A practical analysis of balancing policies for rearranging data replicas in HDFS clusters

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 implemented by the HDFS. The placement of the replicated data across the nodes directly influences replica balancing and data locality, which are essential to ensure high reliability and data availability. The HDFS Balancer is the official solution to perform replica balancing through data redistribution. In this work, we conducted a practical experiment to evaluate different policies for replica rearrangement, namely: datanode, blockpool, and custom. The evaluation results underline the behavior and the effectiveness of each policy. In addition, we investigated the cost of the HDFS Balancer operation and the performance and availability improvements promoted by a balanced replica distribution.*

1. Introduction

Distributed file systems are widely used to support data-intensive applications. One of the widespread adopted file systems is the Hadoop Distributed File System (HDFS) [Foundation 2021]. Besides being the main storage engine of the Apache Hadoop ecosystem, the HDFS is incorporated by several parallel processing frameworks, such as Apache Spark, Drill, Storm, and Flink, and it is used as a storage layer for technologies like Apache Kafka, Phoenix, and HBase [White 2015].

Applications that are compatible with HDFS, in general, deal with large datasets [Foundation 2021]. In this sense, HDFS strives to reliably store the files and provide high availability and throughput access to application data [Turkington 2013]. To this end, the data are automatically replicated and distributed across the file system. Optimizing the placement of the replicas distinguishes HDFS from most other distributed file systems [Foundation 2021]. Therefore, there is an effort to maintain the replicas in a way that improves both availability and performance. Over time, however, the data spread among the nodes can become unbalanced [White 2015].

The HDFS *Balancer* [Shvachko et al. 2010] is the native reactive solution to analyze the replica placement and redistribute the stored data to promote replica balancing. The operation of the tool is guided by a balancing policy that determines when the cluster should be considered balanced. There are two policies implemented by the HDFS *Balancer*, namely, *datanode* and *blockpool* [Cloudera, Inc. 2021]. Furthermore, customized policies designed to improve the balancing process and incorporate new features into the HDFS *Balancer* can be found in the literature [Fazul et al. 2019].

This work presents a practical and comparative investigation of different balancing policies to redistribute data replicas in the HDFS. We highlight the benefits and drawbacks

of using the HDFS *Balancer* with the default configuration and non-standard policies. Moreover, we analyze optimizations in data-intensive operations provided by reactive replica rearrangement in the cluster. The experimental investigation was conducted in a real, multi-rack, and distributed environment running HDFS. To the best of our knowledge, no work in the literature has compared and evaluated the trade-off of the balancing policies available for the HDFS *Balancer*.

The rest of this paper is organized as follows. Section 2 gives some background on the HDFS and its data replication mechanism. Section 3 is dedicated to data locality and replica balancing. Section 4 presents the HDFS *Balancer* and its replica balancing policies in detail. After looking into the replica balancing process in HDFS clusters, Section 5 provides an overview of the related work. Section 6 describes and discusses the evaluation results. Finally, Section 7 concludes the paper and outlines future work.

2. Hadoop Distributed File System (HDFS)

HDFS is a reliable and scalable distributed file system. It is designed to be highly fault-tolerant even when deployed on low-cost hardware [Foundation 2021]. An HDFS cluster follows a server-worker architecture composed of two types of nodes: *NameNode* (NN) and *DataNode* (DN). The NN is the master server that manages the system namespace and metadata, maintains the directory tree, and controls access and distribution of the files. Meanwhile, the DNs are the workers responsible for storing the data and serving write and read requests from the clients [Achari 2015].

Since its creation, HDFS has been optimized to store data on the petabyte scale [Shvachko 2010]. To handle this massive volume of data and support very large files, the HDFS implements a characteristic storage structure [Foundation 2021]. When a file is inserted into the system, instead of saving it in its original form, HDFS splits it into a sequence of data blocks: small data chunks created automatically from the partitioning of the initial file into data segments of fixed size (128MB by default). The data blocks are stored across the DNs machines in the cluster.

The NN keeps a reference to every file and block stored in the filesystem in memory, which means that on very large clusters with many files, the memory becomes the limiting factor for scaling [White 2015]. The prior HDFS architecture (Hadoop version 1.x) allows only a single namespace, managed by a single NN, for the entire cluster [White 2015]. To address this limitation, Hadoop 2 release series introduces the concept of HDFS Federation [Foundation 2021]: a feature that improves the existing HDFS architecture through a clear separation of namespace and storage, enabling a generic block storage layer. In order to scale the name service horizontally, HDFS federation enables support for more than one namespace in the cluster, which improves scalability and isolation as each NN manages an isolated portion of the filesystem namespace.

