PopRing: A Popularity-aware Replica Placement for Distributed Key-Value Store

Denis M. Cavalcante, Victor A. Farias, Flávio R. C. Sousa,
Manoel Rui P. Paula, Javam C. Machado and Neuman Souza
LSBD, Departament of Computer Science, Federal University of Ceara, Fortaleza, Brazil

Keywords: Replica Placement, Consistent Hashing, Data Access Skew, Multi-objective Optimization, Genetic Algorithm.

Abstract: Distributed key-value stores (KVS) are a well-established approach for cloud data-intensive applications, but they were not designed to consider workloads with data access skew, mainly caused by popular data. In this work, we analyze the problem of replica placement on KVS for workloads with data access skew. We formally define our problem as a multi-objective optimization and present the PopRing approach based on genetic algorithm to find a new replica placement scheme. We also use OpenStack-Swift as the baseline to evaluate the performance improvements of PopRing under different configurations. A moderate PopRing configuration reduced in 52% the load imbalance and in 32% the replica placement maintenance while requiring the reconfiguration (data movement) of only 6% of total system data.

1 INTRODUCTION

Distributed key-value stores (KVS) are a well-established approach for cloud data-intensive applications. Their success came from the ability to manage huge data traffic driven by the explosive growth of social networks, e-commerce, enterprise and so on. In this paper, we focus on the particular type of KVS which can ingest and query any type of data, such as photo, image and video. This type of KVS is also called object store, such as Dynamo (DeCandia et al., 2007) and OpenStack-Swift (Chekam et al., 2016). These systems evolved to take advantage of peer-to-peer and replication techniques to guarantee scalability and availability, however, they are not efficient for dynamic workloads with data access skew, once their partitioning technique, based on distributed hash table (DHT) and consistent hashing with virtual nodes (CHT), assumes uniformity for data access (DeCandia et al., 2007) (Makris et al., 2017).

Data access skew is mainly a consequence of popular data (hot data) due to high request frequency. Previous works suggest that popular data is one of the key reasons for high data access latency and/or data unavailability in cloud storage systems (Makris et al., 2017). The authors (Mansouri et al., 2017) affirm that a data placement algorithm should dynamically load balance skewed data access distribution so that all servers handle workloads almost equally. To overcome that limitation, the reconfiguration of replica placement is necessary, although it requires data movement throughout the network. Minimizing load imbalance and replica reconfiguration are NP-hard (Zhuo et al., 2002).

Additionally to the mentioned challenges, there is the replica maintenance of cold data where considerable storage and bandwidth resources may be wasted at keeping too many replicas of data with low request frequency, i.e., unnecessary replicas. To get worse, the authors (Chekam et al., 2016) affirm that the data synchronization of too many replicas is not a good choice due to network overhead.

From the discussed issues of hot and cold data in KVS systems, important questions should be answered: should be data migrated and/or replicated? Which node should be the new host of the replicated/migrated data? Could replica maintenance and reconfiguration costs be minimized while still minimizing the load imbalance of *Get* requests submitted to the system during last observed time?

To the best of our knowledge, our work is the first to answer those questions at investigating the trade-off between load balance, replica maintenance and replica placement reconfiguration for KVS systems based on CHT partitioning.

The major contributions of this paper are as follows:

M. Cavalcante, D., Farias, V., Sousa, F., P. Paula, M., Machado, J. and Souza, N.
PopRing: A Popularity-aware Replica Placement for Distributed Key-Value Store.
DOI: 10.5220/0006703504400447
In *Proceedings of the 8th International Conference on Cloud Computing and Services Science (CLOSER 2018)*, pages 440-447
ISBN: 978-989-758-295-0
Copyright © 2019 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

- A modeling of the multi-objective problem of minimizing load imbalance, replica placement maintenance and replica placement reconfiguration costs on KVS systems based on CHT partitioning;
- The PopRing strategy for replica placement on KVS systems based on CHT partitioning as well its implementation and evaluation against the replica placement of OpenStack-Swift in a simulation-based environment.

