*Computer Systems Science & Engineering*
DOI: 10.32604/csse.2023.036400
***Article***

Tech Science Press

Check for updates

# Enhanced Best Fit Algorithm for Merging Small Files

**Adnan Ali[1], Nada Masood Mirza[1,2] and Mohamad Khairi Ishak[1,*]**

[1]School of Electrical and Electronic Engineering, Universiti Sains Malaysia (USM), Nibong Tebal, Pulau Pinang, 14300, Malaysia
[2]University College, United Arab Emirates University, Al Ain, UAE
*Corresponding Author: Mohamad Khairi Ishak. Email: khairiishak@usm.my
Received: 29 September 2022; Accepted: 13 November 2022

**Abstract:** In the Big Data era, numerous sources and environments generate massive amounts of data. This enormous amount of data necessitates specialized advanced tools and procedures that effectively evaluate the information and anticipate decisions for future changes. Hadoop is used to process this kind of data. It is known to handle vast volumes of data more efficiently than tiny amounts, which results in inefficiency in the framework. This study proposes a novel solution to the problem by applying the Enhanced Best Fit Merging algorithm (EBFM) that merges files depending on predefined parameters (type and size). Implementing this algorithm will ensure that the maximum amount of the block size and the generated file size will be in the same range. Its primary goal is to dynamically merge files with the stated criteria based on the file type to guarantee the efficacy and efficiency of the established system. This procedure takes place before the files are available for the Hadoop framework. Additionally, the files generated by the system are named with specific keywords to ensure there is no data loss (file overwrite). The proposed approach guarantees the generation of the fewest possible large files, which reduces the input/output memory burden and corresponds to the Hadoop framework's effectiveness. The findings show that the proposed technique enhances the framework's performance by approximately 64% while comparing all other potential performance-impairing variables. The proposed approach is implementable in any environment that uses the Hadoop framework, not limited to smart cities, real-time data analysis, etc.

**Keywords:** Big data; Hadoop; MapReduce; small file; HDFS

## 1 Introduction

The digitalization-based advanced technology has changed today's world extensively. With the help of digitalization, the data processing and storing facility gets easier. However, a noticeable factor is that with the wide spread of digital technology, the digitized world has been overloaded by the massive volume of data collected from various sources [1] that need to be processed within seconds. By considering such requirements, the concept of big data has evolved. Big data is often called the humongous, greater variety, and unmanageable amount of data in three categories. The vast amount of data generated daily has become very difficult to analyze using traditional methods as the data lies in those files crossed the

billion mark long ago. The term "Big Data" can be incorporated with a humongous amount of data that has eliminated the existence of older technologies to handle or process this kind of data in terms of size, speed, and type. In this era of big data that is generated at a high pace and quantity, new techniques and systems must be developed to overcome this issue.

Since these big data files are classified into three main subsets "Structure, Semi-Structured and unstructured" [2,3]. Structured data belongs to a specified format it follows strictly and exists in a relational database structured to follow a table pattern where the data is stored in a row and column format [4,5]. The Semi-Structured data doesn't lie in a relational database but follows a structure property. It can be easily migrated to structure data by utilization of specific techniques, which include JSON, XML, and CSV. Unstructured data encloses a massive chunk of big data generated by the Internet of Things (IoT) devices, Rich Media, etc. Although the data is categorized, another characteristic that cooperates with big data is called 3V's (volume, variety, and velocity). Each aspect has a specific reason, such as volume for the amount measured in Zetta Bytes, variety reflects the different types of data associated, and velocity refers to the speed at which data is generated.

In practice, managing such a wide range of data variety and size is extremely difficult and impossible to control using traditional methods. Considering this factor, the Apache foundation introduced the Hadoop framework to overcome the raised issue by enhancing the vast data processing speed [6]. This framework is built on the core components of Google's architecture, including MapReduce and the Google File System. The Apache Organization developed the open-source framework Hadoop to give a platform capable of processing these massive data files while being practical, accessible, and inexpensive. Data can be analyzed and stored over the distributed system using Hadoop. The Hadoop framework has dominated the big data analysis field due to its compatibility with several applications. These applications include the utilization of hardware resources (from single to hundreds of servers), parallel processing of massive amounts of data, network load balancing, and fault tolerance. Additionally, these characteristics have helped this framework gain popularity by processing files of enormous volumes [7].

Although Hadoop works incredibly well when processing these enormous files, unfortunately, the framework performance is hindered when processing files that are vast in number but small in size, leading to several drawbacks in the framework. Due to memory limitations, each small file consumes a specific amount of memory, leading to memory consumption beyond availability. Each metadata holds 150 bytes of NameNode. If these numbers are ramped up, this will cause deadlock at the memory level and exponentially degrade the performance. Recent advancements have somewhat mitigated this issue by utilizing appendable solutions that combine small files into large ones. However, this led to another problem that started to cause deadlock at the data blocks as those files were beginning to hit the block size limit (64 or 128 MB). On top of the current drawbacks, the disk utilization starts to creep high as the MapReduce job requires seeking these many files that delay the overall processing time along with consumption of higher I/O numbers instead of processing data in the available time.

By considering these critical data processing-related issues, this study proposes a new data preprocessing algorithm that utilizes the Best fit merging algorithm to merge multiple small files based on specified criteria. The proposed approach will aid the Hadoop Framework to resolve issues caused by multiple small files not limited to reduced disk read/write times, memory load, function calls, etc. Moreover, this paper also aims to resolve small file management-related issues [8].

This article is in the following structure: Section 2 explains the Hadoops background and issues related to small files. Section 3 discusses the related works done in the past. Section 4 presents the proposed methodology. Section 5 contains the results of the proposed approach, and Section 6 concludes the paper with future works.

