Towards a Better Replica Management for Hadoop Distributed File System

Hilmi Egemen Ciritoglu*, Takfarinas Saber†, Teodora Sandra Buda‡, John Murphy* Christina Thorpe*
*Performance Engineering Laboratory, School of Computer Science, University College Dublin, Dublin, Ireland
hilmi.egemen.ciritoglu@ucdconnect.ie, {christina.thorpe, j.murphy}@ucd.ie
†Natural Computing Research and Applications Group, School of Business, University College Dublin, Dublin, Ireland
takfarinas.saber@ucd.ie
‡Cognitive Computing Group, Innovation Exchange, IBM Ireland, Dublin, Ireland
tbuda@ie.ibm.com

***Abstract*—The Hadoop Distributed File System (HDFS) is the storage of choice when it comes to large-scale distributed systems. In addition to being efficient and scalable, HDFS provides high throughput and reliability through the replication of data. Recent work exploits this replication feature by dynamically varying the replication factor of in-demand data as a means of increasing data locality and achieving a performance improvement. However, to the best of our knowledge, no study has been performed on the consequences of varying the replication factor. In particular, our work is the first to show that although HDFS deals well with increasing the replication factor, it experiences problems with decreasing it. This leads to unbalanced data, hot spots, and performance degradation. In order to address this problem, we propose a new workload-aware balanced replica deletion algorithm. We also show that our algorithm successfully maintains the data balance and achieves up to 48% improvement in execution time when compared to HDFS, while only creating an overhead of 1.69% on average.**

***Keywords*-Hadoop Distributed File System, Replication Factor, Software Performance.**

I. Introduction

In recent years, the exponential growth of data and the consequential need to process tremendous volumes, has resulted in a demand for efficient data analysis systems, i.e., systems that can support large-scale, data-intensive analytics. This led many companies to store and analyse their data on distributed systems. One popular example of this is Hadoop [1], a software framework that has been developed by the Apache Software Foundation to store and process big data in an efficient, reliable, and distributed manner. One of the main components of Hadoop is the Hadoop Distributed File System (HDFS) [2]. HDFS is responsible for storing large data sets on distributed machines. Different processing engines (e.g., MapReduce [3], Spark [4]), or applications (e.g., data warehouse systems such as Hive [5] and Pig [6]) run on top of HDFS. Therefore, optimising HDFS is critical for the performance of the Hadoop ecosystem as any improvement on HDFS will affect the overall system.

Replication is a well-known technique for improving the performance of HDFS [7], [8], as increasing the replication factor is directly linked to increasing data availability. Some proposals in the literature aim at increasing the data availability of big data systems using adaptive replication factor frameworks. These frameworks assign a popularity ratio for each file in the system in either a proactive [9], or dynamic [10], [11], [12] way, and use this popularity ratio to define the replication factor. Hadoop systems are long-running systems; thus the demand for files can change over time. Although varying the replication factor can allow significant gains in performance, the placement of replicas is a crucial problem in clusters [7], [13], [14]. As the replication factor changes, so as the block density of each node, leading to performance degradation. Therefore, in homogeneous clusters, better data placement algorithms split data into equal chunks and distribute them on nodes evenly.

While there has been some work in the literature analysing the impact of increasing the replication factor [7], [8], to the best of our knowledge, there is no work analysing the effects of decreasing it. To the best of our knowledge, our paper is the first to identify a major data unbalancing problem in Hadoop's replica deletion algorithm, which has the potential to significantly degrade the performance of the system. As a solution, we propose a novel Workload-aware Balanced Replica Deletion algorithm (WBRD) to prevent this unbalancing problem on Hadoop clusters. We investigate the performance enhancement of WBRD by conducting a thorough performance evaluation. The contributions of this paper can be summarised as follows: (i) we identified a data unbalancing problem resulting in a major performance degradation when the replication factor is decreased, (ii) we formally defined the replica deletion problem, and (iii) we proposed a new deletion algorithm (WBRD) to address this problem, which improves the performance up to 48% with only a small overhead.

The remainder of this paper is organised as follows: In section II, we provide background information and related work about HDFS. Section III identifies and models the replica deletion problem in HDFS. Section IV details our novel WBRD algorithm. Section V describes the experimental environment. Section VI presents results of our evaluation. Finally, section VII concludes this paper.

II. Background and Related Work

HDFS has a master-slave architecture consisting of two main node types, the master NameNode (NN), and the slave DataNode (DN). The *Heartbeat* is the signal that is used to maintain communication between the NN and the DNs. The DN indicates it is alive by sending periodic heartbeats to the NN. The NN controls block replication, enabled by receiving a heartbeat and block reports from the cluster nodes. As HDFS clusters are designed to run on commodity machines, machine failures are not an exceptional situations [1]. To provide reliability, the blocks storing the files are replicated in at least 3 computers (by default), enhancing data locality and allowing a greater tolerance towards machine failures. The replication factor is both a file-level (i.e., can be changed for any specific file individually) and an on-line (i.e., can be altered at runtime) setting.