The NNs are federated, which means they are independent and do not require coordination with each other. The DNs register with each NN in the cluster and then they can be used as common storage for all the existing block pools: a set of blocks that belong to a single namespace [Foundation 2021]. Each block pool is managed independently, so an NN failure does not prevent the DNs from serving the remaining NNs in the cluster. In addition to NN failures, the chances of DN failures in an HDFS cluster are high since it runs on commodity hardware [White 2015]. Even with this condition, it is necessary

to ensure high availability and reliability, so that the services are maintained and no data is definitely lost. In this sense, Section 2.1 presents the main fault tolerance mechanism implemented by the HDFS.

2.1. Data Replication

A common way to ensure high reliability and data availability in distributed environments is through replication, in which data redundancy is preserved in the system [Lamehamedi et al. 2002]. HDFS uses block replication as the primary mechanism for fault tolerance and the central element of its storage model. With replication, copies of the data are kept at multiple nodes so that a block can be accessed from any DN that maintains its replicas. It is expected that, in the event of failures, at least one of the copies of the data is still available. The number of replicas is defined by the Replication Factor (RF), which is configured per file and has a default value of three [Achari 2015].

The NN is responsible for monitoring the number of data replicas available for each block stored in the file system, ensuring that the specified RF is respected. To this end, the DN processes communicate periodically with the NN through heartbeat messages [Foundation 2021]: a fault tolerance mechanism that allows the detection of operational failures in DNs. If the NN does not receive heartbeats from a DN within a predefined period, it marks this DN as inactive. The data held in an inactive DN are not available for HDFS, which can take the RF of the blocks previously stored in its node to a value below the specified. Since the NN constantly tracks which blocks need to be replicated, it triggers the re-replication of the under-replicated blocks whenever necessary.

The distribution of the replicas, originating from both the initial replication of the blocks and the re-replication process, is essential for reliability and performance. Optimizing the placement of replicas in the cluster distinguishes HDFS from most other distributed file systems [Foundation 2021]. As the master server, the NN is in charge of selecting which DNs will store the replicas of each block. Therefore, it follows a Replica Placement Policy (RPP) [White 2015], that aims to improve system reliability and data availability, as well as the use of network bandwidth according to the cluster architecture. For the common case, when the RF is three, the RPP stores the first replica on the same node as the client (i.e., in the local machine). The second replica is placed on a node, chosen at random, in a different rack from the first (off-rack) and the third replica is placed on a different node in the same remote rack as the second replica.

The strategy implemented by the RPP ensures high reliability since, even if an entire rack fails, no data blocks will be lost. Moreover, with its rack-aware distribution model, the RPP makes it possible to identify the rack closest to the client that stores the requested replica. As a result, local replicas – or closer to the reader – are preferred over remote replicas [Foundation 2021], which tends to decrease the time spent on data transfers. Section 3 explains how the location of the blocks is exploited by the file system to meet high access demands and, in this context, how an unbalanced data distribution can affect the functioning of the file system.

3. Data Locality and Replica Balancing

HDFS considers that the most efficient data processing pattern is the *write once*, *read many* access model. In this sense, to maximize throughput during reading operations,

Hadoop moves the computing tasks to where replicas are kept and, if it is not possible, to nodes that have a faster network path for the DNs that maintain the requested blocks. This feature, known as data locality optimization [White 2015], increases the overall system throughput when processing large volumes of data, in addition to minimizing read latency and network congestion.

Considering that each block is replicated, by default, in three distinct DNs, the chances that a computational task will be able to process most of the data locally are high [Achari 2015]. However, an unbalanced distribution affects data locality, resulting in a large number of intra-rack or even off-rack transfers, since tasks assigned to nodes that do not maintain many replicas are unlikely to have access to local data. Besides increasing the global bandwidth consumption in the cluster, the imbalance can cause some DNs to become full, preventing them from receiving new replicas and, thus, reducing the reading parallelism and leading to performance degradation [Foundation 2021]. Therefore, HDFS works best when the blocks are evenly distributed across the cluster [White 2015].

The replica imbalance in the HDFS may have different causes, such as [Cloudera, Inc. 2021]: (i) the RPP itself, which stores two-thirds of the replicas in the same rack and does not guarantee a balanced distribution of replicas; (ii) the behavior of the client's application that, if executed directly on a DN, always stores one of the replicas on the local node to reduce write bandwidth consumption; (iii) the occurrence of DN failures, since inactive DNs result in the re-replication process of multiple blocks, which is also based on the RPP; and (iv) the insertion of new DNs in the file system, as the existing blocks are not automatically moved and they will be candidates for block placement alongside all the other active DNs in the cluster to receive new replicas, causing under-utilization of computational resources for a significant period [Turkington 2013].

