# ALOCS: An Allocation-Aware Key-Value Repository

Patrick A. Bungama[1], Wendel M. de Oliveira[1], Flávio R. C. Sousa[2], Carmem S. Hara[1]

[1] Universidade Federal do Paraná, Brazil
pabungama, wendel, carmem@inf.ufpr.br
[2] Universidade Federal do Ceará, Brazil
flaviosousa@ufc.br

**Abstract.** Large volumes of data produced every day brought new challenges for developing efficient ways to extract, store and access the data. One approach to support increasing storage capacity is to exploit distributed storage repositories, which can be used to implement the physical layer of a Database Management System (DBMS). One of the difficulties reported by DBMSs that adopted a Distributed Hash Table (DHT) as their storage back-end is the high cost of communication for query processing. This is because DHTs do not offer control over the location of the data, and thus data involved in a query may be spread over different servers. This article presents ALOCS, a distributed data repository which provides to the application control over data locality. ALOCS adopts the key-value data model, and allows a set of key-value pairs to be grouped into buckets, that are stored at a server determined by the application. This control allows data commonly used together to be allocated at the same server, reducing the amount of inter-server communication for processing queries. This ability is essential to improve the performance for processing queries in a distributed setting. We have implemented ALOCS and conducted an experimental study that shows the efficiency of the system.

Categories and Subject Descriptors: H.3 [**Information Storage and Retrieval**]: Information Storage

Keywords: Allocation Control, Fragmentation, Key-Value, Placement, Repository, Storage

## 1. INTRODUCTION

The evolution of computer systems and easy access to the Internet led to the production of large volumes of data [Gantz and Reinsel 2012]. This avalanche of data brought new challenges, such as the definition of efficient ways to collect, store, access and extract this data [Yin and Kaynak 2015].

One approach to support increasing storage capacity is to exploit distributed storage repositories. A distributed repository can be used as the physical layer of a Database Management System (DBMS), as illustrated in Figure 1. In such a *layered* architecture, it is desirable that the interface between the repository and the DBMS components consists of a well-defined, fixed set of operations. This feature creates DBMSs with *pluggable* storage engines, allowing different storage structures to be used, while maintaining the same query language for the user and applications. Such an architecture is adopted by relational DBMSs (MySQL, MariaDB), XML (Zorba), and more recently, NoSQL (MongoDB).

Given that distributed hash tables (DHTs) have been proposed to provide a general purpose interface for indexing and distributing data, some DBMSs have adopted them to implement the underlying distributed storage [de S. Rodrigues et al. 2009] [Arnaut et al. 2011] [Ribas et al. 2011].

One of the difficulties reported by systems that followed this approach is the high cost of communication for query processing. This is because DHTs support a key-value model and offer an interface to access individual values based on their keys (operations get-put-rem). However, queries in DBMSs involve a set of related values. Moreover, in a DHT, each server is responsible for storing an interval

This work has been partially supported by CNPq (Grant number 455214/2014-0).

