Heterogeneity-Aware Data Placement in Hybrid Clouds

Jack D. Marquez(✉), Juan D. Gonzalez, and Oscar H. Mondragon

Universidad Autonoma de Occidente, Cali, Valle del Cauca 760030, Colombia
{jdmarquez,juan_davi.gonzalez,ohmondragon}@uao.edu.co

Abstract. In next-generation cloud computing clusters, performance of data-intensive applications will be limited, among other factors, by disks data transfer rates. In order to mitigate performance impacts, cloud systems offering hierarchical storage architectures are becoming commonplace. The Hadoop File System (HDFS) offers a collection of storage policies that exploit different storage types such as RAM_DISK, SSD, HDD, and ARCHIVE. However, developing algorithms to leverage heterogeneous storage through an efficient data placement has been challenging. This work presents an intelligent algorithm based on genetic programming which allow to find the optimal mapping of input datasets to storage types on a Hadoop file system.

Keywords: Hadoop · HDFS · Integer lineal programming · Genetic algorithm · Data placement

1 Introduction

As the amount of data generated by organizations is taking enormous proportions, more and more companies leverage Big Data analysis to drive business decisions [1,2]. The use of distributed file systems allows to reliably store this massive data and distribute storage and computation across very large clusters, facilitating scalability. In order to optimize the storage and retrieval of data, system software must provide efficient data placement mechanisms [3].

Modern distributed file systems commonly have large hybrid storage capacity through the combination of Solid State Drive (SSD) and Hard Disk Drive (HDD) disks. Data placement algorithms must be aware of these heterogeneous architectures in order to maximize the performance of applications using them. For example, applications with real-time requirements may benefit from using SSD instead of HDD, as disk data transfer rates have been identified as a bottleneck for such applications [4].

One of the most commonly used framework to process big data is Hadoop Apache software [5], which can run tasks that use and produce a large amount of data using a vast quantity of processing cores. Hadoop provides a distributed file system (HDFS) [6] and a framework that allows to analyze and process large amount of data using the MapReduce model [7].

D. Da Silva et al. (Eds.): CLOUD 2019, LNCS 11513, pp. 177–191, 2019.
https://doi.org/10.1007/978-3-030-23502-4_13

From version 2.3, Hadoop supports heterogeneous storage, which enables users to specify the type of storage (RAM_DISK, SSD, HDD, ARCHIVE) to use by setting storage policies (Lazy_Persist, All_SSD, One_SSD, Hot, Warm, Cold) to files and directories [4]. The selection of a storage policy impacts directly applications performance since it determines data locality and the data transfer rate offered to them [8]. We defined the problem of allocating different datasets to storage types resources as an Integer Linear Programming (ILP) problem and propose an intelligent algorithm based on Genetic Programming to solve it. Then, we contrast our genetic algorithm against other solutions, specifically Simplex, Generalized Reduced Gradient (GRG) and Evolutionary algorithms and show a comparison of the throughput achieved for each of them for different sizes of datasets by running each algorithm on a Hadoop cluster.

The contributions of this paper are (I) The formulation of the heterogeneous storage problem as an ILP problem. (II) The implementation of an intelligent algorithm based on genetic programming that solves the ILP problem allowing to find the optimal mapping of each dataset to storage types on a Hadoop file system. (III) A comparison of our solution against alternative algorithms. (IV) A new HDFS storage policy that allows using RAM_DISK without replication in other storage types. (V) A benchmark to test and evaluate data write and read throughputs in HDFS.

The rest of the paper is organized as follows. In Sect. 2, we present a brief background on Hadoop heterogeneous storage support and HDFS storage policies. In Sect. 3, we describe the problem and the proposed ILP model for data placement. Section 4 explains our proposed GA to solve the formulated ILP problem. In Sect. 5, we discuss the experiments performed to validate our model. In Sect. 6, we report related work. Finally, we conclude in Sect. 7.

2 Background

This section describes the main features of the Hadoop heterogeneous storage mechanisms and the HDFS storage policies that support our algorithm.

