UNIVERSIDADE FEDERAL DO CEARÁ
CENTER DE CIÊNCIAS
DEPARTAMENTO DE COMPUTAÇÃO
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
MESTRADO ACADÊMICO EM COMPUTAÇÃO

DENIS MORAIS CAVALCANTE

POPRING: A POPULARITY-AWARE REPLICA PLACEMENT FOR REDUCING LATENCY ON DISTRIBUTED KEY-VALUE STORES

FORTALEZA
2018

DENIS MORAIS CAVALCANTE

POPRING: A POPULARITY-AWARE REPLICA PLACEMENT FOR REDUCING LATENCY ON DISTRIBUTED KEY-VALUE STORES

Dissertation submitted to the Postgraduate Program in Computer Science of the Center of Science of the Universidade Federal do Ceará, as a partial requirement to obtain the Academic Master Degree in Computer Science. Concentration Area: Ciência da Computação.

Advisor: Prof. Dr. Jose Neuman de Souza.

Co-advisor: Prof. Dr. Javam de Castro Machado.

FORTALEZA

2018

Dados Internacionais de Catalogação na Publicação
Universidade Federal do Ceará
Biblioteca Universitária
Gerada automaticamente pelo módulo Catalog, mediante os dados fornecidos pelo(a) autor(a)

C364p Cavalcante, Denis.
PopRing: A Popularity-aware Replica Placement for Reducing Latency on Distributed Key-Value Stores / Denis Cavalcante. – 2018.
71 f. : il. color.

Dissertação (mestrado) – Universidade Federal do Ceará, Centro de Ciências, Programa de Pós-Graduação em Ciência da Computação, Fortaleza, 2018.
Orientação: Prof. Dr. José Neuman de Souza.
Coorientação: Prof. Dr. Javam de Castro Mchado.

1. Distributed Key-value Store. 2. Replica Placement. 3. Load Balancing. 4. Genetic Algorithm. 5. Bayesian Optimization. I. Título.
CDD 005

DENIS MORAIS CAVALCANTE

POPRING: A POPULARITY-AWARE REPLICA PLACEMENT FOR REDUCING LATENCY ON DISTRIBUTED KEY-VALUE STORES

Dissertation submitted to the Postgraduate Program in Computer Science of the Center of Science of the Universidade Federal do Ceará, as a partial requirement to obtain the Academic Master Degree in Computer Science. Concentration Area: Ciência da Computação.

Approved on: September 03, 2018

EXAMINERS COMMITEE

Prof. Dr. Jose Neuman de Souza (Advisor)
Universidade Federal do Ceará (UFC)

Prof. Dr. Javam de Castro Machado (Co-advisor)
Universidade Federal do Ceará (UFC)

Prof. Dr. Leonardo Oliveira Moreira
Universidade Federal do Ceará (UFC)

Prof. Dr. Joaquim Celestino Junior
Universidade Estadual do Ceará (UECE)

Dedicated to my mother and my grandmother for the support and encouragement from the beginning of my life as well as my wife Soraya who accepted to marry me during the same period of time that I started the master’s degree therefore staying at my side even during the most difficult moments.

ACKNOWLEDGEMENTS

Special thanks to my advisors, Prof. Dr. Javam Machado and Prof. Dr. Neuman Souza, for the advice, guidance and opportunities granted that were fundamental to the success of this dissertation.

To the management of the Laboratório de Sistemas e Banco de Dados (LSBD) for the support and confidence that allowed me to enter and accomplish the master's degree.

To all the friends who have already passed and still are in the Laboratório de Sistemas e Banco de Dados (LSBD), for the pleasant coexistence of the last years and for the valuable collaboration during the practice of research and development.

To the friends Manoel Rui, Victor Farias, Serafim Costa, Elvis Teixeira, Sergio Marinho, Fernando Dione, Flávio Sousa, Leonardo Moreira, Eduardo Rodrigues, Paulo Amora, Felipe Timbó and Vanessa Rodrigues for the considerations and discussions throughout my master's degree.

To the Universidade Federal do Ceará for being public and free.

To Lenovo for the funding of master's degree research through employment relationship.

And to the examiners commitee who agreed to evaluate this work.

“Imagination is more important than knowledge.”

(Albert Einstein)

RESUMO

O armazenamento distribuído em chave-valor (KVS) é uma abordagem bem estabelecida para aplicações com uso intensivo de dados na nuvem. Contudo, este tipo armazenamento não foi projetado para considerar cargas de trabalho com acesso desbalanceado aos dados devido principalmente a dados populares. Na presente pesquisa, foi feita uma análise do problema de alocação de réplicas no KVS para cargas de trabalho com acesso desbalanceado aos dados. O problema é definido formalmente como um problema de otimização multiobjetivo, pois além do custo de desequilíbrio de carga, também existem os custos de manutenção e reconfiguração de réplicas, que afetam o desempenho do sistema. Para resolver o problema de alocação da réplica, nós propomos o componente de alocação de réplicas PopRing. Esse componente, baseado em algoritmos genéticos, busca de forma eficiente novas alocações de réplica. Em seguida, a estrutura PopRing foi estendida com um componente de otimização de hiper-parâmetros baseado em otimização bayesiana, de modo a encontrar com eficiência a importância adequada das funções objetivas de desbalanceamento de carga, manutenção de réplica e reconfiguração de acordo com a latência do sistema. Para validar o PopRing, foi implementado um protótipo completo a fim de gerar novos esquemas de alocação de réplica no formato da interface distributed hash table (DHT) de armazenamento de objetos OpenStack-Swift. Em seguida, em nosso ambiente de experimentação, foi configurado um cluster distribuído do OpenStack-Swift, um nó de referência do benchmark e o protótipo PopRing para executar alguns experimentos. A partir da avaliação dos resultados, verificou-se que a solução conseguiu reduzir a latência do sistema para diferentes níveis de desbalanceamento de acesso a dados sem intervenção humana, ou seja, ajustou seus parâmetros automaticamente para encontrar um esquema de alocação de réplica apropriado para um dado cenário.

Palavras-chave: Armazenamento Chave-valor Distribuído. Alocação de Réplica. Balanceamento de Carga. Algoritmo Genético. Otimização Bayesiana.

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 problem because not only load imbalance cost, but replica maintenance and reconfiguration costs affect system performance as well. To solve the replica placement problem, we present the PopRing replica placement component based on Genetic algorithms to find new replica placements efficiently. Next, we extend PopRing framework with a hyper-parameter optimization component based on Bayesian optimization in order to efficiently find the proper importance of load imbalance, replica maintenance, and reconfiguration objectives according to the system latency. To validate our PopRing engine in practice, we implemented a full prototype of PopRing to generate new replica placement schemes in the format of the distributed hash table (DHT) interface of the popular object store OpenStack-Swift. Then, in our lab environment, we deployed a distributed cluster of the OpenStack-Swift, a benchmark node and the PopRing prototype to run some experiments. From results evaluation, we verified that our solution was able to reduce system latency for different levels of data access skew without human intervention, i.e., PopRing auto-tuned its parameters to find a proper replica placement scheme to a given scenario.

Keywords: Distributed Key-value Store. Replica Placement. Load Balancing. Genetic Algorithm. Bayesian Optimization.

LIST OF FIGURES

LIST OF TABLES

GLOSSARY

BO	Bayesian optimization
CHT	Consistent hashing table
DFE	Directory for exceptions
DHT	Distributed hash table
EI	Expected improvement
GA	Genetic algorithm
GP	Gaussian process
KVS	Distributed key-value stores
P2P	Peer-to-peer
RSM	Response surface methodology
UCB	Upper confidence bound

LIST OF SYMBOLS

$\Re^d$	d-Dimensional Euclidean space.
$\mathbf{A} \in \Re^d$	Feasible region
$\mathbf{x} \in \Re^d$	A vector in d-dimensional space
m	Number of objective functions
$f(\vec{x}), f(\vec{x}) = (f_1(\vec{x}), ..., f_m(\vec{x}))^T$	Vector-valued objective function
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.

B	A set of service nodes.
$C1$	It is a PopRing parameter and it weights the total load imbalance cost on PopRing.
$C2$	It is a PopRing parameter and it weights the total replica maintenance cost on PopRing.
$C3$	It is a PopRing parameter and it weights the total replica placement reconfiguration cost on PopRing.
MRT_{bd}	It is a mean response time data point collected between a service node $b \in B$ and a storage node $d \in D$ during a get request.
$AverageMRT(_{C_1,C_2,C_3,t})$	It is the average of all MRT_{bd} collected from all service and storage nodes at a given time $t \in T$ with the system is using a replication scheme with C_1, C_2 and C_3 importance weights.
$Original\_AverageMRT$	It is the $AverageMRT(_{C_1,C_2,C_3,t})$ collected from all service and storage nodes at right before any PopRing adjustment.
$TotalDatapointsCollected$	Total of $AverageMRT(_{C_1,C_2,C_3,t})$ data points collected since the new scheme deployment until system stabilization.
C_{MRT}	Mean response time cost. This is the objective function to be minimized based on system performance.

CONTENTS

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 work, 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 (P2P) networks 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 table (CHT), assumes uniformity for data access (DECANDIA *et al.*, 2007) (MAKRIS *et al.*, 2017b).

Data access skew is mainly a consequence of popular data (hot data) due to high request frequency. Previous works, such as (MAKRIS *et al.*, 2017b) and its references, suggest that popular data is one of the key reasons for high data access latency and/or data unavailability in cloud storage systems. 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 aforementioned 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 make matters 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.

1.1 Contributions

The major contributions of this work are grouped in two works as described in Sections 1.1.1 and Section 1.1.2:

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

- A modeling of the multi-objective problem of minimizing load imbalance, replica placement maintenance and replica placement reconfiguration costs for KVS based on DHT with virtual nodes and consistent hashing;
- A PopRing replica placement component based on Genetic algorithm (GA) to solve our multi-objective replica placement problem;
- An implementation and experimentation of the PopRing replica placement component in a simulated-based environment;
- An investigation of the trade-off regarding the minimization of load imbalance, replica placement maintenance and replica placement reconfiguration objectives;

1.1.2 PopRing Hyper-parameter Optimization

- A modeling of the optimization problem of minimizing system latency while manipulating our replica placement hyper-parameters, i.e., the importance of the load imbalance, replica placement maintenance and reconfiguration objectives;
- A PopRing hyper-parameter optimization component based on Bayesian optimization (BO) to solve our objective of hyper-parameter optimization problem;
- An implementation and evaluation of the PopRing hyper-parameter component in our cloud environment by deploying an OpenStack-Swift cluster;
- An investigation of the minimization of the system latency under different levels of data access skew;

1.2 Scientific Papers

During the development of this work, two papers were published according to Table 1.

Table 1 – Published Papers.

Conference	Title
CLOSER 2018	HIOBS: A Block Storage Scheduling Approach to Reduce Performance Fragmentation in Heterogeneous Cloud Environments
CLOSER 2018	PopRing: A Popularity-aware Replica Placement for Distributed Key-Value Store

1.3 Dissertation Structure

Organization: This paper is organized as follows: Section 2 provides background information. Section 3 discusses related work. Section 4 proposes the replica placement component of the PopRing and evaluates the approach. Section 5 proposes the hyper-parameter optimization component of the PopRing and evaluates the approach. Section 6 presents the final conclusions of the PopRing approach and suggests future work.

2 BACKGROUND

This section 2 aims to prepare the reader with background information for understanding our problem context and formulation as well as our solutions. Firstly, we give an introduction to DHT systems which form the core technology behind distributed key-value stores. Next, we discuss the concept of "load", associated terms and load balancing approaches for DHT systems.

After the minimum background about DHT systems, we specialize our background section to key aspects of distributed key-value store systems. This type of system is the one used to validate our work and its peculiarities regarding partitioning, replication and architecture beyond being important for understanding our problem formulations. Finally, we introduce the usage of artificial intelligence algorithms for optimization problems in which we focus on GA and BO because they are the underlying technology of our solutions.

2.1 Distributed Hash Table

P2P systems represent a radical shift from the classical client-server paradigm, in which a centralized server processes requests from all clients (FELBER *et al.*, 2014). DHT is a type of structured P2P systems which have the property of associating a unique identify (key) to a node of the system, thus making a DHT mapping of keys and nodes.

The space of keys of a DHT is partitioned among the nodes, thus decentralizing the responsibility of data placement look-up. As a result of the DHT management, an overlay network is built on top of the physical network on which the nodes are linked. These links may be based on IP-based P2P systems, wherein a peer may communicate directly with every other peer. While IP-based P2P systems are researched in this work, mobile ad-hoc networks are out of scope. To support this overlay, the DHT component must support two functions:

- Put (key, data): Store data into the node associated with a key;
- Get (key, data): Retrieve data from the node associated with a key.

2.1.1 Object and Request Load

Our understanding of object, request load and resources are similar to the concepts defined by the authors (FELBER *et al.*, 2014):

- Object: a piece of information stored in the system. An object has at least its identification and size as meta-data. The popularity of an object is the frequency at which it is accessed.

- Request load: a number of object requests per time unit in which each object request consumes storage space, processing time, bandwidth and disk throughput resources of the system during the request and after the request is completed.
- Resources: system resources have limited capacity in terms of available storage space, processing time, bandwidth and disk throughput. This work refers Get requests for retrieving data and Put requests for inserting data to the system through a web API component.

2.1.2 Load balancing in DHT systems

The decentralized structure of DHT overlays requires careful design of load balancing algorithms to fairly distribute the load among all participating nodes. For systems based on DHT, name-space, request rate, routing at overlay and underlay are important aspects to understand how DHT designs are affected by load balancing issues and approaches as described in Table 2. Sections 2.1.2.1 and 2.1.2.2 discuss in detail each aspect of our work regarding the literature of load balancing approaches for DHT systems according to the classification proposed by (FELBER *et al.*, 2014):

Table 2 – Important aspects for DHT load balancing.