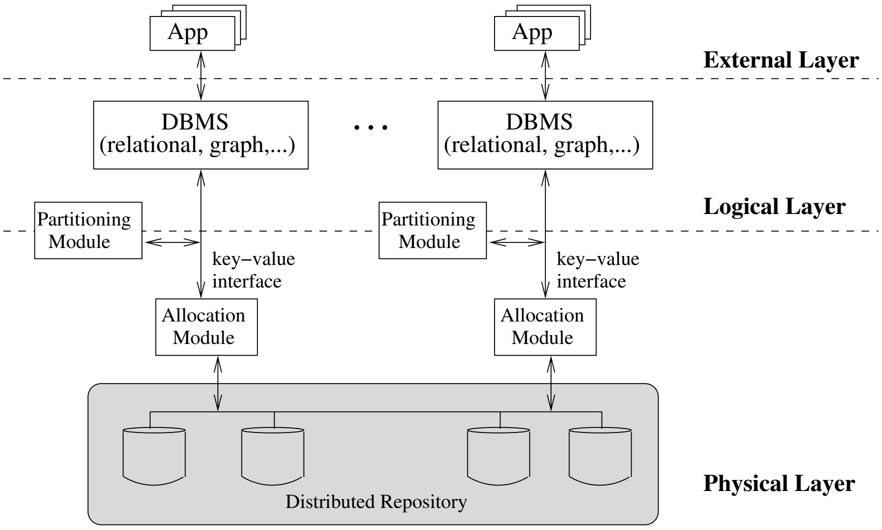


Fig. 1. Layered architecture of a DBMS

of values which are the result of a hash function on the key. In Figure 1, this function is depicted as the Allocation Module. Since the DHT does not offer control over the location, data involved in a query may be spread over different servers and may require communication for retrieving each element individually.

Therefore, a possible alternative is to adopt a Distributed File System (DFS) as the DBMS storage module. They use files as storage units . A DFS provides access to remote files from any computer of a network as if they were local. It is responsible for organizing, storing, getting, sharing and protecting files [Rani et al. 2014]. Metadata are kept to store the files location. The access to metadata, when centralized, generates from 50% to 80% of all network traffic [Ousterhout et al. 1985]. Although the size of the metadata is generally small compared to the system storage capacity, it may become a potential bottleneck. Avoiding this bottleneck is essential in order to achieve high performance. Some DFSs allow applications to specify in which server a given file must be stored; that is, they support control over data location.

The location of a stored object in a distributed environment influences the performance of queries over these data. Therefore, controlling the data location is very important as it can ensure that data usually used together can be allocated at the same server, reducing the amount of communication to process queries. Co-allocation prevents access to multiple servers, and can bring information closer to applications that use them more often. This factor is essential to avoid high latency in a distributed environment as a WAN [Corbett et al. 2013].

Consider as an example a graph database on the domain of product offers as depicted in Figure 2. The data partitioning is based on star structures, where each node is stored with its literal (basic type) properties. The resulting data fragments are depicted enclosed in dashed line rectangles. In the example, there are two *product* fragments, and four *offer* fragments. Suppose that each fragment is mapped to a key-value pair in the distributed repository. If the repository is based on a DHT, there is no guarantee that a *product* fragment is stored in the same server as its *offer*s. However, in a DFS-based repository with data location control it is possible to allocate the fragments as depicted in Figure 2. This configuration reduces the number of inter-server communication for queries involving both *product*s and their *offer*s.

This article presents ALOCS, a distributed data repository that adopts the key-value model, but allows a set of pairs to be grouped in a single structure, called *bucket*, whose location is treated