To mitigate the problems inherent in the imbalance in the distribution of data replicas, maintain maximum cluster health and avoid performance bottlenecks, it is necessary to redistribute the data already stored in the file system [White 2015]. The official solution for replica balancing in HDFS is presented in Section 4.

4. HDFS Balancer

HDFS *Balancer* [Shvachko et al. 2010] is a tool integrated into the Hadoop distribution, aimed at balancing replicas between storage devices on HDFS. It operates iteratively by moving data replicas of DNs that have high utilization (source) to DNs that have a smaller amount of stored data (target) until there is less discrepancy between the data volume held on the nodes. The tool, as a reactive balancing solution, needs to be triggered on demand by the cluster administrator.

Two native policies can be used to determine whether the cluster is balanced, namely *datanode* and *blockpool* [Cloudera, Inc. 2021]. The former is the HDFS *Balancer*'s default policy, which balances the data storage at the DN level (i.e., the cluster is balanced if each DN is balanced). The latter means that the cluster is balanced if each pool in each node is balanced, which is especially relevant to clusters running a federated HDFS service. The *blockpool* is a more strict policy than the *datanode* in the sense that the *blockpool* requirement implies the *datanode* requirement [Cloudera, Inc. 2021], that is, it balances the storage at the block pool level as well as at the DN level. Next, Section 4.1 presents the execution flow of the rebalancing process in HDFS.

4.1. Cluster Balancing Flow

The operation performed by the HDFS *Balancer* is guided by a *threshold*, which has a default value of 10%. Let $G_{i,t}$ represents the group of devices of type t (e.g., disk or SSD) of a DN i. With the standard balancing policy (*datanode*), the *threshold* limits the maximum difference between the utilization of a $G_{i,t}$ ($U_{i,t}$) and the average utilization of the cluster ($U_{\mu,t}$) considering the storage devices of the type t. When the utilization of each DN is within this limit, the cluster is considered balanced. With the *blockpool* policy, in turn, the *threshold* limits the maximum difference between the utilization of a block pool partially stored in a given $G_{i,t}$ and the average utilization of the block pool in relation to the storage capacity of the devices of type t in the cluster. It is important to reinforce that as nodes can store blocks belonging to multiple block pools, balancing each of the pools in each node also implies the guarantees of the *datanode* policy.

The HDFS *Balancer* runs in iterations. Each iteration consists of four main steps [Cloudera, Inc. 2021]. Initially, in the *storage group classification* step, the *Balancer* asks the NN for information about the utilization and occupation of the DNs and then classifies each $G_{i,t}$ as follows: (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 $U_{\mu,t} \geq U_{i,t} \geq U_{\mu,t} - threshold$; or (iv) under-utilized, if $U_{\mu,t} - threshold > U_{i,t}$.

Next, in the *storage group pairing* step, each over-utilized $G_{i,t}$ (source) is paired with one or more under-utilized $G_{i,t}$ (target). For the remaining over-utilized $G_{i,t}$, candidates are selected from the groups classified as below-average. If there is still under-utilized $G_{i,t}$, candidates are chosen among the above-average groups. To reduce bandwidth consumption in the cluster, the pairing strategy initially seeks groups belonging to DNs that reside in the same rack.

Then, in the *block move scheduling* step, the *Balancer* triggers a thread responsible for selecting and moving blocks between each source-target pair. A block in a source DN is a valid candidate to be redistributed if, after the move, its placement continues in accordance with the RPP. Once the block to be relocated has been defined, the DN closest to the target (or less loaded than the source) and that stores a replica of the block to be moved is used as a proxy. This strategy allows reducing the inter-rack network traffic necessary for transferring the data between nodes residing in different racks.

Finally, in the *block move execution* step, the proxy to copy the block to the target DN together with a hint that the block should be deleted from the source DN. If, after the conclusion of all block movements, the cluster is not yet balanced – depending on the defined *threshold* –, a new iteration will start. This strategy ensures, at least, that the utilization of all DNs in the cluster is within the lower ($U_{\mu,t} - threshold$) and upper ($U_{\mu,t} + threshold$) limits.