## 2 Background

Over the years, data has become a valuable commodity, with massive data sets generated due to the explosion of sensors and mobile devices. Corporations, governments, and other organizations have realized the importance of data. However, even though the advantages of big data are immense, there are also some associated concerns. One of the major concerns is that organizations may be unable to store and manage all the data appropriately and do not have the expertise or resources to analyze large quantities of data. The Apache Hadoop framework was introduced, comprised of the following modules, demonstrated in Fig. 1, to understand these factors better.

i) Hadoop MapReduce
ii) Hadoop Distributed File System (HDFS)
iii) Hadoop Common
iv) Hadoop Yarn

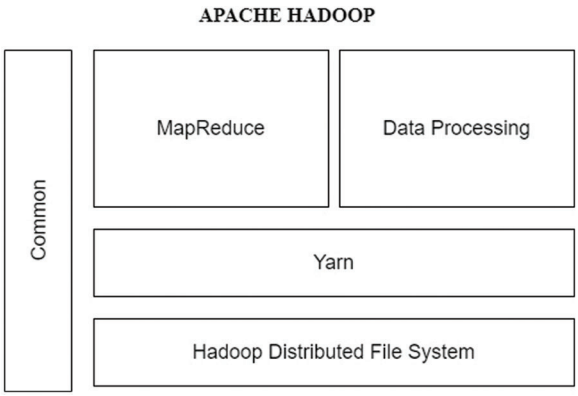


**Figure 1:** Apache hadoop components [9]

It is a notable factor that besides MapReduce, HDFS and Yarn, some related projects have an equal contribution to Apache. The Hadoop platform has been developed with Zookeeper, Chukwa, Avro [10], Apache pig, and HBase, which works as a distributed coordination service provider to Dgraph nodes of big data discovery clusters. These utilities ensure the high availability of the query person by Dgraph nodes in the group. This utility has a unique feature where it creates an overhead file massive number of small files preserved in it. In this process, the name node memory is part of every file, block, and directory in HDFS, which acts as a single entity. The size of the HDFS block is considered 64 MB by default [11].

A large amount of high-speed data develops new challenges that the traditional database management system cannot resolve due to the old inferior concept of data management. Structured data is comparatively easier to handle by such a conventional system. The world has moved on to the production of unstructured data, in which the data management system must be more complicated and trickier to handle the data efficiently [12].

The Hadoop framework is quite efficient in delivering distributed file management and processing a colossal array of chunk data. In this process, the most unified point is that it allows the data partition while processing the data over many machines. Files smaller than the default block size in HDFS are called small files of HDFS. This kind of file does not adequately walk with many small files for the following reasons [13].

- NameNode usually keeps a record of each block and file and keeps these data stored in the memory. Therefore, logically, a more significant number of files would occupy more space in memory. While the NameNode also holds all of its metadata in the RAM for faster access. Now every file, block, and directory in HDFS represents an object in the memory of NameNode. Each of these files usually occupies approximately 150 bytes. Hence, from this calculation, it is clear that the 10 million files using the block are supposed to use 3 GB of memory.
- A single file can be handled only by one block, effectively causing a lot of small block creation that is usually smaller than the configured block. Hence, reading this colossal array of blocks consumes a lot of time [14].

As discussed above, on the NameNode, many small files are usually stored, which demands more storage space. Each small file is generated as a task map; hence, creating too many search maps causes inadequate input. Transforming and storing small-size files in HDFS overburdens the MapReduce program. Due to this fact, the overall performance of the NameNode [15]. A noticeable factor is that when the amount of the small size file gets increased, then a significant fall in Hadoop performance can be seen [16].

### *2.1 High Memory Usage*

The compilation of large files in the HDFS uses a high amount of memory that breaks down the file size of HDFS blocks. On the other hand, the HDFS DataNodes on the C cluster's design infrastructure name node also hold the Metadata relating to each block. Due to search reasons, significant performance degradation is noticeable when a user tries to feed a large number of small-sized data into the system, due to which small and large files are treated differently. For instance, Patel et al. (2015) [17] demonstrate that HDFS uses separated blocks to store each of the files irrespective of the file size. Consequently, using this idea, it can be inferred that a relatively large file of 50 MB saved on HDFS will use the same amount of metadata storage as a similarly sized file of 2 MB.

For this reason, in the case of many small sizes, the name note outgrows the total of the actual file size to maintain the Metadata. This block report is updated regularly with the help of a heartbeat signaling transmission process executed by each block periodically. Zhou et al., in 2015 [18], exclaimed in their journal that the block report is 21,600 s by default. Due to the memory shortage and improper memory management system, the system cannot accept any other operation while updating the procedure for maintaining the synchronization, which consumes more blocks for small files. Blocking multi-functions causes delay and degrades performance, which is noticeable throughout the system. Mukhopadhyay et al., in 2014 [19], proposed that this factor indicates that in the NameNode memory utilization process, a sharp restrictive factor also affects the cluster's capacity.

### *2.2 Node-based Frequent Communication*

A lot of communication happens for accessing a file on Hadoop through frequent communication processes due to the separation of the metadata and the corresponding data across the file system. Matri et al., in 2018 [20]. In such a position, to operate the system properly, the client must establish communication with the NameNode by seeking the file's location first. In response to this call, NameNode will provide the requested file location to the client. Upon receiving the response, the client can establish communication with the desired Data that will proceed further as per the requirements. According to Ahad et al. [21], all communication occurs through a single network, leading to network latencies.

The network latency usually occurs due to the unjustified accessing protocol of HDFC in case of small file transfers. As discussed above, manipulating, and managing a massive amount of small data demands

more time than I/O operations in processing the data. On the other hand, the unavailability of caching and prefetching mechanism in the HDFS also adds to the input-output latency of the file system [22]. Due to this reason, regulation of communication frequencies through the DataNode and Metadata between server and clients is essential.