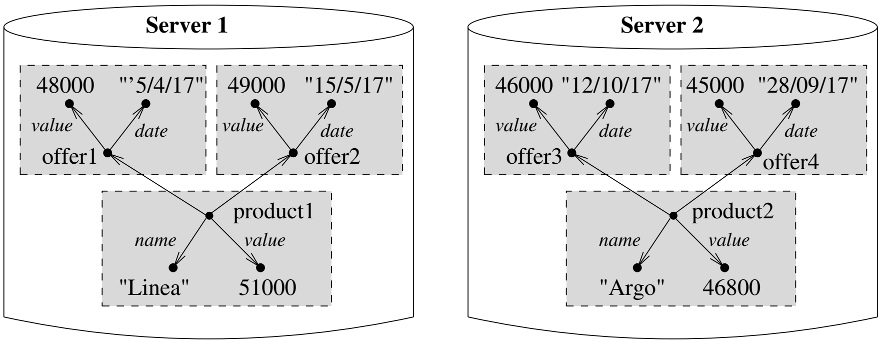


Fig. 2. Data fragmentation and allocation

in a controlled manner and guided by an application-tailored *allocation* module. Thus, although the interface for the application is similar to NoSQL repositories (based on get-put-rem operations), ALOCS extends insertion operations with locality-based parameters. Buckets consist of sets of key-value pairs, and correspond to the data communication unit between servers. Thus, although the user interface is based on individual key-value pairs, a `get(key)` operation involves retrieving the bucket in which the `key` is stored from possibly a remote server. Since ALOCS also supports caching, the *bucket* is kept in the local cache, and thus subsequent accesses to keys in the same *bucket* do not require further communication. By introducing the concept of *buckets*, ALOCS keeps the interface of key-value repositories unchanged, while allowing pairs that are commonly accessed together to be grouped. Here, we apply the same idea of layered DBMS architectures: there is a distinction between the repository data model (key-value), and the physical model (set of key-value pairs – *bucket*). This feature, combined with the ability to control the locality of buckets, allows ALOCS to be used as storage *back-end* to database management systems, providing a model for distributed *clusterization.*

It is important to point out that ALOCS has not been designed for any specific data partitioning or allocation methods. Whether, its goal is to provide the storage infra-structure on which different methods can be developed, as depicted in Figure 2.

ALOCS adopts a DFS as its underlying storage system. As DFSs rely on a metadata structure to link the data to the server where it is stored, ALOCS allows replication of these information, and the number of replicas can be configured according to the volume of read and write operations expected by the application. That is, applications with a heavy load of read operations can have several replicas of the metadata to prevent them from becoming a bottleneck. However, for applications with a heavy load of write operations, the existence of many metadata replicas makes updates a bottleneck and therefore an architecture with a small number of metadata servers is more appropriate.

ALOCS has been implemented following a well-defined interface among its modules. This feature allows both the DFS and the metadata module to be replaced by other implementations and tools, as long as they provide the expected functions.

This article is an extended version of [Bungama et al. 2016], which highlights the distinctive features proposed by ALOCS: (1) a data model that supports access to individual key-value pairs, while grouping them into allocation-aware *buckets* in order to reduce inter-server communication; (2) caching of buckets; and 3) a modular design. To this end, we have included the interface specification of all modules that compose the system, as well as examples to show the flexibility of the proposed storage model, and a new experiment to determine the effectiveness of our design goals. The article is organized as follows: Section 2 discusses related work. Section 3 describes the storage model defined in this article as well as the architecture of the proposed solution. Section 4 presents the experimental results, which show the impact of the metadata system on read and write operations, as well as the effect of data co-allocation on the system performance. Our conclusions and future work are presented in Section 5.

## 2. RELATED WORK

Some important features for comparing distributed storage systems are the methods used to distribute and retrieve data, scalability and availability. Distribution and retrieval methods influence the system scalability, since they have a direct impact on the data retrieval time [Paiva and Rodrigues 2015]. Two techniques are commonly used as NoSQL storage back-end: Distributed Hash Tables (DHTs and Distributed File Systems (DFSs).

Among the DFSs, Hadoop Distributed File System (HDFS) [Shvachko 2010] and Google File System (GFS) [Ghemawat et al. 2003] do not implement mechanisms for applications to define the files location. They fragment files that are allocated to servers with more resources available: less workload in the case of HDFS, and less storage usage for GFS. Ceph DFS [Weil et al. 2006] and PVFS [Ross et al. 2000] are among the DFSs that support mechanisms for applications to control data location. Ceph is based on objects, and maps files to one or more objects. Objects are distributed among servers in a cluster based on a set of user-provided rules that define policies for distribution and replication of objects. This functionality ensures partial control over the location of data, which motivated the use of Ceph for implementing ALOCS. PVFS also fragments data files and allows applications to define the allocation of files. However, it does not support replication.