Aspect	Meaning
Name-space	It refers to how objects are mapped to keys
Request rate	It refers to how request load is distributed over the keys
Overlay routing	It refers to how keys are mapped to the nodes of the system.
Underlay routing	It refers to how the proximity distance of the underlay nodes matches with the proximity distance of overlay nodes.

2.1.2.1 Load balancing issues

- Name-space balancing: When nodes and keys are not uniformly distributed over the identifier space, objects are expected to be shared between nodes in a skewed manner, thus causing heavy load on some nodes. One classical approach to achieving this property is the name-space balancing (e.g., hashing the IP address of a node or an object name). The DHT adopted in this work uses consistent hashing for name-space balancing.

- Request Rate: The overlays are generally designed to be well balanced under a uniform flow of requests, i.e., with objects having equal popularity, all nodes receive a similar amount of requests. In practice, many workloads have not equal popularity as mentioned in Section 1. By this means, skewed request rate is the main issue which our work aims to solve.
- Overlay routing: each node maintains in its "routing table" the list of outgoing links leading to its immediate neighbors. Nodes with a huge number of incoming links are thus expected to receive on more requests than other nodes, thus causing load imbalance as well. In order to fairly share the traffic load in the overlay, the routing tables should be organized in such a way that the number of incoming links per node is balanced. In our context, replicas means multiple links to the same key, thus the request load on each replica is fairly distributed.
- Underlay routing: the underlying topology is also an important aspect for building the routing tables (e.g., immediate neighbors in the overlay routing which are placed in distant regions in the underlay may cause an unexpected delay for those that are not aware of this discrepancy. Underlay routing is out of scope of our work.

2.1.2.2 Load balancing approaches

According to the literature, there are many approaches to handle load balance issues regarding name-space, request rate, overlay and underlay routing as described in Figure 1. Our work focuses on the replication mechanism for solving the object placement.

Object placement basically deals with node and key placement in the identifier space, key to node mapping, the physical location of objects, and the size and popularity of objects. In this work, we generalized the term object placement to replica placement and focus our efforts to improve the replica placement of our system.

2.1.3 Consistent hashing

It is a method of name-space balancing presented by (KARGER *et al.*, 1997). It introduces an appropriate hashing function (e.g., SHA-1) uniformly distributing identifiers and keys over the identifier space. Every node independently uses this function to choose its own identifier. The basic idea of consistent hashing is relatively simple, since each node or object gets its ID by means of the predefined hashing function, e.g., SHA-1. Unfortunately, the usage of

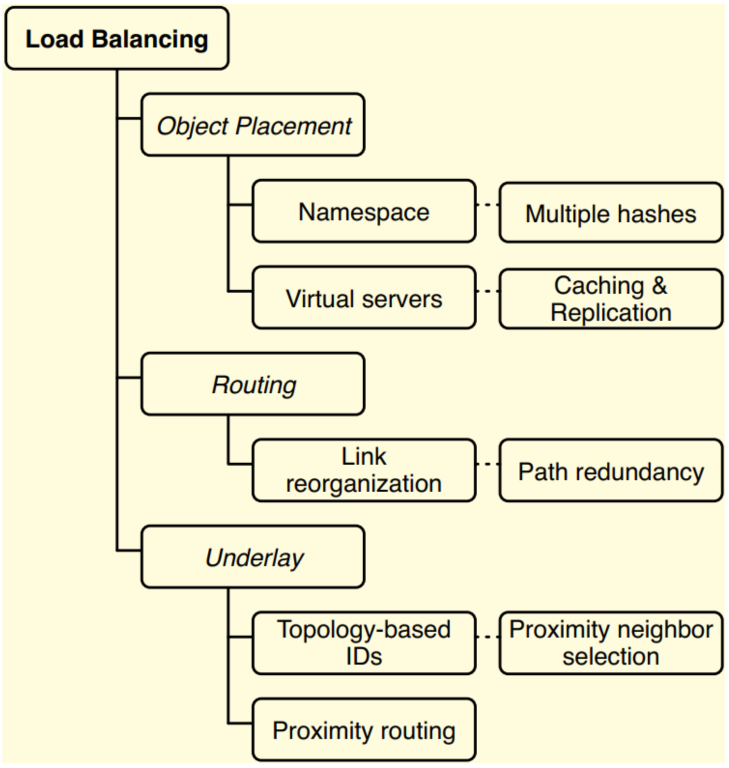


Figure 1 – Load balance solutions for DHT. Source: (FELBER *et al.*, 2014)

only consistent hashing may have the side-effect of having a long interval of keys being managed by a single node. To overcome that, the literature has proposed the usage of consistent hashing with virtual servers.

2.1.4 Virtual servers

Virtual servers or virtual nodes were first introduced by (STOICA *et al.*, 2001) to uniformly distribute identifiers over the addressing space, because the probability of a node to be responsible for a long interval of keys decreases. Beyond that virtual servers give the capacity of activating an arbitrary number of virtual servers per physical node, which is proportional to the peer capacity.

2.2 Distributed Key-Value Store

2.2.1 Partitioning and Replication

The partitioning of the adopted KVS system in this work is based on consistent hash with replicated virtual nodes as shown in Fig. 2. A virtual node or virtual server represents a set of objects mapped to a unique key in the DHT. This way, objects are directly mapped to virtual nodes rather than physical nodes.

Our virtual nodes also have the same concept of the virtual nodes in Dynamo KVS and the partitions in OpenStack-Swift KVS, i.e., they form an abstract layer of data management in which parts of system data can be managed without affecting other data. 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 regarding 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.

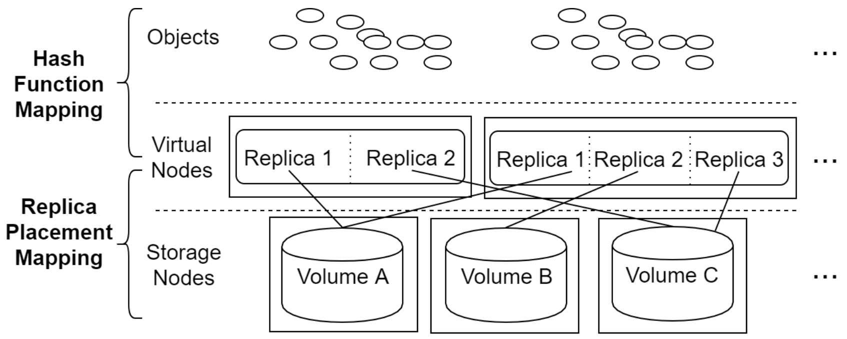


Figure 2 – Objects, virtual nodes and storage nodes mappings.

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. 2. That scheme enables data management operations such as replica creation, migration and deletion. This way, a physical node hosts one or more virtual nodes. As example of load balancing by migration, virtual nodes may be migrated from heavily loaded physical nodes to lightly loaded physical nodes.

2.2.2 Object Hash Collision

As mentioned before, the consistent hashing function maps objects to virtual nodes through hash operation. The hash operation produces hash collision of a set of different objects mapped to the same virtual node key. To proper handle those hash collisions, each storage node is based on traditional file-systems and block storage layers. This way, different objects with the same key are mapped to the same virtual node and stored into storage nodes without issues.

2.2.3 Architecture

The targeted system architecture of our work is composed of 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”. In this work, each service node not only receives a similar number of *Get* requests from the service users. A service node also spread equally the *Get* requests submitted to a virtual node, i.e., each replica of a virtual node is demanded equally.

The coordinator node is a centralized controller responsible for generating new replica placement schemes as well as deploying them into the other nodes. It maintains a copy of the replica placement scheme in-use by the other nodes. It also monitors the total number of *Get* requests per virtual node served by the service nodes as well as the available storage capacity of the storage nodes. The coordinator node works independently of the other nodes and it is not required for the meeting of the new scheme, i.e., the other nodes uses only its copy of the new replica placement scheme to meet the new replica placement. An instance of the architecture is shown in Fig. 3 in which the generated a new replica placement scheme based on

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

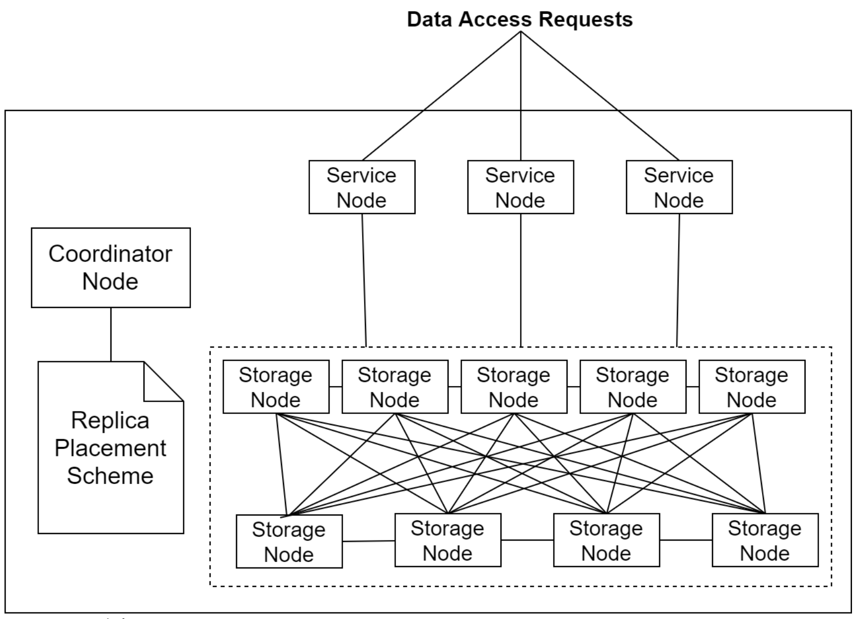


Figure 3 – System architecture.

of a virtual node and by the minimum number of replicas not to reconfigure. This minimum number of replicas not to reconfigure may be used to avoid data unavailability due to aggressive replica placement reconfiguration, i.e., a new scheme which requires all replicas of a highly accessed virtual node to be migrated at the same time.

2.2.4 Consistency Model

Ideally, a distributed system should be an improved version of a centralized system in which fault-tolerance and scalability are proper handled. In fact, this goal is obtained because data may be replicated all over the system, but within the limitations set by the CAP Theorem (GILBERT; LYNCH, 2002). Briefly, CAP Theorem stated that in a distributed system, only two of three following aspects can be met simultaneously: consistency, availability, and partition tolerance. In the case of the type of systems targeted in this work, non-transactional distributed key-value store systems, high availability and partition tolerance are prioritized.

The design decision of prioritizing high availability and partition tolerance may raise a question: how much consistency is sacrificed? According to the literature, there are at least the following levels of consistency:

- Strong Consistency: The gold standard and the central consistency model for non-transactional systems is linearizability, defined by (HERLIHY; WING, 1990). Roughly speaking, linearizability is a correctness condition that establishes that each operation shall appear to be applied instantaneously at a certain point in time between its invocation and its response (VIOTTI; VUKOLIĆ, 2016).

- Weak Consistency: It is the contrary of strong consistency, which does not guarantee that reads return the most recent value written. Last, but no least important, it does not provide ordering guarantees hence, no synchronization protocol is actually required. For example: relaxed caching policies that can be applied across various tiers of a web application, or even the cache implemented in web browsers (VIOTTI; VUKOLIĆ, 2016).
- Eventual Consistency: replicas converge toward identical copies in the absence of further updates. In other words, if no new write operations are performed on the object, all read operations will eventually return the same value (TERRY *et al.*, 1994) (VOGELS, 2008) (VIOTTI; VUKOLIĆ, 2016).

2.2.4.1 Eventual Consistency Implications

As mentioned before, in order to offer high data availability and durability, KVS-like systems such as OpenStack-Swift and Amazon DinamoDB typically replicate each data object across multiple storage nodes, thus leading to the need of maintaining consistency among the replicas.

The eventual consistency is embodied by leveraging an object synchronization protocol to check different replica versions of each object. Consequently, this synchronization process introduces network overhead for already existent replicas as well as replica movement throughout storage nodes due to new reconfiguration of replica placement schemes.

After a new scheme is deployed, virtual nodes may be remapped to different storage nodes, thus requiring the synchronization of virtual nodes. In case of too much data being synchronized, the system may suffer destabilization.

2.2.5 Optimal Replica Placement

The partitioning, replication and architecture supported by the adopted KVS system support virtual server replication and migration. The literature says that only the optimal virtual server relocation problem is NP-Complete (FELBER *et al.*, 2014). This way, global optimization techniques are important techniques for dealing with these type of problems. Next, we present background information on global optimization.

2.3 Global Optimization

We formalize the continuous global optimization problem given by Equation 2.1

$$\begin{aligned} \min \quad & f(\vec{x}) \\ \text{s.t.} \quad & \vec{a} \leq \vec{x} \leq \vec{b} \end{aligned} \tag{2.1}$$

where $f : \Re^n \rightarrow \Re$ is referred to as the objective function while $\vec{a}$ and $\vec{b}$ define the lower and upper bounds of the search space, respectively.

In the discrete case, $\vec{x} \notin \Re^n$, a candidate solution is a discrete data structure, or object such as ordinal integers, categorical variables, binary variables, permutations, strings, trees or graphs in general.

Global optimization aims to find globally the best solution of models even in the presence of multiple local optimums. Global optimization problems may be classified not only as continuous or discrete, but also as linear or non-linear and convex or non-convex etc. In general, due to the absence of structural information and the presence of many local extrema, global optimization problems are extremely difficult to solve exactly (HU *et al.*, 2012).