Organization: This paper is organized as follows: Section 2 discusses related work. Section 3, provide background information. Section 4, defines the system model and formalizes the multi-objective optimization problem. Section 5 explains PopRing's theoretical aspects and implementation of the solution. Section 6 compares PopRing against the baseline. Section 7 presents the conclusions.

2 RELATED WORK

In this section, we contrast our work with existing works on replica placement problem by discussing their characteristics as well as their solutions.

(Long et al., 2014) aimed to find a suitable replica placement to improve five objectives including load imbalance, but they did not consider the minimization of replica placement maintenance and reconfiguration costs. Another difference is that their approach was only evaluated with a few number of nodes and files. (Mseddi et al., 2015) clarify that replica placement systems may result in a huge number of data replicas created or migrated over time between and within data centers. Then, they focused on minimizing the time needed to copy the data to the new replica location by avoiding network congestion and ensuring a minimal replica unavailability. (Liu and Shen, 2016) proposed a solution to handle both correlated and non-correlated machine failures considering data access skew, availability and maintenance cost. (Long et al., 2014), (Mseddi et al., 2015), (Liu and Shen, 2016) did not consider DHT/CHT techniques when proposing their solutions, thus it is not possible to know if their solutions are efficient for KVS based on DHT/CHT.

(Mansouri et al., 2017) evaluated the trade-off between network and storage cost to optimize pricing cost of replication and migration data across multiple cloud providers. Beyond that, their replication factor is the same for all data objects and is not adaptive to data popularity. (Makris et al., 2017) report that response times of *Get* requests quickly degrade in the presence of workloads with power-law distributions for data access. Then, they defined their objective as the minimization of the average response time of the system under a continuously changing load of *Get* requests. They did not consider replica creation and deletion, thus only focusing on data migration.

3 BACKGROUND

In this section, we present the partitioning, the replication and the architecture of our KVS system which substantiate our system model, optimization problem and replica placement strategy.

3.1 Partitioning and Replication

The partitioning of our system is based on consistent hash with replicated virtual nodes as shown in Fig. 1. Our virtual nodes have the same concept of virtual nodes in Dynamo and partitions in OpenStack-Swift which is an abstract layer for managing all system data into smaller parts, i.e., a set of objects.

The placement of every data object is mapped to one virtual node through the consistent hash function mapping. This mapping is the process of hashing the identification of a data object to calculate its modulo using the total number of virtual nodes defined by the system admin. Our hash function outputs hashed values uniformly distributed, thus balancing the number of objects on every virtual node. The hash function mapping between an object and a virtual node is fixed because the hash function is the same during all system operation.

In our system, the system administrator sets the total number of virtual nodes to a large value at the first deployment of the system and never changes it. Otherwise, it would break the property of the consistent hashing technique by creating the side-effect of huge data movements.

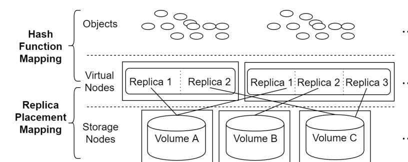


Figure 1: Objects, virtual nodes and storage nodes mappings.

A virtual node can be replicated multiple times into different storage nodes. This mapping of the virtual node replicas and storage nodes is called replica placement scheme where it describes the replication factor and the placement of every virtual node replica

as shown in Fig. 1. That scheme allows dynamically management of data through operations of replica creation, migration and deletion.

3.2 Architecture

Our system architecture is composed by three types of nodes: service node, storage node and coordinator node. The service nodes handle data access requests as a reverse proxy to storage nodes by using the replica placement scheme for data location.

The service nodes accepts write/read operations of data objects by supporting *Put* requests for creating objects creation and *Get* requests for accessing data objects. An object is any unstructured data, i.e., a photo, a text, a video and so on. The system is able to handle any object size and is write-once, read-many. The service nodes use shuffle algorithm for replica selection, thus spreading equally *Get* requests to the virtual node replicas.

The coordinator node is a centralized controller which monitors the total number of *Get* requests per virtual node served by the service nodes. It monitors the available storage capacity of the storage nodes. The coordinator node also maintains a copy of the replica placement scheme in-use by the other nodes.