One of the difficulties of using a DFS as the back-end of a distributed DBMS is that files are their storage unit, which is a much coarser granularity than the units of data considered by DBMSs, based on data items. Given that DHT-based protocols consider key-value pairs as their storage unit, some NoSQL databases, such as Scalaris [Schütt et al. 2008] and Memcachedb [Tudorica and Bucur 2011], use DHTs as their data distribution mechanism. However, DHTs usually do not support data allocation control.

Some DHTs, such as Scalaris[1], propose to overcome this problem by using a hash function that follows the lexicographical order of the keys. This approach, however, requires the application to modify or adapt the keys, such as adding prefixes, in order allocate them in the same server. Nevertheless, there is no guarantee that pairs with the same prefix are indeed stored at the same server. Autoplacer [Paiva et al. 2014] proposes a different approach for minimizing inter-server communication. It is developed on the Infinispan DHT[2], and applies a probabilistic algorithm for associating servers to their most frequently accessed data items. Based on the collected statistical information, it *replicates* data items on servers that require the majority of remote accesses. The focus of Autoplacer is on self-tuning replica placement. In the architecture shown in Figure 2, the allocation module and distributed repository are implemented by the underlying Infinispan DHT. However, there is an additional *replication module* that controls where copies of data items stored in the DHT must be placed. In ALOCS, methods for grouping key-value pairs and buckets and their allocation are orthogonal to the adopted storage model. Thus, it is possible to envision a system in which Autoplacer's placement strategy can be used to implement the allocation module *on top of ALOCS*. Such a system would not need to rely on two systems to control data placement (from the DHT and Autoplacer), but would be able to control the locality of data directly on ALOCS.

PNUTS [Cooper et al. 2008] and Spanner [Corbett et al. 2013] are two distributed repositories based on distributed file systems. Both support the idea of grouping data items into structures that ensure co-allocation (called tablets by PNUTS, and directory by Spanner). However, unlike ALOCS, which is based on a key-value model, they support a simplified relational data model. Moreover, neither is publicly available. PNUTS is used as back-end storage for Yahoo Web applications, and Spanner has been designed to support Google's applications with complex and evolving schemas. The original

[1] http://scalaris.zib.de
[2] http://infinispan.org

motivation for developing ALOCS was the inexistence of a freely available distributed repository with allocation control to support ClusterRDF, a partitioning and allocation system for RDF [Schroeder and Hara 2015]. The first version of ClusterRDF has been developed on Scalaris DHT. Although the system tried to take advantage of the lexicographical ordering of keys, there was no guarantee that fragments that were supposed to be allocated in the server indeed were co-allocated. Such a control is provided by ALOCS, while keeping a simple interface based on key-value pairs, similar to the interface of DHTs.

## 3. ALOCS: A KEY-VALUE DATA STORE WITH LOCALITY CONTROL

This section presents ALOCS, a key-value repository that provides control over data placement. It has been designed to support DBMSs in which the location of data involved in a query is an important performance factor. The following sections describe the storage model (Section 3.1), the architecture (Section 3.2) and the implementation of ALOCS (Section 4).

### 3.1 Storage Model

The storage model supported by ALOCS is based on four concepts:

—*key-value pair*: corresponds to the access unit;
—*bucket*: a set of key-value pairs that corresponds to the communication unit among servers;
—*directory*: a set of *buckets* that defines the replication unit;
—*server*: a set of directories which corresponds to a physical server of the distributed storage system.

The storage model concepts define a hierarchy used to store and locate the data (physical location of a key-value pair) as depicted in Figure 3. They describe a path (`/Server/Directory/Bucket/Key`). ALOCS has been designed to be used as the back-end storage of a DBMS. Due to its flexible hierarchy, different data models can be mapped to ALOCS storage units. Figure 4(b) shows a possible mapping of a relational database depicted in Figure 4(a). Here, tuples are stored in key-value pairs and a set of tuples in buckets. Horizontal and vertical fragmentation, as proposed in [Ribas et al. 2011], can also be mapped to the proposed hierarchy. This dataset is the same as the one depicted in Figure 2 in a graph-based model. In this example, nodes and their adjacency lists are stored in key-value pairs, and connected nodes in distinct pairs are grouped in buckets. It is worth mentioning that the application developed on top of ALOCS is responsible for defining how the database is fragmented into buckets and directories, and for allocating them at physical servers. ALOCS is orthogonal to the fragmentation strategy, but provides a storage framework that allows the application to control data clustering in a distributed setting. Clustering is a traditional database optimization technique. ALOCS supports key-value pairs to store units of information in the logical level, such as relational tuples and graph nodes.