The literature for solving these type of global problems is vast. To improve the understanding of many types of solvers, some authors classify them according to Figure 4 in which deterministic methods are not expanded because the focus of this work is on hard optimization problems. According to the authors of (BOUSSAÏD *et al.*, 2013), hard optimization problems cannot be solved optimally, or to any guaranteed bound, by any exact (deterministic) method within a "reasonable" time limit.

Yet in Figure 4, stochastic methods are expanded into instance-based or model-based algorithms according to the mechanism of generating new candidate solutions. Instance-based algorithms maintain a single solution or population of candidate solutions, and the construction of that candidate solutions depends explicitly on the previously generated solutions such as simulated annealing, genetic algorithms, tabu search, generalized hill climbing, and evolutionary programming. Genetic algorithms are used by the algorithmic infrastructure proposed in Section 4.

In model-based algorithms, new solutions are generated via an intermediate probabilistic model that is updated or induced from the previously generated solutions. Finally, surrogate model techniques may be classified into single surrogate, multi-fidelity and ensemble surrogate. An example of single surrogate is the Bayesian optimization technique which is used by the algorithmic infrastructure proposed in Section 5.

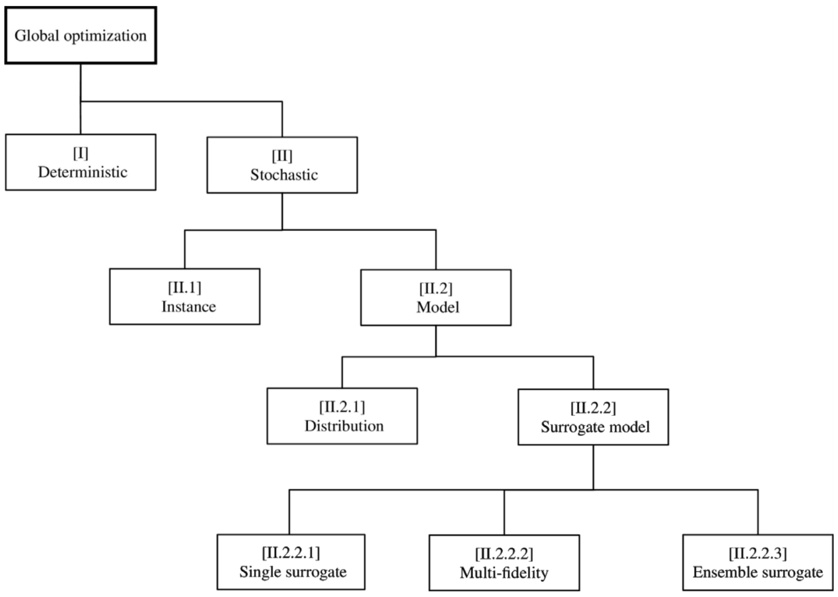


Figure 4 – Global Optimization Techniques Classification. Source: (BARTZ-BEIELSTEIN; ZAEFFERER, 2017).

2.3.1 Exploitation vs Exploration

Due to the difficult to solve a diverse range of hard problems within a "reasonable" time limit, an optimizer will be successful on a given optimization problem if it can provide a balance between the exploration (diversification) and the exploitation (intensification). Exploitation is necessary to identify parts of the search space with high quality solutions. Exploitation is important to intensify the search in some promising areas of the accumulated search experience.

2.3.2 Genetic Algorithms

Genetic algorithms are also classified as meta-heuristics optimization. This class of solvers are designed to solve approximately a wide range of hard optimization problems without having to deeply adapt to each problem. Meta-heuristics are generally applied to problems for which there is no satisfactory problem-specific algorithm to solve them.

Meta-heuristics may be classified in terms of their features with respect to different aspects concerning the search path they follow, the use of memory, the kind of neighborhood exploration used or the number of current solutions carried from one iteration to the next, single-solution based meta-heuristics and population-based meta-heuristics. The last, but not least important, the main difference between the existing meta-heuristics concern the particular way in which they try to achieve the balance between exploitation and exploration according to (BOUSSAÏD *et al.*, 2013). For example, single-solution based meta-heuristics are more exploitation oriented whereas population-based meta-heuristics are more exploration oriented.

Genetic algorithm is a meta-heuristic optimization inspired on the process of natural selection. It works by creating and evolving a population of candidate solutions or individuals. In general, individuals are composed of a chromosome and a fitness value (obviously, after evaluation of its objective function). Chromosomes are often represented as binary of zeros and ones in which binary value is a chromosome. The selection process is guided by evaluating the fitness of each individual and selecting the individuals according to their fitness values. New individuals are then generated using crossover and mutation functions. Even though both the crossover and mutation functions ensure a diversity of solutions, the random property of mutation guarantees convergence to global optimum. GA are mainly characterized by its operators in which the literature, such as (LI *et al.*, 2017), has evaluated many different algorithms for each GA operator as shown in Figure 5.

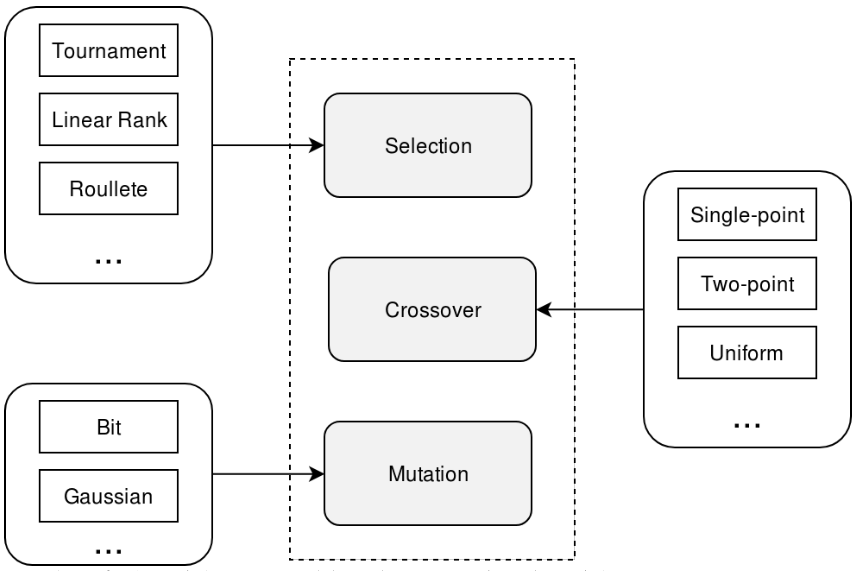


Figure 5 – Example of algorithms used by the genetic algorithm operators.

2.3.3 *Multi-objective Optimization*

Researchers have explored multi-objective problems since the 1970's in various domains for system control, decision making, circuit design, operations research, networking and telecommunications protocol design, and so forth according to the authors (CHO *et al.*, 2017) (PARDALOS *et al.*, 2017). Once, in Section 4, a multi-objective problem is proposed and explored, this work generalizes the global optimization problem to a multi-objective global optimization problem given by 2.1.

$$\begin{aligned} \min_{x \in \mathbf{A}} \quad & f(\vec{x}), f(\vec{x}) = (f_1(\vec{x}), ..., f_m(\vec{x}))^T \\ \text{s.t.} \quad & g_(j) \leq 0, j = 1, 2, ..., J \end{aligned} \tag{2.2}$$

where $f(x) : A \to \Re^m$ is now referred to as a vector-valued objective function. The feasible region $\mathbf{A} \subset \Re^d$ is expressed by a number of inequality constraints: $\mathbf{A} = \{x \in \Re^d | g_j \leq 0, j = 1, 2, ..., J\}$. If all the objective functions and the constraint functions are linear, then 2.2 is called a multi-objective linear programming problem. If at least one of the objective or constraint functions is nonlinear, then the problem is called a nonlinear multi-objective optimization problem. If the feasible region and all the objective functions are convex, then 2.2 becomes a convex multi-objective optimization problem. When at least one of the objective functions or the feasible region is non-convex, then we have a non-convex multi-objective optimization problem.

2.3.3.1 Scalarization Method

There were attempts to reduce multi-objective optimization problems to single-objective ones from the very beginning of their investigation (PARDALOS *et al.*, 2017). The reduction of a problem of multi-objective optimization to a single-objective optimization one is normally called scalarization in which all objective functions are combined to form a single function (MARLER; ARORA, 2004) (PARDALOS *et al.*, 2017).

The linear scalarization method is the weighted sum method where the auxiliary single-objective function is defined by the Equation 2.3:

$$\min \quad F(\vec{x}) = \sum_{i=1}^{m} w_i f_i(\mathbf{x}), w_i \geq 0 \tag{2.3}$$

where the weights w_i are parameters. In the case of non-convex objectives, the selection of a vector $\mathbf{w} = (w_1, ..., w_m)^T$ is not a guaranteed existence of w yielding an arbitrary element of the Pareto frontier. The Pareto frontier is a subset of w in which no improvement can be made to one objective function without worsening other objective function.

2.3.4 Surrogate Model-based

The Response surface methodology (RSM) is typically useful in the context of continuous optimization problems and focuses on learning input–output relationships to approximate the underlying simulation by a surface (also known as a meta-model or surrogate model) (AMARAN *et al.*, 2016).

Classical RSM are local search methods in which there is no guarantee for finding a global optimum.

2.3.4.1 Bayesian Optimization

Different from classical RSM, BO seeks to build a global response surface and is classified as a global optimization method to deal with expensive-to-evaluate black-box objective functions. It is commonly based on techniques such as Gaussian process (GP) regression in which subsequent samples are chosen based on some sort of improvement metric balancing exploitation and exploration.

For continuous functions, BO typically works by assuming the unknown function was sampled from GP and maintains a posterior distribution for this function as different observations are made (MOCKUS, 2012), (SNOEK *et al.*, 2012) (SHAHRIARI *et al.*, 2016).

To pick the hyper-parameters of the next experiment, one can optimize the Expected improvement (EI) over the current best result or the GP Upper confidence bound (UCB). EI and UCB have been shown to be efficient in the number of function evaluations required to find the global optimum of many multi-modal black-box functions (SNOEK *et al.*, 2012).

Acquisition functions trade off exploration and exploitation; their optima are located where the uncertainty in the surrogate model is large (exploration) and/or where the model prediction is high (exploitation) (SHAHRIARI *et al.*, 2016).

3 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.

3.1 MORM: A Multi-objective Optimized Replication Management strategy for cloud storage cluster

The authors (LONG *et al.*, 2014) focus on the replica placement problem. They also formulated their problem as a multi-objective optimization problem of the mean file unavailability, mean service time, load variance, energy consumption and mean access latency as five objectives. In comparison to our work, they use a similar load variance as objective. Considering their file availability objective, instead of using a file availability based on a failure probability, we use a constraint of minimum replicas, because hand-off techniques can be used to maintain the number of replicas electing new nodes as temporary replicas.

Additionally, we also differed from (LONG *et al.*, 2014) by proposing the replica maintenance and reconfiguration objectives. These objectives offer support to a fine-grained control of the amount of data to maintain in terms of store and the amount of data to move in order to adapt to a new configuration. Their multi-objective function is modeled using the weighted-sum scalarization technique. Our replica placement modeling is composed of only three parameters, but it also aims to minimize the data management issues targeted by (LONG *et al.*, 2014), the load imbalance and the number of replicas. This way, our modeling is more efficient because it avoids minimizing redundant objectives based on data access and additionally to the load imbalance objective, it focuses on minimizing the amount of data to maintain and reconfigure.

Similar to our work, (LONG *et al.*, 2014) also used GA-based algorithm to find good replica placements, but some differences can be pointed. For example, during the initialization of their individuals, they create only random individuals, but eliminate those that do not meet storage capacity constraint. Other difference from our work is that they do not perform chromosome repairing during the evolving process to meet the minimal replicas constraint. They also do not improve the scalability of their evaluation function using sparse matrix technology.

The last difference is the fact that they did not use black-box technique for optimizing the importance of their five objective functions as we will present in Section 5 in order to improve our first work in the Section 4. As result, their solution keeps requiring human-intervention for setting satisfactory importance parameters for improving the system response time according to a current scenario.

3.2 A novel object placement protocol for minimizing the average response time of get operations in distributed key-value stores

The authors (MAKRIS *et al.*, 2017a) reported 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. From the possibilities of replica manipulation: creation, migration and deletion, they focused on the migration operation.

The migration operation is then used for moving objects between nodes in run-time with the goal of minimizing very high (based on a threshold identified empirically) response times. In their scope, they reduced the object placement problem to the bin packing problem and formulate their problem as an integer linear optimization problem. To select the keys to be migrated as well as the under-loaded nodes that will host these keys, they based their solution on an offline bin packing heuristic algorithm, the First Fit Decreasing (FFD) algorithm.

They also proposed a combination of consistent hashing and a directory-base look-up for exceptions instead of consistent hashing and virtual nodes. The consistent hashing is a naming technique used for load balancing as explained in the Section 2 and an assumption of our system. The second technique aims to overcome the weakness of the consistent hashing technique by using a second auxiliary directory-base data structure for look-ups. They were aware that such directory-base look-ups become arbitrarily large being difficult to manage and maintain. This way, they proposed a Directory for exceptions (DFE) in which the auxiliary routing structure maps only popular data using the following conditional decision: if a request arrives at a random node in the cluster and this nodes holds the corresponding key, the node serves the client directly. Otherwise, it raises an exception to the directory for updating the DFE and redirects the request to the correct node.

3.3 On optimizing replica migration in distributed cloud storage systems

(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.

Finally, the discussed points of the related work can be summarized according to Table 3.

Table 3 – Related Work Comparison.

