Arabian Journal for Science and Engineering
https://doi.org/10.1007/s13369-021-05929-5

RESEARCH ARTICLE-COMPUTER ENGINEERING AND COMPUTER SCIENCE

The Role of Vertical Elastic Namenode in Handling Big Data in Small Files

Mohammad Al-Masadeh[1] · **Fahad Al-Zahrani**[1]

Received: 8 August 2020 / Accepted: 20 June 2021
© King Fahd University of Petroleum & Minerals 2021

Abstract
Hadoop is a distributed system used exclusively for BigData analysis and processing that is Hadoop distinguished itself through its high performance and a solid availability. Hadoop cluster is available for use at any time and this is one of Hadoop's solid attributes that make it popularly known in data analysis and sciences. However, there are a several factors impacting Hadoop cluster, causing it to be inaccessible. One of these factors is BigData in small files whereby Hadoop's availability shortage accrued when a massive amount of small files dataset is pushed toward a Hadoop cluster. This will harm the cluster's performance, making it unavailable for access and use. This negative factor affects the Namenode itself as the Namenode is a single point of failure. Hence, once it crashes, the whole cluster will be out of service and need to jump again manually. This paper will introduce the elastic Namenode in lieu of the current traditional one. The elastic Namenode has an ability to adapt to the frequent negative factors that are affecting the whole cluster, causing it to become unavailable. The elastic Namenode will adopt the vertical elasticity manner, this type of elasticity will add more memory resources to the Namenode based on a direction from a script that traces the Namenode memory itself. The result will be a cloud elastic Namenode that can be expanded and shrunk upon request, which allows Hadoop cluster to treat BigData in small files without any negative factor or issue.

Keywords Hadoop · HDFS · BigData · Namenode · Cloud · Development · Elastic Namenode

1 Introduction

Data availability and connectivity is a major principle that makes Hadoop attractive to data vendors. This is because the stored data inside are secured and available for access at any time [1]. Since Hadoop's first release in late 2006, apache software (Hadoop Owner) has been used for altering and updating the data availability methodologies to guarantee data integrity and availability. In Hadoop 2.X, apache is used to innovate file replication manner whereby every single file must be replicated two more times (three copies in total) in order to assure data availability and integrity. Only one of these three copies is in charge while the rest are on standby. Therefore, if the running copy is being somehow unreachable, then, other copies will take the lead to keep the data processing job active [2]. In Hadoop 3.X, apache performs a major step in data availability with erasure coding (ER). ER decodes the original data files into data chunks, and each one of these chunks will make another encoded copy called parities. So, when a missing data accrued, both of chunks and parities (called strip) will work together to reconstruct the missing data [3].

Hadoop works excellently with data availability and integrity because both replication and erasure coding keep Hadoop's data on the safe side and away from recovery failure. However, some data vendors reported that Hadoop cluster itself is unavailable for reach owing to several factors, which means that the whole cluster becomes unreachable and all the row data and ecosystems inside cannot be accessed [4]. BigData in small files was one of these negative factors impacting the Namenode with a massive number of metadata that must be hosted in the Namenode's memory. Apache noticed this problem early in Hadoop 2.X, and hence, Apache used to develop several recovery solutions to the shortage of metadata that are heading toward

✉ Mohammad Al-Masadeh
mbmasadeh@uqu.edu.sa

Fahad Al-Zahrani
fayzahrani@uqu.edu.sa

[1] Umm AlQura University, Al-Azizyyah, Mecca, Kingdom of Saudi Arabia

Published online: 07 August 2021

the Namenode. Many of those solutions have been used to customize the HDFS hosting mechanism while the rest of others are appending more complexity to the Hadoop topology which made the solution useless for adoption [5, 6].

2 Current Hadoop Topology

Hadoop cluster is a distributed system based on a client–server topology, and it consists of one Namenode and several thousands of Datanodes. Namenode is the Hadoop manager, and it knows every transaction accrued on the cluster. It also knows all the Datanodes activities and the stored data [7]. Namenode can ride all the clusters using the metadata, and these metadata come from all Datanodes that report the status of each Datanode and the data inside it, and whether the Datanode is up and running (or not) [8]. Furthermore, these thousands of Datanodes are storing the actual data that Hadoop already holds for analysis and insights, and each Datanode comprises a group of storing units called Datablocks [9]. Each Datablock size is 64 MB by default used to host the upcoming files. If the file is bigger than the Datablock size, Hadoop will split it into two or more parts in order to fit the Datablock size. Each Datablock has two more replications in order of data availability, and these replications are hosted into other Datanodes. As such, if one of the Datanodes has a running failure due to power issue or experiences OS failure, the replication will take place to keep the data availability in a high manner. All of Datanodes must send a signal to the Namenode. The signal which is called a Heartbeat [2] informs the Namenode that a given Datanode is up and alive. Also, the Namenode needs to know that all the Datanodes are consistently alive. Hence, if one Datanode stopped sending a signal (heart-beating) to the Namenode, this last one will list all the Datablocks inside that dead Datanode in order to get the backups ready to take over. There is another node in Hadoop cluster called secondary Namenode, and this node is a standby Namenode that is ready to lead the cluster in case the master Namenode suffers a fatal crash and becomes unavailable. The secondary Namenode is fully synchronized with the master Namenode metadata [8]. In normal system, the data processing mechanism involves sending the data to the computing unit in order to ascertain the required information, but with BigData, it is quite impossible to move 10 TB of data to the computing unit. Thus, Hadoop sends the computing script to the data place, whereby the Hadoop data processing involves splitting the large processing job into several small jobs which are then apportioned to the Datanodes [10].