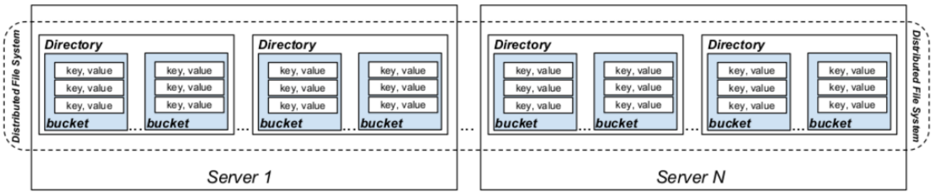
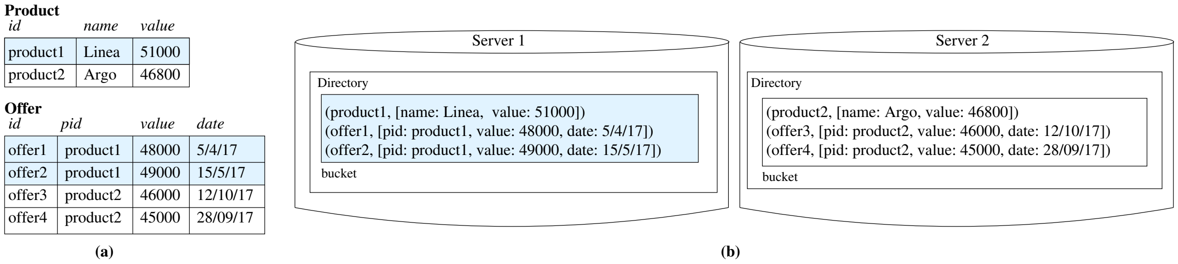


Fig. 3. Storage Model



**Product**

| *id* | *name* | *value* |
|---|---|---|
| product1 | Linea | 51000 |
| product2 | Argo | 46800 |

**Offer**

| *id* | *pid* | *value* | *date* |
|---|---|---|---|
| offer1 | product1 | 48000 | 5/4/17 |
| offer2 | product1 | 49000 | 15/5/17 |
| offer3 | product2 | 46000 | 12/10/17 |
| offer4 | product2 | 45000 | 28/09/17 |

**(a)**

(product1, [name: Linea, value: 51000])
(offer1, [pid: product1, value: 48000, date: 5/4/17])
(offer2, [pid: product1, value: 49000, date: 15/5/17])

(product2, [name: Argo, value: 46800])
(offer3, [pid: product2, value: 46000, date: 12/10/17])
(offer4, [pid: product2, value: 45000, date: 28/09/17])

**(b)**

Fig. 4. ALOCS as a storage back-end for a relational DBMS

Buckets are similar to disk pages, but in a distributed setting. Applications can minimize communication among servers to process queries by storing in the same bucket data that are frequently accessed together. Since buckets correspond to units of communication, the first access of a key-value pair in a bucket requires the bucket to be transmitted from the storage server to the local cache. Once in the cache, subsequent accesses to key-value pairs in the same bucket do not require additional transmissions. The maximum size of a bucket is defined by the application.

Directories are replication units, as well as placement units, since all buckets that compose a directory are stored in the same physical server. The ability to group buckets in a single server is important to support strong consistency. Some NoSQL systems guarantee ACID properties only if all the data involved in a transaction is stored in the same server. The notion of a replication unit is also important for read-intensive applications: directories on high demand can be replicated in several servers in order to provide higher throughput.