2.1 Hadoop Heterogeneous Storage Support

The heterogeneous storage support included in Hadoop 2.3 version changed the storage model from single storage to multiple physical storage media. This allows HDFS to adapt according to the characteristics of the data to store. HDFS supports the following storage types:

- ARCHIVE: archival storage is commonly used for heavy storage and for storing data that is accessed only rarely.
- DISK or HDD: hard disk drives are the default storage type.
- SSD: it is recommended to use solid state drives to store data that needs to be written and recovery with a higher intensity.
- RAM_DISK: this type of storage has the highest I/O performance, but the storage is non-persistent.

2.2 HDFS Storage Policies

HDFS storage policies allow managing different types of storage and the replication factor. As can be seen in the Table 1, Hadoop provides six different storage policies, based on the types of storage supported by HDFS and the combination of them. Block placement property defines the type of storage that will be used to locate the data blocks and its replicas (n). Fallback Storage for Creation property indicates what storage type will be used as an alternative in the case that the main storage type is not available. Fallback Storage for Replication property indicates what type of storage will be used for the replicas alternatively in the case that the main storage type defined for the replicas is not available [9]. HDFS provides the following policies:

Table 1. HDFS storage policies [9]

ID	Name	Block placement (n)	Fallback creation	Fallback replication
15	Lazy_Persist	RAM_DISK: 1, DISK: n-1	DISK	DISK
12	All_SSD	SSD: n	DISK	DISK
10	One_SSD	SSD: 1, DISK: n-1	SSD, DISK	SSD, DISK
7	Hot (default)	DISK: n	<none>	ARCHIVE
5	Warm	DISK: 1, ARCHIVE: n-1	ARCHIVE, DISK	ARCHIVE, DISK
2	Cold	ARCHIVE: n	<none>	<none>

- Lazy_Persist: this is the only one of the policies that allows to use the storage in RAM_DISK and combines it with the storage in DISK. Lazy_Persist always stores the data in RAM_DISK and replicates them in DISK regardless of the replication factor is 1. A replication factor greater than one impacts performance, since only one replica is stored in RAM_DISK while the rest of the replicas are stored in DISK [9].
- All_SSD: as the name implies, it stores all the replicas in SSD and uses the DISK storage as an alternative for creating data blocks and for replicas.
- One_SSD: this policy combines SSD and DISK. It stores one replica in SSD and the rest in DISK.
- Hot: when no policy is set, this is used by default. Both data blocks and replicas are stored on disk and there is no alternative for the storage of the blocks in case of a failure. For replicas, Archive is used as a fallback.
- Warm: Warm policy combines DISK and ARCHIVE. It stores one replica on Disk and the rest on file.
- Cold: Cold stores all replicas in ARCHIVE.

3 Modeling Data Placement on Heterogeneous Storage

In this section we describe the formulation of the problem of mapping input datasets to storage types on a Hadoop Distributed File System (HDFS). We formulate this as an Integer Linear Programming problem, which allows to minimize the time of placement of datasets on HDFS heterogeneous storage by leveraging storage policies.

3.1 Resource Constraints

We consider the problem of allocating N datasets (DS) to M different Storage Types (ST) in a Hadoop cluster. Each storage type has certain resource capacities including data write rate and storage capacity. Correspondingly, each dataset has a size. In any valid DS-to-ST allocation, the capacity constraint must be satisfied. That is, for each resource, the total capacity requested by all the DSs cannot exceed STs' storage capacity.

3.2 Optimization Objective

The optimization objective of our problem is to minimize the data placement time of the DS to the ST. The placement time may be affected by several contributors such as network technology, CPU, memory and storage throughput. For simplicity, we focus on the effect of storage throughput because it is the factor with impacts most data placement time [10]. HDFS storage policies play an important role here since they allow to define the storage types in which files will be stored.

3.3 Data Placement Model

Each one of the DSs to be stored is received from the application and moved directly to one of the STs. We develop an Integer Linear Programming model [11] for this problem which is considered as NP-Hard [12]. We define an objective function that includes the dataset size, the storage type data write rate, and the correct allocation as decision variable. We consider two types of constraints for this problem: storage capacity and the guarantee that each DS must be assigned to one ST. Table 2 describes the variables we use in our model.