Authors.	(LONG *et al.*, 2014)	(MAKRIS *et al.*, 2017a)	(CAVALCANTE *et al.*, 2018)	Cavalcante et al
Research Problem.	Replica Placement.	Replica Placement limited to migration	Replica Placement.	Latency-Adaptive Replica Placement.
Replica Placement Objectives.	Mean file unavailability, and other 4 metrics manly based on access such as load imbalance.	Load imbalance.	Load imbalance, Replica Maintenance and Replica Reconfiguration.	
Data Organization / Scalability.	Hierarchical Directory.	DHT and Hierarchical Directory.	Efficient DHT.	
Load Balance Solution.	Meta-heuristic / GA operators.	Heuristic / First Fit decreasing.	Meta-heuristic / GA operators with chromosome repairing.	
Importance Optimization of the Replica Placement Objectives (hyper-parameter optimization).	No.	No.	No	Yes. Adopt Bayesian Optimization technique for searching satisfactory replica placement parameters.

4 POPRING: A POPULARITY-AWARE REPLICA PLACEMENT FOR DISTRIBUTED KEY-VALUE STORE

In this section 4, we introduce system model of the replica placement component. We also formalize our objectives as a multi-objective optimization problem.

From the discussed issues of hot and cold data in KVS systems addressed in Section 1, the DHT technology used for KVS for balancing data in terms of quantity or size is not effective for dynamic and skew data access workloads. Our approach is not to create technology from scratch, but to take advantage of the current DHT properties while aggregating new capacities able to handle its limitations.

Adopting a strategy of evolving our efforts on demand, the work in Section 4 focuses on elaboration and experimentation of a replica placement component. A good solution design would require to answer the following research questions: should data be migrated and/or replicated? Which node should be the new host of the replicated/migrated data? Which replicas should be removed? 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? The investigation of these questions resulted in the work presented in current Section 4.

4.1 System Model and Replica Placement Optimization Problem

4.1.1 System Model

In this section, our system is modeled, thus making clear the scope of our research.

4.1.1.1 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.

4.1.1.2 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.

4.1.1.3 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 $|P|$.

4.1.1.4 Replica Placement Maintenance Cost

The replica placement maintenance cost indirectly represents the network delay/overhead caused by the data synchronization of already existing, yet 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 equation 4.1. This way, during the evaluation of previous and new schemes, a solution can focus on deleting already existing replicas.

$$C_{maintenance} = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} (o_{ij}s_{ij})(repl\_size_j) \tag{4.1}$$

4.1.1.5 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 equation 4.2. 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 4.2.

$$C_{reconfig.} = \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} (s_{ij} - o_{ij}s_{ij})(repl\_size_j) \tag{4.2}$$

To better understand the differences between replica placement maintenance and reconfiguration costs, we describe examples bellow considering Tables 4 and 5:

Table 4 – Example of Previous Replica Placement Scheme.

$\mathbf{O} =$

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

Table 5 – Example of New Replica Placement Scheme.

$\mathbf{S} =$

	$p=0$	$p=1$	$p=2$	...	**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 4, 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 5, 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 4 and considering the new scheme S in Table 5, 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 $d = 0$.

4.1.1.6 Load Imbalance Cost

The amount of *Get* requests submitted to each storage node $d \in D$ is measured according to the Equation 4.3 in which as mentioned before, M is the number of storage nodes

$|D|$ and N is the number of virtual nodes $|P|$.

$$stor\_node\_get_i = \sum_{j=0}^{N-1} s_{ij}(virt\_node\_get_j/repl_j) \tag{4.3}$$

As we mentioned early, D has similar performance capacities, then the ideal data access per storage node is defined according to the Equation 4.4

$$ideal\_get = (\sum_{i=0}^{M-1} stor\_node\_get_i)/M \tag{4.4}$$

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 Equation 4.5

$$C_{load\_imbalance} = (\sum_{i=0}^{M-1} |stor\_node\_get_i - ideal\_get|)/M \tag{4.5}$$

4.2 Optimization Problem

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 Equation 4.6:

$$\begin{aligned} \min_{S} \quad & C_{load\_imbalance}, C_{maintenance}, C_{reconfiguration} \\ \text{s.t.} \quad & used\_stor_d <= max\_stor_d \\ & repl_p >= min\_repl_p \\ & not\_reconf_p >= min\_not\_reconf_p \end{aligned} \tag{4.6}$$

4.3 Solution

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 in which different objectives may conflict with each other as shown in Table 6. By this means, they require optimal trade-off analyses among the objectives of a system.

The authors (CHO *et al.*, 2017) surveyed many approaches to solve multi-objective (MOO) problems. The weighted sum (WS) method is computationally efficient in generating

Table 6 – Issue / Cost / Modeling.

Issue	Cost	Modeling
Hot Data	Load Imbalance	Skewed data access of get requests.
Data Movement	Replica Placement Reconfiguration	Replica migration/creation
Cold Data	Replica Placement Maintenance	Essential replicas and dispensable replicas

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 Equation 4.7, 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.

$$F = C_1 C_{load\_imb.} + C_2 C_{mainten.} + C_3 C_{reconfig.} \tag{4.7}$$

4.3.1 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 all combinations of F in Equation 4.7 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 randomly generate 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.

4.3.1.1 Initial Population Improvement

An advantage of GA is the possibility of inserting prior knowledge to the initial population to reduce the convergence time of the search. This way, the initial population formed mainly by random individuals may have specials individuals created using some knowledge/guess that clearly guides more rapidly to better results.

In our work, we used two individuals as guesses:

- Previous replica placement scheme. This individual is the actual replica placement scheme used by the system. It increases the chances of reducing the replica migration/creation because this individual has the smallest cost for $C_{reconf.}$
- Reduced replica placement scheme. This individual is based on the actual replica placement in which each virtual node has its replicas randomly reduced until the min_repl_p constraint. At including this reduced replica placement scheme as individual, the randomized search increases the chances of reducing the total replicas, because that individual has the smallest cost for $C_{mainten.}$.

4.3.1.2 Chromosome repairing

The constraints of an optimization problem may turn the search process very difficult to find feasible solutions. One way to solve that is to relax and or remove the constraints. Another way is to use prior knowledge to fix unfeasible individuals by locating and mutating the bad genes to meet the constraint. This way, the cost function of a fixed individual is reevaluated. Then, the unfeasible individual is removed while the fixed individual is added to the population.

When individuals are unfeasible because they have virtual nodes with less replicas than the required by the constraint minimum number of replica min_repl_p, we proposed a simple chromosome repairing algorithm that lists all the storage nodes where the virtual node is not replicated yet, then selects randomly one of them to host a new replica of the virtual node. This process is repeated until the constraint is met.

4.3.2 PopRing Replica Placement Algorithm

In this section, we present the main flow of PopRing replica placement component. When it is the first time the KVS is being deployed and consequently no dataset of data access is available yet, our algorithm creates a replica placement scheme in which the replicas are spread uniformly and randomly among the storage nodes.

Assuming the system is already in normal functioning, i.e., a replica placement scheme is deployed and the system is serving data access operations, the Algorithm 1 can be performed any time to generate a new replica placement scheme.

At line 1, the individuals generated randomly and the two guesses, described in Section 4.3.1.1, are used to form the initial population. At lines 3, 4 and 5, genetic algorithm operators are performed on the current population using Tournament, Two-Point and Bit for selection, cross-over and mutation operators, respectively.

The function cost F of the PopRing Replica placement component, defined at Equation 4.7, is then used to evaluate the cost of each individual, i.e., replica placement scheme candidate as shown at line 6. At lines 7 and 8, any individual candidate that does not meet the minimal number of replicas constraint is fixed using the chromosome repairing idea described in Section 4.3.1.2. Finally, the cost of the fixed individual candidate is reevaluated. [1]

Algorithm 1: PopRing Algorithm of the Replica Placement Component

```
   Input: (Previous replica placement scheme, current data access, $C_1$, $C_2$ and $C_3$)
   Output: New replica placement scheme
 1 Perform the population initialization
 2 while Maximum number of generations do
 3     Perform the Tournament selection algorithm on individuals
 4     Perform the Two-Point crossover algorithm on individuals
 5     Perform the Bit mutation algorithm on individuals
 6     Evaluate $F(c_1, c_2, c_3)$ of the individuals
 7     Perform chromosomes repairing of the unfeasible individuals
 8     Reevaluate $F(c_1, c_2, c_3)$ of the individuals
 9 end
10 Select best individual as new replica placement
```

[1] Note.: In the work presented in the Section 4, we did not perform any chromosome repairing. In contrast, in the work presented in Section 5, the evaluated scenario required the usage of the repairing technique.

4.3.2.1 Sparse Matrix Improvement

The matrix calculations necessary to evaluate F in Equation 4.7 for one replica placement scheme S have $O(mn)$ complexity where m is the number of 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 in the Equation 4.7 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 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 Sparse Row (CSR) format (GROSSMAN *et al.*, 2016) and reduced the time complexity of matrix operations on F in the Equation 4.7 to $O(n)$.

4.4 PopRing Replica Placement Component

After discussing all the techniques used to build our replica placement solver in previous sections, we consolidate that techniques into the PopRing replica placement component. This component is an overview focusing only on the inputs and outputs of our strategy. The previous replica placement scheme is the last scheme adopted by the KVS. An system administrator may keep the previous scheme or deploy a new one. The previous data access is the last observed access of the system objects. The size of the objects in the system is also input to the system, but they are not shown for simplicity. Finally, the importance of the load imbalance cost C_1, replica maintenance C_2 and replica reconfiguration C_3 are proper defined by the system administrator and used as input by the replica placement component as well. The output of the component is to produce a new replica placement scheme as shown by the Figure 6.

4.5 Experimental evaluation

For evaluating our proposed solution PopRing against the default replica placement algorithm of the OpenStack-Swift, our simulated environment is described in Section 4.5.1. Finally, Section 4.5.2 discusses the improvements of our solution under different configurations

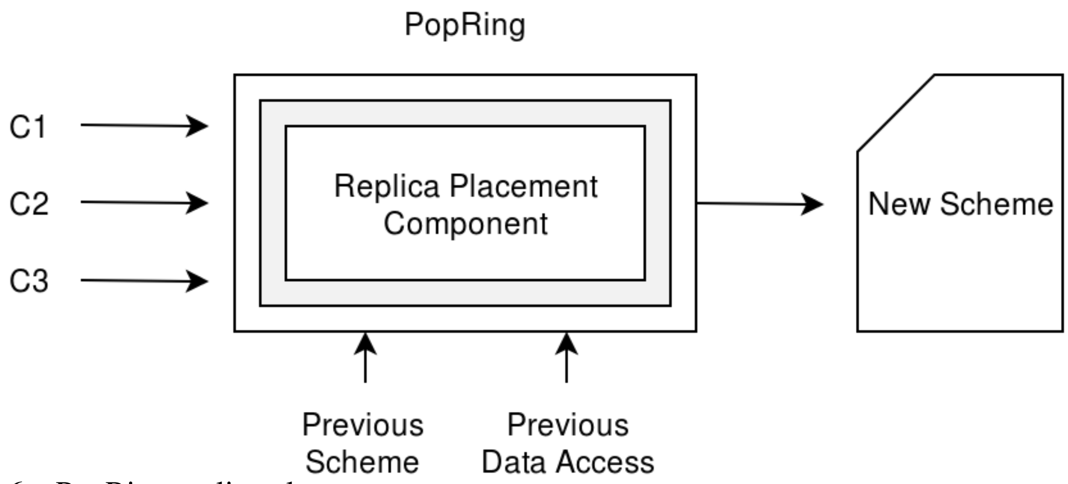
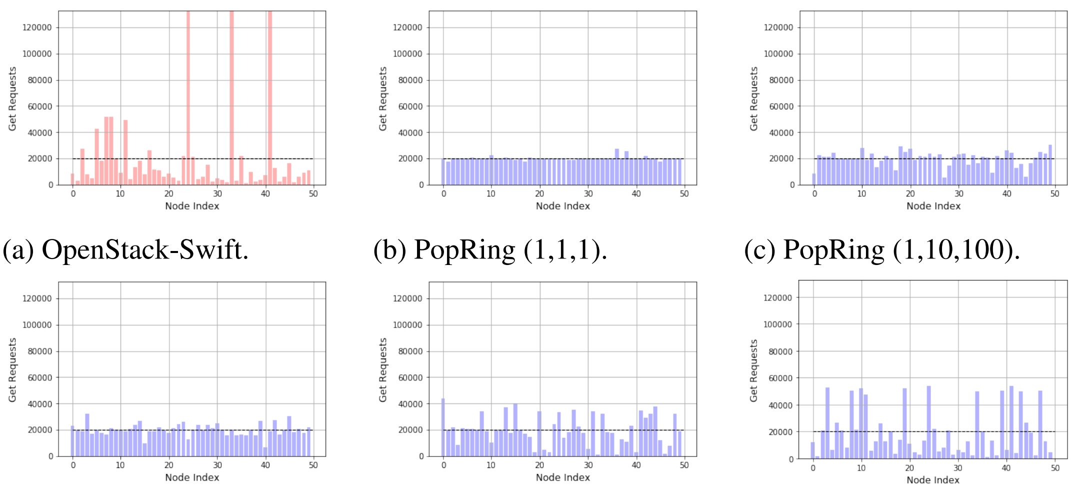


Figure 6 – PopRing replica placement component.

regarding the importance of the objectives.



(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 7 – Total *Get* Requests Per Storage Node Index.

4.5.1 Simulated Environment

First, we setup the default settings of the OpenStack-Swift as 3 and 1024 for replication factor and number of virtual nodes, respectively. We simulated 50 as number of storage nodes. We also 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 4.2, 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 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 7 – 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 observed that for our environment, the Table 7 represents an interval of configurations in which $(1,1,1)$ is the highest importance for reducing load imbalance while $(1,200,200)$ is the contrary, i.e., the lowest importance.