3 Hadoop Availability Issue (BigData in Small Files)

In Hadoop, the actual data place is in the Datanodes and each of these nodes contains several Datablocks storing units that facilitate Hadoop in finding the desired data. The default size of each Datablock in Hadoop is 64 MB, and therefore, any file in Hadoop is considered a big file if the file size is larger than or equal to the Datablock size [11]. Hadoop data processing best practice treats the big file, whereby if the file is bigger than the Datablock, the system will split it to fit several data blocks, and the system will then send the required metadata about each new hosted Datablock to the Namenode. However, the situation is different for BigData in small files. In this regard, small files are any file which is smaller than the Datablock size and these files could be photographs, sound records, documents, sheets and some other models such as log files. Hadoop cluster availability issue is among the factors impacting the cluster performance and availability, making it inaccessible. These factors will be accrued by the delivery of a large number of small files toward the cluster. Here, each file presents a single Datablock and the Datablock cannot host more than one file. Thus, if the file is 1 KB or larger, only one file will be hosted in one Datablock [6, 12]. In order to clarify this issue further, suppose there are two datasets and both of them are 2 GB in size. The first dataset is BigData in big file, and the other one is BigData in small files. Table 1 will uncover the comparison between these datasets.

In Table 1, there are two types of datasets that can be uploaded to a Hadoop cluster, namely dataset 1 and dataset 2. Dataset 1 comprises a BigData in big files and each file size is 51 MB. The Datablock default size is 64 MB but the available size to use is 80% of the Datablock as the other 20% is for Hadoop usage purpose, and so, each Datablock can hold only 51 MB [13]. Thus, 2 GB files require 40 free Datablocks in HDFS (Hadoop distributed file system).

Table 1 Comparison between big files and small files

	Dataset 1	Dataset 2
Type	Big files, 51 Mb/file	Small files, 1 Mb/file
Size	2 GB	2 GB
No. of required Datablocks	40 Dbs	334 Dbs
Total size required to be hosted	2 GB	21.376 GB
Number of Namenode metadata	40 files	334 files

Each Datablock is required to obtain a metadata file that presents the Datablock information to the Namenode. In Dataset 1, there will be 40 metadata files that must head to the Namenode memory.

Dataset 2 presents BigData in small files and each file is 1 MB in size. However, the Datablock cannot hold more than one file inside, and thus, the total required amount of Datablocks in 334. The small files calculation is as follows:

$$\text{Total No of Datablocks} * \text{default Datablock size} : 334 * 64 = 21.376\,\text{GB}$$

Hence, nearly 21 GB is needed to host only 2 GB dataset. However, the problem is not with the wasted memory, but with the generated metadata files. In this situation, 334 Datablocks will generate 334 metadata files to be sent to the Namenode memory. The metadata file volume is 128 Byte, and thus, 334 metadata files will generate 42.752 KB of a metadata. As such, a 2 GB dataset will generate almost 0.4 MB of metadata. Hence, a 1 PB (petabyte) dataset will generate almost 419 GB of metadata that will harm the Namenode memory and force it to shut down and become unavailable [14].

This huge number of metadata must be involved in another calculation in order to achieve data integrity and availability. Accordingly, Figs. 1 and 2 show that every single block must be repeated into two more blocks to enable the application of Hadoop data integrity.

Figure 1 presents the best practice scenario for Hadoop in treating BigData in big files. The used file size is 192 MB which will be split into two Datablocks, followed by the application of the replication manner. Then, the metadata files containing Datablock information are sent to the Namenode. The total metadata files size is 1050 Bytes.

In Fig. 2, the same dataset contains a group of small files, and the size of each file is 1 MB only. However, each file will have a place in a Datablock. Hence, a 192 MB of files will become 192 Datablocks, and each Datablock has to reply 2 more times for data integrity purposes, and finally, a metadata file about each Datablock information is sent to the Namenode. In Table 1, calculation is made, and the 419 GB of metadata will become 1.25 TB of metadata in terms of the replication manner [16]. Normally, the Namenode reading memory must absorb all of this incremental number of metadata, but later on, when the memory is close to being full, the Namenode will face some hardship in completing the processing job. Finally, when the metadata is greater than or equal to the Namenode memory, this node will be harmed and will totally collapse. As the Namenode is a single point of failure; once it collapses, the whole cluster will be unavailable [17].