Optimization Objective. Minimize

$$\sum_{j=1}^{M}\sum_{i=1}^{N}\frac{DSS_i}{B_j}\times X_{ij} \tag{1}$$

In our approach, the optimization objective is to minimize the data placement time of datasets to the available storage types. Each mapping time corresponds to the ratio between the dataset size (MB) and the allocated storage type data

Table 2. Problem model variables

Variable	Description
DS_i	i-th Dataset
DSS_i	i-th Dataset size
B_j	j-th Storage Type Data Write Rate
X_{ij}	Dataset $_i$ stored in Storage Type $_j$
C_j	available storage capacity of the $_j$ Storage Type
M	Number of Storage Types
N	Number of Datasets

write rate (MB/s). The mapping of a dataset i to a storage type j is controlled by the binary variable X_{ij}, which indicates whether DS i is assigned to ST j, with $X_{ij} = 1$ if DS i is assigned to ST j and $X_{ij} = 0$ otherwise.

Constraints

$$\sum_{j=1}^{M} X_{ij} = 1, \quad \forall i \tag{2}$$

$$\sum_{i=1}^{N} DSS_i \times X_{ij} \leq C_j, \quad \forall j \tag{3}$$

(2) is the placement constraint, which ensures that each dataset should be assigned to exactly one ST.

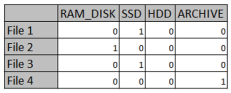
(3) is the storage type capacity constraint. New allocated datasets size cannot exceed the current available storage of a storage type.

4 Solving the Optimization Problem Using Genetic Programming

Computational Intelligence (CI) often is used to address cloud computing resource allocation problems [13]. Our Optimization problem shows an ILP problem behavior and can be solved it using different algorithms such as Simplex [14], GRG [15], Evolutionary and Genetic Algorithms [16], among others. For this problem, we propose a Genetic Algorithm (GA) which is explained in next sections.

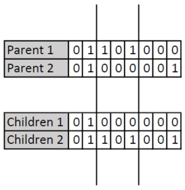
4.1 Chromosome Representation

One of the most important parts of a GA is the chromosomes representation. Each one of the "chromosomes" represents one individual or possible solution to the proposed problem [17]. In our case, each solution consists of a binary vector which represent the mapping of the datasets to any of the four storage types. In Fig. 1 there is an example where the file 1 and 3 are assigned to SSD, file 2 to RAM_DISK, and file 4 to ARCHIVE. In Fig. 1 the chromosome is shown as a matrix but in our algorithm, it is represented as a one dimension vector.

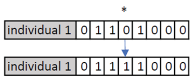


	RAM_DISK	SSD	HDD	ARCHIVE
File 1	0	1	0	0
File 2	1	0	0	0
File 3	0	1	0	0
File 4	0	0	0	1

((a)) Chromosome Representation



((b)) Two-point Crossover



((c)) Single-point Mutation

Fig. 1. GA algorithm steps

The size of this vector is determined by the number of datasets to be placed. One dataset only can be placed in one storage type in order to satisfy the constraint (2). One storage type can store multiple datasets but without exceeding its capacity in order to satisfy the constraint (3).

4.2 Initialization

This is the first stage of the GAs. In this stage the initial population must be created. In our case the population is a number of chromosomes generated randomly. Here, the constraints have to be considered when creating possible solutions (i.e. chromosomes).

4.3 Fitness Function and Selection

The fitness function allows to measure the quality of the solution. Generally, this value is obtained using the objective function. In our case each of the solutions represents the X_{ij} values, therefore we have to multiply these values with the placement time, given by the quotient of the dataset size and the storage type bandwidth. The selection operation is the process in which the best solutions are selected from the population. In our case we are doing a elitist selection which consists in selecting two chromosomes that had the highest value on the fitness function.

4.4 Crossover and Mutation Operators

GAs are based in biological evolution, the crossover (Fig. 1(b)) and mutation (Fig. 1(c)) processes represent the steps in which new individuals, solutions or chromosomes are generated by mixing the best ones (parents) from the previous generation [18]. These new individuals are generated in each generation if the probability of crossover is bigger than the defined. The children usually tend to get a better value for the fitness function.

4.5 Genetic Algorithm Parameters

There is not a standard mechanism to calculate GA parameters [19]. Like conventional genetic algorithms, our chromosome parameters receive random initial values. We consider 100 chromosomes randomly created as initial population, a crossover probability of 0.9 and a mutation probability of 0.1. The number of generations was set to 1000.

