The HDFS Replica Placement Policies: A Comparative Experimental Investigation

Rhauani Weber Aita Fazul, Patrícia Pitthan Barcelos

▶ To cite this version:

Rhauani Weber Aita Fazul, Patrícia Pitthan Barcelos. The HDFS Replica Placement Policies: A Comparative Experimental Investigation. 17th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), Jun 2022, Lucca, Italy. pp.151-166, 10.1007/978-3-031-16092-9_10 . hal-03855562

HAL Id: hal-03855562

https://inria.hal.science/hal-03855562v1

Submitted on 9 Dec 2024

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution 4.0 International License

This document is the original author manuscript of a paper submitted to an IFIP conference proceedings or other IFIP publication by Springer Nature. As such, there may be some differences in the official published version of the paper. Such differences, if any, are usually due to reformatting during preparation for publication or minor corrections made by the author(s) during final proofreading of the publication manuscript.

The HDFS Replica Placement Policies: A Comparative Experimental Investigation

Rhauani Weber Aita Fazul(✉) and Patrícia Pitthan Barcelos

Federal University of Santa Maria (UFSM), Santa Maria, RS, Brazil
{rwfazul,pitthan}@inf.ufsm.br

Abstract. The Hadoop Distributed File System (HDFS) is a robust and flexible file system designed for reliably storing large volumes of data in distributed environments. Its storage model relies upon data replication and one of its central features is to optimize the placement of the replicas across the cluster for fault tolerance, availability, and performance. To this end, the Replica Placement Policy selects which nodes will store the data blocks. This work presents an experimental investigation of the different placement strategies available in HDFS. For a broader analysis, we consider different stages where the placement of the replicas is necessary, such as writing files in the system, re-replicating blocks among the nodes, and balancing the replica distribution in the cluster. The evaluation results allowed a deeper understanding of the behavior of the policies, in addition to highlighting the advantages and drawbacks of the replica placement concerning optimizations in data availability, data locality, write and read throughput, and in the overall performance of the HDFS.

Keywords: Data replication · Block distribution · Replica placement policies · Distributed file systems

1 Introduction

In the current days, it is common to come across scenarios that deal with large volumes of data, of the most varied types, being generated at high speed. In this context, there are demands for scalability, reliability, availability, and data distribution, which can not always be satisfactorily addressed by traditional tools, so specialized solutions become necessary. One of these solutions is the Apache Hadoop framework [3]: an open-source platform dedicated to the efficient storage and processing of big data in distributed environments.

The Hadoop Distributed File System (HDFS), Hadoop's storage engine, is a reliable and scalable file system, which is incorporated as a persistence layer by several technologies, such as Apache Spark, Storm, and HBase [12]. HDFS follows a master-worker architecture composed of a *NameNode* (NN) and multiple *DataNodes* (DNs). The NN is the master server, responsible for maintaining the

© IFIP International Federation for Information Processing 2022
Published by Springer Nature Switzerland AG 2022
D. Eyers and S. Voulgaris (Eds.): DAIS 2022, LNCS 13272, pp. 151–166, 2022.
https://doi.org/10.1007/978-3-031-16092-9_10

system namespace and controlling the access and distribution of the files, while the DNs are the workers that effectively store and retrieve the data.

In order to handle large files, HDFS uses a storage strategy based on blocks, where the files are split into a sequence of data blocks of fixed size (128MB by default). The HDFS was designed to run on commodity hardware and reliably store the data across machines in large clusters [2]. So, the blocks of a file are replicated and maintained by different DNs. During replication, the selection of the DNs to maintain the replicas is a critical factor for the proper functioning of the HDFS. To select the DNs, the NN follows a Replica Placement Policy (RPP). There are five different RPPs integrated into the Hadoop distribution, and one of them is applied by default in the file system. A good replica placement optimizes data availability and reliability, in addition to reducing write bandwidth consumption and increasing read performance [12]. The current RPP implementation is the first effort in this direction, and it is one of Hadoop's goals to validate the policy on production systems, learn more about its behavior, and build a foundation to test and research more sophisticated RPPs [3].

This work presents a practical investigation of the RPPs available on HDFS. To this end, we consider different stages in which the policies for replica placement are necessary in the file system, such as writing files across the cluster, re-replicating blocks after failures, and redistributing blocks during replica balancing. At each stage, we analyze the behavior of the RPPs and measure the optimizations in availability and performance achieved by their placement strategies. The experimental analysis was conducted in a real, distributed, and heterogeneous environment running HDFS.

The paper is organized as follows. Section 2 is dedicated to data replication and balancing on HDFS. Section 3 presents the official policies for replica placement. Section 4 outlines the main related work. Section 5 exhibits and discusses the evaluation results. Finally, Sect. 6 concludes the paper and points out further research directions.

2 Data Replication in HDFS

Data replication is the primary fault tolerance mechanism and the core of the HDFS storage model. It consists of creating redundant copies of the data blocks so that, in the event of a failure, there are still replicas available in the system [12]. The replicated data are stored in different nodes of the cluster in such a way that the blocks can be accessed from any DN that maintains their replicas. The number of replicas is determined by the Replication Factor (RF), which is configured per file at the time of its creation and can be modified later through system utilities. An RF of n avoids data loss even if $n - 1$ DNs fail at the same time. The default RF is three.

The NN controls and makes all decisions regarding the replication of the blocks, which involves selecting the DNs for storing the replicas based on an RPP. This choice is initially performed when writing files in the cluster, and it is also necessary during the re-replication and redistribution of the data already

stored in the file system. In all these moments, the selection of the DNs must be done in order to maintain data availability in the event of failures and improve the system's performance in serving I/O operations over the data. Next, Sect. 2.1 introduces block re-replication, Sect. 2.2 details the redistribution process, and Sect. 3 presents the official RPPs that guide the NN decisions.

