The Node Status as a Prioritization Strategy for Replica Balancing in a HDFS Cluster

Rhauani Weber Aita Fazul[1], Patrícia Pitthan Barcelos[1]
[1]*Post Graduate Program in Computer Science*
Federal University of Santa Maria (UFSM)
Santa Maria – RS, Brazil
{rwfazul, pitthan}@inf.ufsm.br

***Abstract*—Data replication is the main fault tolerance mechanism of HDFS, the Hadoop Distributed File System. Although replication is essential to ensure high availability and reliability, the replicas might not always be placed evenly among the nodes. The HDFS Balancer is an integrated solution of Apache Hadoop that performs replica balancing through the rearrangement of the data blocks stored in the file system. The Balancer, however, demands a high computational effort of the nodes during its operation. This work presents a customization for the HDFS Balancer that considers the status of the nodes as a strategy to minimize the overhead caused by the balancing operation in the cluster. To this end, metrics obtained at runtime are used as a way to prioritize the nodes during data redistribution, making it occurs primarily between nodes with low communication traffic. Also, the Balancer starts to operate aiming at a minimum balance level, reducing the number of data transfers required to even up the data stored in the cluster. The evaluation results showed that the proposed customization allows reducing the time and bandwidth needed to reach the system balance.**

***Index Terms*—data replication, replica balancing, balancing overhead, data locality**

I. Introduction

Apache Hadoop [1] is an open-source framework designed for large-scale, massively parallel, and distributed data processing and storage. One of its main components is the Hadoop Distributed File System (HDFS), a system designed to provide reliable storage even when running on clusters of commodity hardware. The HDFS follows a master-worker architecture composed by a NameNode (NN) and multiple DataNodes (DNs). The NN is the master server responsible for maintaining the system namespace and controlling the access and distribution of the files across the cluster. The DNs, on the other hand, are the workers that store and recover the data.

When a file is inserted in the HDFS, it is split into data blocks of fixed size (128MB by default). To ensure reliability and availability, the HDFS implements data replication as a fault tolerance mechanism. In this way, the blocks are replicated and distributed between the DNs. The number of replicas of each block is defined by the Replication Factor, whose default value is 3. It can be configurable for each file either by the user or the application [2]. A factor of n ensures that data will not be lost even if $n-1$ DNs fail simultaneously.

Besides providing reliability through redundancy, the replication promotes performance improvements as it allows the applications to explore a higher data availability in the cluster. In this sense, a good replica placement is essential to keep the file system working properly. The NN takes all the decisions regarding the data placement and determines which DNs will store the replicas of each block. To do that, the NN follows a *Replica Placement Policy* (RPP) [1] that optimizes the data distribution by using rack awareness. The RPP assures that a node can have only one replica of the same block, and in a rack, a block can be present in a maximum of two nodes [3]. This provides high availability in the event of failures and prevents losing data even when an entire rack fails.

In general, the RPP provides fault tolerance and good performance on reading and writing data by optimizing the network bandwidth utilization [2]. However, it does not ensure a fully-balanced replica distribution [4]. Imbalance also occurs in situations like the addition of new DNs in the cluster as it results in a large discrepancy between the data held on the DNs [5]. The replica imbalance impacts the data locality and can prevent the HDFS to explore the computational resources in an optimized way, degrading its overall performance.

One way to even out the data spread across the cluster is through the HDFS *Balancer* [4], a Hadoop daemon that redistributes replicas from over-utilized to under-utilized DNs. However, it operates in a generalized way to suit different applications running in a variety of contexts, disregarding parameters, such as the communication traffic of the DNs and the volume of data to be transferred. Thus, the *Balancer* may not be optimized to meet some specific usage demands that can be hardly affected by the balancing operation in the cluster.

This work presents a prioritization strategy for the HDFS *Balancer* that aims to reduce the overhead of the balancing operation through metrics recovered in runtime. Based on that, the data redistribution starts to occur primarily between nodes with low communication traffic, minimizing the impact in the other tasks executing in the cluster during the balancing operation. Additionally, an effort is made to reduce the number of data transfers required to balance the data distribution on the cluster. An experimental investigation was lead in order to validate and evaluate the effectiveness of the implementation.