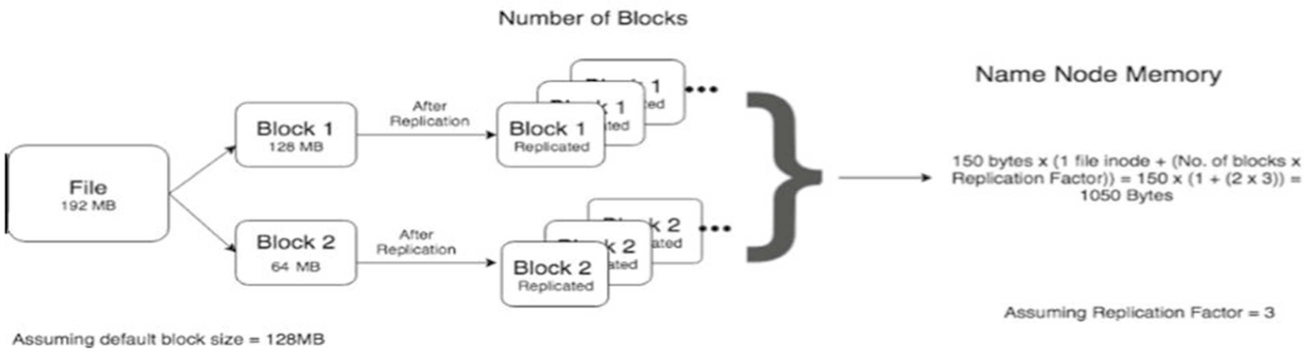
Number of Blocks
File
192 MB
Block 1
128 MB
Block 2
64 MB
After Replication
After Replication
Block 1
Replicated
Block 2
Replicated
Name Node Memory
150 bytes x (1 file inode + (No. of blocks x Replication Factor)) = 150 x (1 + (2 x 3)) = 1050 Bytes
Assuming Replication Factor = 3
Assuming default block size = 128MB

Fig. 1 First scenario with BigData in big files [15]

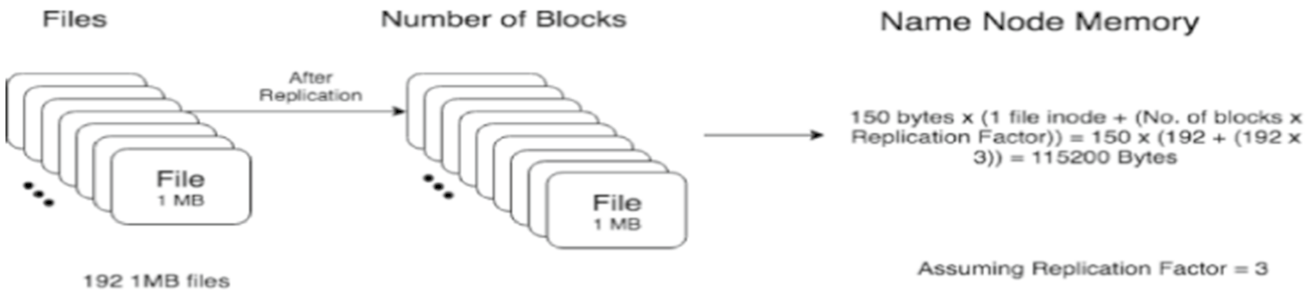
Files
Number of Blocks
Name Node Memory
After Replication
File
1 MB
File
1 MB
150 bytes x (1 file inode + (No. of blocks x Replication Factor)) = 150 x (192 + (192 x 3)) = 115200 Bytes
192 1MB files
Assuming Replication Factor = 3

Fig. 2 Second scenario with BigData in small files [15]

4 Cloud Namenode Topology

This paper focuses on how to transform the Namenode to be an elastic one without customizing the data feeding and processing mechanism on the Datanode. Thus, the Namenode needs to be moved to a cloud service platform with a Haas (Hardware as a service) in order to access it programmatically. Elasticity is the ability to rapidly provide and release resources, and thus, the elastic Namenode must adapt to the upcoming metadata volume in order to keep the Namenode up and running. Figure 3 presents a Hadoop cluster topology that runs a cloud elastic Namenode.

There are two types of elasticity. The first one is horizontal elasticity, and this kind of elasticity is widely used and supported; it refers to the provision and release of nodes to the cluster, and once it is required to add more resources to the network, this elasticity type will be appending more Namenodes in parallel to absorb the extra workload. The second type of elasticity is the vertical elasticity which appends more resources to the same Namenode. However, these resources can be additional RAM, memory, more core processers and so forth [18].

In Hadoop case, and in order to apply the elastic cloud Namenode, it is recommended to adopt the vertical elasticity, owing to the fact that Hadoop rides by a single Namenode, and thus it is not recommended to adopt the horizontal elasticity because it will turn the cluster into HDFS federation case as appeared in Fig. 4 [19]. Specifically, this case relates to a shared Datanodes pool that is linked to a bunch of isolated Namenodes, and each one of these Namenodes controls a part of the Datanodes. Hence, if the first Namenode requests more horizontal resources, each one of the appended nodes will be a Namenode that