The main mission of the coordinator node is to use our replica placement strategy to compute periodically and incrementally a new replica placement scheme. The administrator of the coordinator node sets up a period constant to compute and to apply the new scheme into the other nodes as well. An instance of the architecture is shown in Fig. 2.

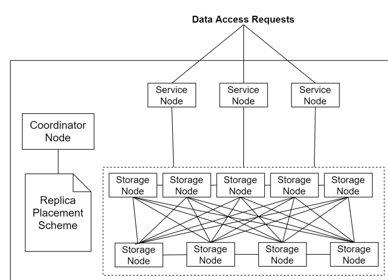


Figure 2: System architecture.

A new replica placement scheme is synchronized by a peer-to-peer asynchronous process in the storage nodes different from the process to serve data access requests. This process aims to synchronize all replicas units of the current replica placement scheme. Every storage node knows exactly which replicas it manages because every node has a copy of the replica placement scheme.

The data availability of the system is maintained by the minimum number of replicas of a virtual node and by the minimum number of replicas to not reconfigure. This last one, it is important to avoid data unavailability due to aggressive replica placement reconfiguration, i.e., all replicas of a virtual node are reconfigured to migrate at the same time.

4 SYSTEM MODEL AND OPTIMIZATION PROBLEM

In this section, we introduce important definitions of our system model and we formalize our objectives as a multi-objective optimization problem.

4.1 System Model

Table 1 gives the meaning of the symbols used in the definitions below.

Definition 4.1 (Storage Nodes Specification)**.** The system is composed of a set of distributed and independent storage nodes D, where each $d \in D$ is a storage node connected to others by a network. Each storage node can receive data until the maximum storage capacity in gigabytes max_stor_d is reached.

Definition 4.2 (Workload Specification)**.** The workload submitted to our system is composed of a set *Get* requests where $virt_node_get_p$ is the total number of *Get* requests targeted to each virtual node $p \in P$, where P is the set of virtual nodes.

Definition 4.3 (Replica Placement Scheme Variable)**.** The replica placement scheme S is a binary matrix of $s_{dp} \in \{0,1\}$ values of size $|D||P|$ where a row represent a storage node $d \in D$ and a column represent a virtual node $p \in P$. A virtual node $p \in P$ is replicated into the storage node $d \in D$ if the value is 1, otherwise is 0. The minimum number of replicas of a virtual node $p \in P$ is min_repl_p and the minimum number of replicas not to be reconfigured of a virtual node $p \in P$ is $min_not_reconf_p$, where both are set by the system administrator.

Our replica placement scheme allows incremental changes to a replica placement scheme already in-use by a KVS system. A smart solution can improve an existent replica placement scheme to evaluate dispensable data redundancy and movement while reducing load imbalance. We use O as a snapshot of a previous replica placement scheme in-use and o_{dp} to represent a cell in O. Both are constant for our model. We also use M as the number of storage nodes $|D|$ and N as the number of virtual nodes $|P|$.

Table 1: Definitions.

Symbol	Meaning
D	A set of storage nodes.
P	A set of virtual nodes.
S	A matrix representation of the replica placement scheme.
O	A snapshot of a previous replica placement scheme in-use.
s_{dp}	A binary cell in the matrix of the replica placement scheme.
o_{dp}	A binary cell of a previous replica placement scheme O.
M	Total of storage nodes $\|D\|$.
N	Total of virtual nodes $\|P\|$.
max_stor_d	Maximum storage capacity in GB of a storage node $i \in D$.
$used_stor_d$	Used storage capacity in GB of a $d \in D$.
min_repl_p	Minimum number of replicas of a virtual node $p \in P$.
$repl_p$	Number of replicas of a virtual node $p \in P$.
$min_not_reconf_p$	Minimum number of replicas of a virtual node $p \in P$ not to reconfigure.
not_reconf_p	Number of replicas of a virtual node $p \in P$ not to reconfigure.
$repl_size_p$	Total size in GB of one replica of a virtual node $p \in P$.
$virt_node_get_p$	Total number of *Get* requests submitted to a virtual node $j \in P$.
$stor_node_get_d$	Total number of *Get* requests submitted to a storage node $d \in D$.
$ideal_get$	Ideal number of *Get* requests to submit to a storage node $d \in D$.
$C_{load_imbalance}$	Total load imbalance cost.
$C_{maintenance}$	Total replica maintenance cost.
$C_{reconfiguration}$	Total replica placement scheme reconfiguration cost.