The work is organized as it follows. Section II presents the main causes and problems inherent by the replica imbalance in HDFS clusters. Section III gives an overview of the related work focused on replica balancing. Section IV describes the proposed solution in details. Section V presents and discusses

the evaluation results. Lastly, Section VI concludes the paper and summarizes possible future work.

II. Replica Balancing

A principle of Hadoop's operation is to move the computational tasks to where the data are stored and, if it is not possible, to nodes with a faster network path to the DNs that maintain the blocks required for the operation. Bringing computing closer to data, the heart of Hadoop processing, is known as *data locality* [2]. This feature increases the performance of the platform in processing large datasets as the block access – since it is local – becomes faster and cheaper in terms of network bandwidth utilization.

As each block is replicated by default in three different DNs, the probability that a computational task is able to process most blocks locally is high [3]. Therefore, the placement of the replicas is critical for data availability and the performance of *I/O bound* applications running on HDFS [4]. Hadoop works hard managing block placement in a way that maximizes both reliability and performance [5]. However, it is not always possible to prevent an HDFS cluster from becoming imbalanced.

Different aspects can contribute to the replica imbalance, such as: (i) the RPP that does not take into account DN storage space utilization [4], thus contributing to inter-DN imbalance, and selects a rack to keep two-thirds of the replicas of a certain block, resulting in inter-rack imbalance; (ii) the client application behavior which, if executed directly in one DN of the cluster and according to the RPP, always stores one of the replicas locally; (iii) the re-replication process that is also under the same initial replication policy; and (iv) the addition of a new DN into the system, as it will be a valid candidate for block placement alongside all the other nodes in the cluster, therefore remaining lightly utilized for a significant period [5].

An imbalanced replica distribution tends to affect the data locality and may result in a higher number of intra-rack or even off-rack transfers, thus consuming valuable cluster bandwidth [6]. Also, the imbalance can cause an overhead in the highly utilized DNs (nodes with more stored blocks), thus leading to performance bottlenecks in the system.

III. Related Work

Two main approaches can be used to mitigate the problems of replica imbalance. The first one is through preventive actions made in the moment of the initial block distribution. With that, it is possible to reduce the chances that the cluster becomes imbalanced. In general, examples of this approach, like the solution presented in [7], involve the development of new replica placement policies for the HDFS that consider the data volume stored in each node as criteria for data distribution. However, in some situations, like when new nodes are added to the system, it is not possible to prevent a high discrepancy between the data held on the cluster nodes, in a way that reactive approaches become necessary. This work is focused on solutions from this latter approach.

The reactive balancing in HDFS acts as a corrective approach that allows the cluster administrator to even out the replica placement between the nodes. In this sense, the blocks already stored in the file system are redistributed across the cluster, aiming at a controlled balancing level. Examples of reactive approaches include [8] that proposes an enhanced algorithm to balance racks based on priority. The algorithm acts mainly on balancing over-utilized racks, minimizing the chances of rack failures caused by overheads, and contributing with more uniform data distribution.

In [9] it is showed a balancing strategy that besides DNs utilization considers variations in the write and read latency of the nodes storage disks to reallocate the data in HDFS. With that, DNs with less disk latency tend to receive a high number of blocks. The authors in [10], in turn, focus on optimizing the redistribution process by exploring the computational power of the nodes. The proposed balancing algorithm is based on the processing capacity of the DNs, where the blocks are redistributed only to specific DNs, selected from an initial classification by their heterogeneity and performance.

Another possible solution for reactive balancing, integrated into Hadoop distribution as a utility, is the HDFS *Balancer* [4]. Since this solution is the basis for the implementation presented in this work, the balancer operation is detailed next.

A. HDFS Balancer

The HDFS *Balancer* [4] is a Hadoop daemon designed for replica balancing across storage devices in HDFS. Following its default operation policy, the HDFS *Balancer* moves data blocks from DNs with high utilization to DNs that store a smaller volume of data. The tool is executed on demand by the cluster administrator.