5 Allocating Storage in a Hadoop Cluster

In the next sections we describe a number of experiments performed in order to validate our model, both using simulations and real hardware. First, we describe the experiment setup used for both kind of experiments. Next, we introduce a new storage policy that we propose in order to analyze the performance of RAM_DISK storage, which differs from the current Lazy_Persist policy in that it uses exclusively RAM_DISK storage. Then, we use our GA model to simulate the placement of datasets to storage and compare our results against alternative algorithms. Finally, in order to study the algorithms performance in a real HDFS environment, we executed some experiments on a cloud computing cluster running Hadoop.

5.1 Experiment Setup

We tested our GA algorithm and compare its performance against Simplex, GRG, and Evolutionary solutions. For that end, we performed experiments using different numbers of datasets as input, which are mapped to storage in a 5-node cluster with a hierarchical storage setup. We used the experiment setup described here to perform both simulated and real hardware experiments. For the latter, we configured our cluster on Chameleon Cloud [20], a large-scale cloud research configurable environment funded by National Science Foundation (NSF). Table 3 shows the experiment setup.

For the real hardware experiments, we used five Chameleon nodes, each one with two CPU, 67.6 GB RAM, x86_64 Platform Type, Intel Xeon 3.00 GHz Processor, 400 GB SSD and 2TB HDD. One of the nodes was configured as master and the rest as workers. Datasets are mapped exclusively to workers nodes. We allocate to HDFS only part of the worker nodes resources: 64 GB of RAM, 1 TB of HDD, and 250 GB of SSD from each node. The cluster architecture is shown in Fig. 2.

Table 3. Experiment setup

Parameter	Element
Number of datasets	15, 35, 50
Number of storage types	3
Total cluster RAM_DISK storage capacity	256 GB
Total cluster SSD storage capacity	1 TB
Total cluster HDD storage capacity	4 TB
RAM_DISK bandwidth	6600 MB/s
SSD bandwidth	2320 MB/s
HDD bandwidth	267 MB/s
Dataset size	$0 \leq$ DSS $\leq$ 200 GB

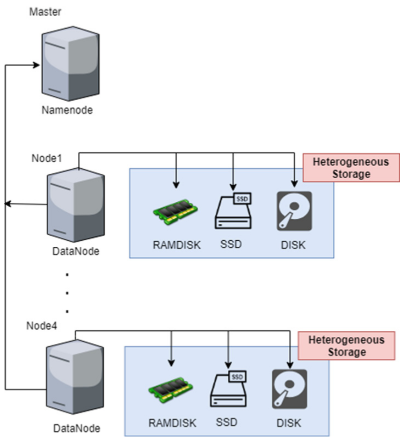


Fig. 2. Hadoop cluster architecture

5.2 A New HDFS Storage Policy

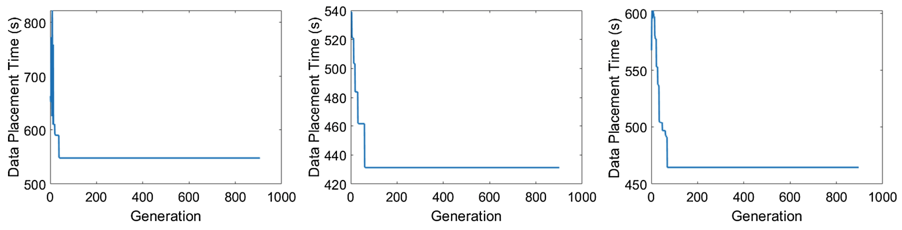
In order to get a sense of RAM_DISK storage performance it was necessary to implement a new storage policy. In HDFS, the only available storage policy that use RAM_DISK is Lazy_Persist and this policy always creates a replica on Disk, even if the replication factor is set as one. We named our new policy All_RAM and differs from Lazy_Persist in that it does not make replications on any other storage type. Table 4 describes All_RAM policy.