Definition 4.4 (Replica Placement Maintenance Cost)**.** The replica placement maintenance cost represents indirectly the network delay/overhead caused by the data synchronization of already existing, but dispensable replicas. To conform with this definition, we give a cost in GB to the enabled cells in the previous scheme O that are still enabled in the new scheme S according to the equation 1. This way, during the evaluation of previous and new schemes, a solution can focus on deleting already existing replicas.

$$C_{maintenance} = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} (o_{ij}s_{ij})(repl_size_j) \quad (1)$$

Definition 4.5 (Replica Placement Reconfiguration Cost)**.** The replica placement reconfiguration cost represents indirectly the network delay/overhead caused by the movement/synchronization of replica creation and migration. To conform with this definition, we give a cost in GB to the disabled cells in the previous scheme O that are now enabled in the new scheme S according to the equation 2. This way, during the evaluation of the previous and new schemes, a solution can focus on avoiding replica creation and migration. The reconfiguration cost of replica placement scheme is defined according to the Equation 2.

$$C_{reconfig.} = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} (s_{ij} - o_{ij}s_{ij})(repl_size_j) \quad (2)$$

To better understand the differences between replica placement maintenance and reconfiguration costs, we describe examples bellow considering Tables 2 and 3:

Table 2: Example of Previous Replica Placement Scheme.

O =	$p=0$	$p=1$	$p=2$	...	**P**
$d=0$	1	0	0	...	
$d=1$	1	1	1	...	
$d=2$	1	1	1	...	
$d=3$	0	0	1	...	
...	...	...	...	...	
D					

Table 3: Example of New Replica Placement Scheme.

S =	$p=0$	$p=1$	$p=2$	...	**P**
$d=0$	0	1	0	...	
$d=1$	1	1	1	...	
$d=2$	1	1	1	...	
$d=3$	1	0	0	...	
...	...	...	...	...	
D					

- **Replica Placement Maintenance Example.** Considering the previous scheme O in Table 2, the virtual node $p=2$ had to periodically synchronize its three replicas at the storage nodes $d=1$, $d=2$ and $d=3$. Considering the new scheme S in Table 3, the maintenance cost of the virtual node $p=2$ was reduced from 3 to 2;
- **Replica Placement Reconfiguration Example.** Considering the previous scheme O in Table 2 and considering the new scheme S in Table 3, virtual node $p=0$ has a replica moved from storage node $d=0$ to $d=3$ and virtual node $p=1$ has a new one replicated at storage node $d=0$.

Definition 4.6. (Load Imbalance Cost): The amount of *Get* requests submitted to each storage node $d \in D$ is measured according to the Equation 3.

$$stor_node_get_i = \sum_{j=0}^{N-1} s_{ij}(virt_node_get_j / repl_j) \quad (3)$$

As we mentioned early, D has similar performance capacities, then the ideal data access per storage node is defined according to the Equation 4

$$ideal_get = (\sum_{i=0}^{M-1} stor_node_get_i)/M \quad (4)$$

Finally, to reduce the overload/underload of *Get* requests on every storage node caused by data access skew, we define the data access cost according to the Equation 5

$$C_{load_imbalance} = (\sum_{i=0}^{M-1} |stor_node_get_i - ideal_get|)/M \quad (5)$$

4.2 Problem Formalization

Given the system model as well as the load imbalance, replica maintenance and replica placement reconfiguration costs that were previously defined, we set the goal of the system as the minimization of the three object functions according to the Equation 6:

$$\begin{aligned} \min_S \quad & C_{load_imbalance}, C_{maintenance}, C_{reconfiguration} \\ \text{s.t.} \quad & used_stor_d <= max_stor_d \\ & repl_p >= min_repl_p \\ & not_reconf_p >= min_not_reconf_p \end{aligned} \quad (6)$$

5 PopRing REPLICA PLACEMENT

PopRing is a replica placement strategy for distributed key-value stores with the ability to automatically create, migrate and delete replicas. PopRing aims to minimize the load imbalance, replica placement maintenance and replica placement costs where these different objectives may conflict with each other, requiring optimal tradeoff analyses among the objectives of a system.