The *Balancer* is driven by a *threshold* (a percentage in the range of 0% to 100%), which is passed as parameter to its execution and it is aware that a single DN may have multiple storage devices of different types. So, considering that $G_{i,t}$ represents the group of storage devices of type t that belongs to DN i, the *threshold* will limit the maximum difference between the utilization of a certain $G_{i,t}$ ($U_{i,t}$), i.e., the ratio of the used space on the node to its total storage capacity, and the average utilization of the cluster ($U_{\mu,t}$), i.e., the ratio of the used space on the cluster to its total capacity [2]. When the utilization of each group is in accordance with the *threshold*, which has the default value of 10%, the cluster is considered to be balanced. Reducing the *threshold* increases the balance of the cluster but requires more effort in terms of data processing and transfer.

The *Balancer* operation can be divided in different stages that are executed iteratively [11]. Initially, each $G_{i,t}$ of the cluster is classified in one of the following categories: (i) *over-utilized*, if $U_{i,t} > U_{\mu,t} + threshold$; (ii) *above-average*, if $U_{\mu,t} + threshold \geq U_{i,t} > U_{\mu,t}$; (iii) *below-average*, if $U_{\mu,t} \geq U_{i,t} \geq U_{\mu,t} - threshold$; (iv) *under-utilized*, if $U_{\mu,t} - threshold > U_{i,t}$. Besides, for each $G_{i,t}$, the data amount (in bytes) needed for taking its $U_{i,t}$ until the $U_{\mu,t}$ is calculated and stored in a variable called *maxSize2Move*.

After, each over-utilized $G_{i,t}$ (source) is paired with one or more under-utilized $G_{i,t}$ (target) in a 1 – n relation (the lists are accessed sequentially, without any predefined order).

The over-utilized groups that have *maxSize2Move* satisfied are removed from their list and will not be paired in the current iteration. For the remaining over-utilized groups, pairs are formed between the $G_{i,t}$ classified as below-average. If there is still some under-utilized $G_{i,t}$, candidates are searched between the remaining $G_{i,t}$ classified as above-average.

In sequence, the replicas that will be moved from the source to the target group are elected. The *Balancer* maintains the same data availability level initially provided by the RPP. So, when selecting a replica to move and determining the target $G_{i,t}$, the *Balancer* assures that this decision will not decrease either the number of replicas or the number of racks [4].

Additionally, to decrease the inter-rack data copying required for the transfers between nodes of different racks, the *Balancer* uses the DN nearest to the target and that has a copy of the block to be moved as a proxy. Thus, after copying the block kept in the proxy to its local storage, the target group sends an alert to the NN that triggers a delete operation of the replica in the source group. If after the conclusion of all block movements the cluster still has over-utilized or under-utilized groups, a new balancing iteration will be started.

Although the *Balancer* was designed to operate in the background without affecting the other clients and applications in the cluster [2], the balancing operation may demand high cost in both processing and bandwidth consumption. Section IV presents a customization for the HDFS *Balancer* that turns the tool aware of the status of the cluster nodes, avoiding inappropriate overheads.

IV. Customized Replica Balancing Policy

Previous work attested that replicas imbalance directly affect the HDFS performance in serving I/O *bound* applications [12]. Motivated by the achieved results, we defined a customized policy for the HDFS *Balancer* dedicated to optimizing the default operation policy of the Hadoop's native balancer.

The customized policy defines a priority system based on the cluster topology and in the metrics of the HDFS. The priorities are grouped into four categories according to their behavior, as shown in Table I. All priorities ensure that the replica placement after the balance will continue to respect the RPP and that the maximum variation of the data volume stored in each DN remains controlled by the *threshold* value. This work focuses on the priorities of the **Node status** category, which are presented in Section IV-A.

TABLE I
Priorities of the customized replica balancing policy.

Category	Priority
Node capacity	Processing capacity, Storage capacity
Node status	**Node utilization**, **Node classification**, **Node load**
Rack status	Rack reliability, Rack utilization
Data distribution	Data availability

A. Node Status Category

The priorities of the Node status category customize the replica balancing policy of the HDFS through differences in metrics obtained in the *Balancer*'s runtime. In the following sections, the Node utilization, classification, and load priorities are described in detail.