5. Related Work

As presented in the previous sections, data replication and the placement of the replicas across the nodes are critical subjects related to fault tolerance, reliability, and data availability in distributed file systems. Many researchers have investigated how replication optimizes data locality in data-intensive systems [Liu et al. 2020] and several studies were carried out to evaluate and improve the data replication mechanism [Rajput et al. 2022] and replica balancing [Shwe and Aritsugi 2019] in HDFS.

In [Yin and Deng 2022], the authors investigated methods for placing replicas in edge-cloud environments. Based on the analyses, a strategy was proposed in which the replicas are placed on edge nodes based in their load and on the cost-effective value. A study to determine if increasing the replication factor for in-demand data can have a positive impact on HDFS performance is presented in [Cao et al. 2022]. With the help of an adaptive replication system, which increases the RF of the most accessed data, it was possible to optimize the overall availability of data and reduce job execution times.

In previous work [Fazul et al. 2019], we proposed a customized balancing policy for the HDFS *Balancer*, which focuses on improving data availability and performance through replica balancing. To this end, a balancing priority, called "data availability", has been incorporated into the *block move scheduling* step executed in each balancing iteration to prioritize block movements that increase the availability of the data stored in the HDFS, that is, place the replicas in as many racks as possible. Besides fault tolerance – as placing block replicas in different racks reduces the chances of data loss due to rack failures – the additional availability can be used to take better advantage of cluster bandwidth for performance improvements.

6. Experiments and Discussion

In order to investigate the behavior and evaluate the effectiveness of the main policies for balancing the replicas in HDFS, we carried out experiments on the GRID'5000[1] platform in an environment composed of 10 nodes belonging to cluster *gros* of the *site Nancy*. Each node (Dell PowerEdge R640) had the following configurations: 1 CPU Intel Xeon Gold 5220 (Cascade Lake-SP, 2.20GHz, 18 cores/CPU), 96GB of RAM, 447GB of storage capacity (SSD SATA Micron), and 2 Ethernet connections with a configured rate of 25Gbps each. All the nodes were running a Debian GNU/Linux 10 (*buster*) distribution.

We set up the Apache Hadoop framework (version 2.9.2) in a fully-distributed operation with 2 NNs (federated cluster with two namespaces) and 10 DNs (one DN per node) grouped by four racks (R_1 to R_4). Racks R_1 and R_2 maintained 3 DNs each. Racks R_3 and R_4, in turn, grouped 2 DNs each. The test scenarios we built considered four different policies to distribute the data blocks across the cluster, as follows:

- *initial RPP*, in which the replicas are placed immediately after writing the files based on the standard Replica Placement Policy of the HDFS (i.e., without reactive balancing), as presented in Section 2.1;
- *datanode*, in which the redistribution of the blocks is done by the HDFS *Balancer* configured with the default balancing policy (Section 4);
- *blockpool*, in which the rearrangement of the replicas is performed by HDFS *Balancer* configured with the *blockpool* policy (Section 4); and
- *custom*, in which the replica balancing process in the file system is conducted by the HDFS *Balancer* customized with the "data availability" priority presented in [Fazul et al. 2019], as pointed out in Section 5.

In the experiments presented later in this section, the *initial RPP* is used as a baseline for comparison as it generates the default replica distributions of every file written on HDFS. As for the three balancing policies (*datanode*, *blockpool*, and *custom*), the

[1] `https://www.grid5000.fr`

HDFS *Balancer* – configured with the default *threshold* of 10% – is executed right after the blocks are replicated and stored in the file system. Next, Sections 6.1 and 6.2 present and discuss the evaluation results obtained in two test scenarios with variation in the volume of data stored on the file system.

6.1. First Scenario

In this first scenario, 20 files of 20GB each and with a standard RF of three replicas per block were written. Since the cluster is federated, the load was equally divided into the two namespaces (NS_1 and NS_2), that is, 10 files of 20GB each under the responsibility of each of the NNs. The data load was performed by *TestDFSIO* (version 1.8) [White 2015]: a benchmark that tests HDFS performance by executing parallel and intensive I/O tasks. After the writing operations were complete, the total volume of data in the file system was approximately 1.17TB, which is equivalent to 3,200 blocks of 128MB each (9,600 replicas). The average utilization of the cluster ($U_{\mu,SSD}$) was 32.41%.