### *2.3 Overheads Processing*

As this project deals with a massive number of small files, it indicates that processing all of those files generates a vast number of InputSplits (Small processing units) against every file. In the next stage, execution of the mapper takes place for each generated InputSplits. On the other side, a context switch also occurs between each invoke. Hence, creating and closing multiple processes continuously results in a lot of overhead in the corresponding context switches. These kinds of issues cause a severe reduction in the overall performance of the system. Therefore, in the case of processing multiple small files, the map phase may need to deal with the single InputSplit [18].

In the general case, a reducer's main job is to combine the various outputs of multiple mappers, shrink their size, and produce a more precise result. In contrast, only one mapper was used in this study, indicating that the reducer's output is available during the map phase. As a result, the additional overhead will be compulsory to retrieve the map phase output and use it as the reduced phase input [23].

### *2.4 Cross-node Transmission File*

Many petitions may take place to retrieve a large number of small files in a single system, leading to a loT of hopping from one particular DataNode to another DataNode. Such factors promote excessive congestion in the network system and reduce the system's performance throughout. In practice, the time spent retrieving small files from the system cannot be of higher priority over processing these files [24].

## 3 Related Work

Multiple techniques have been proposed to effectively merge multiple files, irrespective of type and size. Such practices usually lead to an imbalance in the framework processing criteria. The small files problem in Hadoop HDFS is the main focus factor that needs improvement. In this context, more research has been done and figured that the standard size of a block of data is more significant than any other file system, i.e., n * 64 MB (Where, n = 1, 2, 3…). According to Renner et al., in 2017 [25], those files smaller than the block size to a reasonable extent can be considered small files. The fact that Hadoop is ineffective at managing smaller files is brought to light much more by this factor. As a result, numerous methods for guiding the answer have been suggested [24].

### *3.1 Hadoop Archive*

In the range of archiving techniques, the Hadoop Archive (HAR) is a fast technique that packs several small files into HDFS blocks more efficiently. In the case of HAR files, additional information is not essential as this file can be accessed directly (HAR is different from zipping, RAR) from the main memory. The other advantage of HAR is that it reduces the data storage overhead on MapReduce and NameNode operations to enhance performance [26,27].

Accessing the file through the HAR layout technique requires two indexes file-based read operations and one data file-based read operation. For example, when accessing a client file, the request must be sent first to the metadata index so that the requested data is accessible from the metadata archive [27]. However, it is notable that the file reading in HAR is less efficient and slower than the HDFS reading process. Access the files through one data file read operation, and two indexes file-based read operations are needed to execute along with slight overhead in the referencing file. The required file is accessible by requesting the

index metadata. Due to slow operation, the mapping process fails to operate on all pre-allocated files in the HR code resident on an HDFS block. On the other side, upgrading the HAR, the HDFS architecture needs to change, which is quite tricky in practice [13,26,28].

### *3.2 Improved HDFS*

The improved HDFS model is often referred to as the index-based model due to its statistical-based advanced techniques. The burden of the NameNode is reduced by keeping the file in the same directory and merging the data into big files generating the index file. Thus, eliminating excessive useless junk in the index file for smoothing the NameNode. On the other side, to raise the efficiency of reading the small files, the cache manager is placed at the DataNode, known as cache policy.

Hence, checking cache data must be done before reading small-size data every time. If the resources are not present in the cache, then the smaller size is sorted based on the current data [27]. To appropriately execute the sorting technique, integration of all required files with the big file is necessary, which is part of an individual index file. The workings file size must be less than the block size to store big files in a single block. In case the file size is greater than the block size, multiple blocks need to be used [27]. Fig. 2 shows the basic architecture of HDFS.

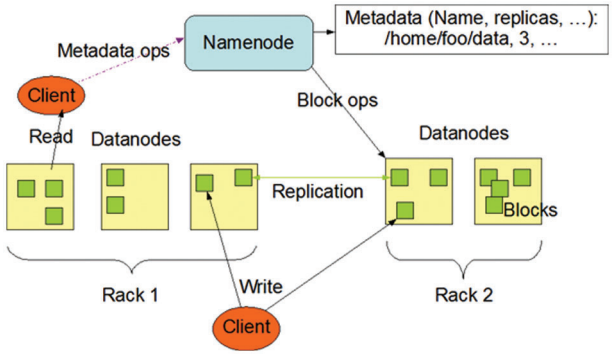


**Figure 2:** HDFS architecture [29]

File integration design: Every big file usually comprises the index files in which there is the length and the offset of the given original small file.

The process of file integration: Shortening the smaller files under the directory base, and later those files are again written into the big file one by one.

Sum of small files determination: These primary functions are to initially calculate the size of the big file and then compare the result with the HDFS block's standard size.

Thus, based on the paradigm mentioned above, the sequence and offset of each file in a big file can be identified and usually built according to the index file, which ends the integration process [13].

Table 1 shows different techniques proposed in the past. Most of the past strategies suggested for solving the file merging problem present new ideas for doing so by combining files in novel ways that improve the functionality of the frameworks and guarantee efficiency. Reducing disk input/output and memory use has been a common theme throughout all previously suggested technologies.

The fundamental issue still exists since none of the recommended methodologies consider the file type and a technique that would improve block sizing balance, which might further influence the processing mechanism of the framework.