*1) **Node utilization priority**:* This priority reorders the lists of the storage device groups considering the amount of data stored in each group. This increases the chances that the $G_{i,t}$ with the highest utilization in the cluster will be primarily paired with the $G_{i,t}$ with the lowest utilization and so on. As the definition of the source-target pairs follows a $1 - n$ relation, it tends to reduce the value of n, optimizing the matching of the source and target groups during the balancing process.

To this end, it is necessary to order the lists that keep the storage groups before the pairing stage starts. Thus, we implemented a new method called *orderStorageGroups*, which sorts the lists according to the used or the available space of the groups. If the used space (attribute *dfsUsed*) is the configured metric, the lists of the source groups (over-utilized and above-average) are sorted in descending order and the lists of the target groups (below-average and under-utilized) are sorted in ascending order. On the other hand, if the metric is the available space (attribute *remaining*), the lists of the source groups are sorted in ascending order and the lists of target groups in descending order.

This priority can be used simultaneously alongside other priorities of the customized policy. The algorithm of this priority was omitted due to its simplicity (i.e., numeric list sorting). It is worth mentioning, however, that the group lists are originally kept in Java collections, and thus they do not have a predefined order. So, it becomes necessary to convert the lists to a sortable implementation (e.g., *LinkedList*) before applying the comparisons to order the storage groups.

*2) **Node classification priority**:* This priority aims at a minimum balance in function of the configured *threshold*. Thus, the utilization of the over-utilized and under-utilized groups are taken only until the balancing limits to the cluster be considered balanced. With that, it is possible to reduce the execution time and the number of data transfers required for balancing the replicas stored in the file system.

To allow this behavior, we implement a set of methods for integration into the source code of the HDFS *Balancer*. The first of them, exhibited in Figure 1, is used to calculate a new variable, called *minSize2Move*. The value of *minSize2Move* represents the amount of data required to make the utilization of a given group ($U_{i,t}$) to be in concordance with the balancing *threshold*. For a group classified as above-average or below-average, this value will be zero as its utilization already complies with the *threshold* (line 5). On the other hand, for an over-utilized $G_{i,t}$ the value of *minSize2Move* will represent the data amount (in bytes) required to take the $U_{i,t}$ of the group until the upper balancing limit (line 9), i.e., $U_{\mu,t} + threshold$. Similarly, for an under-utilized $G_{i,t}$, this value will be the volume necessary to take the $U_{i,t}$ of the group to the lower balancing limit (line 12), i.e., $U_{\mu,t} - threshold$.

After defining *minSize2Move* for each $G_{i,t}$ it is possible to determine a new value for the *maxSize2Move* variable. It is worthwhile to mention that, in some cases, new validations

```
1: procedure CALCMINSIZE2MOVE
2:   utilizationDiff ← utilization − average
3:   thresholdDiff ← |utilizationDiff| − threshold
4:   if thresholdDiff ≤ 0 then                    ▷ above or below-average G_{i,t}
5:     minSize2Move ← 0
6:   else
7:     if utilizationDiff > 0 then                ▷ over-utilized G_{i,t}
8:       supLimDiff ← utilization − (average + threshold)
9:       minSize2Move ← |supLimDiff| × capacity / 100
10:    else                                       ▷ under-utilized G_{i,t}
11:      infLimDiff ← utilization − (average − threshold)
12:      minSize2Move ← |infLimDiff| × capacity / 100
13:    end if
14:  end if
15:  return minSize2move
16: end procedure
```

Fig. 1. Method to calculate the auxiliary variable *minSize2Move*.

```
1: procedure COMPUTELOAD
2:   for each r ∈ DataNodeStorageReports do
3:     key ← r.getDatanodeInfo().getDatanodeUuid()
4:     xceiverCount ← r.getDatanodeInfo().getXceiverCount()
5:     xceiverMap.put(key, xceiverCount)
6:   end for
7:   min ← min(xceiverMap.values())
8:   max ← max(xceiverMap.values())
9:   for each r ∈ DataNodeStorageReports do
10:    key ← r.getDatanodeInfo().getDatanodeUuid()
11:    xceiverCount ← xceiverMap.(get(key))
12:    weight ← 0.5
13:    if (max − min) ≠ 0 then
14:      load ← (xceiverCount − min) / (max − min)
15:    end if
16:    loadMap.put(key, load)
17:  end for
18: end procedure
```

Fig. 2. Method used to quantify the load of the storage groups.

become necessary. Suppose that the HDFS has multiples over-utilized $G_{i,t}$ and none under-utilized $G_{i,t}$. If *minSize2Move* is attributed directly to *maxSize2Move*, the balancing operation will be frozen because the target groups (below-average, in this case) will not allow data transfers as the new value of *maxSize2move* of these groups is zero.