The standard deviations (σ) of the occupation of the DNs with the *initial RPP*, *datanode*, *blockpool*, and *custom* policies were, respectively, 57.06GB (σ of 14.47% considering the utilization of the nodes), 22.77GB (5.78%), 19GB (4.82%), and 30.08GB (7.63%). Thereby, with the replicas being distributed based on the *initial RPP*, there is a high discrepancy in the volume of data maintained in the storage devices of the DNs. It is also interesting to note the uttermost differences of the data volume kept in the nodes, which attest that the RPP does not consider the utilization of the nodes when selecting the DNs and, thus, does not guarantee a balanced data distribution across the cluster.

In relation to the balancing policies, we can see how the rearrangement of the replicas contributes to cluster balancing. The result of the balancing operation is attested by the reduction of the standard deviations of the occupation and utilization of the DNs. Besides that, the HDFS *Balancer* strove to maintain the utilization of each DN in an interval controlled by the lower and upper balancing limits. As the *blockpool* policy is more rigorous, it was able to ensure the highest level of balance when compared to the others, closely followed by the *datanode* policy. Since the *custom* policy prioritizes specific block movements during the balancing operation, it achieved a reduced balance level as shown by the slightly higher standard deviation when compared with the two balancing policies, but it still followed the defined *threshold*.

To facilitate the visualization of the volume of data spread across the file system's racks, Table 1 displays the accumulated occupation in GB ($O_{Ri,SSD}$) and utilization percentage ($U_{Ri,SSD}$) at the rack level. It is important to note that racks R_3 and R_4 (2 DNs per rack) had less total storage space than racks R_1 and R_2 (3 DNs per rack). Thus, it is expected proportional differences in the average occupation among the racks. The utilization of the racks, on the other hand, provides values relative to the storage capacity of each rack. The discrepancy in the data volume stored in each rack with the *initial RPP*, among other reasons, is explained by its default choosing strategy of placing one-third of the block replicas in one rack and two-thirds in a second rack, which can promote inter-rack imbalance in the cluster. Also, based on the standard deviations, we can see that the replica rearrangement performed by the HDFS *Balancer* with the *datanode* and *custom* policies help to achieve a better balance at the rack level. The *blockpool* policy, in turn, focuses on balancing at the block pool level, which does not always contribute to inter-rack balancing in the HDFS cluster.

Table 1. Data stored in each rack with the four policies in the first scenario.

Rack	*initial RPP*		*datanode*		*blockpool*		*custom*	
	$O_{Ri,SSD}$	$U_{Ri,SSD}$	$O_{i,SSD}$	$U_{Ri,SSD}$	$O_{Ri,SSD}$	$U_{Ri,SSD}$	$O_{Ri,SSD}$	$U_{Ri,SSD}$
R_1	350.80	29.66	356.15	30.12	321.51	27.19	354.97	30.02
R_2	343.30	29.03	372.65	31.51	377.69	31.94	359.29	30.38
R_3	263.55	33.43	247.05	31.34	258.13	32.74	236.97	30.06
R_4	252.20	31.99	234.57	29.75	258.01	32.73	261.53	33.17
Standard deviation (σ)	51.78GB	2.05%	71.85GB	0.88%	57.61GB	2.67%	63.11GB	1.52%

To further analyze the placement of the replicas considering an availability perspective, we used the HDFS utility *fsck* (*filesystem check*) [White 2015] to retrieve the locations of the block stored in each rack. With the *initial RPP*, as expected, all the blocks were placed in exactly two racks. In contrast, after redistributing the replicas with the balancing policies, we noticed that 11.34% of the block movements carried out with *datanode* policy made the blocks reach maximum availability considering the RF (i.e., block with replicas placed in three unique racks). With the *blockpool* policy that value was 7.78%. With the *custom* policy, on the other hand, 85.65% of the movements increased data availability. This is justified by the behavior of the "data availability" priority implemented by the *custom* policy that prioritizes transfers that place the replicas in a larger number of racks. This is especially useful in scenarios with two or more racks going down at the same time, as placing replicas on only two racks will cause data loss.

Regarding the balancing operation, the *datanode* policy resulted in 90.38GB of data being moved in three balancing iterations that required 4083.25 seconds to be performed. With the *blockpool*, 137.13GB were moved in five iterations that took 8641.5 seconds. The *custom* policy moved 81.78 GB in six iterations, causing the balancing operation to complete in 8165.56 seconds. Naturally, the placement of the replicas in the scenario with the *initial RPP* does not require data redistribution as it places the replicas during the write operation and the initial replication. Nevertheless, the *custom* policy, with a higher number of inter-rack transfers to place the replicas in three different racks, demanded a greater effort in terms of balancing time and bandwidth consumption for data transfers. The *blockpool* policy, in turn, needed to move a larger volume of data to perform the replica balancing at the block pool level. Therefore, the default policy (*datanode*) caused a lower overhead compared to the other two policies even though the block movements performed by all the balancing policies also follow the standard RPP.