Table 4. All_RAM storage policy

ID	Name	Block Placement (n)	Fallback creation	Fallback replication
14	All_RAM	RAM_DISK: n	DISK	DISK

5.3 Using the Genetic Algorithm Model to Simulate Data Placement

Figure 3 shows the results of using the genetic algorithm model we describe in Sects. 3 and 4 to simulate the data placement of different numbers of datasets in a HDFS cluster, according with the experiment setup described in Sect. 5.1. Throughout the generations, the fitness value is getting better, this is because of the selection implemented method, elitism. This method ensures that the parents of each generation are going to be the best in order to likely get better children.



((a)) 15 Datasets ((b)) 35 Datasets ((c)) 50 Datasets

Fig. 3. Best fitness curves using elitism with a 0.9 crossover rate and a 0.1 mutation rate. The initial population is 100 chromosomes.

Each generation must accomplish the ILP problem constraints. In algorithms like simplex or GRG it is possible to explicitly include the constraints, but in the case of an Evolutionary algorithm and for our GA, mechanisms such as *penalization* have to be used in order to make sure that each of the chromosomes satisfies all the constraints. To accomplish that each chromosome satisfies the constraint (2) we have to guarantee that there always exist a storage type storing the dataset, even during the mutation process. Also, to achieve the fulfillment of constraint (3) we implement a penalty process, which sets a high penalization value in the objective function to the chromosomes that do not comply with the restriction. This penalty value is used to ensure that a bad chromosome is not going to be selected as a parent in the selection operation [21,22]. For all the test, the GA found the optimal value long before the generation number 1000. As shown in Fig. 3, for all the cases, the GA reached the equilibrium point before generation number 100.

Comparing Against Alternative Algorithms. Our purpose in this part is to evaluate the simulated data placement time achieved for alternative algorithms and compare against our GA algorithm results. The resulting data placement execution times are shown in Fig. 4.

Figure 4 shows that our GA outperforms the rest of studied algorithms. From that figure, we can see that the data placement times obtained for the generic evolutionary algorithm are significantly higher than the others, even though they have the same parameter values used for GA, this is because the evolutionary

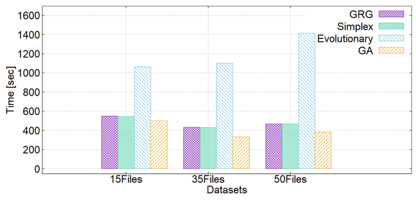


Fig. 4. Data placement execution time (s)

Table 5. Algorithms execution time (s)

	15 Files	35 Files	50 Files
GRG	5.2	50	46
Simplex	1.3	1.2	2.5
Evolutionary	37	34	34
GA	0.6	0.9	1.3

algorithm does not consider the specific constraints for our ILP problem. Most of the GA fitness values were better than the results of the other algorithms even when those methods take into consideration the constraints automatically, except for the evolutionary algorithm [23].

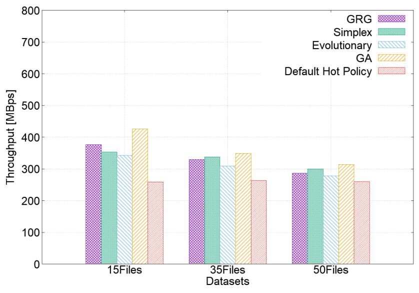
When using few datasets as input, Simplex, GRG and our genetic algorithm have almost the same fitness values. For 35 datasets our GA shows the best fitness value, while GRG and Simplex place the input data in approximately the same time. For 50 datasets the difference between GA and the others algorithm is bigger, but GRG and Simplex are still close.

Table 5 shows the time used for the algorithms under analysis to calculate the data placement allocations. GA was the fastest followed by the Simplex algorithm, while GRG and evolutionary take substantially more time to execute, especially for datasets with a higher number of files.

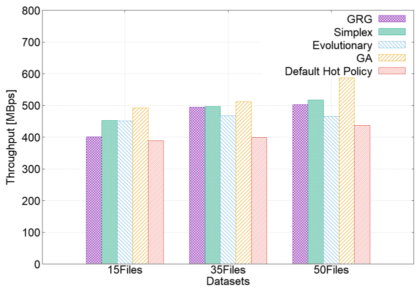
5.4 Placing Datasets into a Real HDFS Cluster

We wrote a synthetic benchmark which generates different numbers of datasets and load them to HDFS according to the allocation made by the algorithms. We performed two experiments using replication factors of one and three. We use replication factor 3 because this is the default replication factor to provide fault tolerance and is the most common replication factor in Hadoop clusters. Additionally, we experimented with a replication factor one in order to contrasts against our simulation results. We also compare these results with the ones obtained from the allocation of datasets using the default Hot policy, in order to evidence how the use of the studied algorithms and heterogeneous storage improve substantially the writing and reading throughput.