To solve this problem, a set of auxiliary methods was created to compensate the value of *maxSize2Move* in order to allow the data amount that exceeds the *threshold* in over-utilized groups (*overLoadedBytes*) to be redistributed between the remaining below-average groups. The same strategy is used to the above-average groups that, when it is necessary, can transfer blocks to compensate the data amount below the lower balancing limit (*underLoadedBytes*). The solution to this adjustment is summarized, in general, as the mapping – in each balancing iteration – of the above and below-average groups with smaller and bigger utilization in the cluster. The *maxSize2Move* variable is then updated to allow the utilization of these groups ($U_{i,t}$) to be either extended or reduced.

*3) **Node load priority**:* This priority considers the number of active connections in the nodes to determine the volume of data to be moved between the storage device groups of the HDFS. In addition to redistributing the minimum amount of blocks required for balancing, the node load priority tends to prevent further impacts on the performance of other tasks running on the cluster during the *Balancer* execution.

This priority is used in association with the other two previously presented priorities. Initially, the lists that kept the groups according to their classifications are sorted with the *orderStorageGroups* method of the Node utilization priority. Then, the load of each $G_{i,t}$ is quantified by a method called *computeLoad*, as showed in Figure 2. The first part of the method (lines 2 to 6) fills the *xceiverMap* structure, that relates each *DatanodeStorageReport* with the value of the attribute *xceiverCount* of the given node. This attribute represents an estimated number of executing *threads* in the target node, which is used by Hadoop to check the communication traffic of the DNs and mark them as *busy* in order to perform some actions regarding load balance issues during data placement.

In the second part of the *computeLoad* method (lines 7 to 17), some calculations are made to fill the *loadMap* structure, which registers for each DN a value that represents its load (L'_i) based on the value of the *xceiverCount* of the node ($L_{i,t}$). This value was established from the *min-max* normalization[1] that turns the minimum value of a set into 0, the maximum value into 1 and any other value in a proportional number between 0 and 1. We get the normalization through the equation $L'_i = (L_{i,t} - min)/(max - min)$.

After computing the normalized value for each $G_{i,t}$ (line 14), it is created an entry in the *loadMap* structure (line 16) that is accessed later in a new method called *calcMaxSize2MoveBasedOnNodeLoad*, exhibited in Figure 3. Firstly, it is calculated the required quantity of bytes to take the $U_{i,t}$ of the group until the $U_{\mu,t}$, this amount is saved in the *bytes2Avg* variable (line 3). Then, the *minSize2Move* variable is defined using the *calcMinSize2Move* method (line 4), already presented as a part of the Node classification priority.

```
1: procedure CALCMAXSIZE2MOVEBASEDONNODELOAD
2:   utilizationDiff ← utilization − average
3:   bytes2Avg ← |utilizationDiff| × capacity / 100
4:   minSize2Move ← calcMinSize2Move(capacity, utilization, average)
5:   key ← r.getDatanodeInfo().getDatanodeUuid()
6:   loadBasedBytes ←
           (bytes2Avg − minSize2Move) × (1 − loadMap.get(key))
7:   maxSize2Move ← minSize2Move + loadBasedBytes
8:   if thresholdDiff < 0 then                    ▷ target G_{i,t}
9:     maxSize2Move ← min(remaining, maxSize2Move)
10:  end if
11:  return min(maxSize2Move, max)                ▷ max = 10GB
12: end procedure
```

Fig. 3. Method to define *maxSize2Move* based on the node load.