When data is uploaded onto HDFS, first it will be divided into blocks of a predefined size (by default 128MB). These blocks will be distributed over the cluster. By default, HDFS creates exactly 3 replicas of each data block and places these blocks according to HDFS block placement algorithm. The first replica is placed on a DN in the local rack (preferably on the client itself), the second replica is put on another DN on a different rack, and the third replica is placed on a different DN that is in the same rack as the second replica. This placement strategy provides rack awareness and reduces network traffic between racks.

When a job is submitted, the data must be first located before it is processed. Each part of data used in a job is called a "split", and can be composed of one or more blocks. It is always more efficient to process the data locally rather than requesting the data from a remote node. In this way, Hadoop will always try to process the data locally instead of transferring the data from a different node. Hence, the block placement is crucial for system performance.

There are three modes of task execution related to data locality that are shown in Figure 1.

- **Local access:** when a data split is stored in the same node running the task, e.g., Node 1 processes Block A.
- **Same rack access:** when a data split is stored in another node, but within the same rack as the node running the task, e.g., when Node 4 needs to process Block B, it requests the block from Node 5 (which is in the same rack).
- **Off rack access:** when a data split is stored in another node and in another rack, probably with a slower path through the network, e.g., when Node 3 needs to process Block C, it requests the block from Node 6.

Since Hadoop always tries to run tasks locally, its resource manager (YARN) may even delay the start of a task while waiting for a local node to become available [15]. Achieving a better data locality improves the reading performance and reduces the network consumption, consequently leading to lower execution time.

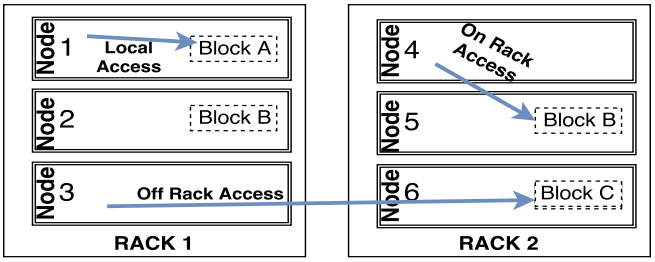


Figure 1. Data locality in Hadoop jobs

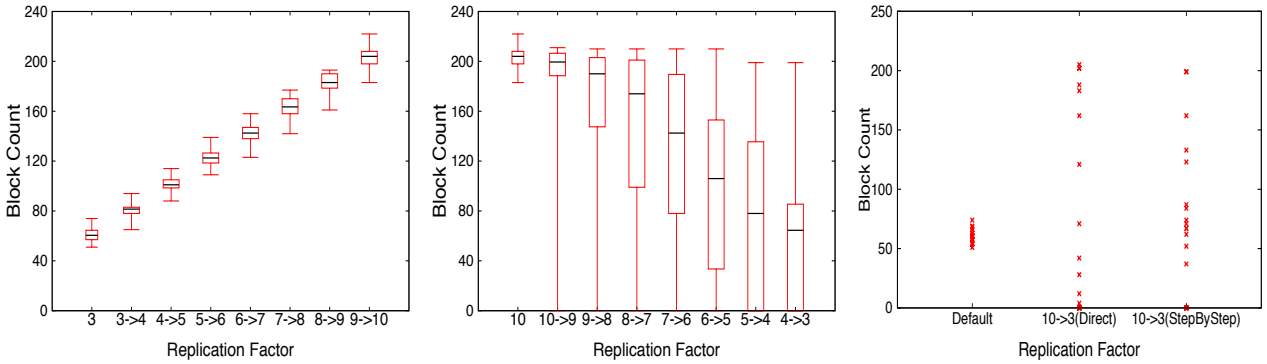
There is a lot of research on data placement as a means to achieving a better data locality in Hadoop [16], [13], [7]. There are also several work that exploit the replication factor and put forwards frameworks [10], [11], [12] with various approaches to determine the 'ideal' replication factor. For instance, Wei et al. propose CDRM, a cost-effective dynamic replication management scheme for cloud storage cluster [10]. The authors propose a model, which links the replication factor with the data availability that they then use to determine the minimum replication factor. Abad et al. [11] propose DARE, an adaptive data replication for efficient cluster scheduling. DARE uses an ageing algorithm and a probabilistic approach to determine a suitable replication factor. Cheng et al. [12] propose ERMS, an elastic replication management system for HDFS. The Active/Standby storage model is used to increase the number of replicas for "hot data" (in-demand data) to active nodes while storing less replicas for "cold data".

Although the proposed replica management frameworks improve data availability and locality, none of the related works to-date study the main drawback of changing the replication factor dynamically in Hadoop. That is, the data unbalancing problem that is introduced when the replication factor is decreased. The resulting performance degradation may not be avoidable in the long-term as, in an adaptive system, data popularity changes over time and the replication factor could be decreased. Our work details a new deletion algorithm, WBRD, to overcome this shortcoming.

III. Replica Deletion Problem

Block distribution is the critical issue in distributed computing, where all machines collaborate with each other to achieve a common goal. It is typically better to evenly-distribute data equally over the cluster rather than storing all the data in fewer nodes. If the data distribution is skewed, a few nodes will keep more data than others, thus becoming hot spots (they will process more data). Furthermore, the rest of the nodes will be less likely to store the particular blocks they have to process and will end up requesting them from hot spot nodes.