2.1 Block Re-replication

Active monitoring is a vital requirement for assuring resilience and fault tolerance in the HDFS. In addition to the initial placement of the blocks performed when writing files, it is necessary that NN constantly monitor the state of the replicas and which data blocks need to be re-replicated. The necessity for data re-replication may arise due to different reasons, such as [3]: (i) the corruption of one or more replicas; (ii) a failure in one or more of the DN storage disks; (iii) the increase of the RF of a file; or (iv) DNs becoming unavailable, either due to network partition that causes some subset of DNs to lose connectivity or due to crash-failures.

Even when running on clusters of commodity hardware, HDFS is designed to maintain reliability and data availability in scenarios with consecutive failures. Therefore, the NN must control the number of existing replicas of each block, ensuring compliance with the RF [11]. To this end, the DNs processes communicate periodically with the NN through heartbeat messages: a fault tolerance mechanism that allows detecting operational failures in DNs [2]. If the NN does not receive heartbeats from a DN within a predefined period[1], it marks the DN as dead and does not forward any new requests to that node. The data in a dead DN is not available to HDFS, which can cause the RF of the blocks previously stored in its node to fall below the specified value.

Since the NN determines the mapping of blocks to DNs and constantly tracks which blocks need to be replicated, it can trigger the re-replication of the under-replicated blocks whenever necessary. To re-replicate a data block, the NN selects a source DN that contains one of its remaining replicas and a target DN that will receive the new copy of the replica stored in the source. As with the initial replication, this selection is performed transparently by HDFS and must be in accordance with the defined RPP.

2.2 Replica Rearrangement

HDFS is built around the idea that the most efficient data processing pattern for files is the write once, read many (WORM) access model. In this sense, a principle of Hadoop – and the reason for its good performance – is to move the computational tasks to where the data are stored and, if it is not possible, to

[1] The timeout to set a DN dead is relatively long (over 10 min by default) to avoid replication storms caused by state flapping of DNs [3]. To better suit performance-sensitive workloads, it is possible to configure a shorter interval to mark DNs as stale and exclude their nodes in I/O operations.

the nodes that have a faster network path for the DNs that maintain the blocks needed for the operation. This feature, known as data locality optimization [12], increases the overall throughput when processing large datasets and minimizes read latency and network congestion.

An unbalanced replica distribution tends to affect the locality of the data, resulting in an increased number of intra-rack and off-rack transfers, since tasks assigned to nodes that do not maintain many replicas will possibly not access local data. In addition to increasing the consumption of network bandwidth, the imbalance may cause some nodes to become full and prevent them from receiving new blocks, reducing their read parallelism and leading to performance degradation [3]. Therefore, HDFS works best when the blocks are evenly spread across the cluster. Over time, however, the cluster may become unbalanced, with a large discrepancy in the data volume stored in the nodes.

The main causes of replica imbalance are [5]: (i) the replica placement strategy that, in general, does not consider the node utilization; (ii) the re-replication procedure, which follows the RPP; (iii) the behavior of the client application that, if executed directly on a DN, stores one of the replicas in its node to preserve data locality; (iv) the addition of new DNs to the system, since they will be candidates for replica placement alongside all other DNs [11].

To maintain maximum cluster health and avoid performance bottlenecks, it is necessary to redistribute the data. For this purpose, there is a tool, integrated into the Hadoop distribution, designed for replica balancing: the HDFS *Balancer* [9]. By analyzing the positioning of the blocks, the *Balancer* makes decisions about the redistribution of data between the storage devices in the cluster. The *Balancer* daemon – which should be triggered by the administrator – operates until the utilization of each DN differs from the average utilization of the cluster by no more than a given threshold percentage, which default value is 10% [12]. To this end, it will move replicas from over-utilized to under-utilized DNs, while adhering to the configured RPP.

3 Replica Placement Policies

HDFS instances are commonly spread across multiple racks. In this sense, the placement of replicas on HDFS uses rack awareness[2], both for fault tolerance and performance. The former is achieved by placing replicas of the same block in at least two different racks, assuring data reliability and availability even if an entire rack fails (this could happen, for instance, due to network switch failure or partition within the cluster). The latter is optimized since it is possible to reduce network bandwidth utilization when writing files and to use the bandwidth of multiple racks when reading the data.

There are different RPP implementations available in the Hadoop distribution. They all follow the same interface to select the desired number of targets for

[2] HDFS tries to satisfy a read request from a block that is closer to the reader so that local replicas are preferred over remote data. This reduces global bandwidth consumption and read latency [3].

placing block replicas. Next, the five RPPs currently being supported in HDFS are presented.