The difference between *bytes2Avg* and *minSize2Move* is weighted based on the normalized value stored in the *loadMap* structure (L'_i), resulting in the *loadBasedBytes* variable (line 6). For the over-utilized and under-utilized groups, the dif-

[1]If the DNs do not show differences in the *xceiverCount*, L'_i will have a average value of 0.5 (line 12 of the *computeLoad* method), making the subsequent definition of *maxSize2Move* similar to the value that is already adopted by the standard balancing policy, i.e., $|U_{i,t} - U_{\mu,t}|$.

ference between *bytes2Avg* and *minSize2Move* represents the amount of data (in bytes) required to take the $U_{\mu,t}$ until the upper and lower balancing limits, respectively. For the above-average and below-average groups, *minSize2Move* will be zero, keeping the *bytes2Avg* value unchanged. Notice that to reach the expected value of *maxSize2Move*, it is important to sum *minSize2Move* with *loadBasedBytes* (line 7), ensuring that the $U_{i,t}$ of over-utilized and under-utilized groups will be taken, at least, to the balancing limits determined by the *threshold*. With these changes, the HDFS *Balancer* starts considering the node loads, avoiding busy nodes from being further impacted by the replica balancing operation.

Considering the goals of the node load priority, it may be appropriate to personalize the execution settings of the HDFS *Balancer*. The properties for *Background Mode* in [11] are recommended for clusters serving other jobs and applications at the same time of the *Balancer* execution.

V. Experimentation and Discussion

The experiment presented in this section uses the three priorities defined in the **Node status** category of the customized balancing policy. The tests were performed on the GRID'5000 platform, using Hadoop (version 2.9.2) in a fully-distributed mode with 10 DNs configured on the *ecotype* cluster of the *Nantes* site over a Debian 9.9 distribution. Each configured node had 2 Intel Xeon E5-2630L v4 (10 cores/CPU) processors, 128GB of RAM, 372GB of storage capacity (SSD) and two *Ethernet* connections of 10Gbps each. In total, the HDFS cluster had 3.19TB of storage capacity.

To validate the implementation and evaluate the effectiveness of our customization, we consider the state of HDFS before replica balancing (i.e., data placement entirely based on the default RPP) and after running the HDFS *Balancer* with the priorities. Although the works of [8], [9], and [10] present different balancing solutions for the HDFS, we understand that standard PPR is the most appropriate basis of comparison for this work. Our solution differs from the others precisely because it is based on a customization for the HDFS *Balancer* and it acts as a reactive prioritization strategy that considers the load of the nodes during the process of replica balancing.

The data load was performed by the *TestDFSIO* [2], a benchmark that measures HDFS performance trough intensive I/O operations. *TestDFSIO* was used to write 10 files of 25GB each with a default Replication Factor of 3 replicas per block, resulting in a total of 775.87GB of data ($U_{\mu,SSD}$ in 23.16%).

Table II shows the occupation in GB ($O_{i,SSD}$) and the utilization percentage ($U_{i,SSD}$) of each DN before and after running the HDFS *Balancer* with the Node utilization, classification, and load priorities and a *threshold* of 5%. Initially, DN_{01}, DN_{02}, DN_{03}, DN_{05}, DN_{07}, and DN_{10} were under-utilized, with utilization below 18.16% ($U_{\mu,SSD} - $ *threshold*, i.e., $23.16\% - 5\%$). On the other hand, DN_{04}, DN_{06} , and DN_{08} were over-utilized, with utilization above 28.16% ($U_{\mu,SSD} +$ *threshold*, i.e., $23.16\% + 5\%$).

The L'_i column in Table II exhibits the calculated values obtained by the *min-max* normalization based on the communication traffic of the nodes (determined by the *xceiverCount* variable). Highlighted are the DNs that were considered *busy* by having a high flow of communication in their nodes (L'_i at 1.0). Due to its high communication traffic, the DN_{01} (under-utilized) received only as many blocks as necessary to be classified as below-average by bringing its utilization up to approximately the lower balancing limit ($U_{01,SSD} > U_{\mu,SSD} -$ *threshold*). Similarly, the DN_{06} (over-utilized) only transferred the approximate number of blocks to be classified as above-average by bringing its utilization down to approximately the upper balancing limit ($U_{06,SSD} < U_{\mu,SSD} +$ *threshold*).

TABLE II
Cluster state before and after replica balancing with the proposed customization for the HDFS Balancer.