In this case, even if nodes are available to process data, they still need to delay the start of map tasks while they wait for data transfer. After these blocks are transferred to



(a) RF is increased step by step

(b) RF is decreased step by step

(c) Default (after the upload) vs RF is decreased step by step and directly

Figure 2. The block distribution when the replication factor (RF) is changed

available nodes, computing can proceed. This will introduce a bottleneck. Consequently, the network consumption and the disk input/output will increase, CPU cycles will be wasted, thus causing performance degradation.

A. Empirical Illustration

As a motivational example and in order to understand the issue with decreasing the Replication Factor (RF) in terms of data imbalance, we analyse HDFS's behaviour when the RF is changed (i.e. increased and decreased).

Figure 2 shows the block distribution per node in a cluster of 20 nodes (see details of the setup in Section V) when the replication factor is modified. Fig 2(a) shows the initial block distribution (i.e., with RF = 3) and its evolution when increasing RF from 3 to 10 step-by-step (increasing RF by 1 every time). Fig 2(b) shows the opposite, as it shows the block distribution with an initial RF of 10 and its evolution when decreasing RF step-by-step from 10 to 3. Fig 2(c) shows individual block counts per node with each point being a block count for a node (i) Default: with an initial RF of 3, (ii) 10->3 Direct: when the RF is decreased in one step from 10 to 3, and (iii) 10->3 StepByStep: when the RF is decreased step-by-step from 10 to 3.

When RF is increased, it can be seen from Fig 2(a) that all of the minimum values are above zero; that is, all the nodes are involved in storing data. In fact, the minimum and maximum values are always close to the median and also the inter-quartile range (mid-spread) is quite narrow. Additionally, the standard deviation is approximately 7 blocks only. This means that Hadoop successfully distributes data equally over the cluster when RF is increased.

However, when RF is decreased, the inter-quartile range gets wider after each step as shown in Fig 2(b). Immediately after RF is decreased from 10 to 9, the minimum value goes to zero. Consequently, at least one of the nodes is not involved in storing data at all. After further inspection of the results, it can be seen that the number of nodes not hosting any data blocks is 1, 2, 4, 5 and 7 when the RF is decreased to 9, 7, 5, 4, and 3, respectively. It can also be seen that there is an increase in the inter-quartile range as the RF is decreased. After reaching an RF of 3, only 65% of all nodes are responsible for storing data. However, only 4 nodes are storing 50% of the entire data. We clearly see from Fig 2(b) that Hadoop does not effectively handle the decrease in RF, as it creates an unbalanced data.

The replication factor can be decreased from 10 to 3 directly or step-by-step. Fig 2(c) shows how blocks are distributed for each reduction methodology. While we see in the 'Default' scenario that blocks are well distributed with $\sim$57 blocks per node and a standard deviation of 5, it is not the case when RF is decreased from 10 to 3. However, we see a larger disparity in block distribution for the direct decrease in RF with a standard deviation of 80 compared to 60 in step-by-step. Furthermore, only 55% of nodes are involved in storing data when reducing RF directly compared to 65% in the step-by-step case.

It is worthwhile mentioning that although we used Hadoop version 2.7.3 in our experiments, we have seen the same trend in other versions (e.g., 3.0.0 released on Dec. 2017). In this paper, we have identified that decreasing the replication factor could create a bottleneck on the system, which has the potential to cause a significant system performance degradation. Considering that Hadoop clusters are long-running systems, the consequence of this could be pernicious.

B. Formal Definition

Let's consider a cluster $\mathcal{C}$ composed of a set of slave machines $\mathcal{M}$ (a.k.a., datanode) contained in a set of racks $Racks$, such that every machine $m \in \mathcal{M}$ is contained in a rack $Rack(m) \in Racks$. The cluster hosts a set of files $\mathcal{F}$ at the root path $Root$. Each file $F_i \in \mathcal{F}$ at the path $Root$ is split over blocks B_i of a predefined size (by default $128MB$). Each block $b_{ij} \in B_i$ is replicated $k_i \in \mathbb{N}^*$ times (i.e., the replication factor of file F_i).

The replica of each block b_{ij}^u with $u \in \{1, .., k_i\}$ is hosted by a machine $M(b_{ij}^u) \in \mathcal{M}$. We denote the block count by BC. Each machine $m \in \mathcal{M}$ hosts a number of blocks $U_{BC}(m)$ (i.e., the block count on m). A machine $m \in \mathcal{M}$

also has an average CPU utilisation $U_{CPU}(m,t)$ during the last given time duration t.

We also define a partial block count $U_{PBC}(m,P)$ for each machine given a path $P \subseteq Root$ which might be different from $Root$ (i.e., not including all the files in the distributed system). To measure this partial block count, we only count blocks $b^u_{i,j}$ that are part of any file F_i located at the given path P. Note that if P is equal to $Root$, then, $U_{BC}(m,P) = U_{PBC}(m)$.

We introduce a binary variable x^u_{ij} for each block replica b^u_{ij} that takes the value 1 if the block is kept after reducing the replication factor, and 0 if it is deleted.