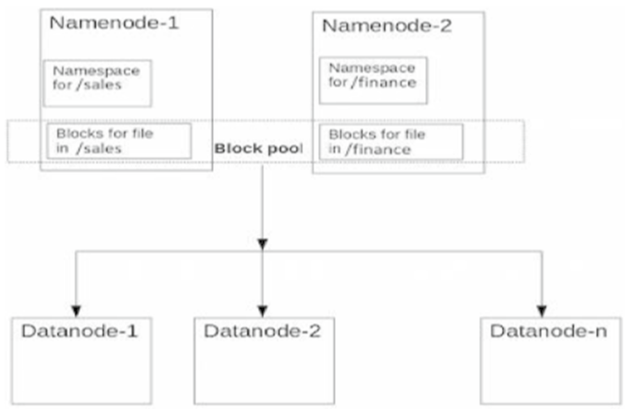

Fig. 4 HDFS federation case [20, 21]

controls a part of Datanodes and is fully isolated from other Namenodes. After the extra nodes are no longer needed, the cluster will shrink again to release these new Namenodes by shutting them down. This step will make a huge part of the cluster to become unavailable [20, 21].

4.1 Applying the Vertical Elasticity

To apply the vertical elasticity Namenode, the primary Namenode must be moved to the cloud service. This is because cloud service can offer unlimited memory resources supported by server-less functions in riding the elasticity mechanisms. Vertical elasticity can be applied on both Cloud platform service and local cluster, but in the case of local clusters (Oracle VM, VM-Box, Docker, etc.), the resources will be limited and need to be upgraded every couple of times. However, it is not that simple to move the Namenode machine to any cloud service platform because some functions are done differently there.

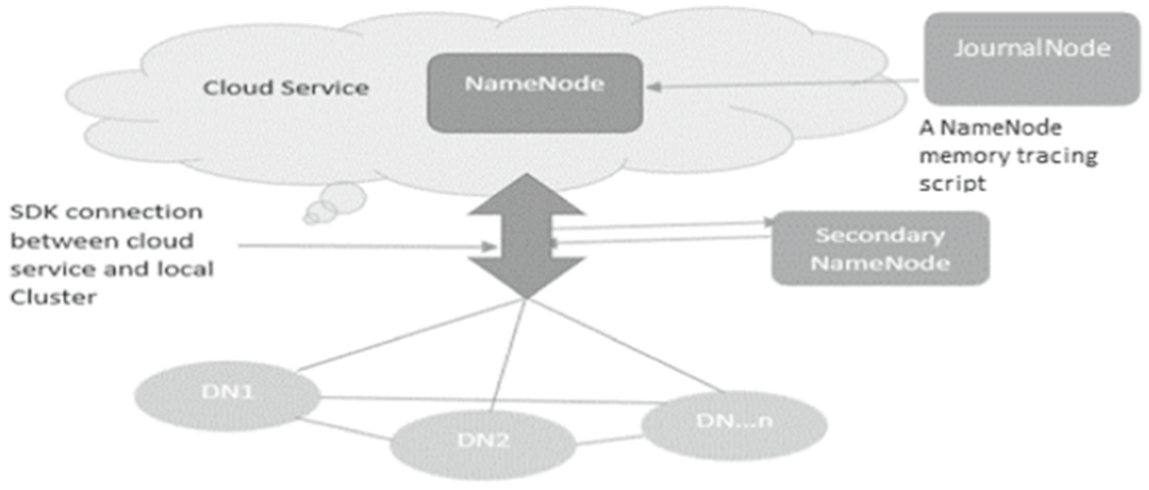
Cloud Service
NameNode
JournalNode
A NameNode memory tracing script
SDK connection between cloud service and local Cluster
Secondary NameNode
DN1
DN2
DN...n

Fig. 3 Cloud elastic Namenode

One of these functions is the capacity planning for the targeted Namenode VM, and based on [22, 23], the memory allocation of the Namenode Virtual Machine (VM) is not only for metadata absorption, but there are some components that must be taken into consideration in Namenode allocation. These components are:

- Host OS usage or reserve: It encompasses the memory for the Virtual Machine Operation System (VM-OS).
- Infrastructure services: These services are utilized for azure stack (31 Virtual Machines).
- Resiliency reserve: This reserve is to keep the Virtual Machine in the save zone in case the memory and resources are required unconditionally.
- Tenant VMs: These are the tenant Virtual Machines created by Azure Stack users.
- Add-on RPs: They encompass Virtual Machines deployed for the Add-on RPs like SQL and MySQL [18].

Based on the contingencies above, the resiliently reserve formula will be:

$$H + R * ((N - 1) * H) + V * (N - 2)$$

where: H = Size of single host memory; N = Size of Scale Unit (number of Hosts-Machines); R = Operating system reserve/Memory used by the Host OS, which is 0.15 in this formula; V = Largest VM in the scale unit.