The authors (Cho et al., 2017) surveyed many approaches to resolve multi-objective (MOO) problems. The weighted sum (WS) method is computationally efficient in generating a strong non-dominated solution (Cho et al., 2017). We chose WS to minimize the multiple objective functions defined in the previous section by using the weighted sum method to transform the multi-objective optimization problem into the minimization of a unique function F.

By using the WS method, any user has individual control of the importance of each objective as shown in the Equation 7, where C_1, C_2 and C_3 are the importance constants corresponding to the objective functions $C_{load_imbalance}$, $C_{maintenance}$ and $C_{reconfiguration.}$, respectively. This way, it is possible to customize F to adapt the optimization to be computed and applied to the storage nodes periodically with small time intervals between iterations to reduce huge data movements, for example.

$$F = C_1 C_{load_imb.} + C_2 C_{mainten.} + C_3 C_{reconfig.} \quad (7)$$

5.1 Randomized Search

Given a replica placement scheme matrix S with each cell element $\{0,1\}$ and dimension size of $m \times n$ where m is the number storage nodes and n is the number of virtual nodes, the worst-case time complexity for performing brute-force search to evaluate F and find the optimum replica placement has exponential time complexity $O(2^{mn})$.

To substantially reduce the search time while not getting stuck into local optimum at minimizing our function F, we decided to use operators of genetic algorithms (G.A.) such as selection, crossover and mutation to guide the search process. The usage of these operators simulates the survival of the fittest from Darwin's evolutionary theory and generates useful solutions for optimization (Li et al., 2017).

The work (Li et al., 2017) surveyed many different approaches for each genetic algorithm and ranked them according to the most used by the literature. Considering the most popular approaches, PopRing uses the binary coding, the tournament, the single-point, the bit inversion, the total number generation methods for coding, selection, crossover, mutation and termination, respectively. These genetic operators are used by PopRing traditionally according to the literature to generate randomly a population of individuals and update that population during a number of generations to guide the search process to find the best individual, i.e., a new replica placement scheme

To reduce convergence time and maintain population diversity, we add two special individuals to the population to give clues when exploring the search space to find good solutions as quickly as possible. The first one is an individual based on the exact actual replica placement scheme used by the system to reduce the replica migration/creation once this individual has the smaller cost for $C_{reconf.}$ The second one is an individual is based on the actual replica placement, but with randonmly reduced replicas until the min_repl_p. At including this second individual, the randomized search reduces the total replicas of a

new replica placement, because this individual has the smaller cost for $C_{mainten.}$.

5.1.1 Sparse Matrix Improvement

The matrix calculations on the replica placement scheme S have $O(mn)$ complexity where m is the number storage nodes and n is the number of virtual nodes. The total number of virtual nodes may be too high such as 1024, 65536, 1048576 and so on, thus resulting in huge dimensions for the replica placement scheme. These huge dimensions slow the evaluation of F at performing mathematical operations on matrix/vectors structures.

Our approach reduces dispensable data redundancy and reconfiguration, then the percentage of the average of non-zeros in S is very low when the population of individuals is getting closer to the optimum. Near to the optimum, the number of enabled replicas is much lower than the number of virtual nodes $|P|$.

This way, we converted our matrix to the Compressed Row (CSR) format (Grossman et al., 2016) and reduced the time complexity of matrix operations to $O(n)$.

6 EXPERIMENTAL EVALUATION

For evaluating our proposed solution PopRing against the OpenStack-Swift, our simulated environment is described in Section 6.1. Finally, the improvements of our solution under different configurations regarding the importance of the objectives are discussed in Section 6.2.