Testing Using Replication Factor 1. Figure 5 shows the writing and reading throughput obtained for each dataset, using a replication factor of 1. The GA algorithm shows the best writing and reading performance. Furthermore, it can be evidenced that the throughput using the Simplex and GRG algorithms are very similar. Finally, the Evolutionary algorithm throughput is the slowest of the four. It is inferred that the throughput changes substantially compared with the throughput obtained in simulation, due to the different processes that HDFS performs for the placement of the files. One of those processes is the division



(a) Write

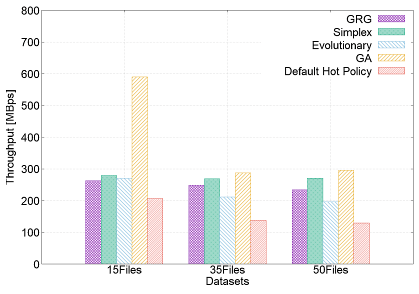


(b) Read

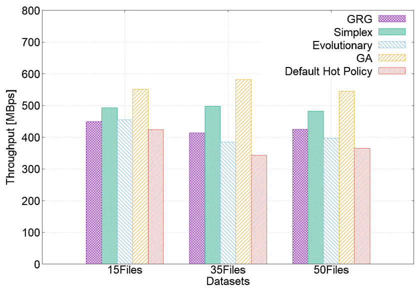
Fig. 5. Write and read throughput using replication factor 1.

of the files into blocks and the distribution of them through the cluster nodes. Other factor that influences the disk writing performance is available cache.

Testing Using Replication Factor 3. Figure 6 shows the writing and reading throughput obtained for each dataset, using a replication factor 3. The data placement used in this test was different to the one used for the test with replication factor 1, because it was necessary to recalculate the size of the files to ensure that the files along with their replicas fit into the HDFS. As in the first test, the GA algorithm kept being the most efficient and the Evolutionary algorithm the slowest of the four.



(a) Write



(b) Read

Fig. 6. Write and read throughput using replication factor 3.

In our datasets, as the number of files increases, the file sizes go smaller. Thus, the files belonging to the 50-files dataset are smaller than the files belonging the 15-files dataset. In the test with replication factor 3, it was possible to observe that HDFS is more efficient for big files. Figure 6 shows that for the dataset with 15 files the throughput was better than the dataset with 35 and 50 files.

6 Related Work

Several works identify data storage and access as a bottleneck in Hadoop Clusters [4,24–27]. Integrating SSD as an effective storage tier in addition to HDD in HDFS, has been pointed out as an alternative to store data while the performance improves. Some works show this integration [25–30].

In [31], authors propose a new HDFS multi-tier storage design that integrates different heterogeneous technologies such as HDD and SSD. In order to exploit the advantage of each storage type, they propose data placement policies that consider SSD high cost and HDD low transmission rate to make placement decisions.

In [26], authors propose a dynamic data management system for Hadoop, allowing workloads that are expected to benefit from SSD to be able to use it as a cache for the HDDs too. They integrate in their data management system a component called Popularity Predictor, which is responsible for analyzing the number of accesses to different files in order to decide which files should be moved to SSD and get less access time to them.

Pan et al. [27], also propose a data access strategy for Hadoop. This strategy focuses in establish some categories for the tasks, taking into account data locality and available storage types. Both, different speeds of storage types and reduced access times thought data locality, allow reduce the applications execution time.

Other authors focused in managing the replication factor in order to guarantee the storage availability in HDFS [32]. Other techniques have been used to improve heterogeneous storage performance [4,33], and some works consider infrastructure heterogeneity and try to find the best data placement according with specific machine capabilities [34].