We aim in our work to reduce the replication factor for each file $F_i \in \mathcal{F}$ in the path P from k_i to k^{new}_i, with $k_i > k^{new}_i > 0$. This is enforced by the equation (1).

$$\sum_{u=1}^{k_i} x^u_{ij} = k^{new}_i, \quad \forall\ F_i \in P, \quad \forall\ j \in \{1,...,|B_i|\} \qquad (1)$$

While deleting replicas, we also aim at maintaining rack awareness as expressed in equation (2). In the case of a multi-rack environment and a new replication factor larger than 2, the replicas should be spread over two or more racks. $\forall\ F_i \in P, \quad \forall\ j \in \{1,...,|B_i|\}$:

$$\left|\left\{Rack(M(b^u_{ij})) \mid u \in \{1,...,k_i\}\ and\ x^u_{ij} = 1\right\}\right| \geq 2 \quad (2)$$

We also introduce a variable $PBC_m \in \mathbb{R}^+$ for each machine $m \in \mathcal{M}$ which will set to the partial block count of m after the reduction in replication factor (non-deleted blocks). Every variable PBC_m takes its value following equation (3). $\forall m \in \mathcal{M}$:

$$PBC_m = \sum_{i \in \{1,...,|\mathcal{F}|\}} \ \sum_{j \in \{1,...,|B_i|\}} \ \sum_{\substack{u \in \{1,...,k_i\} \\ \wedge\ M(b^u_{ij}) = m}} x^u_{ij} \qquad (3)$$

For a given path $P \subseteq \mathcal{P}$, the objective of our work is to reduce the replication factor to the desired one while satisfying the rack awareness constraints and minimising the maximum PBC of all machines of the cluster as expressed in equation (4).

$$minimise \ \max_{m \in \mathcal{M}}(PBC_m) \qquad (4)$$

IV. Towards a Better Replica Deletion

The overall objective of our work is to develop a dynamic framework to configure replication factor, as described in our previous work [8]. However, while investigating the performance benefits of adaptive replication factor, we identified a critical cluster unbalancing problem upon replica deletion using the current Hadoop replica deletion algorithm. The purpose of the work in this paper is to improve the performance of Hadoop by providing a workload-aware balanced replica deletion algorithm. Using the proposed algorithm, we expect to see an even data distribution. Therefore, the data locality takes advantage of well-distributed data to achieve a better performance. The resource manager can easily assign a map task to an available node that is also storing data. This will help to reduce the network traffic because data need not be requested remotely.

Replication factor is a file-level setting which can be changed on a per file or per path basis. The Following command can be used for setting the replication factor:

hadoop fs -setrep [-R] [-w] <numReplicas> <path>

Whenever the replication factor is modified, the method *setReplication* in BlackManager.java is called. Then, if the new replication factor is less than the old replication factor, *processOverReplicatedBlocks* method will run for each block of the dataset to manage deletion by calling the *chooseExcessReplicates* method. First, it looks for which blocks are associated with which nodes, then *chooseReplicasToDelete* in BlockPlacemenPolicyDefault.java is called to determine which replicas need to remove from the cluster.

There are two problems in the current algorithm: The first problem is the *chooseReplicasToDelete* algorithm is only concerned about balancing the overall cluster, not the dataset itself. The second problem is that even if Hadoop is concerned about balancing the cluster, the deletion of each replica is based on the initial state of deletion and it does not update metrics after each deletion. Therefore, there is no tight connection between each delete operation, which leads to imbalanced data.

A. Workload-aware Balanced Replica Deletion (WBRD)

In light of these two problems, we are proposing a workload-aware balanced replica deletion algorithm, as shown in Algorithm 1. Our main objective is balancing the data distribution as evenly as possible across the cluster when the replication factor is decreased. The Secondary objective is trying to reduce the CPU load of a node. If two different machines $(m_i, m_j) \in \mathcal{M}^2$ are storing the same amount of partial blocks $U_{PBC}(m_i,P) = U_{PBC}(m_j,P)$, the algorithm will remove the replica from the machine which is over-utilised (i.e., highest CPU utilisation for the last given t time period). First, WBRD calculates the partial block count for each node in the cluster. After that, if an environment is multi-rack, the *getNonRackAware* method returns the set of replicas that will violate one of the rack awareness constraints (as in equation (2)), if any of the replicas is deleted. Deleting non-rack aware replicas can cause a violation of rack awareness, therefore, we do not remove them. In each iteration, WBRD tries to delete the replica that is stored on the node with the highest partial block count and highest CPU utilisation.

B. WBRD Implementation

We implemented our algorithm on top of Hadoop (version 2.7.3). We only altered the HDFS module and more specifi-