After setting up the cloud Virtual elastic Machine for the Namenode, the cloud service platform offers several choices for contacting with the local Datanodes as a one geographical cluster, such as site-to-site connection and the static IP address. For the first creation of the Namenode VM, the IP address must become static so that if the service somehow shuts down, it will run again with the same static IP. Applying the elasticity mechanism can be done by using the packages offered by the cloud service platform. In Microsoft Azure, there are packages presented in NuGet manager in visual studio that can ride the elastic memory programmatically. One of them is “Microsoft.Azure.Management.Fluent,” and this package is used to create and maintain the Azure VM remotely. This package is able to keep an eye on the current Namenode resources status in order to increase it when needed. However, Azure is not the only choice in the application of Elastic Namenode as there are various NuGet libraries offered by Microsoft that can perform similar task into AWS (Amazon Web Service). Appositely, the package above is used to write the hard code lines for the following steps:

- The hard code is installed in another cloud machine called JournalNode.
- This Journal node is connected to the Cloud Namenode.
- The JournalNode will run a process that keeps sending a signal to the Namenode memory to check if the memory close to full or not. The signal returns are as follows:
- 1 for the full memory
- 0 for the normal case
- 2 for the useless memory
- The signal will be sent in every time unit of 10 s.

Figure 5 presents the vertical elasticity Namenode. As can be observed, the green area (A) is the current allocated memory. Meanwhile, the hard code script that is already written and uploaded to the JournalNode is listening to the memory using a frequent signal and consistently returns one of these values of 0, 1 and 2. Once the signal returns a 1, the JournalNode will run another process to expand the memory of the Namenode. This process is called AM. However, the JournalNode is free to expand the Namenode memory depending on the upcoming signal value. Relevantly, process A is run using a proportion calculation as follows:

$$\mathrm{UM} = \mathrm{CM} * 80\% \tag{1}$$

$$\mathrm{If}(\ \mathrm{UM} + \mathrm{CM} <= \mathrm{Full\,Memory}) \tag{2}$$

$$\mathrm{CM} = \mathrm{CM} + (\mathrm{CM} * 20\%) \tag{3}$$

In function 1, UM is the used memory and CM is the current one. Thus, once the PCM touched 80% of the current Namenode memory, the JournalNode will assign more memory to the node, and process A will work again depending on the new memory volume. Meanwhile, function 2 makes a comparison between both UM and CM and determines if the total is greater than or equal to the full memory. On the other hand, function 3 shows the new memory volume that will be appended to the current Namenode memory.

Point B presents the limit of the free memory. In this regard, if the JournalNode assigns all the available memories between A and B to the Namenode, then, there will be

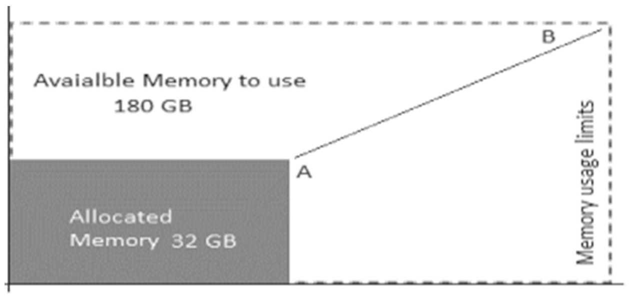

Fig. 5 Suggested vertical elasticity

another process namely process PM that will replace process AM. Both processes, namely AM and PM, differ in a sense that the former appends memory that is already purchased and free for use, while the latter will purchase more memory and push it to be in charge. Hence, once this new memory becomes useless, process PM can release it and return to the maximum limit which is point B. The goal of making both processes (AM and PM) work in a reversible manner is to free the memory for the purpose of reducing the annual cost. The reversible manner will work using these functions:

$$\mathrm{WM} = \mathrm{CM} * 80\% \tag{4}$$

$$\mathrm{If}(\mathrm{WM} + \mathrm{CM} < \mathrm{Full\,Memory}) \tag{5}$$

$$\mathrm{CM} = \mathrm{CM} - (\mathrm{CM} * 20\%) \tag{6}$$

In function 4, WM is the wasted memory, and it is the result of 80% of the current memory. Function 5 determines if both of WM and CM are less than the registered total memory (or not). Function 6 will run if Function 5 is true and will cut off 20% of CM.

5 Experiments

To apply the experiments on the proposed method above, comparison has to be made between the method and one of the most popular methods in big data in small files handling methods, namely nHAR. nHAR refers to new Hadoop archiving which is the second generation of HAR. Both HAR and nHAR compress the small files into one archive file called HAR. This new archive with one reference metadata file heading to the Namenode can contact the small files inside each archive file by using a private index of each desired file. In nHAR, the archiving begins by sending some HDFS commands to the Datanode; these commands include the archiving command and the desired files to be archived. Table 2 summarizes the difference between nHAR and the proposed method in this study.

Based on Table 2, each one of the techniques has a different area to run and handling big data in small files. However, the comparison above shows that:

- The traditional nHAR is a solid behavior, and a human intervention is required once the Namenode requests some extra memory resources. On the other hand, the proposed method is fully auto-elastic with the use of a server-less functions that are riding the elasticity method without human intervention.
- nHAR runs beyond HDFS function in order to read/create the archive files. The proposed method doesn't require a HDFS commands to apply elasticity.