6.1 Simulated Environment

First, we setup replica placement settings of the OpenStack-Swift as 3, 50, 1024 for replication factor, number of storage nodes and number of virtual nodes, respectively. Then, we simulated the creation of 300 thousand objects using Zipf law with its exponent 1.1 for object size and the submission of 1 million *Get* requests using Zipf distribution to represent different data popularity levels according to (Liu et al., 2013). For the problem constraints described in Section 4, we used the maximum storage capacity of storage nodes, the minimum replication factor of virtual node and minimum replicas not to reconfigure are set to 500 GB, 2, 1 respectively.

For the setup of the evolutionary parameters of PopRing, we used 1000, 50, 3, 0.5, 0.1 and 0.0005 for generation size, population size, tournament size, cross-over rate, mutation rate and gene mutation rate, respectively. We used the versions 1.0.2 of DEAP, 0.19.1 of scipy libraries and Mitaka to perform evolutionary algorithms, matrix calculations and OpenStack-Swift baseline, respectively. Our algorithm was been processed on a desktop computer with core i7 3.40GHz and 16GB memory, but it required much less computer resources than the maximum capacity and took less than 2 minutes to finish.

Table 4: Popring Parameters.

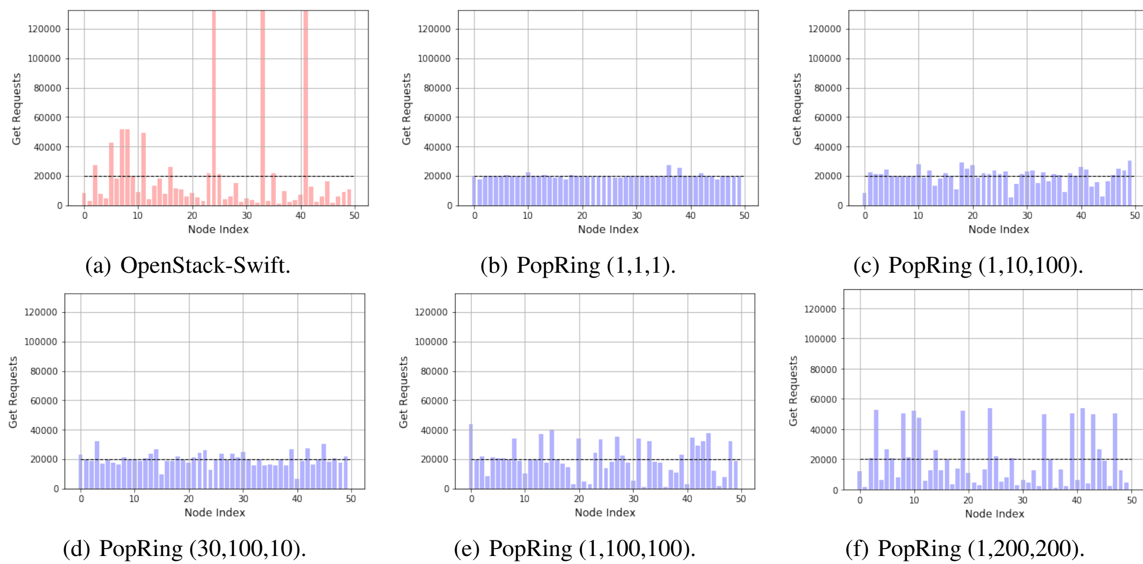
(C_1, C_2, C_3)	**Importance**
(1, 1, 1)	**Low** maintenance and reconfiguration.
(1, 10, 100)	**Low** maintenance and **moderate** reconfiguration.
(1, 100, 10)	**Moderate** maintenance and **low** reconfiguration.
(1, 100, 100)	**Moderate** maintenance and reconfiguration.
(1, 200, 200)	**High** maintenance and reconfiguration.

To evaluate our strategy, we experimented PopRing under different configurations. We organized these configurations by keeping constant the importance of the load imbalance cost and varying the importance of other costs. Low, moderate and high represent the level of importance of each objective. The values of function costs are not normalized, thus we adjusted C_1, C_2 and C_3 to represent the levels described at the Table 4.