To investigate possible performance improvements and optimizations in data locality promoted by the distribution of the replicas with the policies, we considered 20 different executions of *TestDFSIO* to read all the data files stored in each namespace of the HDFS. Table 2 presents key metrics of the I/O operations, separated by the two namespaces (NS_1 and NS_2), considering the arithmetic means of the benchmark in the 20 executions regarding the read time (i.e., the total execution time of the benchmark), read throughput, and read average I/O rate. The read throughput is given by the ratio of the total data volume processed (in MB) to the sum of times (in seconds) spent by each task (due to parallelism, this value may be greater than the execution time of the job). The read average I/O rate is the ratio between the transfer speed obtained by each map task to the total number of mappers. By default, the number of mappers that will be executed is

equivalent to the number of files read by the benchmark.

Table 2. HDFS performance in reading the data in the first scenario.

Metric	*initial RPP*		*datanode*		*blockpool*		*custom*	
	NS_1	NS_2	NS_1	NS_2	NS_1	NS_2	NS_1	NS_2
Read time	102.34	106.02	77.86	91.95	88.89	77.33	68.04	85.95
Percentage change (%)	-	-	-23.92	-13.27	-13.14	-27.06	-33.52	-18.93
Read throughput	389.18	387.69	463.26	438.68	445.08	485.09	558.18	473.37
Percentage change (%)	-	-	19.03	13.15	14.36	25.12	43.42	22.10
Read avg. I/O rate	468.76	445.50	550.27	499.61	596.05	590.11	646.73	573.24
Percentage change (%)	-	-	17.39	12.15	27.15	32.46	37.97	28.67

For each of the metrics, we present the percentage change given by $((T_b - T_a) / T_a \times 100)$. The terms T_a and T_b represent, respectively, the arithmetic mean of the metric under analysis in the 20 runs of the benchmark with the *initial RPP* (baseline) and with the evaluated balancing policy. The average reading time (in seconds) considering both namespaces with the data distribution based on the *initial RPP* was 104.18s. With the *datanode*, *blockpool*, and *custom* policies, the values were, respectively, 84.91s, 83.11s, and 77s. Thus, the average percentage change was -18.5% with the *datanode* policy, -20.22% with the *blockpool* policy, and -26.09% with the *custom* policy. The negative variations represent the reduction obtained in the reading times after running the HDFS *Balancer* in relation to the *initial RPP*.

The average read throughput considering both namespaces with the *initial RPP* was 388.43MB/s. With the *datanode*, *blockpool*, and *custom* policies, there was an increase to 450.97MB/s (percentage change of 16.1%), 465.08MB/s (19.73%), and 515.77MB/s (32.78%). Similarly, the average read I/O rate with the *initial RPP* was 457.13MB/s and, with the balancing, it goes to 524.94MB/s (14.83%), 593.08MB/s (29.74%), and 609.99MB/s (33.44%). Thereby, when compared with the *initial RPP*, there were performance gains using the HDFS *Balancer*, regardless of the policy adopted in the balancing process.

6.2. Second Scenario

For the second scenario, we consider a higher volume of data stored in the file system. In this sense, we used the *TestDFSIO* to write 30 files of 30GB each with the standard RF. The load was equally divided into the two namespaces (NS_1 and NS_2). The total data volume was approximately 2.64TB, which is equivalent to 7.200 blocks of 128MB each and 21.600 replicas in total. The average utilization of the cluster ($U_{\mu,SSD}$) was 72.96%.

The standard deviations of the occupation of the DNs with the *initial RPP*, *datanode*, *blockpool*, and *custom* policies were, respectively, 60.06GB (σ of 15.24% considering the utilization), 25.10GB (6.37%), 24.92GB (6.32%), and 34.19GB (8.67%). Similar to the results obtained in Section 6.1, the replica distribution based on the *initial RPP* caused a high level of cluster imbalance. As for the balancing policies, the *datanode* and *blockpool* policies provided a very similar variance in terms of data volume stored in the DNs (as shown by the proximity of the standard deviations), followed by the *custom* policy, which apart from the balancing, also operates to improve the overall data availability.