Table 2 A Comparison between nHAR and the proposed method in handling BigData in small files

Key	nHAR	Proposed method
Target	Datanodes	Namenode environment
Namenode data absorbing	Limited	Unlimited
Cost	Free	Pre-paid
Cloud required	No	Yes
HDFS command	Yes	No
Built in functions	No	Yes (Server-less elasticity)
Elasticity	Solid	Vertical elastic
Automatic	No (Human intervention)	Yes

- The proposed method targets the Namenode running environment which requires no extra customization in its application.
- The proposed method requires a special setup to host the Namenode on the cloud and then connect it with the local Datanodes via cloud VPN service or site-to-site platform, which also requires a cloud pre-paid service for the Namenode host machine, site-to-site platform and server-less functions. On the other hand, nHAR requires only the traditional Hadoop setup.
- The proposed method has an unlimited memory resources and is able to apply nHAR within it. However, the unlimited memory is applicable based on the pre-paid cloud service package. nHAR treats a solid-state Namenode memory resources. Thus, if the nHAR files filled the Namenode memory, then, the cluster needed to be shut down to upgrade the Namenode resources, and jump it up again.

Figures 6, 7 and 8 accordingly show the three scenarios of big data in small files data feeding. However, Fig. 6 shows a Hadoop cluster that is natively running without a big data in small files handling tool and with a 120 GB Namenode Memory. The cluster will shut down after a while. Regarding big data in small files issue, the Namenode will collapse and become unavailable due to the huge number of metadata that must be saved in the Namenode memory.

Figure 7 runs the same cluster data feeding but with nHAR. nHAR will group the small files into archives. Thus, every archive will have the same Datablock size which is 64 MB, so, nHAR will delay the collapsing point. Hence, the current Datablocks number is fair enough to absorb the small files, but when it is required to add more Datanodes to host the incremental small files on the same Namenode resources, the cluster will collapse again and become unavailable.

Figure 8 shows a native Hadoop cluster that is running the proposed solution. The elastic Namenode memory

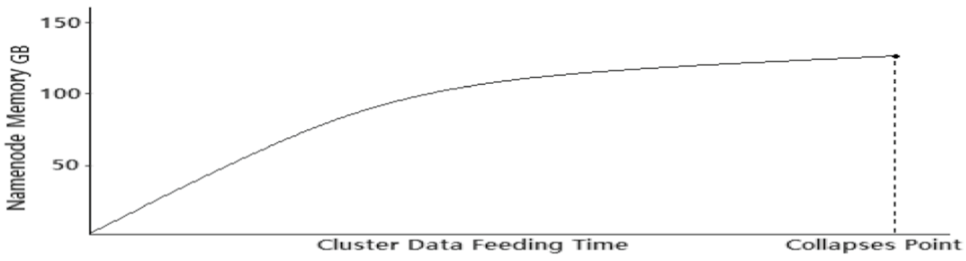

Fig. 6 Native Hadoop cluster collapses

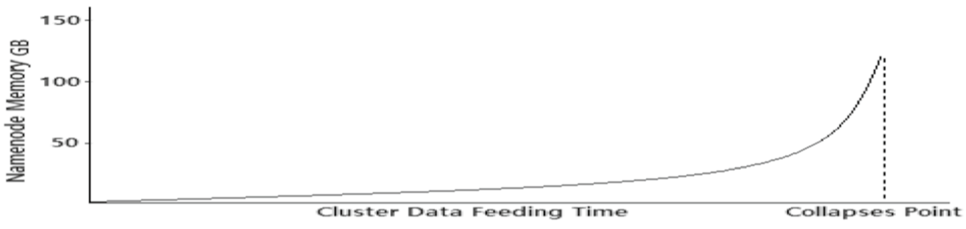

Fig. 7 Hadoop cluster with nHAR

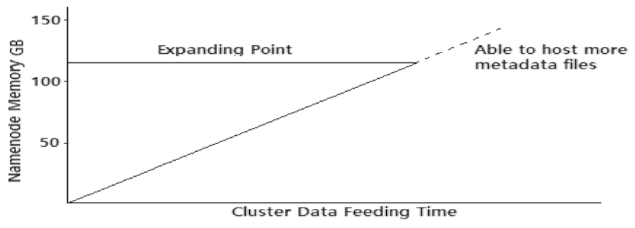

Fig. 8 Applying the elastic Namenode

resources will keep expanding and shrinking, depending on the upcoming small data on the local Datanodes.

There are no collapses points in the proposed solution. Rather, it is just an elastic resource. Namenode gains more resources when required and releases them once the resources become useless. The expanding point is where the Namenode starts gaining more memory resources. There could be a multi-expanding point depending on the purchased cloud memory plans. The expanding point are similar to the shrinking point that the elasticity methods will release to save cost.

6 Results

The obtained result in this study is a running elastic Hadoop cluster that can perfectly address the negative impact of BigData in small files. Accordingly, Fig. 9 presents two scenarios: the first one (a) is the vertical elastic HDFS, and the second one (b) is the standard one. Scenario (b) shows four shortage points (A, B, C and D) that happen because the node is not tracing the right changes of the incoming data. On the other hand, scenario (a) shows that the node is tracing the data volume changes perfectly without any shortage point. Moreover, in scenario (a), the elasticity will be applied between the elastic provisioning and the over provisioning. While the elasticity is accruing in between them, there is no need to append more vertical resources (RAM Memory) to the Namenode. However, once the elasticity diagram touches the over provisioning line, the suggested elasticity function in Fig. 5 will append more vertical resources to the Namenode to absorb the incremental need of metadata hosting.