Algorithm 1 Workload-aware Balanced Replica Deletion

Input: $\mathcal{M}$: Set<Machine>, P: Path, k_{new}: $\mathbb{N}^*$

$\mathcal{F} \leftarrow getFilesFromPath(P)$
$PBC \leftarrow calculatePartialBlockCount(\mathcal{M}, \mathcal{F})$
for all $F_i \in \mathcal{F}$ **do**
  $B_i \leftarrow getBlocksOfFile(F_i)$
  for all $b_{ij} \in B_i$ **do**
    $\mathcal{R} \leftarrow \{b_{ij}^u \mid u \in \{1, .., k_i\}\ \}$ // gets replicas
    while $|\mathcal{R}| > k_{new}$ **do**
      if $k_{new} \geq 2$ and Env = Multi-Rack **then**
        //replicas critical for rack-awareness
        $\mathcal{R}_{NRA} \leftarrow getNonRackAware(\mathcal{R})$
        $R \leftarrow R - \mathcal{R}_{NRA}$
      $M_{\mathcal{R}} \leftarrow \{M(b_{ij}^u) \mid b_{ij}^u \in \mathcal{R}\}$ // get hosts
      $s_m \leftarrow highestLoadMachine(M_{\mathcal{R}}, PBC)$
      // delete replica
      $deleteBlockAtMachine(b_{ij}, s_m)$
      $PBC(s_m) \leftarrow PBC(s_m) - 1$
      // remove replica from list
      $\mathcal{R} \leftarrow \{b_{ij}^u \in \mathcal{R} \mid M(b_{ij}^u) \neq s_m\}$

Algorithm 2 Select Machine with Highest Partial Block Count and Highest CPU Utilisation(highestLoadMachine)

Input: $\mathcal{M}$: Set<Machine>, PBC: Map<Machine, $\mathbb{N}^+$ >
Output: $Machine$

$m_s \leftarrow \phi$ // selected machine
$maxPBC \leftarrow -1$ // maximum partial block count
for all $m \in \mathcal{M}$ **do**
  if $maxPBC < PBC(m)$ **then**
    $maxPBC \leftarrow PBC(m)$
    $m_s \leftarrow m$
  else if $maxPBC = PBC(m)$ **then**
    if $U_{CPU}(m_s, t) < U_{CPU}(m, t)$ **then**
      $m_s \leftarrow m$
return m_s

cally, the block placement policy and block manager for the server. Whenever an administrator or a replica optimising framework decides to decrease the replication factor for files in a given directory, we create a hashset which matches blocks with the nodes that store them. Next, we calculate the partial block count for each node. Then, our algorithm performs the replica deletion while maintaining a balanced partial block count. After each replica deletion, we update the partial block count. WBRD iterates over the hashset that has already been created to determine the partial block count after deletion. This calculation may create an overhead on the system, which we have measured and will discuss in Section VI. Additionally, in our implementation, we set the time period t during which the CPU utilisation is measured to one day, however, this parameter can be tuned by the user depending on the type/shape of workload.

V. Experimental Setup

In this section, we detail our experimental setup and describe both the testbed and the benchmarks.

A. Experimental Environment

Our experimental environment is a combination of both Hardware and Software configurations.

1) Hardware Configuration: Our experiments were conducted on the Performance Engineering Laboratory's research cluster (in University College Dublin). The cluster consists of 21 dedicated machines (1 master and 20 slaves). Nodes are connected to each other with a Gigabit Ethernet switch(single rack). All slave (DNs) computers are identical: Intel Core i5 (5th generation) processors, 8 GB of RAM and 1 TB hard drive. The master node (NN) has the following specification: Intel Core i7 (6th generation) processors, 16 GB of RAM and 1 TB hard drive.

2) Software Configuration: The operating system selected was Lubuntu, which runs on kernel Linux 4.4.0-31-generic, and the java version 1.8.0_131 was installed. All tests were run on Hadoop version 2.7.3 (native and our implementation). Default Hadoop and YARN Resource manager configurations were used. Hive version 1.2.2 was selected for concurrency tests on TPC-H. Ganglia was used for monitoring the cluster load and the network traffic.

B. Methodology

In this section, we discuss the methodology that was used for experiments. First, we imported data to HDFS; subsequently, the replication factor is increased from 3 to 10. After this, the replication factor is decreased back to 3. We exported and used the same data set for both WBRD and Hadoop's algorithm. Once the benchmark was run on the system, we removed the data and formatted the cluster. Each test run was executed ten times for statistical soundness; we normalised values by using the average of these runs. However, we indicate the range of results in each graph.

The reasoning behind this methodology is that there are different solutions for replica management. All of them try to find the optimal replication factor for 'hot' data, with different methodologies. The 'hot' data can become 'cold' data after some time interval. Regardless of how the replication factor is altered, we should still see an even distribution for homogeneous clusters, similar to the distribution when the data is initially uploaded or replicated.

C. Benchmarks

A distributed system such as Hadoop has several performance bottlenecks that can be stressed (e.g. disk I/O, CPU, network), thus, a comparison between WBRD and Hadoop's replica deletion algorithm would not be fair unless it stresses most of them. While there exists several benchmarks on which we could perform our comparison, each of them stresses a different aspect of the distributed system. For

instance, TestDFSIO stresses disk I/O performance, whereas, Terasort stresses the whole system at once with a large network, disk and CPU utilisation. We therefore selected three benchmarks that stress different parts of the systems: (i) TestDFSIO, (ii) Grep, and (iii) Terasort.

Hadoop clusters are multi-tenant systems with users able to run multiple jobs at the same time. As it has been underlined in [17], query-like frameworks with multiple concurrent users (e.g. Hive and Pig) are heavily used in production environments. Therefore, we also include TPC-H on Hive as a benchmark and test concurrency to represent these production domains.