The amount of data kept in each rack, considering the accumulated occupation in GB ($O_{Ri,SSD}$) and the final percentage of utilization ($U_{Ri,SSD}$) of the racks in the cluster, can be seen in Table 3. The standard deviations of the occupation and utilization of the racks with the balancing policies are reduced when compared to the *initial RPP*. In this context, all the balancing policies contributed to rack-level balancing. In fact, the HDFS *Balancer* operates to take the utilization of the nodes to the interval defined by the lower limit ($U_{\mu,SSD} - threshold$) and the upper limit ($U_{\mu,SSD} + threshold$), which, in general – but not always, as shown by the *blockpool* policy in Table 1 of Section 6.2 – will provide a better level of inter-rack balancing than the *initial RPP*.

Table 3. Data stored in each rack with the four policies in the second scenario.

Rack	*initial RPP*		*datanode*		*blockpool*		*custom*	
	$O_{Ri,SSD}$	$U_{Ri,SSD}$	$O_{i,SSD}$	$U_{Ri,SSD}$	$O_{Ri,SSD}$	$U_{Ri,SSD}$	$O_{Ri,SSD}$	$U_{Ri,SSD}$
R_1	851.00	71.96	799.10	67.57	785.36	66.41	796.82	67.38
R_2	729.76	61.71	822.95	69.59	792.35	67.00	782.72	66.19
R_3	551.83	70.00	582.03	73.83	592.49	75.15	602.69	76.45
R_4	590.12	74.85	519.04	65.84	553.56	70.21	538.94	68.36
Standard deviation (σ)	136.90GB	5.64%	152.89GB	3.44%	125.65GB	4.00%	129.19GB	4.66%

Concerning the balancing process performed with the three balancing policies, the percentage of blocks movements that place the replicas in unique racks was 8.21% with the *datanode*, 6.9% with the *blockpool*, and 86.27% with the *custom* policy. This demonstrates that the requirement of placing blocks in exactly two racks is relaxed during the HDFS *Balancer* operation though the *custom* policy is the only one that uses the rearrangement of the replicas as a strategy to increase data availability in HDFS.

As for the balancing operation, with the *datanode* policy, the HDFS *Balancer* moved 109.13GB of data in four iterations, which took 7716.96 seconds to complete. With *blockpool* policy, in turn, 186.25GB of data were moved in ten iterations and 10762.12 seconds. With the *custom* policy, a data volume of 115.63GB was redistributed in twelve iterations, demanding 11140.54 seconds to be performed. The high number of balancing iterations required by the *custom* policy is due to its prioritization strategy, which results in a more limited number of possible block movements that allow acting in favor of balancing the cluster and increasing data availability.

To evaluate the performance in subsequent reading operations, we executed *TestDFSIO* 20 more times in read mode. Table 4 shows the metrics collected by the benchmark separated by the namespace. The increase in the overall read times in relation to the values obtained in the first stage presented in Section 6.1 is due to the greater volume of data stored in the cluster. The average of the reading times considering both namespaces was 344.4s with the *initial RPP*, 297.04s with the *datanode* policy, 292.28s with the *blockpool* policy, and 278.41s with the *custom* policy. The percentage change in the times obtained using the balancing policies in relation to the *initial RPP* were -13.75%, -15.13%, and -19.16%, respectively.

In relation to the read throughput, the *initial RPP* achieved 125.49MB/s. With the *datanode*, *blockpool*, and *custom* policies the read throughput was raised to, respectively, 143.6MB/s (percentage change of 14.43%), 149.62MB/s (19.23%), and 157.28MB/s (25.33%). The average I/O rate was 133.26MB/s with the *initial RPP*. With the balancing

Table 4. HDFS performance in reading the data in the second scenario.

Metric	*initial RPP*		*datanode*		*blockpool*		*custom*	
	NS_1	NS_2	NS_1	NS_2	NS_1	NS_2	NS_1	NS_2
Read time	344.37	344.43	307.26	286.82	294.20	290.36	275.70	281.11
Percentage change (%)	-	-	-10.78	-16.73	-14.57	-15.70	-19.94	-18.38
Read throughput	124.80	126.18	135.64	151.56	141.59	157.64	151.78	162.79
Percentage change (%)	-	-	8.69	20.11	13.45	24.93	21.62	29.01
Read avg. I/O rate	133.34	133.18	140.63	169.99	158.07	191.30	160.64	205.77
Percentage change (%)	-	-	5.47	27.64	18.55	43.64	20.47	54.51

policies, the average values increased to 155.31MB/s (16.55%), 174.68MB/s (31.08%), and 183.21MB/s (37.48%). These results reinforce that replica balancing provides significant performance gains regardless of the balancing policy used with the HDFS *Balancer*.