**Table 1:** Comparison of proposed techniques in the past

| Paper's outcome | Data management procedures | Research's novelty | Evaluation parameters |
|---|---|---|---|
| Handling of numerous small files [30] | • DQFS-small files of different sizes are inserted into dynamic queues.<br>• A new index is built over the small files that were previously combined.<br>• The use of caching and prefetching improves access. | • In the DQFS, queues are built under different categories.<br>• Access to small files is made possible by logical hierarchical interaction.<br>• Analytical models are used to identify the size of dynamic queues. | • Accessing time of small files.<br>• Memory utilization for name node.<br>• Metadata size.<br>• Data set upload time.<br>• The overall working of DQFS. |
| Improving small files' input/ output [31] | • Data is referred to directly by using address variables.<br>• With the aid of a cache mechanism, data blocks can be accessed quickly.<br>• Null-sized files are eliminated using the inline data concept. | • Decrease in disk input and output operations.<br>• Reduce application-level lag time.<br>• Increased throughput for processing multiple files. | • Accessing time of small files.<br>• Disk usage.<br>• Cumulative throughput. |
| A solution to the problem of HDFS small files [32] | • Merging's improved method.<br>• Techniques like prefetching and caching are used to reduce delay | • Accessing small files takes less time.<br>• Utilizing a faster and more accessible index file system allows for the maintenance of small file metadata. | • Accessing time of small files.<br>• Memory utilization for name node. |
| Optimized system for small files [33] | • A suggestion for a new file system that can work with different types of storage.<br>• Use of consistency in simple metadata management. | • Access and storage enhancement of online apps is beneficial.<br>• By gaining access to the disk's raw data, creation of the index file for metadata. | • Critical point identification.<br>• Size of metadata.<br>• The ratio of files written.<br>• Comparison with other files systems<br>• Access of throughput |

### *3.3 Best Fit Algorithm (BFA)*

The Best Fit Algorithm is a memory allocation algorithm that allocates the smallest amount of memory the process needs. The term "smallest" means that the partition has a minimum size so that the requesting process can allocate maximum-size objects.

The memory scans from start to end, searching for a group of holes. This method significantly improves memory usage over other memory allocation algorithms. It takes a bit more time since it needs to look for a suitable collection of holes—however, less internal fragmentation results from this effort than from the first

fit. For instance, if an item has a size of 1000 bytes, the best-fit partition will have a size of 1000 or less. The free/busy lists are kept in size order, from least in size to most significant, by the best fit method [34].

The following are the steps involved in the best fit algorithm.

- Track down every single hole that has existed since the memory's inception.
- On completing all processes selection in order, it starts choosing them one by one.
- From the group of holes, choose the one with the smallest diameter.
- If the chosen hole equals the size of the current process, then assign the present hole to the running process and remove it from the set
- If there is no hole available, then mark the procedure as unsatisfied [35]

## 4 Proposed Methodology

This section describes the overall methodology of the proposed method by making the small file issue the main focus area. As discussed, the NameNode's overheads are a prevalent issue when processing small files in Hadoop. This technique frequently involves keeping separate metadata for all types of small-size file that effectively uses more mappers. The proposed approach will utilize the EBFM algorithm to resolve the raised issue. The Best Fit merging algorithm ensures utilization of the available resources to the max.

In contrast, the other approaches don't fit our merging criteria as they are not memory efficient and consume more resources which are against our goal. Through this process, merging small files into one massive file close to the threshold is achievable, which utilizes the maximum size of the data block to speed up the processing. As per the NameNode architecture designing process, the files more significant than the threshold will be readily allowed to pass through directly, skipping the proposed systems processing.

Fig. 3 shows the system flowchart. This study has developed an algorithm combining many small files in terms of size and format. In the next stage, the small files will be placed in a specific directory to smoothen the preprocessing program readability. The program starts to run when these files are available in the designated directory, creating a loop through all types of files. An illustration of the system flowchart is listed below for better understanding.

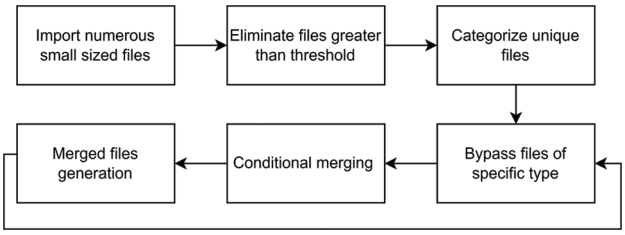


**Figure 3:** System flowchart [8]

As per the system flowchart, the initial step is to run the program to scan all the files from the specified directory. For the smooth execution of the process, the threshold value point is set dynamically in the program, at which point the program will start to discard files that exceed the specified threshold. Every time a file is discarded, the user is routinely informed. After elimination, the unique function executes to retrieve all files, which later helps filter files of similar type. The next step is to select a specific file type, and all files of that type will be processed. All file types are selected until there is no longer a file type in the unique function.

After possessing these files, the system executes a subset generator function that generates all the possible subsets used in merging. Each subset may contain one to many files depending on the number of files that are of the same type delivered to this function. On completion of subset generation, the system then processes these subsets. In any case, if the subset contains only a single file, then the subset will be ignored during the execution of the merging stage, which happens when most of the files from the subset are merged or so-called at the end phase of merging.

The Best Fit function processes the subset that contains more than one file. On passing these subsets of files, the function loops through each subset individually, and the system pulls each file's size from the subset and simulates the merging of files. It will calculate the sum of the file size selected from the subset and validate it by comparing it with the simulated file size, the threshold, and the merging of the current file is greater than the previous file (closer to the threshold value). The function will append this file to merge upon fulfilling the above conditions. Otherwise, the function will skip the file and proceed to the next one. In the following way, the system merges the selected files and removes them from the subset generation to ensure new subsets do not contain the merged files. The system will pass those files selected for merging to the merging function, which will merge the files and place them in a specified directory.

Thus, the above will repeatedly run until all the files of the specific types are merged. Upon merging all files, the loop becomes empty. It will request more files of the same type from the unique function to merge and produce an output process. On completion of program execution, the user can obtain and collect the merged files from the defined directory at the final stage. Demonstrating the program's execution is described in this part, Table 2 contains the pseudo-code for the proposed approach.