### 3.2 Architecture

The architecture of ALOCS considers a set of nodes (physical machines), as illustrated in Figure 5. In addition to support distributed storage of data, the system is responsible for maintaining the *metadata*, which contains information about the hierarchy of servers, directories and buckets.

Nodes that compose the distributed repository can act only as data servers (such as the nodes at the bottom of Figure 5) or data and metadata servers (such as the nodes at the top). Metadata are replicated in all metadata servers. Data servers are designed to support the storage of large volumes of data. As nodes can assume different roles, it is possible to create a larger set of data servers without incurring on costly metadata replication in all of them. Thus, it is possible to adjust the amount of data and metadata servers according to the volume of read and write operations of each application. The number of data servers should be enough to store the data, and answer read operations. The number of metadata servers, on the other hand, must be defined based on the volume of write operations. High volume of updates may impact the performance of the system because they may require updates on the metadata, which is replicated on all metadata servers.

The main components of the system are: the control module, the storage module and the metadata module. They have been designed to provide a standard interface in order to be "pluggable" components of the system. This feature is important because several implementations of the modules can be developed based on different tools and data management systems, as long as they provide the module standard interface. These modules can then be combined to generate different instantiations of ALOCS. Section 4 describes one such implementation. In the following sections, we present details of each module.

3.2.1 *Control Module.* The control module is responsible for providing the user interface of the system. It receives requests from the application programs and communicates with the metadata and storage modules in order to obtain the required information. For example, consider that the

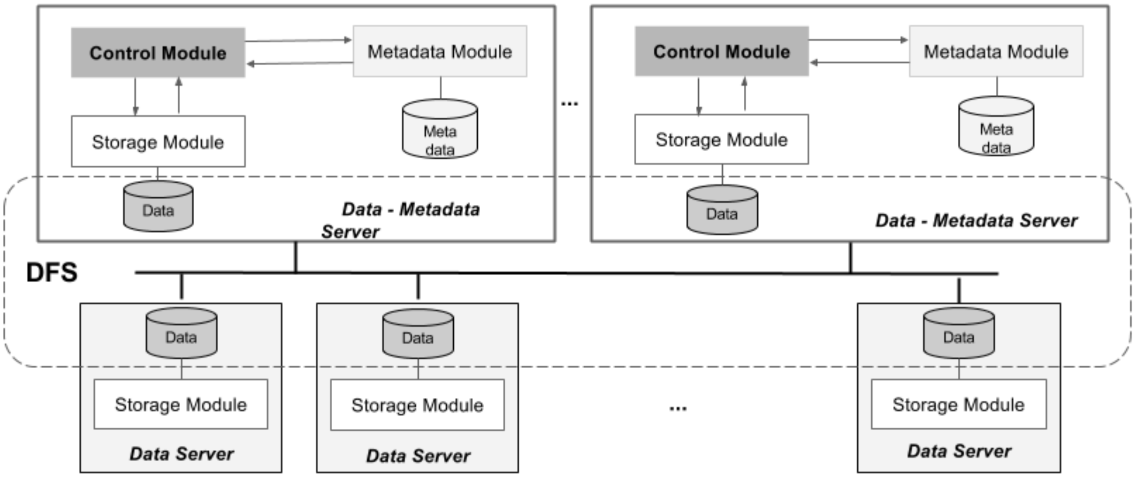


Fig. 5. ALOCS architecture

application submits an operation to store a key-value pair `put(k,v)`, as illustrated in Figure 6. First, it has to determine the location to store the information. Thus, it contacts the metadata module to request the information, which in turn returns the path server/directory/bucket (`s/d/b`) in which the key `k` must be stored. With this information in hand, the key-value pair is sent to the storage module to be stored in the proper location.