1) TestDFSIO: is a well-known benchmark for evaluating the reading and writing performance of HDFS. It is a disk-intensive benchmark, which can measure reading/writing throughput. We believe that having a good data distribution should improve reading performance, hence, execution time will be reduced. To test this hypothesis, we used TestDFSIO to generate a 50 GB data set.

2) Grep: is a benchmark that is similar to the "grep" Linux command-line utility. Simply, it searches for certain regular expressions (keywords) in plain-text data and prints the number of lines that contain the keyword as output. The Grep benchmark is both CPU and disk intensive because it needs to read each line individually and check that the line contains the regular expression. We used the NOAA data set [18] from the National Centres for Environmental Information (between 2008/05-2016/04, 47.3 GB). We selected the keyword string as *'2010,1,'* in our experiments.

3) Terasort: is one of the most well-known benchmarks for assessing Hadoop clusters' performance. It measures the system performance by trying to sort data. The sorting is CPU intensive, the reading is disk intensive, passing the data between map and reduce tasks is network intensive, therefore, it stresses the whole system by trying to sort the data that is created from *teragen* (50 GB in our case).

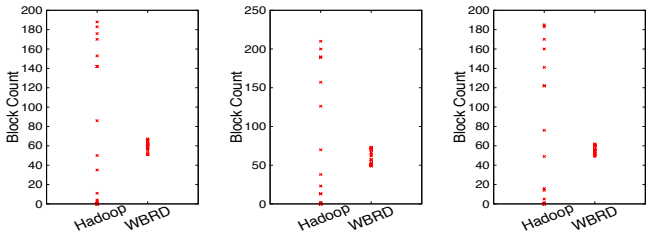
4) TPC-H: is a decision support benchmark [19]. Although, it was designed for testing relational databases; it became popular once the SQL-on-Hadoop frameworks reached significant penetration in industry and academia [20]. We generated a 30 GB TPC-H data set and uploaded it to HDFS. We evaluated the system performance with four different queries: *Q1, Q3, Q6, Q14*. The Queries' key characteristics are described in Table I. We also conducted concurrency tests with *Q6* and measured data locality, average map tasks time, network consumption and ultimately, the system performance. Different numbers of concurrent users are selected as *users* = $\{1, 2, 4, 8, 16, 32\}$. A 1-second interval is used between each user query run.

VI. Evaluation

To evaluate performance, we conducted a comprehensive set of experiments. Firstly, we observed the data distribution because it was our primary objective. WBRD tries to reach an even distribution. The goal is to eliminate hot spots, improve reading throughput, and consequently, WBRD will reduce execution time. To investigate, we inspected the evaluation of the reading throughput, the data locality, the network utilisation, the concurrency and ultimately the overall system performance.

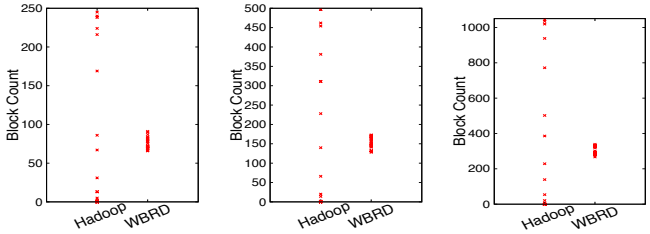
Table I
TPC H Query Characteristics

Query	Key Characteristic	Tables
Q1	aggregation, where, orderby, groupby, 1 table	lineitem
Q3	aggregation, where, orderby, groupby, 2 join, 3 table	customer, lineitem, orders
Q6	aggregation, where, 1 table	lineitem
Q14	aggregation, case, where, 1 join, 2 table	lineitem, part



(a) TestDFSIO (b) Grep (c) Terasort

Figure 3. Data distribution during Hadoop benchmarks



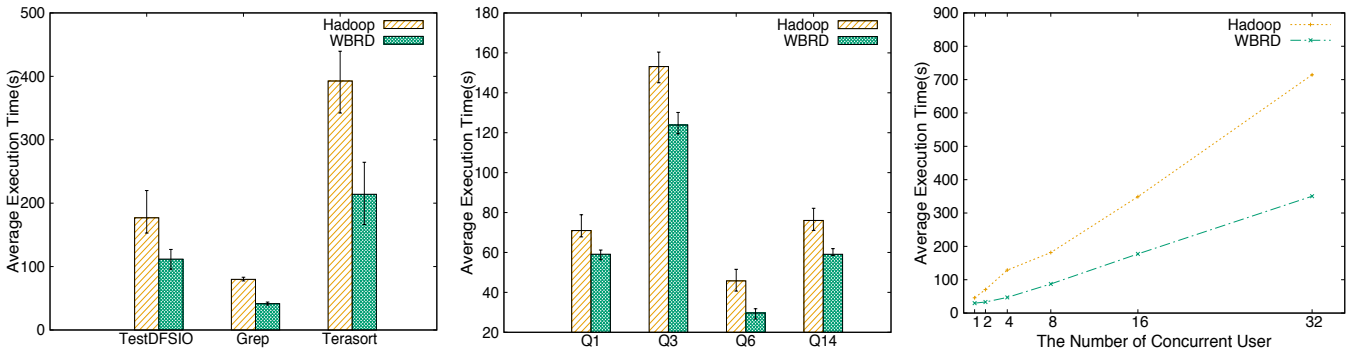
(a) 64GB (b) 128GB (c) 256GB

Figure 4. Data Size - Block Distribution

A. Block Distribution After Deletion

The Figure 3 shows how the blocks are distributed over the cluster when the replication factor is decreased. Each point in the graph represents each node and the number of blocks associated with that node. We noted that standard deviation is approximately 78 for Hadoop, but it is only 4 for WBRD. This result clearly shows that the current Hadoop approach causes data unbalancing after replication factor reduction whereas WBRD maintains an even distribution.