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 in Table 7.

4.5.2 Results

Fig. 7 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 4.4. Fig. 7a 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. 7b. The configurations (1, 10, 100) and (1, 100, 10) had similar load imbalance of 3980.15 and 3246.52 as shown in figures 7c and 7d. The configurations (1, 100, 100) and (1, 200, 200) obtained 8984.42 and 14667.23 of load imbalance, respectively as shown in figures 7e and 7f. 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. 8 represents the percentage of the amount of data according to their replication factor. The configuration (1, 1, 1) has the highest 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 required almost 20% of virtual nodes to increase their number of replicas.

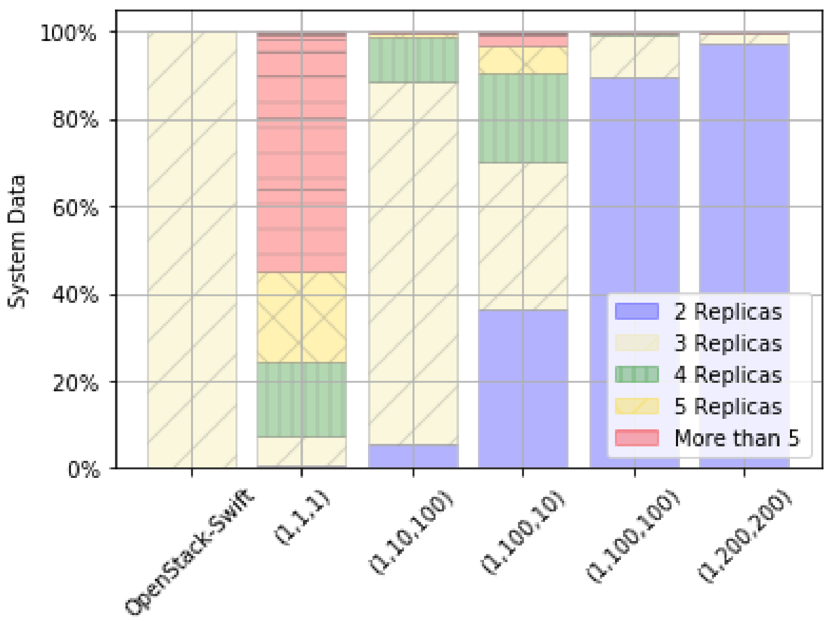


Figure 8 – Replication Factor Evaluation.

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. 9. 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. 10. These results make possible to understand that the performance of load imbalance cost is impacted by the other two objectives.

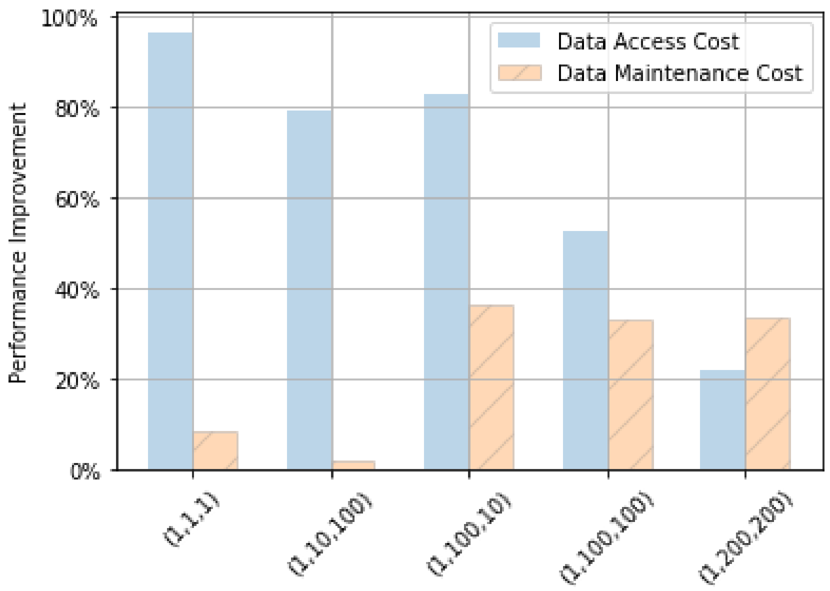


Figure 9 – Relative Costs: How much load imbalance (*Get* requests) and replica maintenance costs were reduced in comparison to the original replica placement of the OpenStack-Swift.

In Fig. 9, it is possible to notice a decline in the load balance performance and a rising in the replica maintenance performance. The same applies to replica placement reconfiguration as shown in Fig. 10. The increase in the importance of replica maintenance and replica reconfiguration make the load imbalance more difficult to minimize. Figures 9 and 10 shown the trade-offs among load imbalance, replica placement maintenance and replica placement reconfiguration objectives.

4.6 Conclusion

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.

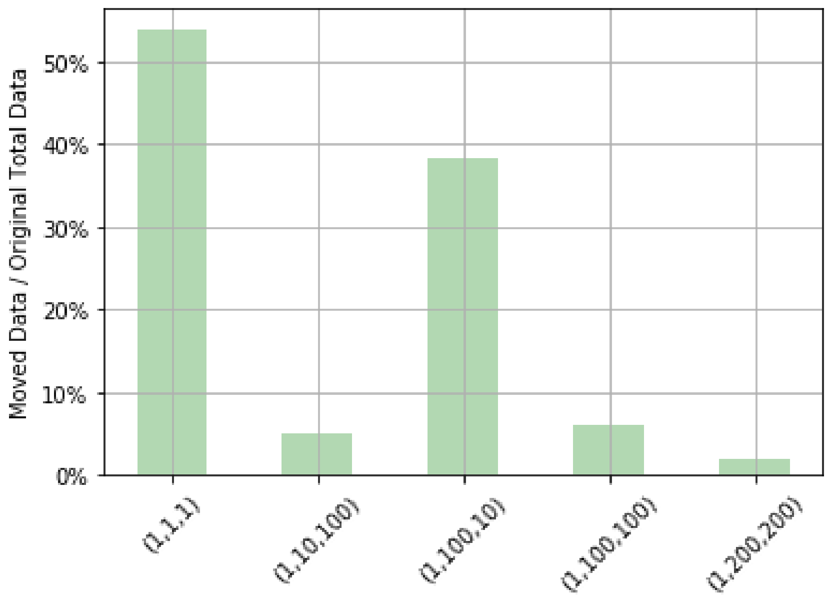


Figure 10 – Total data movement relative to the total previous replica placement scheme.

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.

5 POPRING HYPER-PARAMETER OPTIMIZATION

In this section, we introduce system model of the hyper-parameter optimization component. We also formalize our objective as a single objective hyper-parameter optimization problem.

5.1 System Model and Optimization Problem

In the previous system described in Section 4, the PopRing parameters C_1, C_2 and C_3 are useful for tuning PopRing to give more importance to one objective(s) rather than other(s). The replica placement component aimed to obtain satisfactory results considering the trade-off between the three objectives, but it had limitations that this new work aims to solve proposing another second component. The limitations of the replica placement component were noted from the fact that it always requires prior knowledge and human intervention to tune its parameters to a proper configuration able to improve the performance metrics of the system such as latency of data access.

This way, new research questions should be necessary answered to support an autonomous objective function based on latency. These research questions are: given that the trade-off between the three objectives described in Section 4 may vary according to the current scenario, how to find a configuration C_1, C_2 and C_3 which is able to improve system latency? Is it possible to tune these parameters autonomously? And more important, due to the high cost of evaluating each possible combination, is it possible to evaluate as few scenarios as possible for finding satisfactory parameters values. The investigation of these questions resulted in the work presented in Section 5.

Additionally, the main difference in this current work from Section 4 is that it does not require human intervention with prior knowledge to set proper values for them because our optimization system will search them autonomously to minimize the system latency for any given workload.

5.1.1 System Model

In this section, our system is modeled, thus making clear the scope of our research.

5.1.1.1 Storage and Service 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 stores data according to replication scheme explained in our prior work at Section 4. Additionally, each storage node has a maximum capacity in terms of rate of object requests and in case this limit is exceeded, a significant decrease on its performance occurs.

The system is also composed of a set of distributed and independent service nodes B. For a given *Get* request submitted to our distributed key-value store, a service node $b \in B$ communicates with a storage node $d \in D$ to retrieve data through the network according to the replication scheme.

5.1.1.2 Workload Specification

Any workload submitted to our system is composed of a set of *Get* requests and *Put* requests, but even though these are important metrics for the system optimization of our prior work 4, this current system ignores these metrics and instead considers the metric MRT_{bd} metric.

MRT_{bd} metric is a mean response time data point collected between a service node $b \in B$ and a storage node $i \in D$ during a get request. $AverageMRT_(C_1, C_2, C_3, t)$ is the average of all MRT_{bd} collected from all service and storage nodes at a given time t under a replica scheme setup with the parameters C_1, C_2 and C_3.

5.1.1.3 New Scheme Updating and Stabilization

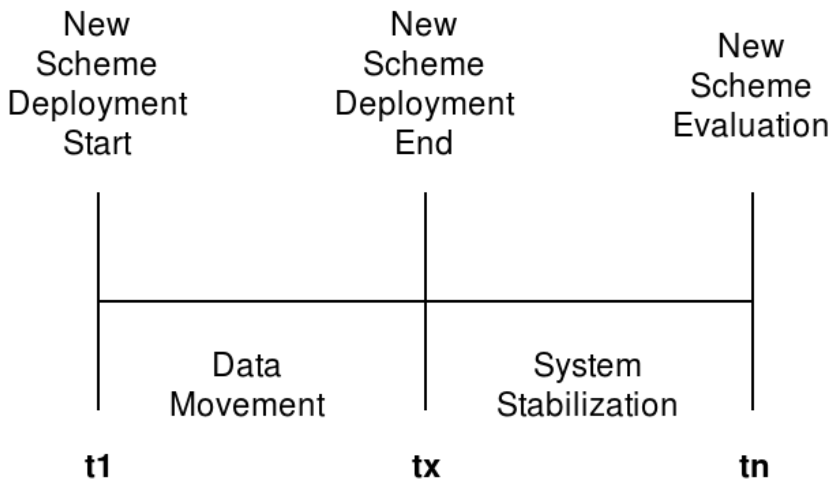


Figure 11 – Key moments in which a bunch of $AverageMRT(C_1, C_2, C_3, t)$.

A deployment of a new replication scheme causes the system to perform data movement during the new scheme deployment. Depending on the new scheme, more or less data movement may occur and cause performance degradation before any improvement to the system performance. As soon as the deployment of the new scheme has been finished, a little more time is given until system performance is stabilized. The amount of $AverageMRT(_{C_1,C_2,C_3,t}) \in T$ data points collected since the new scheme deployment with parameters C_1, C_2 and C_3 until the system stabilization is defined by the $TotalDatapointsCollected$. The Figure 11 illustrates key moments in which a bunch of $AverageMRT(_{C_1,C_2,C_3,t}) \in T$ is collected. The x variable in tx represents a value of t in which the new scheme is totally synchronized. tx is variable, because it depends of, for example, how much data is migrated.

5.1.1.4 Mean Response Time Cost

The mean response time cost C_{MeanRT} is the objective function of our hyper-parameter optimization problem. Given a workload, C_{MeanRT} minimization guides the search to the most adequate PopRing hyper-parameters. C_{MeanRT} is evaluated according to the Equation 5.1.

$$C_{MeanRT} = \sum_{t=0}^{T-1} (AverageMRT(_{C_1,C_2,C_3,t}))/T, T = TotalDatapointsCollected. \tag{5.1}$$

5.2 Optimization Problem

Given the system model as well as the mean response time cost previously defined, we set the goal of the current system as the minimization of the single object function according to the Equation 5.2:

$$\min_{C_1,C_2,C_3} \quad C_{MeanRT} \tag{5.2}$$

5.3 Solution

In this work, we propose an approach for searching satisfactory C_1, C_2 and C_3 parameters of our replica placement component to find satisfactory values for reducing the average internal latency of the system.

Given the set of all possible environment scenarios and the possibility of values for each parameter C_1, C_2 and C_3 not be discrete, it would take unfeasible time to evaluate the best parameters combination using brute-force or grid search techniques. Additionally, our cost function is time consuming and expensive to evaluate because it requires data movement to be performed among the storage nodes of the KVS. While most optimization techniques assume that the objective function is quick to evaluate such as the genetic algorithm optimization we have used in our prior work of the replica placement component, that is not the case for our hyper-parameter optimization.

A good choice for our case is the BO which is very data efficient, i.e., it requires few samples to find good results. This is particularly useful in situations like these where evaluations of the unknown function are costly (SHAHRIARI *et al.*, 2016). For continuous functions, Bayesian optimization typically works by assuming the unknown function was sampled from a Gaussian process and maintains a posterior distribution for this function as different C_1, C_2 and C_3 are observed.

5.3.1 PopRing Hyper-parameters Optimization Algorithm

In this section, we present the main flow of the PopRing Hyper-parameters Optimization Algorithm. At line 1, the BO is configured to optimize the parameters C_1, C_2 and C_3 of the PopRing replica placement component in which their continuous values are bounded by its minimum and maximum value defined by $C_{1\_boundary}$, $C_{2\_boundary}$ and $C_{3\_boundary}$, respectively. We also, set the number of initialization points the BO uses to randomly sample the C_{MeanRT} cost before fitting the Gaussian Process. Finally, the number of BO sample/trials used to verify the termination condition of the algorithm. This number of trials may not be necessarily sufficient to find good parameters, but we verify that stop condition for simplification as shown at line 2.