- ***BlockPlacementPolicyDefault***: this is the standard replica placement policy used in HDFS. Considering an RF of three replicas per block, if the writer (client) is running on a DN, it puts the first replica of a block on the local machine, otherwise, an arbitrary DN of the cluster is selected. The second and third replicas are placed in the same remote rack – different from the rack of the first replica – on two distinct nodes. In the case of a higher RF, the next nodes are randomly chosen, while avoiding placing too many replicas in DNs on the same rack by keeping the number of replicas per rack below the upper limit given by (replicas − 1) / racks + 2. The other four RPPs extend this default policy by adding a variety of behaviors to meet specific usage demands.
- ***BlockPlacementPolicyRackFaultTolerant***: this policy focuses on placing replicas in as many racks as possible. Considering the standard RF, the local rack is always preferred to store the first replica. In contrast, the second and third replicas are stored in separate remote racks. For this, the cluster must have enough racks (i.e., racks $\geq$ RF). In the end, the difference in the number of replicas for every two racks is no more than one. This allows data operations to take advantage of the bandwidth of multiple racks, in addition to providing greater availability in the event of a rack failure.
- ***AvailableSpaceBlockPlacementPolicy***: this policy aims at a balanced placement of the blocks according to the storage space available in the nodes. In this sense, an effort is made to prioritize DNs to receive replicas based on the used space in their storage devices. This prioritization is controlled by a parameter that represents a fraction of balancing preference, which can have values between 0 and 1. If a value below 0.5 is used, DNs with more space in use will receive more block allocations. By default, this fraction is set to 0.6, which prioritizes DNs with a lower occupation and promotes a balance in terms of the volume of data stored between the nodes.
- ***BlockPlacementPolicyWithNodeGroup***: this policy was designed for environments with a node-group layer, that is, an extra layer of locality/failure groups (contained by racks), which maintains logical nodes. This is particularly useful to represent a cluster with a 4-layers hierarchical network topology, where the leaves represent DNs (computers) and inner nodes represent switches/routers that manage traffic in/out of data centers, racks, or physical host (with virtual switch). With this RPP, the placement strategy is adjusted to put the first replica on the local node (or if it is not possible, on the local node-group or the local rack). If the writer is not on a DN, a random DN is selected. The second replica is placed on a DN that is on a different rack. The third replica is placed on a DN which is on a different node-group but the same rack as the second replica node.
- ***BlockPlacementPolicyWithUpgradeDomain***: this policy selects nodes for placing block replicas that honor the upgrade domain policy. Upgrade domains allow grouping cluster hosts for optimal performance during restarts.

This RPP follows the same placement strategy as the default one while assuring that all replicas have unique upgrade domains. To this end, it distributes data across a set of hosts in the system (potentially larger than a single rack) that can be updated or restarted at once without compromising service and data availability. This feature is useful for very large clusters, or for clusters where rolling restarts may happen frequently.

Regarding the overhead in the writing process, which involves storing multiple replicas of the same block, HDFS applies a replication pipeline technique. Figure 1 illustrates a possible block distribution of a file based on the *BlockPlacementPolicyDefault*, considering a cluster formed by two racks with, respectively, two and three DNs and using the standard RF of three. As can be seen, for each block ($b1$ to $b4$), the same DN maintains a maximum of one of its replicas and, in the same rack, a maximum of two of the three replicas of the block is contained.

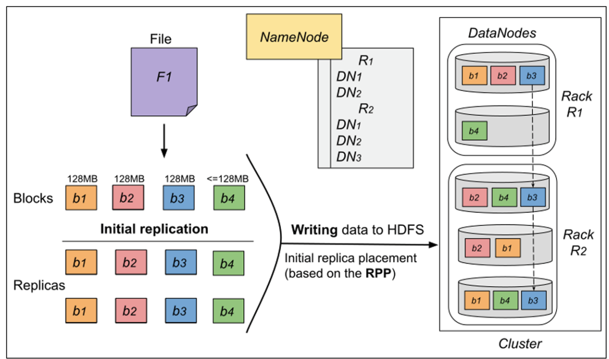

Fig. 1. Standard strategy for block distribution on an HDFS cluster.

In the pipeline of block $b3$ represented in Fig. 1, the NN retrieves the list of the three DNs that will store the data block based on the replication target choosing algorithm implemented by the RPP. Then, the first DN in the pipeline (first DN of rack 1) starts receiving the data in portions, writes each portion to its local repository, and transfers the portion to the second DN in the pipeline (first DN of rack 2). This DN, in turn, writes the received portion to its repository and flushes that portion to the next DN (third DN of rack 2). Therefore, the established writing pipeline allows nodes to simultaneously receive and forward data. Besides improving the write operation, it allows the entire replication process to be transparent to the client, who only needs to interact with a single node when writing data in the file system, regardless of the configured Replica Placement Policy.

The cluster administrator must select which RPP to apply in the file system[3] according to the environment composition and the needs of applications and clients. It is important to mention that the mapping of the cluster topology, which involves racks, node-groups, and upgrade domain groups, is specific to the cluster layout and must also be assigned to each host DN by the administrator (this mapping may or may not reflect the physical network topology of the cluster [5]). The NN uses these definitions to distribute blocks when writing the files and to orchestrate necessary actions, such as rolling restarts and upgrades, block re-replication, and replica rearrangement.

4 Related Work

Several studies have been conducted to investigate the data replication mechanism and the replica distribution strategy of HDFS. The work of [1], for example, provided a theoretical analysis of different approaches for writing data blocks across the DNs, namely: default pipeline, parallel broadcast, and parallel server-worker. The study describes the technical specification, features, and specialization for each approach along with its applications. The authors in [6], on the other hand, presented an improved replica placement policy, which is specifically designed for heterogeneous clusters. The proposed policy satisfies all the selection requirements imposed by the standard RPP while striving to ensure a balanced replica distribution.

A study of the replication factor was presented in [4] to determine if changes in its default value allow for performance enhancements in HDFS. Through 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. The work of [10], in turn, proposed a re-replication scheme that takes into account performance and reliability perspectives. The scheme aims to balance the workload among the nodes during re-replication and reduce the impact and execution time of the re-replication procedure. To this end, the data blocks are divided into priority groups to balance the system and the DNs for storing the replicas are selected based on the utilization of the storage devices in the cluster.