Most of the approaches mentioned above only integrate SSD to the multi-tier heterogeneous storage, without including RAM_DISK. We incorporated it and designed a new policy for its access. RAM_DISK can outperform SSD due to its transmission rate, but it has limited storage capacity. Authors in [33] and [4], include the RAM_DISK in order to improve HDFS performance. Subramanyam [4] consider data temperature to store it, but temperature only means the access frequency without considering storage types capabilities. Islam [33] assign a random priority to each dataset to be stored. We instead include the implementation of an intelligent algorithm based on genetic programming that solves the ILP problem, allowing to find the optimal mapping of each dataset to storage types on a HDFS, taking in account the size of datasets and the write rate of storage types.

7 Conclusions and Future Work

In this paper we have shown how leveraging the Hadoop heterogeneous storage support in combination with intelligent algorithms to perform data placement can help to optimize the performance of applications by improving reading and writing rates in HDFS. All the studied algorithms outperformed the default hot storage policy used by HDFS, while our proposed GA solution showed the best results among all of them.

Genetic algorithms are not generally considered to work in a model with constraints. This is because the search or selection operators, crossover and mutation does not take into count constraints. Hence, there is no guarantee that if a parent chromosome satisfies the constraints the children will satisfy them as well. It is for this reason that some authors suggest not to work with GAs in problems with constraints [35]. We decided to tackle this by working with a penalization system which does not allow wrong chromosomes to be selected as parents due to its resulting fitness function value. This mechanism also provides faster solutions that chromosome repairing methods. As a result of this strategy, our proposed GA took less than 100 iterations to find the best value for our objective function even though it was configured to iterate 1000 times.

One direction for future work is to propose a complete model for data placement in HDFS clusters considering different characteristics of the input data such as arrival frequency, size, priority, among others; becoming a multi-objective optimization problem. Considering internal HDFS operations such as the division of the files into blocks and the distribution and replication of them through cluster nodes would make the model more accurate. Another direction for future work is to include in the model the monetary cost that generates the use of each one of the storage types as a criterion to decide mapping.

Acknowledgements. Results presented in this paper were obtained using the Chameleon testbed supported by the U.S. National Science Foundation.

References

1. Zhou, K., Fu, C., Yang, S.: Big data driven smart energy management: from big data to big insights. Renew. Sustain. Energy Rev. **56**, 215–225 (2016)
2. Li, H., Li, H., Wen, Z., Mo, J., Wu, J.: Distributed heterogeneous storage based on data value. In: 2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), pp. 264–271 (2017)
3. Bezerra, A., Hernandez, P., Espinosa, A., Moure, J.C.: Job scheduling in Hadoop with shared input policy and RAMDISK, pp. 355–363 (2014)
4. Subramanyam, R.: HDFS heterogeneous storage resource management based on data temperature, pp. 232–235 (2015)
5. Welcome to apache hadoop!
6. 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 (2010)
7. Dean, J., Ghemawat, S.: MapReduce: simplified data processing on large clusters. Commun. ACM **51**(1), 107 (2008)
8. Xiong, R., Luo, J., Dong, F.: Optimizing data placement in heterogeneous hadoop clusters. Clust. Comput. **18**(4), 1465–1480 (2015)
9. Archival storage, SSD & memory