In order to observe the relationship between the block distribution and the data set size after replica deletion, we created all three different data set sizes by using TestDFSIO. Then, we conducted tests with different data set sizes:$\{64GB, 128GB, 256GB\}$, detailed in Figure 4. The standard deviation of the default Hadoop deletion algorithm's is 101 for 64GB, 202 for 128GB, 414 for 256GB.



(a) Benchmarks (b) TPC-H Test with a single user (c) TPC-H Q6 concurrent user test

Figure 5. Performance Tests

Table II
READING THROUGHPUT(MB/S)

	Average	Max	Min
Hadoop	16.948	18.755	14.655
WBRD	29.132	31.805	25.825

On the other hand, it is 7, 14 and 23, respectively for WBRD. Experiments highlighted that the unbalanced replica distribution becomes more critical as the standard deviation increased significantly when testing with the larger data set.

B. Reading Throughput

Since Hadoop was designed for data-intensive computing, the reading throughput is critical for the performance. Moreover, given the data distribution and reading throughput is strongly coupled, we measured reading performance using TestDFSIO. The results are shown in Table II. From the graph, we can note that there has been a significant improvement in reading performance from 17 Mb/sec in Hadoop to 29 Mb/sec in WBRD. Thus, WBRD can significantly improve reading performance by reaching an even distribution. The performance gain is almost 100%, highlighting that the presence of hot spot nodes degrades the system performance.

C. Average Execution Time

Figure 5(a) compares the average execution time with the default Hadoop approach and WBRD of the basic Hadoop benchmarks. It can be seen from fig 5(a) that the proposed solution can significantly reduce the execution time with improvements of approximately 37% for TestDFSIO, 48% for GREP and 45% for Terasort on average.

Figure 5(b) plots the result of the experiment on TPC-H using different queries (*Q1, Q3, Q6, Q14*) with one user. We noted that the performance gain is consistent and approximately 16% for Q1, 19% for Q3, 35% for Q6, and 22% for Q14. Improvements are dependant on the type of jobs. Q6 can be expressed with one MapReduce job only. Therefore, the best improvement has been seen in this case. If several MapReduce jobs are needed to run for a certain query, improvements are limited due to the fact that the network is the bottleneck. Similar to fig 5(a), the performance gains can be attributed to the reading performance through better data locality.

Table III
DATA LOCALITY & AVERAGE MAP TASKS TIME FOR TPC-H Q6 CONCURRENCY TEST

	Hadoop	WBRD
Data Locality	75.7%	89.6%
Average Mapping Time (s)	19.898	14.003

D. Testing with concurrent users

Hadoop clusters are typically multi-tenant systems, therefore, we evaluated a scenario with concurrent users. We measured data locality, network utilisation, average map tasks time, and execution time using *Q6* to provide comprehensive in-depth analysis. Figure 5(c) reports average execution time on TPC-H using *Q6*. The results show that the performance gains become more compelling when the data is concurrently (heavily) queried. Improvements in average execution time are between 35% and 63% when data is heavily queried.

Table III presents the data locality and the average time of map tasks during concurrency tests with *Q6*. Note that the average map task execution time includes reading time and is expected to be higher with a lower data locality. Data locality is measured by $\frac{|DataLocalTasks| * 100}{|AllTasks|}$. It is approximately 75%, however, this increased to 89% for WBRD. Achieving a better data locality means that the more nodes will process local data rather than needing to transfer it from a remote node. Therefore, WBRD can reduce the network consumption and the time spent on map tasks.

By enhancing data locality, the average mapping time can be reduced. Better block distribution helps the resource manager to run map tasks locally. Having more data-local map tasks will help to reduce network consumption. The figure 6 plots network utilisation for only 32 concurrent users. The average network consumption is 275Mbps for Hadoop; however, it is 175Mbps for WBRD. We noted that even though, Figure 6 plots the scenario with 32 users and shows the same trend as with different concurrent users.

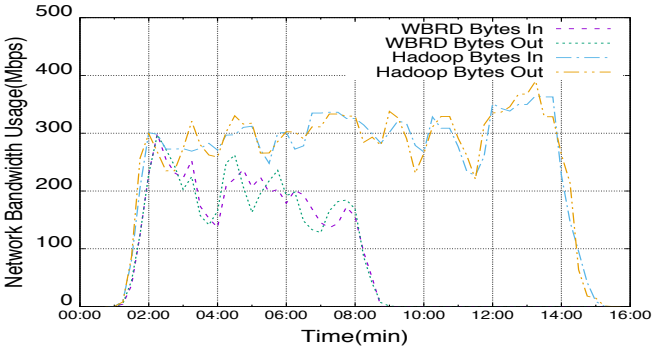


Figure 6. Network utilisation for 32 concurrent users on Q6

Our results suggest that WBRD improves the performance significantly by taking advantage of data locality. Running 'data local' map tasks improves the reading performance and reduces the network utilisation. Thus, map tasks can run more efficiently instead of waiting for the data transfer.