At line 3, we reset the system to the original/previous replica placement scheme, i.e., the one deployed before any BO process. This way, the BO component may able to find the best replica placement for the current state. At line 4, our algorithm requests C_1, C_2 and C_3 from the BO in the search of the global optimum that, by definition, is approached by iteratively maximizing a so-called acquisition function, that balances the exploration and exploitation effect of the search.

At lines 5, we collect the last data required by the PopRing replica placement component. At line 6, PopRing replica placement component is executed with the following inputs: the current C_1, C_2 and C_3 suggested by the PopRing Hyper-parameter optimization component, the previous data access and previous replica placement scheme. For each workload, the level of data access skew remained the same during the whole algorithm. Then, at the next lines, the new scheme generated is deployed and system response time metrics are collected until system stabilization. Finally, at lines 9 and 10, the cost function of the PopRing Hyper-parameter is evaluated and used as a sample performance to update the BO model.

After the maximum number of trials, the parameters C_1', C_2', C_3' which obtained the best cost are used as the final replica placement scheme for the current scenario.

Algorithm 2: PopRing Algorithm of the Hyper-parameter Optimization

```
   Input: ($C_{1\_boundary}$, $C_{2\_boundary}$, $C_{3\_boundary}$, previous_scheme, current_data_access)
   Output: $C_1'$, $C_2'$, $C_3'$
1  Initialize BO parameters: boundary of inputs ($C_{1\_boundary}$, $C_{2\_boundary}$, $C_{3\_boundary}$), the number of
     initialization points and the number of trials.
2  while number of BO trials do
3      Deploy the previous_scheme to reset system to initial state
4      Request $C_1$, $C_2$, $C_3$ new points from BO model
5      Collect last data access from monitoring node
6      Run PopRing Replica Placement Component using $C_1$, $C_2$, $C_3$, previous_scheme and
         current_data_access
7      Deploy new scheme
8      Collect system response time metrics until system stabilization
9      Evaluate C_MeanRT
10     Update BO model with C_MeanRT
11 end
12 Return the parameters of the best trial $C_1'$, $C_2'$, $C_3'$
```

5.4 PopRing Bayesian optimization component

Now, we consolidate the techniques discussed in previous sections into the new hyper-parameter optimization component in which the main goal of that component is to solve in few iterations the optimization problem described in Equation 5.2. This component has as input the minimum and maximum importance values of the load imbalance, replica maintenance and replica reconfiguration costs.

The second input is the number of trials/iterations. This input defines how many times the BO will setup new values for the parameters C_1, C_2 and C_3 to execute the replica placement component. Then, the new schemes are generate and deploy into the KVS system. At minimizing the system response time C_{MeanRT}, the hyper-parameter optimizer component evaluates combinations of C_1, C_2 and C_3 based only on measured response time of the system.

Our PopRing strategy is a black-box solution because it does not know how the system response time behaviors given the usage of a replica placement scheme. At the end, our solution picks the best one and labels a combination of C'_1, C'_2 and C'_3 as the most appropriate for the current scenario as shown by the Figure 12.

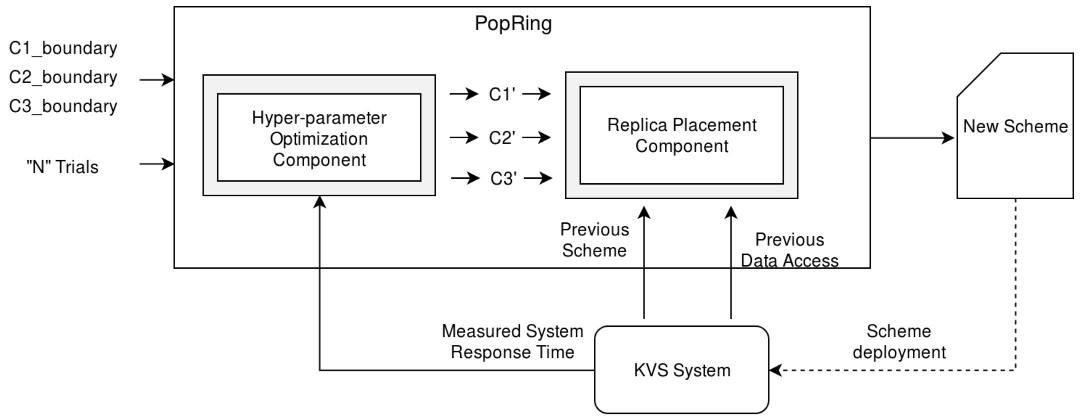


Figure 12 – PopRing with the replica placement component and the new hyper-parameter optimization component.

5.5 Experimental Evaluation

For evaluating our PopRing BO component, we compare different executions of the whole Bayesian optimization process described in Section 2.3.4.1 under different scenarios of our experimental environment using real benchmark tool and distributed key-value store deployments. Finally, the improvements of our solution under different scenarios regarding the Zipf exponent, i.e., level of data access skew are discussed in Section 4.5.2.

5.5.1 Distributed Key-Value Store

Our PopRing BO component has been evaluated using the OpenStack-Swift. Swift is a highly available, distributed, eventually consistent object/blob store. Organizations use Swift to manage lots of data efficiently, safely, and cheaply. Swift was chosen for our experiments due to our prior knowledge on deployment and upgrading their source code.

Even though other distributed key-value stores such as Cassandra and Dynamo could have been used for evaluation purpose once they are based on distributed hash table techniques, they are not strictly required for validating our solution.

5.5.2 OpenStack-Swift

Swift architecture can hold many different web service nodes and background processes whereas each component can be scaled multiple times. The web service nodes are classified in two main categories: proxy server type and storage node type (composed by account server, container server and object server). The background processes are responsible for data replication, data reconstruction, data updating and data auditing. The proxy server node takes requests from a client and forwards them to the account, container and object server nodes in order to persist/retrieve data objects and its metadata to/from disks. Our architecture and partitioning techniques adopted in Section 2.2.1 are intended to be represented by our Swift deployment.

Read and write requests from/to Swift obeys the following rules: for reading requests, a single node is enough replica to retrieve the object and return success to the client; for writing requests, it is enough the quorum of half of total nodes plus one to save the object and return success to the client. For our environment, we configured the replication scheme generated by OpenStack-Swift baseline to provide three replicas for each stored object.

5.5.3 Benchmark

In order to evaluate our approach, we used *Cloud Object Storage Benchmark* (COSBench) (ZHENG *et al.*, 2013). COSBench provides support to many different cloud object storage solutions on the market like Swift, Amazon S3, and Ceph thus making it easier for any future comparison among those cloud object storage solutions (ZHENG *et al.*, 2013).

COSBench tool provides seven benchmark metrics: operations count, bytes count, average response time, average process time, throughput, bandwidth and success ratio. In this paper, we evaluated only system internal metrics for performance evaluation, thus using COSBench only for stress purpose.

5.5.3.1 Workload

For our workload, we set the read/write ratio to 90/20 and the number of workers to one. For object sizes, we defined a uniform distribution between 1-1024KB. We also upgraded COSBench with the Zipf distribution for data access to evaluate different levels of data access skew, by varying the Zipf exponent to 0.1, 0.5, 1.1., 1.5, 2.0 and 30.0.

More detailed benchmark configuration is described at the following: we used single container because replication scheme at container level is out of scope for this current work. We also used 100 objects, where 50 were exclusively for read requests and other 50 were exclusively for write requests, respectively. Other benchmark specific properties like Acceptable Failure Rate (AFR), ramp-up and ramp-down were set to 200000, 15, 15, respectively.

5.5.4 *Software/Hardware*

For our software configuration, we chose the Ubuntu server 14.04 version for operating system. Mitaka version for OpenStack-Swift and 0.4.0.2.c4 version for COSBench. For our hardware configuration, we deployed 11 different nodes. One node for COSBench benchmarking, one for graphite monitoring, one for proxy node and eight storage nodes with similar performance for a minimalist Swift cluster according to Table 8.

Table 8 – Hardware Configuration

Node	**RAM (GB)**	**IOPS limit**[1]	**CPU**	**Storage Size**
COSBench	N/A[3]	N/A[3]	N/A[3]	N/A[3]
Proxy Node	N/A[3]	N/A[3]	N/A[3]	N/A[3]
Graphite	N/A[3]	N/A[3]	N/A[3]	N/A[3]
SN[2]	1	50	1	N/A[3]

1. IOPS limit was handled by virtualization technology.
2. Same configuration for all storage nodes.
3. Resource capacity is not considered in current experiment and they were setup to not be bottleneck.

5.5.5 *PopRing Configuration*

For the setup of the evolutionary parameters of PopRing replica placement component, we used the same parameters of our prior work, i.e., 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. Additionally to the replica placement component, this work varies the importance parameters of PopRing dynamically and without human-intervention using BO.

For evaluating the importance parameters of PopRing, we used BO with 5, 25, Upper Confidence Bound (UCB) and 2.576, respectively for initialization points, number of iterations, acquisition function and kappa. Initialization points are the number of randomly chosen points to sample the target function before fitting the Gaussian process. Number of iterations is the total number of times the process is to be repeated. Kappa is a user-defined parameters for specifying the trade-off between exploration and exploitation. We have used UCB, because it has been shown to be efficient in the number of function evaluations required to find the global optimum of many multimodal black-box functions (SNOEK *et al.*, 2012)

5.5.6 Results

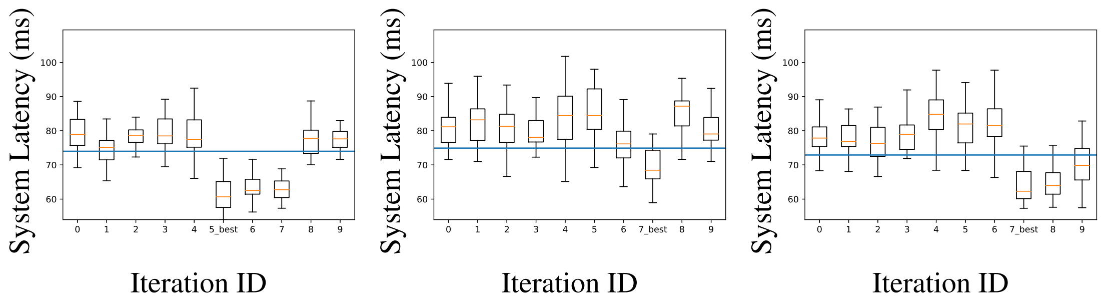
For better understanding of our results, we evaluate three executions of each workload under different aspects. In the Section 5.5.6.1, we discuss *Original_AverageMRT* in the blue horizontal line, the distribution of the $AverageMRT(_{C_1,C_2,C_3,t}) \in T$ data-points collected during each iteration of each Bayesian optimization execution while the best/worst C_1, C_2, C_3 parameters obtained for each execution are compared in Section 5.5.6.2.

In Section 5.5.6.3, we discuss in details the PopRing hyper-parameters C_1, C_2 and C_3 under the best and worst executions of our Bayesian optimization component where the best executions are colored in red and the worst executions are colored in blue. We also discuss the raw values of the best executions as well as the behavior of $AverageMRT(_{C_1,C_2,C_3,t}) \in T$ data-points collected during the deployment process of a new replication scheme. Finally, we present a table to support an overview of the performance of the best executions of our Bayesian optimization component.

5.5.6.1 Bayesian Optimization Iterations/Trials

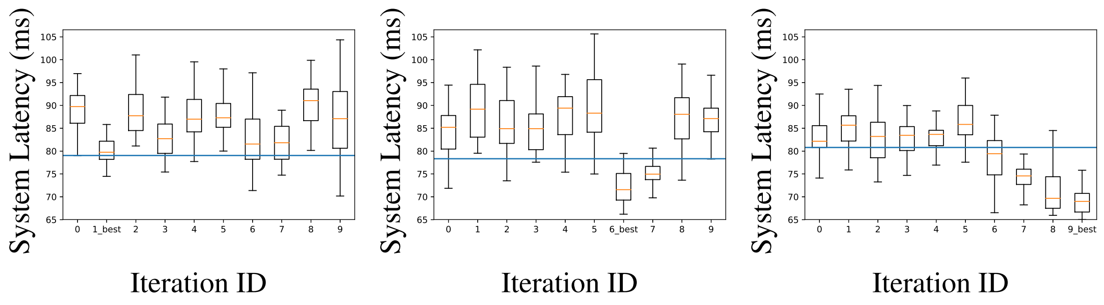
For evaluating the system response time of each combination of C_1, C_2 and C_3 applied by the Bayesian optimization solver, the obtained mean response time cost defined at Equation 5.2 of the Zipf exponents equal to 0.1, 1.5 and 30.0 are presented in Figures 13, 14 and 15. The other Zipf exponents equal to 0.5, 1.1 and 2.0 are not presented because they had proportionally similar behavior.

In Figures 13, 14 and 15, it is possible to verify that the previous average response time of the system *Original_AverageMRT* was around 75 ms for Zipf exponent equals to 0.1, 80 ms for Zipf exponent equals to 2.0 and 115 ms for Zipf exponent equals to 30.0 as shown by the blue horizontal line of all three Bayesian optimization executions. The first trial of any execution rarely obtained the lowest latency, which is reasonable because the first trial is totally random. From a total number of 10 iterations/trials for each execution under Zipf exponent 0.1, it took at best five iterations and at worst 7 iterations to find C_1, C_2 and C_3 values able to reduce the



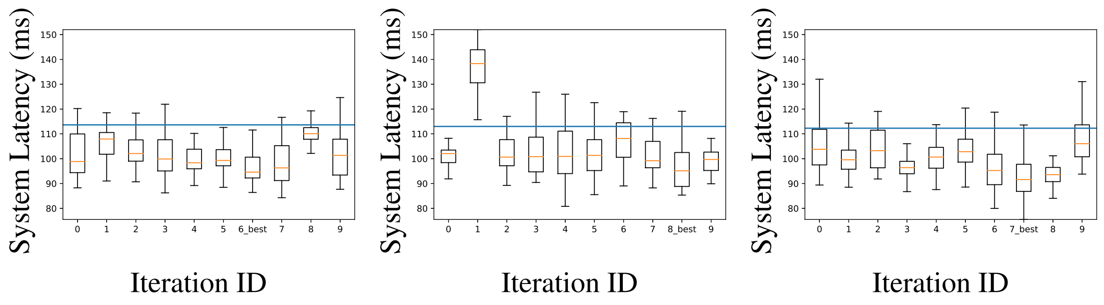
(a) Zipf 0.1 / Exec. 0 (b) Zipf 0.1 / Exec. 1 (c) Zipf 0.1 / Exec. 2

Figure 13 – Bayesian optimization executions / Zipf exponent 0.1.