10. Yoon, M.S., Kamal, A.E.: Optimal dataset allocation in distributed heterogeneous clouds. In: Globecom Workshops (GC Wkshps), 2014, pp. 75–80. IEEE (2014)
11. Klein, D., Hannan, E.: An algorithm for the multiple objective integer linear programming problem. Eur. J. Oper. Res. **9**(4), 378–385 (1982)
12. Apers, P.M.: Data allocation in distributed database systems. ACM Trans. Database Syst. (TODS) **13**(3), 263–304 (1988)
13. Guzek, M., Bouvry, P., Talbi, E.G.: A survey of evolutionary computation for resource management of processing in cloud computing. IEEE Comput. Intell. Mag. **10**(2), 53–67 (2015)
14. Nelder, J.A., Mead, R.: A simplex method for function minimization. Comput. J. **7**(4), 308–313 (1965)
15. Lasdon, L., Waren, A.: Generalized reduced gradient software for linearly and nonlinearly constrained problems. Graduate School of Business, University of Texas at Austin Austin, TX (1977)
16. Coello, C.A.C., Lamont, G.B., Van Veldhuizen, D.A., et al.: Evolutionary Algorithms for Solving Multi-objective Problems, vol. 5. Springer, Boston (2007). https://doi.org/10.1007/978-0-387-36797-2
17. Gen, M., Cheng, R.: Genetic Algorithms and Engineering Optimization, vol. 7. Wiley, Hoboken (2000)
18. Srinivas, M., Patnaik, L.M.: Adaptive probabilities of crossover and mutation in genetic algorithms. IEEE Trans. Syst. Man Cybern. **24**(4), 656–667 (1994)
19. Chiroma, H., Abdulkareem, S., Abubakar, A., Zeki, A., Gital, A.Y., Usman, M.J.: Correlation study of genetic algorithm operators: crossover and mutation probabilities. In: Proceedings of the International Symposium on Mathematical Sciences and Computing Research, pp. 6–7 (2013)
20. About Chameleon | Chameleon
21. Gen, M., Cheng, R.: A survey of penalty techniques in genetic algorithms. In: Proceedings of IEEE International Conference on Evolutionary Computation, pp. 804–809. IEEE (1996)
22. Michalewicz, Z., Janikow, C.Z.: Handling constraints in genetic algorithms. In: ICGA, pp. 151–157 (1991)
23. Kolen, A.: A genetic algorithm for the partial binary constraint satisfaction problem: an application to a frequency assignment problem. Stat. Neerl. **61**(1), 4–15 (2007)
24. Li, H., Li, H., Wen, Z., Mo, J., Wu, J.: Distributed heterogeneous storage based on data value. In: 2017 IEEE 2nd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), pp. 264–271. IEEE (2017)
25. Krish, K., Anwar, A., Butt, A.R.: hatS: a heterogeneity-aware tiered storage for Hadoop. In: 2014 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, pp. 502–511. IEEE (2014)
26. Krish, K., Iqbal, M.S., Butt, A.R.: VENU: orchestrating SSDs in Hadoop storage. In: 2014 IEEE International Conference on Big Data (Big Data), pp. 207–212 IEEE (2014)
27. Pan, F., Xiong, J., Shen, Y., Wang, T., Jiang, D.: H-scheduler: storage-aware task scheduling for heterogeneous-storage spark clusters. In: 2018 IEEE 24th International Conference on Parallel and Distributed Systems (ICPADS), pp. 1–9. IEEE (2018)
28. Krish, K., Wadhwa, B., Iqbal, M.S., Rafique, M.M., Butt, A.R.: On efficient hierarchical storage for big data processing. In: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), pp. 403–408. IEEE (2016)

29. Kambatla, K., Chen, Y.: The truth about mapreduce performance on SSDs. In: 28th Large Installation System Administration Conference (LISA14), pp. 118–126 (2014)
30. Narayanan, D., Thereska, E., Donnelly, A., Elnikety, S., Rowstron, A.: Migrating server storage to SSDs: analysis of tradeoffs. In: Proceedings of the 4th ACM European Conference on Computer Systems, pp. 145–158 ACM (2009)
31. Kang, S.H., Koo, D.H., Kang, W.H., Lee, S.W.: A case for flash memory SSD in Hadoop applications. Int. J. Control. Autom. **6**(1), 201–210 (2013)
32. Wei, Q., Veeravalli, B., Gong, B., Zeng, L., Feng, D.: CDRM: a cost-effective dynamic replication management scheme for cloud storage cluster. In: 2010 IEEE International Conference on Cluster Computing, pp. 188–196. IEEE (2010)
33. Islam, N.S., Lu, X., Wasi-ur Rahman, M., Shankar, D., Panda, D.K.: Triple-H: a hybrid approach to accelerate HDFS on HPC clusters with heterogeneous storage architecture. In: 2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), p. 101. IEEE (2015)
34. Xiong, R., Luo, J., Dong, F.: Optimizing data placement in heterogeneous Hadoop clusters. Clust. Comput. **18**(4), 1465–1480 (2015)
35. Coello, C.A.C., Montes, E.M.: Constraint-handling in genetic algorithms through the use of dominance-based tournament selection. Adv. Eng. Inform. **16**(3), 193–203 (2002)