We also measured the overhead for WBRD: Decreasing the replication factor takes approximately 312 seconds, but the overhead is only approximately 5.3 seconds for 30GB TPC-H. When it is scaled up, a linear increase in time can be seen and the overhead is always less than 1.75%. Therefore, we can claim that the overhead of WBRD is significantly low when compared to the total time.

VII. Conclusion

While several works in the literature use the replication factor as a means to improve the performance of Hadoop Distributed File Systems, this paper identifies that adapting this parameter creates a critical performance problem. We found that the current Hadoop replica deletion algorithm leads to imbalanced data, thus generates hot spot nodes. To address this issue, we formally defined the replica deletion problem and proposed a workload-aware balanced replica deletion algorithm. WBRD successfully overcomes the shortcomings of Hadoop's algorithm by maintaining the data balance and avoiding the creation of hot spot nodes during the replica deletion. We have shown in this work the direct impact of block distribution on the performance of a Hadoop cluster using well-known benchmarks with various bottlenecks (e.g., I/O intensive, CPU intensive). We have shown experimentally that WBRD achieves improvements up to 48% in execution time on average while only creating a low computation overhead of 1.69% on average. Future work will extend this work to heterogeneous clusters with resource aware replica placement and ultimately, we will develop an adaptive replication factor framework.

Acknowledgment

This work was supported with the financial support of the Science Foundation Ireland grant 13/RC/2094.

References

[1] Hadoop. [Online]. Available: https://hadoop.apache.org

[2] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *MSST*, 2010, pp. 1–10.

[3] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, pp. 107–113, 2008.

[4] Apache spark. [Online]. Available: http://spark.apache.org/

[5] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy, "Hive: a warehousing solution over a map-reduce framework," *VLDB*, pp. 1626–1629, 2009.

[6] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: a not-so-foreign language for data processing," in *ACM SIGMOD*, 2008, pp. 1099–1110.

[7] M. Y. Eltabakh, Y. Tian, F. Özcan, R. Gemulla, A. Krettek, and J. McPherson, "Cohadoop: flexible data placement and its exploitation in hadoop," *VLDB*, pp. 575–585, 2011.

[8] H. E. Ciritoglu, L. Batista de Almeida, E. Cunha de Almeida, T. S. Buda, J. Murphy, and C. Thorpe, "Investigation of replication factor for performance enhancement in the hadoop distributed file system," in *ICPE Companion*, 2018, pp. 135–140.

[9] G. Ananthanarayanan, S. Agarwal, S. Kandula, A. Greenberg, I. Stoica, D. Harlan, and E. Harris, "Scarlett: coping with skewed content popularity in mapreduce clusters," in *EuroSys*, 2011, pp. 287–300.

[10] Q. Wei, B. Veeravalli, B. Gong, L. Zeng, and D. Feng, "Cdrm: A cost-effective dynamic replication management scheme for cloud storage cluster," in *CLUSTER*, 2010, pp. 188–196.

[11] C. L. Abad, Y. Lu, and R. H. Campbell, "Dare: Adaptive data replication for efficient cluster scheduling," in *CLUSTER*, 2011, pp. 159–168.

[12] Z. Cheng, Z. Luan, Y. Meng, Y. Xu, D. Qian, A. Roy, N. Zhang, and G. Guan, "Erms: An elastic replication management system for hdfs," in *CLUSTER*, 2012, pp. 32–40.

[13] H. Jin, X. Yang, X.-H. Sun, and I. Raicu, "Adapt: Availability-aware mapreduce data placement for non-dedicated distributed computing," in *ICDCS*, 2012, pp. 516–525.

[14] T. Saber, J. Marques-Silva, J. Thorburn, and A. Ventresque, "Exact and hybrid solutions for the multi-objective vm reassignment problem," *IJAIT*, p. 1760004, 2017.

[15] M. Zaharia, D. Borthakur, J. Sen Sarma, K. Elmeleegy, S. Shenker, and I. Stoica, "Delay scheduling: a simple technique for achieving locality and fairness in cluster scheduling," in *EuroSys*, 2010, pp. 265–278.

[16] J. Xie, S. Yin, X. Ruan, Z. Ding, Y. Tian, J. Majors, A. Manzanares, and X. Qin, "Improving mapreduce performance through data in heterogeneous hadoop clusters," in *IPDPSW*, 2010, pp. 1–9.

[17] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *VLDB*, pp. 1802–1813, 2012.

[18] Noaa. [Online]. Available: www.ncdc.noaa.gov/data-access

[19] M. Poess and C. Floyd, "New tpc benchmarks for decision support and web commerce," *ACM Sigmod*, pp. 64–71, 2000.

[20] A. Floratou, U. F. Minhas, and F. Özcan, "Sql-on-hadoop: Full circle back to shared-nothing database architectures," *VLDB*, pp. 1295–1306, 2014.