(a) Zipf 1.5 / Exec. 0. (b) Zipf 1.5 / Exec. 1. (c) Zipf 1.5 / Exec. 2.

Figure 14 – Bayesian optimization executions / Zipf exponent 1.5.



(a) Zipf 30.0 / Exec. 0. (b) Zipf 30.0 / Exec. 1. (c) Zipf 30.0 / Exec. 2.

Figure 15 – Bayesian optimization executions / Zipf exponent 30.0.

C_{MeanRT} to reach 60 ms. This improvement is interesting because even though the Get requests are not skew, our solution still could adjust the replication scheme to reduce the C_{MeanRT}.

In summary, at least one trial reduced C_{MeanRT} to a value smaller than the *Original_AverageMRT*, which proves that the Bayesian optimization component could optimize hyper-parameters regardless of the tested scenario.

5.5.6.2 *Best/Worst Trials of Each Execution*

The Figures 16, 17 and 18 show the best and worst iterations/trials of each Bayesian optimization execution for Zipf exponents equal to 0.1, 1.5 and 30.0. The other Zipf exponents equal to 0.5, 1.1 and 2.0 are not presented because they had proportionally similar behavior. In the Figures 16, 17 and 18, it is possible to verify that at least in two executions out of three, the best trials, i.e., the smallest C_{MeanRT} were obtained by similar PopRing C_1, C_2 and C_3 settings.

For Zipf exponent 0.1, it is verified that the parameter C_3 is not necessarily high, thus giving the idea that data movement is not a big issue to the system at minimizing the C_{MeanRT}. Also, load imbalance cost is small and low data movement is needed for proper handling the a non-skewed workload as shown in Figure 16.

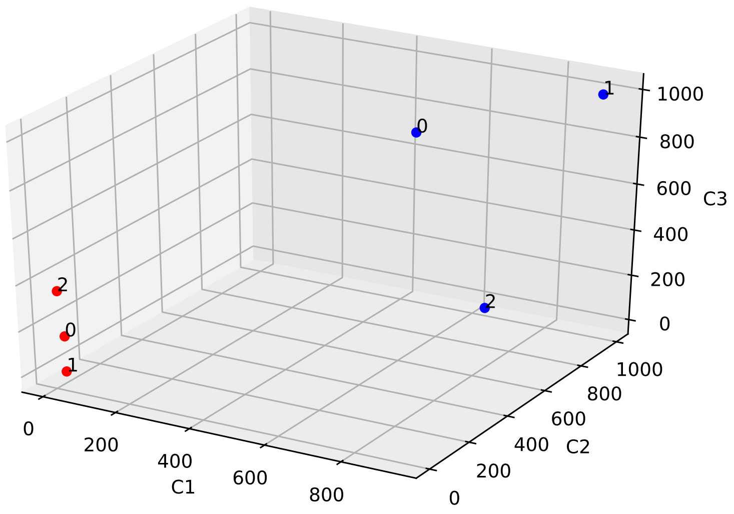


Figure 16 – C_1,C_2,C_3 of the best and worst iterations/trials for Zipf exponent: 0.1.

For Zipf exponent 1.5, it is possible to verify that C_1 converged to low values. More interesting is the C_3 parameter which converged around the highest value of the possible range we defined 1000. This result shows that for the skewed workload of our test-bed, it is necessary to adjust the replication scheme to fix the load imbalance of data access, but it is important to be cautious about data movement due to the negative impact of lots of data movement.

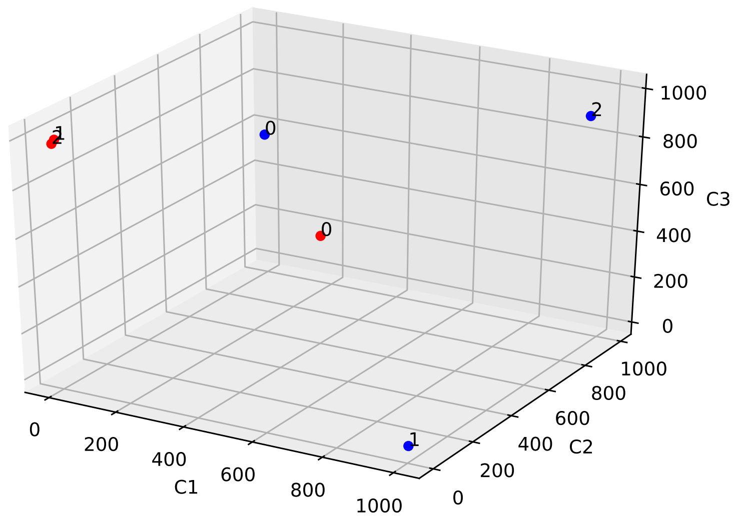


Figure 17 – C_1,C_2,C_3 of the best and worst iterations/trials for Zipf exponent: 1.5.

For Zipf exponent 30.0, the cautions on data movement begins to reduce, because the parameter C_1 is getting higher. In the Section 5.5.6.3, we enter in more detail about the results obtained by our Bayesian optimizer component.

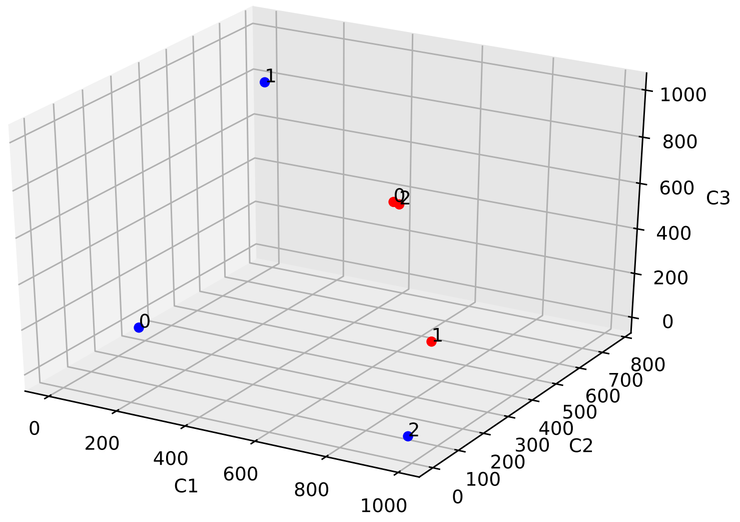


Figure 18 – C_1,C_2,C_3 of the best and worst iterations/trials for Zipf exponent: 30.0.

5.5.6.3 *Detailed Best/Worst Trials*

To better understand the impact of the C_1, C_2 and C_3 parameters to the C_{MeanRT}, we present two analysis:

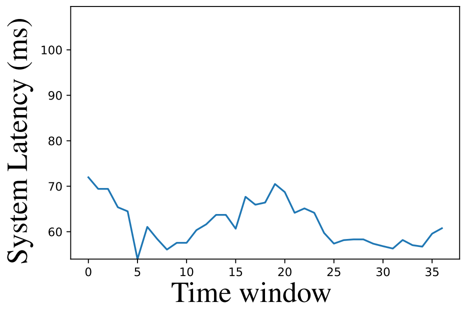
- The raw values for C_1, C_2 and C_3 and their C_{MeanRT} value in Tables 9, 10 and 11;

- The $AverageMRT \in t$ data-points collected from the new scheme deployment until the system stabilization in Figures 19, 20 and 21.

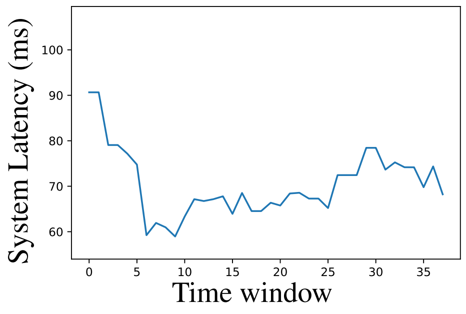
In the evaluation of the workload with Zipf exponent equals to 0.1, while the first and third execution reduced the C_{MeanRT} below of 70 ms, the second execution increased C_{MeanRT} to 90 ms in the beginning by the Figure 19. This happened because differently from the other executions, the second one set less importance to the data movement as shown by Table 9. As a result, node throughput capacity was overloaded causing higher latency.

Table 9 – Summary for Zipf Exponent = 0.1

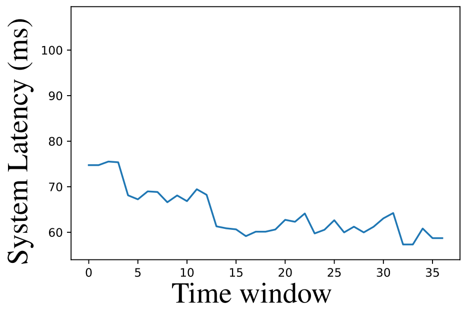
Execution ID	**Best** C_{MeanRT}	C_1	C_2	C_3
0	61.6949	8.9987	30.7529	151.7824
1	70.4973	17.8264	17.4694	6.4869
2	64.0585	10.0192	5.5387	362.4164



(a) Execution ID = 0.



(b) Execution ID = 1.



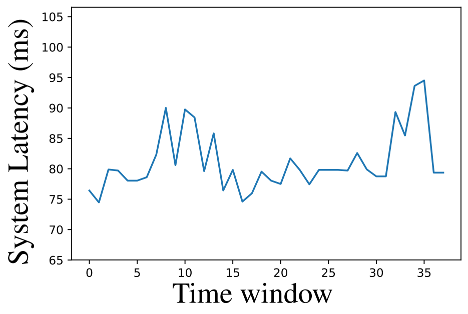
(c) Execution ID = 2.

Figure 19 – System latency behavior of the best iterations for Zipf Exponent = 0.1.

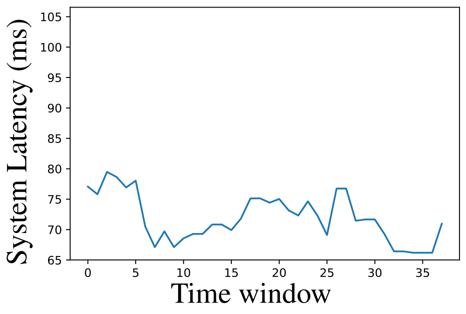
In the evaluation of the workload with Zipf exponent 1.5, different values for PopRing parameters were found as shown in Table 10. The Bayesian optimizer had to decrease the values for C_1 and increase the values for C_3. Despite that modification from the best parameters for Zipf exponent 0.1, a similar proportional setting $C_1 < C_2 < C_3$ remained as the best one at collecting the smallest $AverageMRT \in t$ data-points as shown by the Figure 20. Finally, it is interesting to note that the first execution maintained the CMeanRT on 80 ms or above. This happened because differently from the other executions, the importance of the load imbalance was set too high.

Table 10 – Summary for Zipf Exponent = 1.5.

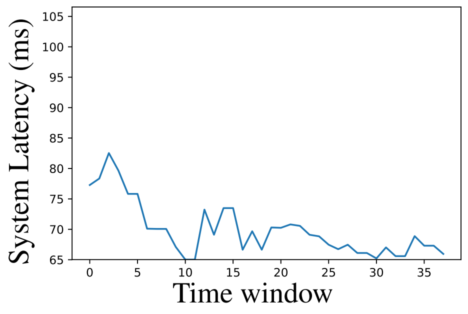
Execution ID	**Best** C_{MeanRT}	C_1	C_2	C_3
0	81.1461	679.0934	129.8970	727.4305
1	71.9033	0.5835	14.2799	986.5002
2	69.8881	0.3616	2.5198	975.4972



(a) Execution ID = 0.



(b) Execution ID = 1.



(c) Execution ID = 2.

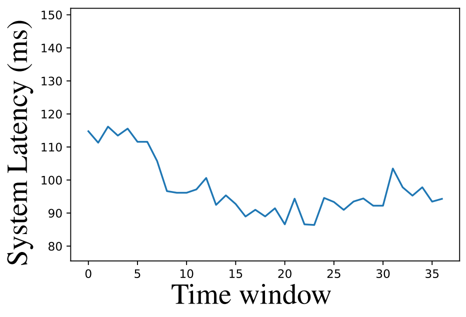
Figure 20 – System latency behavior of the best iterations for Zipf Exponent = 1.5.

In the evaluation of the workload with Zipf exponent 30.0, i.e., high skewed workload, totally different adjustment were required by the Bayesian optimizer as shown in Table 11. Now, higher values for C_1 are required once the load imbalance is degrading the overall response time of the system. Also that results implies that the negative impact of the data movement is less important than the negative impact of the load imbalance.