Regarding data redistribution, in previous work [8], we automated the decision-making process for configuring and triggering the HDFS *Balancer*. Besides that, we modified the balancing policy to take into account reliability and availability attributes. The solution maintains the balance of replicas in the system while redistributing the replicas according to the propensity of node failures in the cluster racks.

In [7], on the other hand, we proposed a customized balancing policy for the HDFS *Balancer*, which focuses on improving data availability through replica balancing. To this end, the balancer starts to prioritize block movements that increase the number of racks in which the blocks are placed. This improves

[3] The definition of the RPP to be used in the file system is made in a configuration file (*hdfs-site.xml*), setting the parameter *dfs.block.replicator.classname* with the corresponding classpath for the desired policy.

reliability since placing block replicas in different racks reduces the chances of data loss due to rack failures. Besides that, the additional availability can be used as a way to take better advantage of data locality, thus improving the overall I/O performance of the cluster. The customized policy behaves similarly to the *BlockPlacementPolicyRackFaultTolerant*, however, it is specifically designed for the HDFS *Balancer*. Therefore, it is exclusive to the balancing process and does not interfere with the global RPP used in the file system.

5 Experimentation

The experiments were carried out on the GRID'5000[4] platform, with Apache Hadoop (version 2.9.2) in a fully-distributed operation over 10 nodes of the *site Rennes*. In order to provide a heterogeneous environment, the HDFS instance was set up in two clusters, with 5 nodes in cluster *paravance* (represented by C_1) and 5 nodes in *parapluie* (C_2).

The clusters were configured with two racks each. The racks in C_1, namely R_1 and R_2, kept, respectively, 3 (DN_{01}, DN_{02}, and DN_{03}) and 2 (DN_{04} and DN_{05}) nodes, each one with the following configurations: 2 Intel Xeon E5-2630 v3 CPUs (2.40 GHz, 8 cores/CPU), 128 GB of RAM, 558 GB of storage capacity (HDD), and 2 Ethernet connections of 10 Gbps each. The racks in C_2, namely R_3 and R_4, also maintained 3 (DN_{06}, DN_{07}, and DN_{08}) and 2 (DN_{09} and DN_{10}) nodes, with: 2 AMD Opteron 6164 HE CPUs (1.7 GHz, 12 cores/CPU), 48 GB of RAM, 232 GB of HDD, 1 Ethernet connection of 1 Gbps, and 1 InfiniBand connection of 20 Gbps. The nodes in both clusters were running a Debian GNU/Linux 10 (*buster*) distribution.

To deeply understand the behavior of the RPPs, the test scenario we built considered different situations in which the data distribution in HDFS is affected. To this end, the scenario is divided into 3 stages executed in sequence. In the first stage, detailed in Sect. 5.1, we analyze the placement of replicas during the initial replication resulting from writing files. In the second stage, discussed in Sect. 5.2, we look into the re-replication procedure. In the third stage, presented in Sect. 5.3, we explore the redistribution of data through replica balancing in HDFS.

5.1 First Stage: Data Load

In the first stage, the distribution of the blocks based on the RPPs is done by writing files in HDFS during the initial data load. For this, we used *TestDFSIO* [12] (version 1.8): a distributed I/O bound benchmark that measures HDFS performance through the execution of parallel tasks. An individual experiment was conducted for each RPP and, in every experiment, 20 files of 15 GB each and with an RF of 3 replicas per block were written through a node in R_1 (local rack), totaling a data volume of approximately 900 GB (2400 blocks of 128MB

[4] https://www.grid5000.fr.

each and 7200 replicas in total). The average utilization of the cluster after the data load was 28.01%.

Table 1 shows the HDFS status using the RPPs in the first stage. The amount of data distributed among the nodes is displayed through the occupation in GB (O_{GB}) and the utilization percentage ($U_{\%}$) of each DN. After the initial data distribution with all RPPs, there is a high discrepancy in the volume of data stored in the nodes. This can be seen by the elevated *Standard deviation (σ)* of the occupation and utilization of the DNs. In relation to the *Default* and *NodeGroup* policies, this is explained by their 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.

Table 1. HDFS status after loading data with all Replica Placement Policies.

Rack	*DataNode*	*Default*		*RackFaultTolerant*		*AvailableSpace*		*NodeGroup*		*UpgradeDomain*	
		O_{GB}	$U_{\%}$	O_{GB}	$U_{\%}$	O_{GB}	$U_{\%}$	O_{GB}	$U_{\%}$	O_{GB}	$U_{\%}$
R_1	DN_{01}	105.93	21.01	89.19	17.69	112.65	22.34	96.96	19.23	63.37	12.57
	DN_{02}	74.16	14.71	71.87	14.25	115.39	22.89	82.88	16.44	76.16	15.10
	DN_{03}	75.55	14.98	100.75	19.98	134.63	26.70	80.03	15.87	104.31	20.69
R_2	DN_{04}	108.39	21.50	129.51	25.68	78.17	15.50	95.96	19.03	155.84	30.91
	DN_{05}	107.87	21.39	83.27	16.51	59.97	11.89	111.16	22.05	57.20	11.34
R_3	DN_{06}	66.90	36.50	68.26	37.24	94.10	51.33	65.79	35.89	76.44	41.70
	DN_{07}	105.27	57.43	74.58	40.69	95.79	52.26	97.12	52.98	68.23	37.22
	DN_{08}	80.38	43.85	87.56	47.76	69.23	37.77	94.14	51.36	77.86	42.47
R_4	DN_{09}	74.25	40.50	125.74	68.59	96.36	52.57	93.97	51.26	126.79	69.17
	DN_{10}	113.60	61.97	77.75	42.42	52.00	28.37	93.69	51.11	101.67	55.46
Standard deviation (σ)		18.32GB	17.29%	21.57GB	17.42%	26.14GB	15.40%	12.25GB	16.59%	31.28GB	19.31%