**Table 2:** EBFM algorithm

Algorithm 1: Algorithm to Merge Numerous Small Files

**Input:** *Files in specified merging directory*
**Output:** *Merged files*
**Initialisation:** *threshold, filesToMerge, allFiles, uniqueType*

```
1       Scan and retrieve all files and type them in the directory
2       Retrieve all unique files type
        LOOP Process
3       for i <= uniquetype do
4           Call mergeFiles(uniqueType, allFiles, filesToMerge)
5           Call bestFit(filesToMerge, threshold)
6           Initialize variables: simFileSize, currentFileSize, FilesMerge, subsetFiles
7           while (count(filesToMerge) >= 1)
8               if(count(subsetFiles) <= 2)
9                   Retrieve subsets simulated size
10                  if (simFileSize < maxSize)
11                      merge the files
12                  end if
13              Else
```

(Continued)

**Table 2 (continued)**

| Algorithm 1: Algorithm to Merge Numerous Small Files | |
|---|---|
| | ***LOOP Process*** |
| 14 | **for** *i <= count(subsetFiles)* **do** |
| 15 | *Simulate merged file size of subset(i) with the subset (i + 1)* |
| 16 | **if** *(currentFileSize < simFileSize && currentFileSize < maxSize && simFileSize < maxSize)* |
| 17 | *remove all files from filesToMerge* |
| 18 | *insert new files to the filesToMerge* |
| 19 | **end if** |
| 20 | *send filesToMerge for merging* |
| 21 | *remove files from allFilesMerge* |
| 22 | *reinitialize all defined variables* |
| 23 | **end for** |
| 24 | **end loop** |
| 25 | **end if** |
| 26 | **end while** |
| 27 | **end for** |
| 28 | **End** |

According to the proposed approach, the EBFM algorithm generates a few large files from those numerous small files by merging them. By delivering these files, the Hadoop framework will have fewer jobs to complete and will not worry about the small files consuming all of its NameNode memory. Overall, it should improve the framework's performance because it considerably decreases the issues raised by the small files, as explained previously.

To evaluate the effectiveness of the proposed system by setting some critical parameters directly related to the overall framework's performance metrics. These parameters include the disk input/output, time spent in the occupied slots, initiated map tasks, virtual and physical memory consumption, memory consumed by the map and reduce functions, CPU and garbage collection time, and Heap memory consumption.

## 5 Results

This section provides a brief overview of the system resources utilized and compares resource consumption with other methodologies. Results are classified based on critical elements affecting the framework's processing during the small files. The deployment of the proposed system is on a virtual machine with Ubuntu 20.04, 6 GB of memory, 60 GB of storage, and Apache Hadoop version 2.10. The deployment of the suggested system in a virtual environment may result in some performance variances. However, this is ineffective in our situation since a single server with the necessary data can handle the test without introducing any additional factors. Lets assume the following:

Number of files in traditional approaches = $N_o$

Number of files in the proposed approach = $N_m$

Overall sum of file size (traditional approach) = Overall sum of file size (proposed approach) = $S$

Time taken by traditional approaches = $T_o$

Time taken by proposed approach = $T_m$

Average file size = Sum of all file sizes/total number of files = $\frac{S_t}{S}$

As per the conditions applied in the proposed approach, it can be easily stated that:

$$N_{m=}\ N_o\ -\left(\frac{S_t}{S}\right) * 0.05 \tag{1}$$

Which leads to the fact that

$$N_m < N_o \tag{2}$$

Knowingly, the time taken is directly proportional to the number of files, thus

$$T_o \propto N_o \tag{3}$$

$$T_m \propto N_m \tag{4}$$

By comparing Eq. (2) with Eqs. (3) and (4) it can be easily stated that

$$T_m < T_0 \tag{5}$$

Which leads to an increase in the efficiency of Hadoop framework which is calculated by Eq. (6)

$$E_f = T_m/T_o \tag{6}$$

The simulation is executed five times to determine an anticipated average. Results are compared based on graphs generated by processing these files in three batches, by the proposed system, BFM [6], and the baseline Hadoop. The input for testing is Twelve text documents containing random phrases and numbers varied in size from 1 kilobyte to 24 kilobytes.

Although the result indicates minor execution time differences due to the small data set, it covers the primary concern of processing small files. Comparatively small documents weigh less than 60% of the upper threshold. The merge limit/upper threshold is set by taking an average of all small files and reducing it by 5%; hence the largest file size that will be determined to merge will likely be closer to the upper limit.

Fig. 4, shows how long the map and reduce function takes to process the key-value pairs in occupied slots before passing them. Based on the generated graph, it is clear that processing files by the proposed system took less time than those passed directly. The baseline Hadoop consumed 126449 ms in the occupied slots by the map function and 2652 ms in occupied slots by the reduce function, whereas the BFM consumed 72645 and 1861 ms, respectively. EBFM reduced the time to 65380.5 and 1674.9 ms.

Fig. 5, shows the distribution of the system's input/output demand. The baseline Hadoop consumed 181302 bytes to read files and 3074521 bytes to write, whereas the BFM reduced these numbers to 90654 and 1641525 bytes, respectively. In contrast, the most gains are visible when the proposed system processes these files, drastically decreasing the consumed time to 90654 and 1641525 bytes, respectively. The proposed system considerably impacted the amount of disk I/O used.