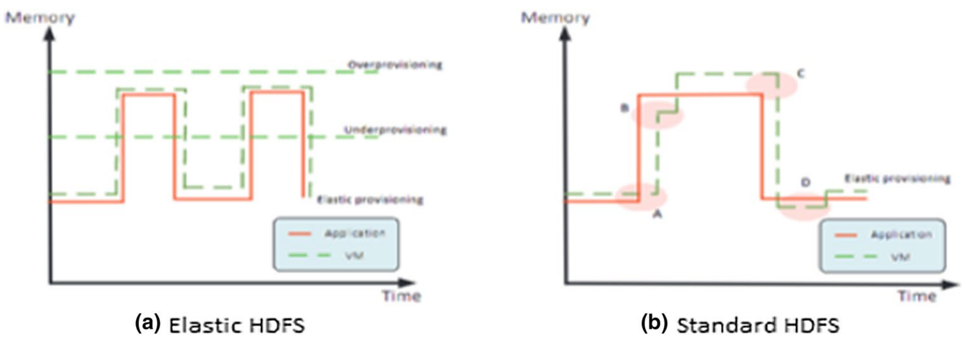

Fig. 9 Vertical elasticity

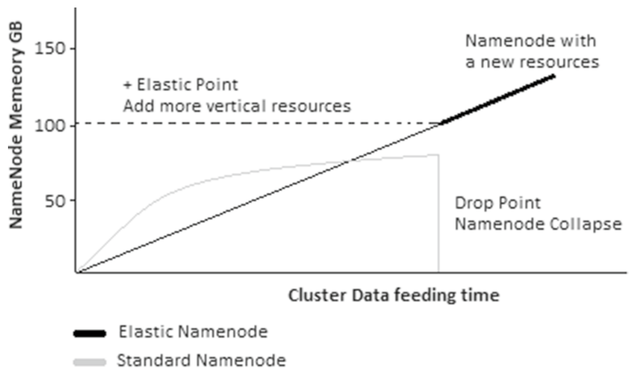

Fig. 10 Two HDFS clusters running in parallel, the gray line is referring to HDFS running standard Namenode solution and the black one is running with the suggested elastic Namenode

Each time the Namenode is used to gain more vertical resources, the over provisioning limit will be expanding to match the newly gained memory, and once again, if the Namenode released a redundant memory, the over provisioning limit will be shrinking to match the previous status.

The elasticity can work in a reversible manner, whereby once the elasticity functions in the JournalNode found out that the Namenode is running with a redundant vertical resource (the over provisioning line edge is not in use anymore), the Namenode vertical resources will shrink and release the redundant vertical resources and save more cost. In Fig. 10, a dataset will be run with two scenarios, whereby the first scenario is with the standard HDFS applying nHAR and the second scenario is with the improved elastic Namenode.

As shown in Fig. 10, two HDFS clusters are running in parallel. The gray line refers to HDFS running standard Namenode solution and the black one runs with the suggested elastic Namenode. In Fig. 10, the same dataset runs on two different HDFS clusters. The dataset size is the same, the Namenode vertical resources are the same, and the only difference is that the first HDFS cluster runs with a standard Namenode while the second one applies the elastic Namenode. The standard Namenode collapses once the in-use memory is totally booked and there is no more memory for the upcoming metadata. However, on the same collapse point, the second elastic Namenode is used to apply the elastic methods and functions to add more vertical resources and keep it going to accept more metadata.

7 Conclusion

This paper discussed the role of vertical elasticity on the Hadoop Namenode performance. The vertical elasticity appends more vertical resources (Memory and a possible CPU) capacity to the allocated resources that the Namenode already has. The elasticity manner already applied on the Namenode vertical resources must be engaged with a cloud-based service that offers unlimited vertical resources plans in order to append them to the current working Namenode once required. On the other hand, vertical elasticity can shrink the Namenode memory once the allocated memory is redundant.

The target treats the negative impact of big data in small files datasets. However, whatever the BDSF dataset volume could be, the vertical resources elasticity can treat any vertical resources shortage in order to keep the Namenode working efficiently and prevent any collapsing chance, while all the datanodes are still running locally in order of data privacy. Thus, this technique can adapt the Namenode perfectly in absorbing the huge number of metadata and reducing the chances of unconditional Namenode shutting down.

The new term is “Local cloud” and it can be a future work, involving the datanodes run on a local cloud. Local cloud is a cloud service that runs inside the country borders and provides its services to the same country. Thus, by moving the datanodes from on-premise demand to a local cloud while keeping the Namenode running on an international cloud, this new structure will guarantee a data privacy with an excellent BDSF treating on the top of the Namenode and also a secure and stable connection between the Namenode and the datanodes.

Acknowledgements The work is funded by grant number 12-INF2970-10 from the National Science, Technology and Innovation Plan (MAARIFAH), the King Abdul-Aziz City for Science and Technology (KACST), Kingdom of Saudi Arabia. We thank the Science and Technology Unit at Umm Al-Qura University for their continued logistics support.