7. Conclusions and Future Work

Data replication is critical to the proper functioning of the Hadoop Distributed File System (HDFS) and optimizing the distribution of replicas across the cluster is one of the most important features of the file system. In this work, we delve deeper into the behavior of the two native balancing policies of the HDFS *Balancer* called *datanode* and *blockpool*. In our analysis, we used the standard strategy for placing replicas on HDFS as a baseline and, for a broader view, we compare the policies with a solution of the literature that we developed in previous work, which consists of a customized replica balancing policy that implements a priority to improve data availability in the file system.

Through practical experimentation, we evidenced the inherent imbalance of data replication on HDFS. All balancing policies showed similar results in terms of reducing the discrepancy in the volume of data maintained in the nodes. Considering performance gains in operations over the stored data, the customized policy presented the best results. This demonstrates that its focus on improving the data availability in HDFS allows taking better advantage of data locality. However, the balancing operation with the customized policy had the highest overhead in terms of balancing time and bandwidth consumption. The *datanode* and *blockpool* policies provided reduced (and similar) optimizations in performance after the balancing but with a lower cost. In particular, the *blockpool* achieved a slightly higher level of balance (trade-off with balancing performance) than the *datanode* and higher performance in serving the I/O applications executed in the experiments. In highlighting the advantages and drawbacks of the evaluated balancing policies we hope to support cluster administrators in decisions regarding replica balancing in HDFS.

For future work, we intend to evaluate the behavior and performance of the balancing policies considering faulty scenarios, different classes of applications running in the cluster, and HDFS instances deployed on heterogeneous environments. In addition, supported by the results presented in this work, we plan to automate the policy selection during the execution of the HDFS *Balancer*. To that end, the decision will be delegated to an event-driven architecture capable of making dynamic adaptations to the balancer settings based on the context of the computational environment and its applications.

Acknowledgment

This work was developed with the support of CNPq - National Council for Scientific and Technological Development – Brazil. 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

Achari, S. (2015). *Hadoop Essentials*. Packt Publishing Ltd, Birmingham, 1st edition.

Cao, X.-y., Wang, C., Wang, B., and He, Z.-x. (2022). A method to calculate the number of dynamic hdfs copies based on file access popularity. *Mathematical Biosciences and Engineering*, 19(12):12212–12231.

Cloudera, Inc. (2021). Managing data storage. [Online]. Available: https://docs.cloudera.com/runtime/7.2.12/scaling-namespaces/topics/hdfs-balancing-data-across-hdfs-cluster.html. [Accessed: Jun 27, 2022].

Fazul, R. W. A., Cardoso, P. V., and Barcelos, P. P. (2019). Improving data availability in hdfs through replica balancing. In *2019 9th Latin-American Symposium on Dependable Computing (LADC)*, pages 1–6, New York. IEEE.

Foundation, A. S. (2021). HDFS Architecture. [Online]. Available: https://hadoop.apache.org/docs/r3.3.4/hadoop-project-dist/hadoop-hdfs/HdfsDesign.html. [Accessed: Mar 27, 2022].

Lamehamedi, H., Szymanski, B., Shentu, Z., and Deelman, E. (2002). Data replication strategies in grid environments. In *5th International Conference on Algorithms and Architectures for Parallel Processing*, pages 378–383, New York. IEEE.

Liu, K., Peng, J., Wang, J., Liu, W., Huang, Z., and Pan, J. (2020). Scalable and adaptive data replica placement for geo-distributed cloud storages. *IEEE Transactions on Parallel and Distributed Systems*, 31(7):1575–1587.

Rajput, D., Goyal, A., and Tripathi, A. (2022). Priority-based replication management for hadoop distributed file system. In *Congress on Intelligent Systems*, pages 549–560. Springer.

Shvachko, K., Kuang, H., Radia, S., and Chansler, R. (2010). The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, New York. IEEE.

Shvachko, K. V. (2010). Hdfs scalability: The limits to growth. *USENIX*, 35(2):6–16.

Shwe, T. and Aritsugi, M. (2019). Preventing data popularity concentration in hdfs based cloud storage. UCC '19 Companion, page 65–70, New York, NY, USA. Association for Computing Machinery.

Turkington, G. (2013). *Hadoop Beginner's Guide*. Packt Publishing Ltd, Birmingham.

White, T. (2015). *Hadoop: The Definitive Guide*. O'Reilly Media, Inc., Sebastopol.

Yin, Y. and Deng, L. (2022). A dynamic decentralized strategy of replica placement on edge computing. *International Journal of Distributed Sensor Networks*, 18(8):9.