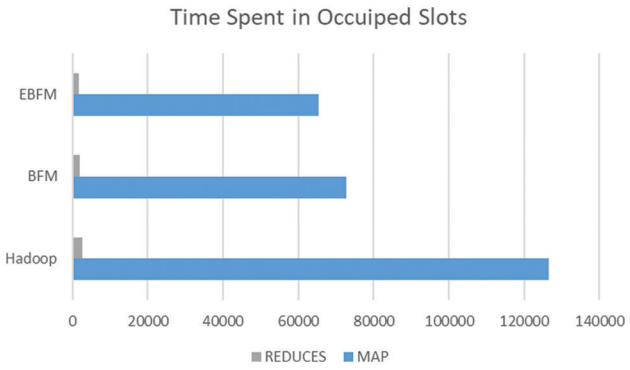


**Figure 4:** Total time spent in occupied Slots

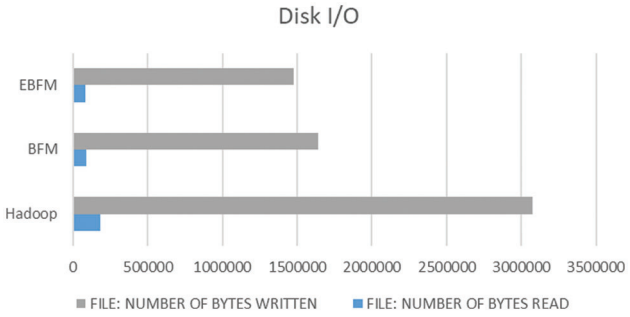


**Figure 5:** Disk I/O utilization

Fig. 6, shows the Total virtual and physical memory consumed. The baseline Hadoop consumed the most, where the virtual memory consumption is 24.5 GB which is 3.2 times more than the available memory, and physical memory consumption stands at 3.5 GB. In contrast, the BFM approach reduced the numbers to 13.3GB for virtual and 1.8 GB for physical memory. The proposed system significantly reduced the numbers to 11 GB for virtual and 1.5 GB for physical memory, which ensures that the file processed used a fraction of the system's resources compared to unprocessed files and depicted the enormous inefficiencies in processing caused by the sheer volume of small files.

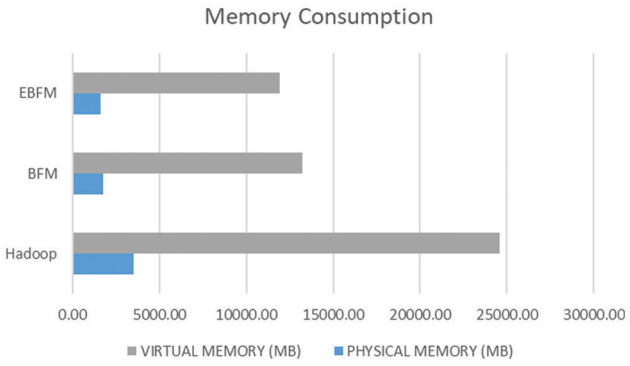


**Figure 6:** Memory consumption

Fig. 7 shows a graph representing the number of map function jobs required to process a file. When the proposed system did not handle the files, the Launched and Data-Local map jobs used considerable

resources. At the same time, the BFM approach was able to drop these numbers to 50% (6 calls) less than the initial one (12), but the proposed system substantially decreases these numbers by 60% (5 calls), where the framework only started half as many Maps tasks.

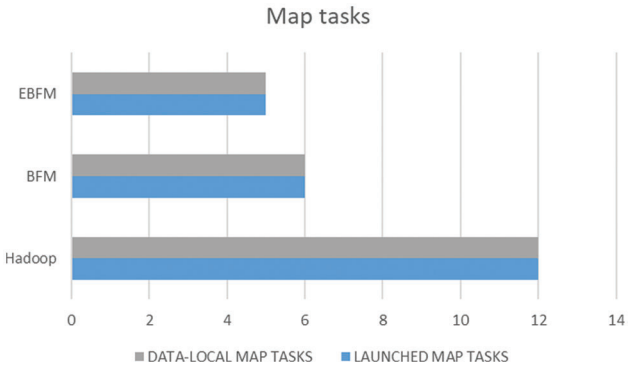


**Figure 7:** Map Tasks

Fig. 8, displays the total Megabyte consumed to process these files by all maps and reduce processes. According to the graph, files processed by the proposed system required incredibly little memory because they could take advantage of the framework's core strength while processing them. The baseline Hadoop consumed 126449 MB in all-map tasks and 2652 MB in all-reduce tasks, whereas the BFM consumed 72645 and 1861 MB, respectively. EBFM reduced the memory utilization to 65380.5 and 1674.9 MB. The baseline required much memory because these split files required more megabytes to be processed, which required more memory to hold information related to those files.

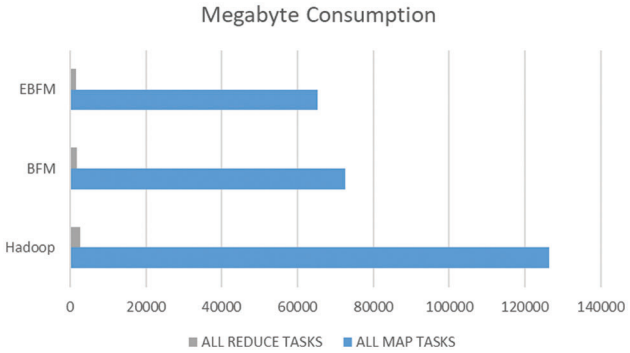


**Figure 8:** Megabyte consumption

Fig. 9, displays the CPU and Garbage Collection (GC) times. CPU time is the total time a job spends before completion. The time a garbage collector spends picking up abandoned or broken items is known as garbage collecting. Maintaining the trash collection time short is vital since it will enhance system performance. According to the results, the framework could function considerably more effective, when the proposed system processed these files with a significant decrease in time. The baseline Hadoop consumed 3059 ms for GC and 8120 ms for CPU, whereas the BFM consumed 1715 and 6260 ms, respectively. EBFM reduced the time to 1542 and 5634 ms. Most of the time consumed by the baseline was spent processing files without a solid reason.