6.2 Results

Fig. 3 shows the percentage of *Get* requests each storage node has to handle in comparison to the total *Get* requests submitted to the system. For our experiment, the ideal load per storage node is 20000 *Get* requests according to the Equation 4. Fig. 3(a) shows that the Swift baseline overloads three storage nodes by submitting to them around 30% of the system total load while the majority of the storage nodes manages each one less than 1% of total system load.

Considering the configuration (1, 1, 1), it is possible to verify that PopRing obtained a replica placement with only 746.95 *Get* requests of load imbalance, i.e., almost the ideal line of *Get* requests per storage node. This performance on load balance is obtained because PopRing configuration is able to dedicate much more importance to the load imbalance problem than the replica placement maintenance and reconfiguration as shown in Fig. 3(b). The configurations (1, 10, 100) and (1, 100, 10) had similar load imbalance of 3980.15 and 3246.52 as shown in figures 3(c) and 3(d). The configurations (1, 100, 100)



(a) OpenStack-Swift. (b) PopRing (1,1,1). (c) PopRing (1,10,100).

(d) PopRing (30,100,10). (e) PopRing (1,100,100). (f) PopRing (1,200,200).

Figure 3: Total *Get* Requests Per Storage Node.

and (1, 200, 200) obtained 8984.42 and 14667.23 of load imbalance, respectively as shown in figures 3(e) and 3(f). The most conservative PopRing configuration (1, 200, 200) still had good performance at reducing the three most overloaded nodes to less than 50% of their previous loading.

Fig. 4 represents the percentage of the amount of data according to their replication factor. The configuration (1, 1, 1) has the higher increase for replication cost due to the low importance given to replica maintenance and replica placement reconfiguration costs. The configuration (1, 10, 100) decreased only less than 5% of virtual nodes to only two replicas and required less than 10% of virtual nodes to increase their number of replicas. In contrast, the configuration (1,100,10) reduced almost 20% of virtual nodes to only two replicas and and required almost 20% of virtual nodes to increase their number of replicas.

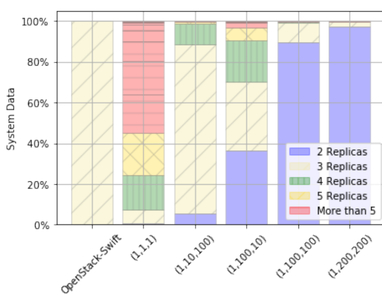


Figure 4: Replication Factor Evaluation.

Our system has a minimum replication factor which limits the amount of data redundancy which can be reduced. It is possible to confirm that limit at comparing (1,100,100) and (1,200,200), where the performance improvement of data maintenance cost has not changed significantly.

PopRing reduced the load imbalance in 96%, 79%, 83%, 52% and 22% while reducing the maintenance cost of current replicas in 8%, 2%, 36%, 33% and 33% for the configurations (1, 1, 1), (1, 10, 100), (1, 100, 10), (1, 100, 100) and (1, 200, 200), respectively as shown at Fig. 5. PopRing also required the reconfiguration of 54%, 5%, 38%, 6%, 1% of total system data for the configurations (1, 1, 1), (1, 10, 100), (1, 100, 10), (1, 100, 100) and (1, 200, 200)., respectively as shown at Fig. 6. These results make possible to understand that the performance of load imbalance cost is impacted by the other two objectives.

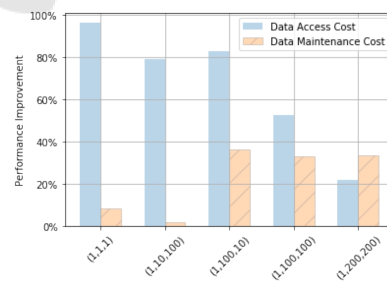


Figure 5: Relative Costs: How much load imbalance (*Get* requests) and replica maintenance costs were reduced in comparison to the replica placement of the OpenStack-Swift. And how much data movement relative to the total previous total storage was needed by our replica placement scheme.

In Fig. 5, it is verified a decline in the load balance performance and a rising in the replica main-

tenance performance. The same applies for replica placement reconfiguration as shown in Fig. 6. The increase in the importance of replica maintenance and replica reconfiguration make the load imbalance more difficult to minimize. Figures 5 and 6 shown the trade-offs among load imbalance, replica placement maintenance and replica placement reconfiguration objectives.