The *RackFaultTolerant* and *UpgradeDomain* policies, on the other hand, place the replicas in unique racks, however, as the RF is less than the number of racks in the cluster, the distribution of the blocks is also not fully balanced. The highest level of balance considering node utilization was achieved by the *AvailableSpace* policy, which ensures that the replicas are positioned in DNs with less used percent based on their storage capacity. Regarding the data volume maintained by each DN (occupation), the *NodeGroup* policy allowed for a better balance since the configured capacity of the nodes in $C1$ is greater than that of the nodes in $C2$.

To better visualize the occupation status of the cluster at a rack level, Fig. 2 illustrates the data stored in each rack. The *AvailableSpace* policy demonstrates its distinguished behavior by storing the largest volume of data in the rack that maintains the nodes with the greatest storage capacity. It is also noted that its choosing algorithm prioritized the local rack R_1, instead of the remote rack R_2, which also belongs to cluster $C1$. The other four RPPs showed similar results between them, with a slight divergence in the volume maintained in each rack.

Table 2 presents key metrics of I/O operations performed in HDFS with the data distribution based on the RPPs. The performance of the write operation is represented by the *Write time* and the *Write throughput*. The *Default* RPP

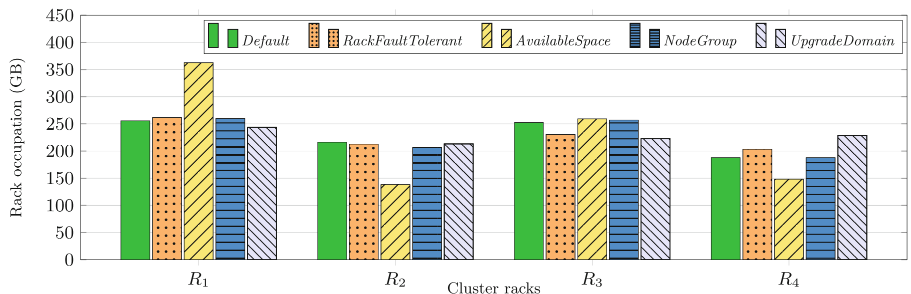

Fig. 2. Data volume stored in each rack after writing the files.

reduces the aggregate network bandwidth used when reading data since most blocks are placed in only two racks rather than three. This improves write performance, however, it does not ensure the block placement with the highest resilience and data availability. The *RackFaultTolerant* and *UpgradeDomain* (each rack was configured in a different domain) policies, in turn, allow for a higher data availability but increase the cost of writing operations since they transfer blocks to more racks.

Table 2. HDFS performance with all Replica Placement Policies in the first stage.

Metric	*Default*	*RackFaultTolerant*	*AvailableSpace*	*NodeGroup*	*UpgradeDomain*
Write time	1907.99s	2235.48s	2485.32s	2272.73s	2354.21s
Write throughput	14.76MB/s	20.33MB/s	18.20MB/s	12.48MB/s	16.17MB/s
Blocks in unique racks	0.00%	100.00%	12.79%	0.00%	100.00%
Read time	877.18s	715.54s	843.74s	870.77s	791.21s
Read throughput	46.01MB/s	87.53MB/s	44.34MB/s	57.21MB/s	71.27MB/s
Read avg. I/O rate	397.71MB/s	384.74MB/s	298.39MB/s	345.90MB/s	231.85MB/s

To further analyze the placement of the replicas considering an availability perspective, we used the HDFS utility *fsck* (*filesystem check*) [12] to retrieve the locations of the blocks stored in each rack. The *Blocks in unique racks* row in Table 2 displays the percentage of blocks that were placed in the largest possible number of racks (i.e., three, given the RF) after writing the files with each RPP. The *RackFaultTolerant* and *UpgradeDomain* policies are the only ones that have this concern, thus ensuring that 100% of the blocks achieve maximum availability. 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. However, as the chance of rack failure is far less than node failures, placing replicas in only two racks tends not to impact data reliability and availability so the other RPPs prioritize writing performance. It is worth mentioning that the *AvailableSpace* policy, which placed 12.79% of the blocks in three racks, focuses on a space balanced distribution, and thus it may place the replicas in unique racks when suitable.

In order to investigate possible performance improvements and optimizations in data locality promoted by the RPPs, we considered 10 executions of *TestDFSIO* – with its default configuration – to read the data stored in the HDFS with each policy. At the bottom of Table 2, we can see the performance of the read operations regarding the arithmetic means of the *Read time* (i.e., the total execution time of the benchmark), *Read throughput*, and *Read average I/O rate*. In general, the *RackFaultTolerant* and *UpgradeDomain* policies performed best, since they enable the applications running on the cluster to use the network bandwidth of one additional rack when operating over the data. In contrast, the other policies stored the replicas in only two racks and resulted in a longer execution time for the benchmark to read the data replicas in the file system.

5.2 Second Stage: Block Re-replication

At this stage, the RPPs are evaluated based on their placement strategies during block re-replication. To emulate a faulty behavior, we insert crash failures in the DNs through the Linux *kill* command. We selected one arbitrary node for racks R_1 and R_3, so that all racks, after the induction of failures, keep exactly two active DNs. When noticing the failure of the faulty nodes (DN_{03} and DN_{08}) for not receiving heartbeats, the NN creates new copies of the under-replicated blocks.