References

1. Prabhu, Y.: Transformation of Hadoop: a survey. Int. J. Sci. Technol. Eng. **4**(8), 97–101 (2018)
2. Koh, S.; et al.: Exploring fault-tolerant erasure codes for scalable all-flash array clusters. IEEE Trans. Parallel Distrib. Syst. **30**(6), 1312–1330 (2019)
3. Mohan, L.J.; Caneleo, P.I.S.; Parampalli, U.; Harwood, A.: Geo-aware erasure coding for high-performance erasure-coded storage clusters. Ann. Telecommun. **73**(1–2), 139–152 (2018)
4. Singh, A.; Choudhary, S.; Kumari, M.: HADOOP ecosystem analytics and big data for advanced computing platforms. Int. J. Adv. Sci. Technol. **29**(5), 6633–6642 (2020)
5. Chandrasekar, S.; Dakshinamurthy, R.; Seshakumar, P.G.; Prabavathy, B.; Babu, C.: A novel indexing scheme for efficient handling of small files in hadoop distributed file system. In: 2013 International Conference on Computer Communication and Informatics. IEEE (2013)
6. Ularu, E.G.; Puican, F.C.; Apostu, A.; Velicanu, M.; Student, P.: Perspectives on big data and big data analytics. Database Syst. J. **3**(4), 3–14 (2012)
7. Korat, V.G.; Pamu, K.S.: Reduction of data at namenode in HDFS using harballing technique. Int. J. Adv. Res. Comput. Eng. Technol. **1**(4), 635–642 (2012)

8. Deshpande, P.P.: Hadoop distributed filesystem: metadata management. Int. Res. J. Eng. Technol. **10**, 2395–2456 (2017)
9. Demir, I.; Sayar, A.: Hadoop optimization for massive image processing: case study face detection. Int. J. Comput. Commun. Control **9**(6) 664–671 (2014)
10. Mrozek, D.; Daniłowicz, P.; Małysiak-Mrozek, B.: HDInsight-4PSi: boosting performance of 3D protein structure similarity searching with HDInsight clusters in Microsoft Azure cloud. Inf. Sci. (Ny) **349–350**, 77–101 (2016)
11. Guan, Y.; Ma, Z.; Li, L.: HDFS optimization strategy based on hierarchical storage of hot and cold data. Procedia CIRP **83**, 415–418 (2019)
12. Approach, A.N.; Undestand, T.; Files, S.; In, P.: A review on small files in Hadoop. (5), 6585–6588 (2017)
13. Chethan, R.; Chandan, K.; Jayanth, K.: A selective approach for storing small files in respective blocks of Hadoop. Int. J. Adv. Netw. Appl. (IJANA) 461–465 (2010)
14. Ahad, M.A.; Biswas, R.: Dynamic merging based small file storage (DM-SFS) architecture for efficiently storing small size files in Hadoop. Procedia Comput. Sci. **132**, 1626–1635 (2018)
15. Naik, S.; Gummalla, B.: Small files, big foils: addressing the associated metadata and application challenges (2019). https://blog.cloudera.com/small-files-big-foils-addressing-the-associated-metadata-and-application-challenges
16. Mohanty, A.; Ranjana, P.; Subramanian, D.V.: Small files consolidation technique in Hadoop cluster. Int. J. Simul. Syst. Sci. Technol. **19**(6), 311–315 (2018)
17. Xiong, A.; Ma, J.: HDFS distributed metadata management research. In: Proceedings of the 2015 International conference on Applied Science and Engineering Innovation. Advances in Engineering Research, pp. 956–961, Atlantis Press (2015). https://doi.org/10.2991/asei-15.2015.190
18. Moltó, G.; Caballer, M.; Romero, E.; De Alfonso, C.: Elastic memory management of virtualized infrastructures for applications with dynamic memory requirements. Procedia Comput. Sci. **18**, 159–168 (2013)
19. Aishwarya, K.; Arvind Ram, A.; Sreevatson, M. C.; Babu, C.; Prabavathy, B.: Efficient prefetching technique for storage of heterogeneous small files in Hadoop distributed file system federation. In: 2013 Fifth International Conference on Advanced Computing (ICoAC). IEEE (2014)
20. Anshudeep.: HDFS federation in Hadoop framework. (2018). [Online]. Available: https://netjs.blogspot.com/2018/02/hdfs-federation-in-hadoop-framework.html.
21. Apach.: HDFS Federation. (2019). [Online]. Available: https://hadoop.apache.org/docs/stable/hadoop-project-dist/hadoop-hdfs/Federation.html.
22. Venkatraman, K.: Virtual machine memory allocation and placement on Azure Stack. *Microsoft Azure*, (2019). [Online]. Available: https://azure.microsoft.com/en-in/blog/virtual-machine-memory-allocation-and-placement-on-azure-stack/.
23. He, S.; Guo, L.; Guo, Y.: Real time elastic cloud management for limited resources. In: *Proc. - 2011 IEEE 4th Int. Conf. Cloud Comput. CLOUD 2011*, 622–629 (2011)