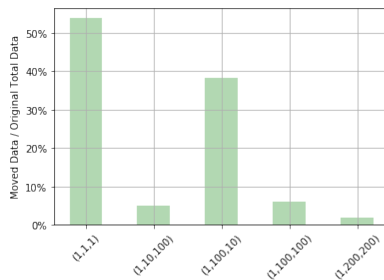


Figure 6: Total data movement relative to the total previous replica placement scheme.

7 CONCLUSION AND FUTURE WORK

In this work, we analyzed the problem of replica placement on KVS systems based on consistent hashing with virtual nodes for workloads with data access skew. We formally defined our problem as a multi-objective optimization and presented the PopRing approach based on genetic algorithm to solve the multi-objective optimization.

Finally, we compared PopRing against the OpenStack-Swift replica placement under different configurations. In most configurations, PopRing could balance workloads with data access skew while reducing unnecessary data redundancy and movement. A moderate PopRing configuration reduced in 52% the load imbalance and in 32% the replica placement maintenance while requiring the reconfiguration (data movement) of only 6% of total system data. As future work, we intend to evaluate PopRing not only on simulated environment, but on real deployments as well while extending it to consider dynamic workloads with restrictive agreements for service quality.

ACKNOWLEDGEMENTS

This work was partially funded by Lenovo, as part of its R&D investment under Brazil's Informatics Law, and also by LSBD/UFC.

REFERENCES

Chekam, T. T., Zhai, E., Li, Z., Cui, Y., and Ren, K. (2016). On the synchronization bottleneck of openstack swift-like cloud storage systems. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pages 1–9. IEEE.

Cho, J.-H., Wang, Y., Chen, R., Chan, K. S., and Swami, A. (2017). A survey on modeling and optimizing multi-objective systems. *IEEE Communications Surveys & Tutorials*.

DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W. (2007). Dynamo: amazon's highly available key-value store. *ACM SIGOPS operating systems review*, 41(6):205–220.

Grossman, M., Thiele, C., Araya-Polo, M., Frank, F., Alpak, F. O., and Sarkar, V. (2016). A survey of sparse matrix-vector multiplication performance on large matrices. *arXiv preprint arXiv:1608.00636*.

Li, T., Shao, G., Zuo, W., and Huang, S. (2017). Genetic algorithm for building optimization: State-of-the-art survey. In *Proceedings of the 9th International Conference on Machine Learning and Computing*, pages 205–210. ACM.

Liu, J. and Shen, H. (2016). A low-cost multi-failure resilient replication scheme for high data availability in cloud storage. In *High Performance Computing (HiPC), 2016 IEEE 23rd International Conference on*, pages 242–251. IEEE.

Liu, S., Huang, X., Fu, H., and Yang, G. (2013). Understanding data characteristics and access patterns in a cloud storage system. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 327–334. IEEE.

Long, S.-Q., Zhao, Y.-L., and Chen, W. (2014). Morm: A multi-objective optimized replication management strategy for cloud storage cluster. *Journal of Systems Architecture*, 60(2):234–244.

Makris, A., Tserpes, K., Anagnostopoulos, D., and Altmann, J. (2017). Load balancing for minimizing the average response time of get operations in distributed key-value stores. In *Networking, Sensing and Control (ICNSC), 2017 IEEE 14th International Conference on*, pages 263–269. IEEE.

Mansouri, Y., Toosi, A. N., and Buyya, R. (2017). Cost optimization for dynamic replication and migration of data in cloud data centers. *IEEE Transactions on Cloud Computing*.

Mseddi, A., Salahuddin, M. A., Zhani, M. F., Elbiaze, H., and Glitho, R. H. (2015). On optimizing replica migration in distributed cloud storage systems. In *Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on*, pages 191–197. IEEE.

Zhuo, L., Wang, C.-L., and Lau, F. C. (2002). Load balancing in distributed web server systems with partial document replication. In *Parallel Processing, 2002. Proceedings. International Conference on*, pages 305–312. IEEE.