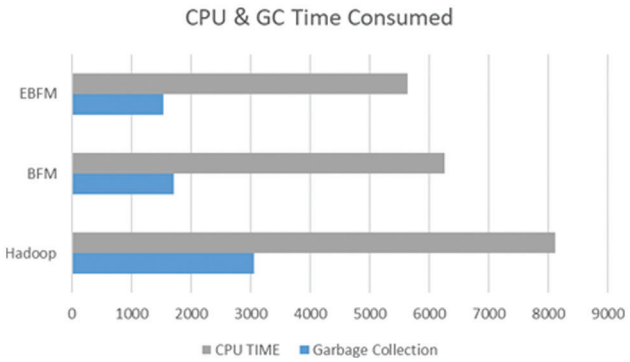


**Figure 9:** CPU and garbage collection time

Fig. 10 shows the Heap memory consumption. The Java Virtual Machine's usable memory is called heap memory. The produced data show that the unmerged small files utilized about 41% (2484.08) of the heap memory, although the smallest in size. Files processed by the BFM approach consumed 21% (1286.6) of heap memory. However, files handled by the proposed system required 19% (1157.94) of the heap memory, which significantly enhanced the framework's processing.

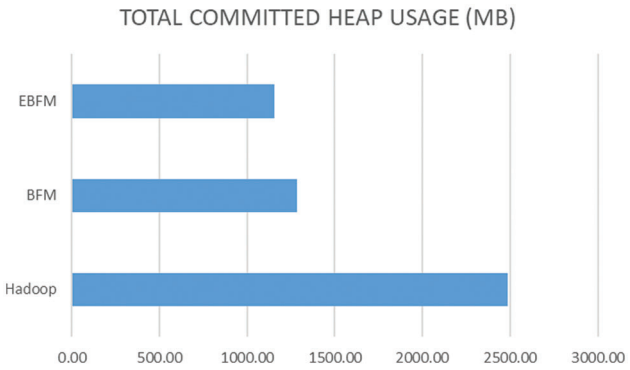


**Figure 10:** Heap Utilization

## 6 Conclusion

One of the significant problems in the modern period is the vast amount of data collected from numerous sources, termed "Big Data". This data not only comes with certain constraints but also varies in size. The small files retrieved from multiple sources contain valuable information that needs analysis. Apache organization proposed the Hadoop framework to process Big Data, but unfortunately, it struggled when it came to numerous small files. Developing New tools and procedures is necessary to solve the issues that prevent the present frameworks from handling this large number of small-sized files. This research proposes a novel merging strategy named EBFM. The proposed approach will dynamically merge all small files within a specified range determined by the threshold based on the number of files of a specific type. The merged files will be incredibly close to the threshold, which aids the framework by eliminating any imbalance on all the blocks and enhancing the read/write operations. It also improves the Hadoop performance while reducing disk utilization, memory load, data, name-node, number of mappings, function calls, and response time by approximately 64%. Testing of the proposed system is limited to small data sets, implementing/experimenting on a single node, and executing in a simulated environment have proven to resolve all the problems mentioned earlier.

In the future, the proposed approach can be adaptable over an HDFS cluster, a more comprehensive range of file formats with substantial data sets, introduced into a streaming system where small files are available and introduced in an intelligent city system.

**Acknowledgement:** The authors would like to thank Universiti Sains Malaysia(USM) and the Ministry of Higher Education Malaysia for providing the research grant, Fundamental Research Grant Scheme (FRGS-Grant No: FRGS/1/2020/TK0/USM/02/1) that helped to carry out this research.

**Funding Statement:** This research was supported by the Universiti Sains Malaysia (USM) and the ministry of Higher Education Malaysia through Fundamental Research Grant Scheme (FRGS-Grant No: FRGS/1/2020/TK0/USM/02/1).

**Conflicts of Interest:** The authors declare that they have no conflicts of interest to report regarding the present study.

## References

[1] S. Vengadeswaran and S. R. Balasundaram, "CORE-An optimal data placement strategy in Hadoop for data intensive applications based on cohesion relation," *Computer Systems Science and Engineering*, vol. 34, no. 1, pp. 47–60, 2019.

[2] A. C. Anadiotis, O. Balalau, C. Conceição, H. Galhardas, M. Y. Haddad *et al.,* "Graph integration of structured, semistructured and unstructured data for data journalism," *Information Systems*, vol. 104, pp. 101846, 2022.

[3] W. N. Chan and T. Thein, "Sentiment analysis system in big data environment," *Computer Systems Science and Engineering*, vol. 33, no. 3, pp. 187–202, 2018.

[4] M. Naeem, T. Jamal, J. D. Martinez, S. A. Butt, N. Montesano *et al.,* "Trends and future perspective challenges in big data," In *Advances in Intelligent Data Analysis and Applications*, vol. 253, pp. 309–325, Singapore: Springer, 2022.

[5] A. Gandomi and M. Haider, "Beyond the hype: Big data concepts, methods, and analytics," *International Journal of Information Management*, vol. 35, no. 2, pp. 137–144, 2015.

[6] R. Beakta, "Big data and Hadoop: A review paper," *International Journal of Computer Science & Information Technology*, vol. 2, no. 2, pp. 13–15, 2015.

[7] F. Mehdipour, H. Noori and B. Javadi, "Energy-efficient big data analytics in datacenters," *Advances in Computers*, vol. 100, pp. 59–101, 2016.

[8] A. Ali, N. M. Mirza and M. K. Ishak, "A new merging numerous small files approach for Hadoop distributed file system," in *2022 19th Int. Conf. on Electrical Engineering/Electronics, Computer, Telecommunications and Information Technology (ECTI-CON)*, Hua Hin, Thailand, pp. 1–4, 2022.

[9] A. Abualkishik, "Hadoop and Big data challenges," *Journal of Theoretical and Applied Information Technology*, vol. 97, no. 12, pp. 3488–3500, 2019.

[10] M. Kerzner and S. Maniyam, "Hadoop illuminated," *Hadoop Illuminated, LLC. Retrieved on November*, vol. 12, pp. 2017, 2013.

[11] K. G. Srinivasa, G. M. Siddesh and H. Shreenidhi, "Hadoop," In: K. G. Srinivasa, G. M. Siddesh and H. Shreenidhi (Eds). *Network Data Analytics: A Hands-on Approach for Application Development*, Cham: Springer International Publishing, pp. 29–53, 2018.