DN	L'_i	before balancing		after balancing	
		$O_{i,SSD}$ (GB)	$U_{i,SSD}$ (%)	$O_{i,SSD}$ (GB)	$U_{i,SSD}$ (%)
DN_{01}	1.0	57.68	17.68	60.72	18.61
DN_{02}	0.0	53.67	16.45	65.51	20.08
DN_{03}	0.0	58.08	17.80	66.52	20.39
DN_{04}	0.0	128.75	39.46	86.68	26.56
DN_{05}	0.0	58.33	17.88	70.17	21.51
DN_{06}	1.0	115.67	35.45	91.75	28.12
DN_{07}	0.0	55.18	16.91	79.87	24.48
DN_{08}	0.0	97.76	29.96	86.68	26.56
DN_{09}	1.0	75.84	23.24	75.84	23.24
DN_{10}	0.0	54.93	16.83	78.35	24.01

It is important to notice that the DN_{09}, given it is within the *threshold* (node classified as above-average) and has L'_i at 1.0, was not matched in any balancing iteration, i.e., did not send or receive data. It reinforces the effort of the Node load priority to minimize the additional overhead caused by the balancing process on the DNs considered as busy due to the high number of open connections in their nodes.

To investigate possible performance improvements provided by balancing the cluster, we run the *TestDFSIO* in reading mode 15 times before and after running the HDFS *Balancer*. Figure 4 shows the reading times (in seconds) in each run of the benchmark. Before the balancing, the average of the execution times was 965.46s and, after balancing, it was reduced to an average of 826.46s. It is equivalent to a percentage change of -14.4%, which represents a reduction in the time needed to read the data stored in the HDFS after replica balancing.

In addition to reducing read times, the data locality optimization in a balanced cluster can provide other performance optimizations on HDFS, such as increased throughput and I/O rate. The throughput is given by the ratio of the total volume of data processed (in MB) to the sum of the times (in seconds) spent by each task (due to parallelism, this value is greater than the total execution time of the job). The average throughput was 33.1MB/s without balancing and 41.11MB/s after using the *Balancer*. The obtained percentage change was 24.2%, indicating an increase in application throughput. The average data transfer rate (i.e., I/O rate), in turn, is the ratio between the transfer speed obtained by each map task to the total number mappers. With *TestDFSIO*, the default number of mappers that

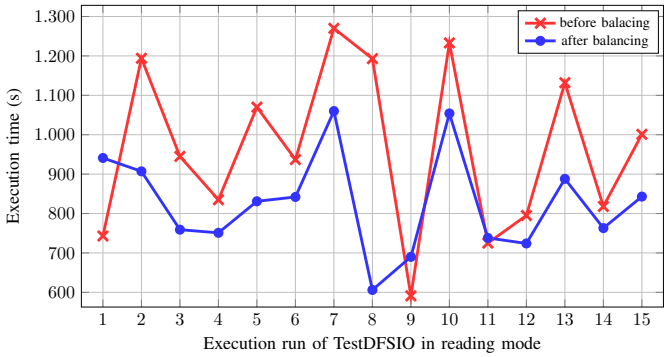

Fig. 4. Read times before and after replica balancing.

will get executed is equivalent to the number of files read by the benchmark (10 in this experiment). The average I/O rate in the 15 runs was 37.24MB/s before balancing and 43.77MB/s after balancing. It shows an increase of 17.53% in the read transfer with replica balancing.

In this experiment, the execution of the HDFS *Balancer* with the proposed customization took about 43 minutes to complete. In total, 5 balancing iterations were performed and 73.5GB of data were moved between the HDFS storage devices. If the *Balancer* was running with their default balancing policy, the data volume to be moved (*bytesLeftToMove*) would be about 231.09GB, which would demand (proportionally) 135 minutes to be relocated and would have taken at least 8 balancing iterations. Thus, the combination of Node utilization, Node classification, and Node load priorities allowed optimizations in the *Balancer* operation by reducing the time and bandwidth required to perform replica balancing on the HDFS cluster.

VI. Conclusion and Future Work

When storing files in the Hadoop Distributed File System (HDFS), they are split into fixed-size data blocks, which are independently replicated and distributed across the cluster. Replication increases reliability and data availability, although the placement of the block replicas between the nodes is critical to system performance. Over time, data might not always be kept uniformly stored. An imbalance in the cluster affects data locality and can put a high strain on the nodes with more stored replicas, degrading HDFS performance.