Table 3 shows the state of the cluster after the failures and how the distribution of the new replicas affected the file system. The occupation and utilization of the DNs demonstrate that the imbalance of replicas was further aggravated in the cluster (there was an increase in *Standard deviation* of all RPPs when compared to the first stage), which indicates an unbalanced placement of the re-replicated blocks. It should be noted that the dead DNs are omitted from the table, as they are decommissioned from the cluster. The average utilization of the cluster after the re-replication was 34.87%.

Table 3. HDFS status after re-replication with all Replica Placement Policies.

Rack	*DataNode*	*Default*		*RackFaultTolerant*		*AvailableSpace*		*NodeGroup*		*UpgradeDomain*	
		O_{GB}	$U_{\%}$	O_{GB}	$U_{\%}$	O_{GB}	$U_{\%}$	O_{GB}	$U_{\%}$	O_{GB}	$U_{\%}$
R_1	DN_{01}	127.21	25.23	118.53	23.51	137.07	27.18	125.71	24.93	96.37	19.11
	DN_{02}	107.84	21.39	104.59	20.74	139.08	27.58	118.41	23.48	109.72	21.76
R_2	DN_{04}	129.67	25.72	146.99	29.15	110.36	21.89	118.68	23.54	172.72	34.26
	DN_{05}	128.88	25.56	97.64	19.36	96.76	19.19	129.76	25.74	74.28	14.73
R_3	DN_{06}	82.52	45.02	96.50	52.64	116.54	63.57	83.55	45.58	102.61	55.97
	DN_{07}	119.50	65.19	104.34	56.92	116.91	63.78	114.77	62.61	98.14	53.54
R_4	DN_{09}	89.81	48.99	145.91	79.60	119.31	65.08	106.96	58.35	140.31	76.54
	DN_{10}	122.71	66.94	93.73	51.13	71.06	38.76	109.86	59.93	113.64	61.99
Standard deviation (σ)		18.36GB	18.67%	21.69GB	21.72%	21.89GB	20.09%	14.24GB	17.92%	30.30GB	22.86%

The amount of data kept in each rack after the re-replication can be seen in Fig. 3. We note an interesting feature of the *AvailableSpace* policy. In the first

stage, this policy had prioritized the nodes in rack R_1 to maintain the replicas. After the failure of DN_{03}, however, the policy chose to store the new replicas in another rack (note the significant decrease in the volume of data stored in R_1 in Fig. 3 compared to Fig. 2). This occurred because the remaining nodes in R_1 (DN_{01} and DN_{02}) were no longer the nodes with the largest available storage space in the cluster and, therefore, it was possible to find more suitable nodes to store the replicas in other racks.

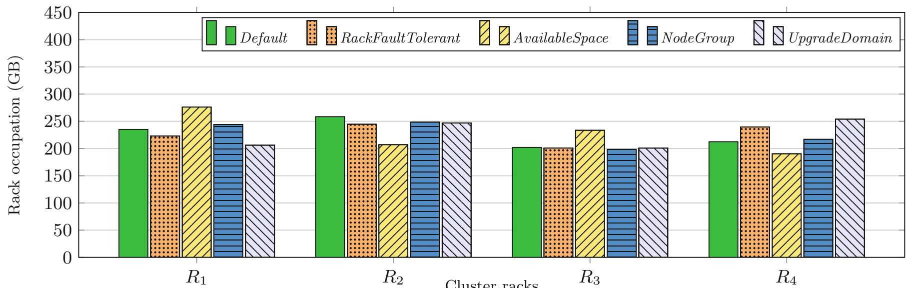

Fig. 3. Data volume stored in each rack after the re-replication of the blocks.

Metrics of the re-replication and subsequent read operations are presented in Table 4. The *Re-replication time* row is equivalent to the elapsed time from the beginning to the end of the re-replication process. We noted that the *Default* policy achieved the highest speed for re-replicating the blocks on component failure, although the performance of the RPPs for storing the new replicas of the under-replicated blocks was not very different from each other. Based on the block mapping, we see that the percentage of *Blocks in unique racks* after the failures has not changed from the first stage, with the exception of the *AvailableSpace* policy, which re-replicated the blocks originally stored in the failed DNs to nodes in a new rack.

Table 4. HDFS performance with all Replica Placement Policies in the second stage.

Metric	*Default*	*RackFaultTolerant*	*AvailableSpace*	*NodeGroup*	*UpgradeDomain*
Re-replication time	585.00s	740.00s	607.00s	819.00s	774.00s
Blocks in unique racks	0.00%	100.00%	48.84%	0.00%	100.00%
Read time	1333.76s	1106.51s	1575.18s	1320.06s	1304.59s
Read throughput	23.36MB/s	42.25MB/s	21.97MB/s	35.16MB/s	34.36MB/s
Read avg. I/O rate	127.89MB/s	148.15MB/s	82.71MB/s	134.66MB/s	95.51MB/s

Regarding the performance of the read operations, we execute *TestDFSIO* 10 more times in reading mode. The increase in the overall *Read time* in relation to the values obtained in the first stage is due to the reduced number of active

DNs (less parallelism and available bandwidth). Again, the higher availability provided by the *RackFaultTolerant* and *UpgradeDomain* policies enables the applications running on the cluster to utilize the bandwidth of one additional rack when operating over the data. However, the *AvailableSpace* RPP, even with 48.84% of the blocks with maximum availability, had the longest reading time and the lowest throughput, which can be justified by the elevated data imbalance after the failures.