[12] T. El-Sayed, M. Badawy and A. El-Sayed, "Impact of small files on hadoop performance: Literature survey and open points," *Menoufia Journal of Electronic Engineering Research*, vol. 28, no. 1, pp. 109–120, 2019.

[13] S. Bende and R. Shedge, "Dealing with small files problem in Hadoop distributed file system," *Procedia Computer Science*, vol. 79, pp. 1001–1012, 2016.

[14] P. Gohil and B. Panchal, "Efficient ways to improve the performance of HDFS for small files," *Computer Engineering and Intelligent Systems*, vol. 5, no. 1, pp. 45–49, 2014.

[15] 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.

[16] W. Jing and D. Tong, "An optimized approach for storing small files on HDFS-based on dynamic queue," in *Proc. -2016 Int. Conf. on Identification, Information and Knowledge in the Internet of Things, IIKI 2016*, Beijing, China, pp. 173–178, 2016.

[17] A. Patel and M. A. Mehta, "A novel approach for efficient handling of small files in HDFS," in *Souvenir of the 2015 IEEE Int. Advance Computing Conf., IACC 2015*, Bangalore, India, pp. 1258–1262, 2015.

[18] F. Zhou, H. Pham, J. Yue, H. Zou and W. Yu, "SFMapReduce: An optimized MapReduce framework for small files," in *2015 IEEE Int. Conf. on Networking, Architecture and Storage (NAS)*, Boston, MA, USA, pp. 23–32, 2015.

[19] D. Mukhopadhyay, C. Agrawal, D. Maru, P. Yedale and P. Gadekar, "Addressing name node scalability issue in Hadoop distributed file system using cache approach," in *Proc.-2014 13th Int. Conf. on Information Technology, ICIT 2014*, Bhubaneswar, India, pp. 321–326, 2014.

[20] P. Matri, M. S. Perez, A. Costan and G. Antoniu, "TýrFS: Increasing small files access performance with dynamic metadata replication," in *Proc.-18th IEEE/ACM Int. Symp. on Cluster, Cloud and Grid Computing, CCGRID 2018*, Washington, DC, USA, pp. 452–461, 2018.

[21] M. A. Ahad and R. Biswas, "Handling small size files," In *Hadoop: Challenges, Opportunities, and Review*, *in Soft Computing in Data Analytics*, *Advances in Intelligent Systems and Computing*, vol. 758, pp. 653–663, Singapore: Springer, 2019.

[22] Y. Mao, B. Jia, W. Min and J. Wang, "Optimization scheme for small files storage based on Hadoop distributed file system," *International Journal of Database Theory and Application*, vol. 8, no. 5, pp. 241–254, 2015.

[23] K. Ji and Y. Kwon, "New spam filtering method with hadoop tuning-based MapReduce naïve Bayes," *Computer Systems Science and Engineering*, vol. 45, no. 1, pp. 201–214, 2023.

[24] R. Aggarwal, J. Verma and M. Siwach, "Small files' problem in Hadoop: A systematic literature review," *Journal of King Saud University-Computer and Information Sciences*, vol. 34, no. 10(A), pp. 8658–8674, 2021.

[25] T. Renner, J. Müller, L. Thamsen and O. Kao, "Addressing Hadoop's small file problem with an appendable archive file format," in *ACM Int. Conf. on Computing Frontiers 2017, CF 2017*, Siena, Italy, pp. 367–372, 2017.

[26] M. A. Mir and J. Ahmed, "An optimal solution for small file problem in Hadoop," *International Journal of Advanced Research in Computer Science*, vol. 8, no. 5, pp. 321–325, 2017.

[27] J. Chen, D. Wang, L. Fu and W. Zhao, "An improved small file processing method for HDFS," *International Journal of Digital Content Technology and Its Applications*, vol. 6, no. 20, pp. 296, 2012.

[28] V. G. Korat and K. S. Pamu, "Reduction of data at NameNode in HDFS using harballing technique," *International Journal of Advanced Research in Computer Engineering & Technology*, vol. 1, no. 4, pp. 635–642, 2012.

[29] S. Sharma, U. S. Tim, J. Wong, S. Gadia and S. Sharma, "A brief review on leading big data models," *Data Science Journal*, vol. 13, Committee on Data for Science and Technology, pp. 138–157, 2014.

[30] W. Jing, D. Tong, G. Chen, C. Zhao and L. Zhu, "An optimized method of HDFS for massive small files storage," *Computer Science and Information Systems*, vol. 15, no. 3, pp. 533–548, 2018.

[31] H. Kim and H. Yeom, "Improving small file I/O performance for massive digital archives," in *2017 IEEE 13th Int. Conf. on e-Science (e-Science)*, Auckland, New Zealand, pp. 256–265, 2017.

[32] Y. Lyu, X. Fan and K. Liu, "An optimized strategy for small files storing and accessing in HDFS," in *2017 IEEE Int. Conf. on Computational Science and Engineering (CSE) and IEEE Int. Conf. on Embedded and Ubiquitous Computing (EUC)*, Guangzhou, Guangdong, China, vol. 1, pp. 611–614, 2017.

[33] S. Fu, L. He, C. Huang, X. Liao and K. Li, "Performance optimization for managing massive numbers of small files in distributed file systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 12, pp. 3433–3448, 2014.

[34] A. K. Mandal, D. K. Baruah, J. Medak, I. N. Gogoi and P. Gogoi, "Critical scrutiny of memory allocation algorithms: First Fit, best Fit and worst Fit," *Turkish Journal of Computer and Mathematics Education*, vol. 11, no. 3, pp. 2185–2194, 2020.

[35] L. W. Htun, M. M. M. Kay and A. A. Cho, "Analysis of allocation algorithms in memory management." *International Journal of Trend in Scientific Research and Development*, vol. 3, no. 5, pp. 1985–1987, 2019.