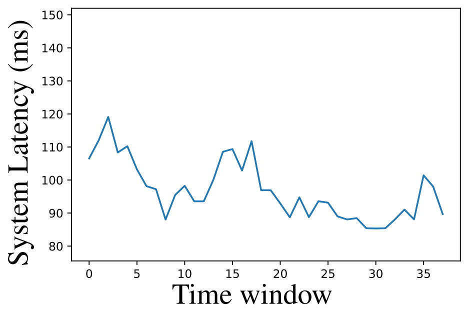
Also, it is interesting to note that the second execution set almost zero importance do C_3, thus suffering from high data movement costs, while the other executions had better C_{MeanRT} while setting similar importance to C_1 and C_3 as shown by the Fig. 21.

Table 11 – Summary for Zipf Exponent = 30.0

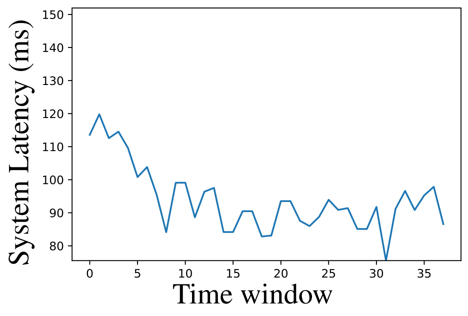
Execution ID	**Best** C_{MeanRT}	C_1	C_2	C_3
0	97.7086	911.7298	41.0751	975.2969
1	96.8466	696.3859	504.3661	0.5935
2	93.9902	951.4281	6.6938	998.5954



(a) Execution ID = 0.



(b) Execution ID = 1.



(c) Execution ID = 2.

Figure 21 – System latency behavior of the best iterations for Zipf Exponent = 30.0.

The summary of the best execution for each level data access skew as shown in Table 12. It is possible to note that, different parameters were found as the best ones for each Zipf exponent workload. As expected, the best parameters are those that require less data movement to the extent which the negative impact of load imbalance is more worth than the negative impact of data movement. On the other hand, it is also true that when data access skew is too high, such as Zipf equals to 30.0 for our environment, a higher importance should be given to the load imbalance cost.

Finally, it is important to remember that the best found parameters are optimized for reducing the system average latency, which in practise, eliminates internal bottlenecks due to overloaded/underloaded nodes. Also, it important to remember that the improvement in response time obtained by our solution should improve the system user as long as no external bottleneck are acting on the system.

Table 12 – Summary of the best BO executions according to the Zipf exponent used in the workload.

Zipf Exponent	**Best** C_{MeanRT}	C_1	C_2	C_3
0.1	61.6949	8.9987	30.7529	151.7824
0.5	66.2489	24.7884	18.8236	993.3504
1.1	69.8604	3.9913	10.2706	991.4982
1.5	69.8881	0.3616	2.5198	975.4972
2.0	75.4091	15.4536	12.3539	967.3205
30.0	93.9902	951.4281	6.6938	998.5954

6 CONCLUSION AND FUTURE WORK

6.1 Conclusion

The combinatorial aspect on the huge search space of our replica placement problem brought theoretical scalability issues for using the brute force algorithm while the non-linearity and non-convexity reduced the space of techniques able for solving the problem in an efficient time manner. Fortunately, meta-heuristic techniques, such as GA, were found in the literature with the property of finding sufficiently good solutions for our type of problem.

The experimental simulations of the PopRing replica placement component, based on GA in Section 4, have proved that our mathematical modeling is able to support the search for a replica placement scheme able to load balance the get requests in order to reduce data access skew. Additionally, our modeling also considered the side-effect of data maintenance and movement by minimizing replica placement maintenance and reconfiguration objectives. These last ones represented the cost of already existent replicas and replica movements.

The multi-objective problem of minimizing load imbalance, replica placement maintenance and reconfiguration was modeled as single-objective using weighted-sum technique. This modeling has the advantage of supporting flexible importance for each objective in order to adapt to different environment necessities. In contrast, this modeling required prior knowledge to find the proper importance setups for each environment necessity. Also, the optimum is difficult or even unfeasible to reach when it is not possible to improve one objective function without worsening other objectives.

To overcome that prior knowledge requirement and optimum difficult, we defined a new object function based on a system performance metric and proposed a black-box hyper-parameter optimization component for finding a proper importance setup of the PopRing parameters according to a system performance metric. This one, defined as the internal mean response time of the get requests between the service and storage nodes.

Different from the other three objective functions, the performance-based objective function was very costly to evaluate since it would be necessary to deploy new replica placement schemes in order to collect the system performance metric. This way, beyond black-box property, the optimizer should be sample efficient to reduce the number of trials to find good PopRing parameters. Fortunately, the literature has been researched black-box optimization techniques, such as BO, to tune hyper-parameters of a unknown function.

To verify the new component in practice, a OpenStack-Swift KVS was deployed in our laboratory environment. Then a PopRing hyper-parameter optimization component based on BO was used to setup the other three objective functions of the PopRing replica placement component based on GA to find a new replica placement scheme without any human intervention under different levels of data access skew.

As result, after few trials of using the replica placement component to generate different replication schemes, our PopRing hyper-parameter optimization component was able to find satisfactory parameters to minimize the performance-based objective function as described in Section 5.

6.2 Future Work

6.2.1 Heterogeneous Storage Node

The PopRing replica placement component uses a load imbalance objective function to measure the level of the data access skew on the storage nodes. Unfortunately, this measure considers that all the storage nodes have the same performance capacity. As consequence, in theory, a heterogeneous setup in which some storage nodes are more robust than others regarding performance metrics such as IOPS, CPU, memory etc would reduce the efficient of the load imbalance objective function.

As future work for supporting heterogeneous performance capacity of storage nodes, the load imbalance objective function could be improved to leverage the characteristics of each storage node. In the beginning, all storage nodes would have the same weight for the load imbalance evaluation, but during the functioning of the system, new weight would be learned and setup for each storage. These new weights could be evaluated using techniques that do not require prior human knowledge such BO and reinforcement learning.

6.2.2 Efficient Deployment

Other limitation is found in the PopRing hyper-parameter optimization component in which some trials must be performed before applying the most adequate replica placement scheme. These trials require unnecessary data movement while trying to find a good replica placement scheme to the current system state.

As future work for avoiding unnecessary deployment trials, the performance-based objective function could be used to label the correct parameters of PopRing replica placement component for different system states. Then, a model could be trained and generalized from these labeled samples in order to support the querying of adequate parameters given any system state.

BIBLIOGRAPHY

AMARAN, S.; SAHINIDIS, N. V.; SHARDA, B.; BURY, S. J. Simulation optimization: a review of algorithms and applications. **Annals of Operations Research**, Springer, v. 240, n. 1, p. 351–380, 2016.

BARTZ-BEIELSTEIN, T.; ZAEFFERER, M. Model-based methods for continuous and discrete global optimization. **Applied Soft Computing**, Elsevier, v. 55, p. 154–167, 2017.

BOUSSAÏD, I.; LEPAGNOT, J.; SIARRY, P. A survey on optimization metaheuristics. **Information Sciences**, Elsevier, v. 237, p. 82–117, 2013.

CAVALCANTE, D. M.; FARIAS, V. A. de; SOUSA, F. R.; PAULA, M. R. P.; MACHADO, J. C.; SOUZA, J. N. de. Popring: A popularity-aware replica placement for distributed key-value store. In: **CLOSER**. [S.l.: s.n.], 2018. p. 440–447.

CHEKAM, T. T.; ZHAI, E.; LI, Z.; CUI, Y.; REN, K. On the synchronization bottleneck of openstack swift-like cloud storage systems. In: IEEE. **Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on**. [S.l.], 2016. p. 1–9.

CHO, J.-H.; WANG, Y.; CHEN, R.; CHAN, K. S.; SWAMI, A. A survey on modeling and optimizing multi-objective systems. **IEEE Communications Surveys & Tutorials**, IEEE, 2017.

DECANDIA, G.; HASTORUN, D.; JAMPANI, M.; KAKULAPATI, G.; LAKSHMAN, A.; PILCHIN, A.; SIVASUBRAMANIAN, S.; VOSSHALL, P.; VOGELS, W. Dynamo: amazon's highly available key-value store. **ACM SIGOPS operating systems review**, ACM, v. 41, n. 6, p. 205–220, 2007.

FELBER, P.; KROPF, P.; SCHILLER, E.; SERBU, S. Survey on load balancing in peer-to-peer distributed hash tables. **IEEE Communications Surveys and Tutorials**, v. 16, n. 1, p. 473–492, 2014.

GILBERT, S.; LYNCH, N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. **Acm Sigact News**, ACM, v. 33, n. 2, p. 51–59, 2002.

GROSSMAN, M.; THIELE, C.; ARAYA-POLO, M.; FRANK, F.; ALPAK, F. O.; SARKAR, V. A survey of sparse matrix-vector multiplication performance on large matrices. **arXiv preprint arXiv:1608.00636**, 2016.

HERLIHY, M. P.; WING, J. M. Linearizability: A correctness condition for concurrent objects. **ACM Transactions on Programming Languages and Systems (TOPLAS)**, ACM, v. 12, n. 3, p. 463–492, 1990.

HU, J.; WANG, Y.; ZHOU, E.; FU, M. C.; MARCUS, S. I. A survey of some model-based methods for global optimization. In: **Optimization, Control, and Applications of Stochastic Systems**. [S.l.]: Springer, 2012. p. 157–179.

KARGER, D.; LEHMAN, E.; LEIGHTON, T.; PANIGRAHY, R.; LEVINE, M.; LEWIN, D. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In: ACM. **Proceedings of the twenty-ninth annual ACM symposium on Theory of computing**. [S.l.], 1997. p. 654–663.

LI, T.; SHAO, G.; ZUO, W.; HUANG, S. Genetic algorithm for building optimization: State-of-the-art survey. In: ACM. **Proceedings of the 9th International Conference on Machine Learning and Computing**. [S.l.], 2017. p. 205–210.

LIU, S.; HUANG, X.; FU, H.; YANG, G. Understanding data characteristics and access patterns in a cloud storage system. In: IEEE. **Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on**. [S.l.], 2013. p. 327–334.

LONG, S.-Q.; ZHAO, Y.-L.; CHEN, W. Morm: A multi-objective optimized replication management strategy for cloud storage cluster. **Journal of Systems Architecture**, Elsevier, v. 60, n. 2, p. 234–244, 2014.

MAKRIS, A.; TSERPES, K.; ANAGNOSTOPOULOS, D. A novel object placement protocol for minimizing the average response time of get operations in distributed key-value stores. In: IEEE. **Big Data (Big Data), 2017 IEEE International Conference on**. [S.l.], 2017. p. 3196–3205.

MAKRIS, A.; TSERPES, K.; ANAGNOSTOPOULOS, D.; ALTMANN, J. Load balancing for minimizing the average response time of get operations in distributed key-value stores. In: IEEE. **Networking, Sensing and Control (ICNSC), 2017 IEEE 14th International Conference on**. [S.l.], 2017. p. 263–269.

MANSOURI, Y.; TOOSI, A. N.; BUYYA, R. Cost optimization for dynamic replication and migration of data in cloud data centers. **IEEE Transactions on Cloud Computing**, IEEE, 2017.

MARLER, R. T.; ARORA, J. S. Survey of multi-objective optimization methods for engineering. **Structural and multidisciplinary optimization**, Springer, v. 26, n. 6, p. 369–395, 2004.

MOCKUS, J. **Bayesian approach to global optimization: theory and applications**. [S.l.]: Springer Science & Business Media, 2012. v. 37.

MSEDDI, A.; SALAHUDDIN, M. A.; ZHANI, M. F.; ELBIAZE, H.; GLITHO, R. H. On optimizing replica migration in distributed cloud storage systems. In: IEEE. **Cloud Networking (CloudNet), 2015 IEEE 4th International Conference on**. [S.l.], 2015. p. 191–197.

PARDALOS, P. M.; ŽILINSKAS, A.; ŽILINSKAS, J. **Non-convex multi-objective optimization**. [S.l.]: Springer, 2017.

SHAHRIARI, B.; SWERSKY, K.; WANG, Z.; ADAMS, R. P.; FREITAS, N. D. Taking the human out of the loop: A review of bayesian optimization. **Proceedings of the IEEE**, IEEE, v. 104, n. 1, p. 148–175, 2016.

SNOEK, J.; LAROCHELLE, H.; ADAMS, R. P. Practical bayesian optimization of machine learning algorithms. In: **Advances in neural information processing systems**. [S.l.: s.n.], 2012. p. 2951–2959.

STOICA, I.; MORRIS, R.; KARGER, D.; KAASHOEK, M. F.; BALAKRISHNAN, H. Chord: A scalable peer-to-peer lookup service for internet applications. **ACM SIGCOMM Computer Communication Review**, ACM, v. 31, n. 4, p. 149–160, 2001.

TERRY, D. B.; DEMERS, A. J.; PETERSEN, K.; SPREITZER, M. J.; THEIMER, M. M.; WELCH, B. B. Session guarantees for weakly consistent replicated data. In: IEEE. **Proceedings of 3rd International Conference on Parallel and Distributed Information Systems**. [S.l.], 1994. p. 140–149.

VIOTTI, P.; VUKOLIĆ, M. Consistency in non-transactional distributed storage systems. **ACM Computing Surveys (CSUR)**, ACM, v. 49, n. 1, p. 19, 2016.

VOGELS, W. Eventually consistent. **Queue**, ACM, v. 6, n. 6, p. 14–19, 2008.

ZHENG, Q.; CHEN, H.; WANG, Y.; ZHANG, J.; DUAN, J. Cosbench: cloud object storage benchmark. In: ACM. **Proceedings of the 4th ACM/SPEC International Conference on Performance Engineering**. [S.l.], 2013. p. 199–210.

ZHUO, L.; WANG, C.-L.; LAU, F. C. Load balancing in distributed web server systems with partial document replication. In: IEEE. **Parallel Processing, 2002. Proceedings. International Conference on**. [S.l.], 2002. p. 305–312.