The HDFS *Balancer* is a tool integrated into the Hadoop distribution designed for replica balancing. The *Balancer* is good at providing an overall balancing in the placement of the replicas by redistributing the data across the cluster. However, its operation does not consider the status of the computational nodes in the cluster, which may be experiencing overhead periods. To make the balancing operation more flexible in different environments that run Hadoop, we propose a customization for the native balancer of HDFS based on balancing priorities.

The strategy presented in this paper aims to reduce the overhead caused for the balancing operation by using metrics retrieved during runtime. In this sense, an effort is made to prevent nodes that are already experiencing overhead periods with high communication traffic (i.e., busy nodes) from being further impacted by the balancing operation, causing the data redistribution to be performed primarily between nodes with low communication traffic. Also, the *Balancer* starts to operate aiming at the minimum balance, moving as little data as possible to make the utilization of each node to be within the balancing *threshold* and so minimizing the number of data transfers required to balance the data distribution across the cluster. The evaluation results showed that our customization allowed us to optimize and make the HDFS *Balancer*'s operation more flexible by reducing the time and bandwidth consumption required to achieve the system balance.

Future work involves analyzing the behavior of the customization presented in this paper on Hadoop instances running over heterogeneous environments and considering the induction of node failures. In addition, we intend to conduct an in-depth experimental investigation involving the use of different stress testing benchmarks for the HDFS, which will allow us to evaluate the effectiveness of the solution in scenarios with nodes experiencing periods of high overhead during replica balancing using the HDFS *Balancer*.

Acknowledgment

Experiments presented in this paper were carried out using the Grid'5000 testbed, supported by a scientific interest group hosted by Inria and including CNRS, RENATER and several Universities as well as other organizations (see https://www.grid5000.fr).

References

[1] Apache Software Foundation. (2020) HDFS Architecture. [Online]. Available: https://hadoop.apache.org/docs/r3.3.0/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. [Accessed: May, 2020].

[2] T. White, *Hadoop: The Definitive Guide*, 4th ed. Sebastopol: O'Reilly Media, Inc., 2015.

[3] S. Achari, *Hadoop Essentials*, 1st ed. Packt Publishing Ltd, 2015.

[4] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. IEEE, 2010, pp. 1–10.

[5] G. Turkington, *Hadoop Beginner's Guide*, 1st ed. Birmingham: Packt Publishing Ltd, 2013.

[6] Z. Guo, G. Fox, and M. Zhou, "Investigation of data locality in mapreduce," in *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (ccgrid 2012)*. Ottawa: IEEE Computer Society, 2012, pp. 419–426.

[7] I. A. Ibrahim, W. Dai, and M. Bassiouni, "Intelligent data placement mechanism for replicas distribution in cloud storage systems," in *IEEE International Conference on Smart Cloud (SmartCloud)*. New York: IEEE, 2016, pp. 134–139.

[8] K. Liu, G. Xu, and J. Yuan, "An improved hadoop data load balancing algorithm," *Journal of Networks*, vol. 8, no. 12, pp. 2816–2822, 2013.

[9] J. Dharanipragada, S. Padala, B. Kammili, and V. Kumar, "Tula: A disk latency aware balancing and block placement strategy for hadoop," in *International Conference on Big Data*. IEEE, 2017, pp. 2853–2858.

[10] A. Shah and M. Padole, "Load balancing through block rearrangement policy for hadoop heterogeneous cluster," in *2018 International Conference on Advances in Computing, Communications and Informatics (ICACCI)*. Bangalore: IEEE, 2018, pp. 230–236.

[11] Hortonworks Data Plataform. (2019) Scaling namespaces and optimizing data storage. [Online]. Available: https://docs.cloudera.com/HDPDocuments/HDP3/HDP-3.1.5/data-storage/content/balancing_data_across_hdfs_cluster.html. [Accessed: June 03, 2020].

[12] R. Fazul, P. V. Cardoso, and P. P. Barcelos, "Improving data availability in hdfs through replica balancing," in *2019 9th Latin-American Symposium on Dependable Computing (LADC)*. IEEE, 2019, pp. 1–6.