Syed Ahmad received B. Applied Science (Hons) Computer Modelling and M. Sc Mathematics from Universiti Sains Malaysia (USM). Her PhD is in Software Engineering and Intelligent System from University of Alberta, Canada. She is expert in computer aided geometric design (CAGD) and neural networks. Sharifah Sakinah is currently an Associate Professor in the Department of Intelligent Computing and Analytics (ICA), Faculty of Information and Communication Technology, Universiti Teknikal Malaysia Melaka (UTeM). She received her bachelor's and Masters degree of Applied Mathematics in School of Mathematics from the University Science Malaysia. Following this, she received her Ph.D. from the University of Alberta, Canada in 2012 in Intelligent System. Her research in graduate school focused on granular computing and fuzzy modeling. Her current research work is on the granular fuzzy rule-based system, evolutionary method, modeling and data science. She can be contacted at email: sakinah@utem.edu.my.