5.3 Third Stage: Replica Rearrangement

The third stage evaluates the behavior of the RPPs during the redistribution of the blocks already stored in the system achieved by balancing the replica placement on the cluster. For this purpose, the HDFS *Balancer* daemon was triggered with a default balancing threshold of 10%. Table 5 displays the state of the cluster after running the HDFS *Balancer*. The level of balance achieved in the cluster is evidenced by the reduction of the *Standard deviation* of the utilization of the nodes in relation to their respective values before the replica balancing in the first and second stages.

Table 5. HDFS status after replica balancing with all Replica Placement Policies.

Rack	*DataNode*	*Default*		*RackFaultTolerant*		*AvailableSpace*		*NodeGroup*		*UpgradeDomain*	
		O_{GB}	$U_{\%}$	O_{GB}	$U_{\%}$	O_{GB}	$U_{\%}$	O_{GB}	$U_{\%}$	O_{GB}	$U_{\%}$
R_1	DN_{01}	168.93	33.50	156.22	30.98	152.69	30.28	154.13	30.57	142.49	28.26
	DN_{02}	131.03	25.99	137.07	27.19	163.65	32.46	145.51	28.86	152.70	30.28
R_2	DN_{04}	153.83	30.51	166.58	33.04	151.92	30.13	141.10	27.98	166.68	33.06
	DN_{05}	155.97	30.93	124.98	24.79	143.31	28.42	167.79	33.28	130.27	25.84
R_3	DN_{06}	76.47	41.72	77.99	42.54	77.36	42.20	77.48	42.27	76.60	41.79
	DN_{07}	71.31	38.90	73.07	39.86	78.99	43.09	73.83	40.28	78.49	42.82
R_4	DN_{09}	78.62	42.89	93.23	50.86	76.35	41.65	75.34	41.10	81.39	44.40
	DN_{10}	75.47	41.17	78.37	42.75	71.06	38.76	77.73	42.41	78.62	42.89
Standard deviation (σ)		42.46GB	6.29%	37.55GB	8.94%	41.55GB	6.16%	41.38GB	6.29%	38.40GB	7.58%

The occupation of the racks after the balancing is exhibited in Fig. 4. It is noticed that the rearrangement of the replicas executed by the HDFS *Balancer* does not aim at inter-rack balance. The tool operates to take the utilization of the nodes to an interval controlled by a lower limit (average utilization of the cluster minus the balancing threshold) and an upper limit (average utilization of the cluster plus the threshold). Thus, as we are running the HDFS instance on a heterogeneous environment, the racks of the nodes with less storage capacity maintain a proportionally smaller volume of data.

Table 6 shows the metrics of the replica balancing process and read operations after running the HDFS *Balancer*. Based on the *Balancing time* we can see that the *UpgradeDomain* and *RackFaultTolerant* policies were the most costly in respect of the execution time of the HDFS *Balancer* in the file system. This is caused by the high number of *Balancing iterations* needed to transfer the

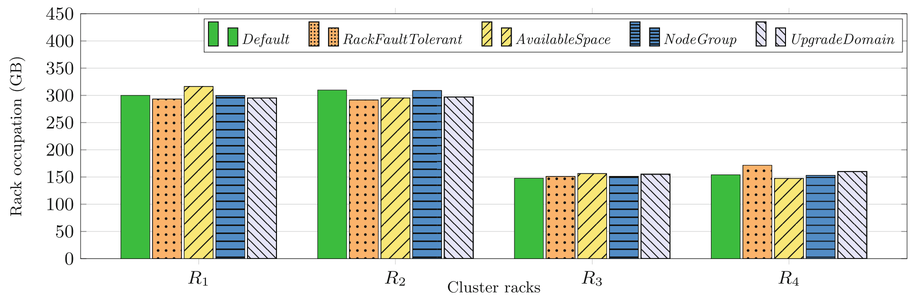

Fig. 4. Data volume stored in each rack after cluster balancing.

replicas between the nodes across the cluster (the amount of data redistributed is represented by the *Data volume moved* row). The *NodeGroup* RPP, on the other hand, had the best performance in the balancing operation, followed by the *AvailableSpace* and *Default* policies. Similarly, these three RPPs do not guarantee that redistribution will store the replicas in three separate racks.

Table 6. HDFS performance with all Replica Placement Policies in the third stage.

Metric	*Default*	*RackFaultTolerant*	*AvailableSpace*	*NodeGroup*	*UpgradeDomain*
Balancing time	6263.17s	6740.23s	5490.20s	4524.93s	8982.66s
Data volume moved	111.25GB	116.38GB	119.13GB	116.00GB	144.63GB
Balancing iterations	9	46	8	7	54
Blocks in unique racks	16.42%	100.00%	62.09%	18.59%	100.00%
Read time	824.39s	879.77s	739.66s	946.54s	770.20s
Read throughput	68.86MB/s	74.19MB/s	85.89MB/s	53.88MB/s	70.84MB/s
Read avg. I/O rate	154.17MB/s	176.25MB/s	186.15MB/s	162.59MB/s	176.64MB/s

The percentage of *Blocks in unique racks* after the balancing shows that, apart from respecting the strategies of the *RackFaultTolerant* and *UpgradeDomain* policies, the requirement of placing blocks in exactly two racks of the other policies is relaxed during the HDFS *Balancer* operation. Therefore, with the balancing, all RPPs maintained some replicas in the largest number of racks allowed by the RF. As mentioned in Sect. 2.2, a balanced cluster can take better advantage of the data locality. To investigate this, we run 10 new executions with *TestDFSIO* aimed at reading the data stored in the file system.

Considering the percentage change given by $((T_b - T_a) \,/\, T_a \times 100)$, where T_a and T_b represent, respectively, the arithmetic means of the metric under analysis in the 10 runs of the benchmark before (i.e., second stage) and after the balancing, the change in the *Read time* with the *Default*, *RackFaultTolerant*, *AvailableSpace*, *NodeGroup*, and *UpgradeDomain* policies was −38.19%, −20.49%, −53.04%, −28.30%, and −40.96%. These values represent the reduction in the execution time of *TestDFSIO* in the third stage in reference to the times obtained without the balance in the second stage.

In addition, there was an increase in the *Read throughput* of 194.78%, 75.60%, 290.94%, 53.24%, and 106.17%, respectively for each RPP. In the *Read average I/O rate*, the increase was 20.55%, 18.97%, 125.06%, 20.74%, and 84.94% after replica balancing with all RPPs. These results reinforce that replica balancing is beneficial for HDFS health regardless of the replica placement strategy used in the file system.

6 Conclusions and Future Work

In HDFS, the blocks are replicated and distributed among different nodes in the cluster. The choice of the *DataNodes* to maintain the block replicas is essential to data reliability, availability, and overall system performance. Optimizing block placement distinguishes HDFS from other distributed file systems. In this sense, HDFS supports the configuration of five pluggable Replica Placement Policies (RPPs). The system administrator can choose the policy based on the cluster infrastructure and the usage requirements of the clients and their applications.

This work presents an experimental and comparative analysis of the strategies used by the official policies for placing replicas in HDFS. Based on the evaluation results we could understand, in-depth, different characteristics of the behavior of the RPPs. To the best of our knowledge – except for the default policy – no work in the literature has investigated the behavior and performance of the specialized block placement strategies of HDFS. In highlighting the trade-off between fault tolerance, write and read bandwidth of each placement strategy, we hope to support the decision-making process of HDFS cluster administrators in choosing the ideal RPP.

We reinforce that, although the experimentation presented in this work is based on a native Hadoop instance, it applies to other processing frameworks that use HDFS as a persistence layer. In this regard, the *BlockPlacementPolicyDefault*, the standard choice in all HDFS distributions, cuts the inter-rack write traffic, which generally improves write performance, without compromising data reliability or read performance. With the *BlockPlacementPolicyRackFaultTolerant*, on the other hand, we can place replicas to more than two racks. This ensures the block placement with the best data reliability and availability even in case of racks failing simultaneously.

The *AvailableSpaceBlockPlacementPolicy*, in turn, extends the default policy so that the selection of DNs starts to be made based on the used space in the storage devices of the nodes, which allows interesting results in heterogeneous clusters. *BlockPlacementPolicyWithNodeGroup* introduces a node group level that fits well with infrastructures running on virtualized environments since it guarantees that, in case of node group failure, only one replica will be lost at the maximum as it will never place more than one replica on the same physical host mapped to a node group. In contrast, the *BlockPlacementPolicyWithUpgradeDomain* addresses the limitation of the default policy on rolling upgrade by adding the concept of upgrade domains into HDFS in which we can group nodes in a new dimension based on the cluster layout in addition to the existing rack-based grouping.

Future research work comprehends an analysis of the behavior and performance of the replica placement policies considering different classes of applications running in the cluster. Besides that, motivated by the achieved results in this work, we plan to validate an alternative for the current RPPs in order to incorporate a temporal perspective of the state of the nodes and the cluster into the data distribution process on the HDFS.

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.

References

1. Abead, E.S., et al.: A comparative study of HDFS replication approaches. Int. J. IT Eng. **3**, 5–11 (2015)
2. Achari, S.: Hadoop Essentials. 1st edn. Packt Publishing Ltd, Birmingham (2015)
3. Apache software foundation: apache hadoop. https://hadoop.apache.org/docs/r3.3.1/ (2021) Accessed 27 Sep 2021
4. Ciritoglu, H.E., et al.: Investigation of replication factor for performance enhancement in the hadoop distributed file system. In: Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, pp. 135–140 (2018)
5. Cloudera Inc: Scaling namespaces and optimizing data storage. https://docs.cloudera.com/runtime/7.2.6/scaling-namespaces/topics/hdfs-balancing-data-across-hdfs-cluster.html (2020). Accessed 3 Sep 2021
6. Dai, W., Ibrahim, I., Bassiouni, M.: An improved replica placement policy for hadoop distributed file system running on cloud platforms. In: 2017 IEEE 4th International Conference on Cyber Security and Cloud Computing (CSCloud), pp. 270–275. IEEE (2017)
7. Fazul, R., Cardoso, P.V., Barcelos, P.P.: Improving data availability in HDFS through replica balancing. In: 2019 9th Latin-American Symposium on Dependable Computing (LADC), pp. 1–6. IEEE (2019)
8. Fazul, R.W.A., Barcelos, P.P.: Automation and prioritization of replica balancing in HDFS. In: Proceedings of the 36th Annual ACM Symposium on Applied Computing, pp. 35–38 (2021)
9. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST), pp. 1–10. IEEE (2010)
10. Shwe, T., Aritsugi, M.: A data re-replication scheme and its improvement toward proactive approach. ASEAN Eng. J. **8**(1), 36–52 (2018)
11. Turkington, G.: Hadoop Beginner's Guide, 1st edn. Packt Publishing Ltd, Birmingham (2013)
12. White, T.: Hadoop: The Definitive Guide, 4th edn. O'Reilly Media Inc, Sebastopol (2015)