One of the goals of ALOCS is to keep its interface as simple as traditional key-value repositories, which is based on put-get-remove operations, but with the ability to control the location of the data. Associating each individual key to a server/directory/bucket may be costly, since every operation would require an access to the metadata module in order to get the location of the pair. To minimize this cost, we associate each bucket with an *interval* of keys, defined at the time of the bucket creation. Recall that the maximum size of a bucket is a configuration parameter given by the application. Thus, whenever a bucket is full, inserting a new key-value pair requires the bucket to be split, moving half of the pairs to a new bucket and updating the intervals associated with the original and new buckets. In order to maintain the user-defined location, the new bucket is created at the same server and directory of the original one. The interface of the Control Module that provide this functionality include the operations listed below.

—for key-value pairs:

```
get_pair (key)
put_pair(key, value)
rem_pair(key)
```

These functions have the same interface as traditional key-value repositories. As a consequence, an application that have already been designed to store data as key-value pairs can be easily modified to adopt ALOCS, by creating directories and buckets with appropriate key intervals.

—for buckets:

```
create_bucket(dirName, idBucket, initialKey, finalKey)
drop_bucket(dirName, idBucket)
isEmpty(dirName, idBucket)
```

Each bucket is associated with an interval of keys (`[initialkey, finalkey]`) with the `create_bucket` function. Insertion of a key-value pair requires finding the bucket associated to the interval that

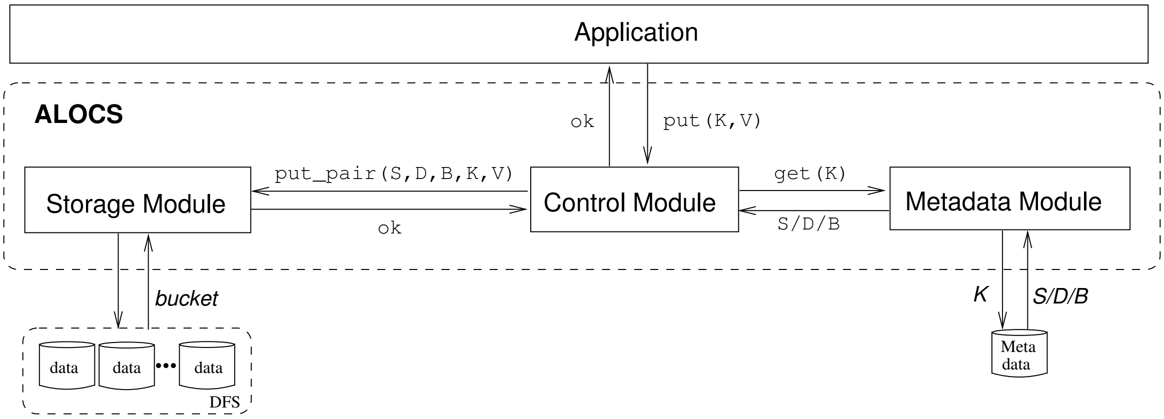


Fig. 6. Processing `put(key, value)`

contains the `key`, and checking whether the bucket is full.

—for directories:
`create_dir(serverName, dirName)`
`replicate_dir(serverOri, dirName, serverTarget)`
`drop_dir(serverName, dirName)`
`isEmpty(serverName, dirName)`

Directories are created in a specific server by the `create_dir` function. `replicate_dir` replicates an existing directory at server `serverOri` in `serverTarget` in order to support increasing volumes of read operations. Function `drop_dir` removes a directory from a server. If this is the last copy of a directory, the function only has an effect if the directory is empty; in other words, only empty directories are allowed to be completed deleted.

3.2.2 *Metadata Module.* The metadata module is responsible for storing the information about all the components of the system. For servers, it keeps the information of which ones are data servers and/or metadata servers, and the list of directories stored in each of them. For directories, it stores in which servers they are replicated; and for buckets, it keeps the corresponding key interval and the directory to which it belongs. The metadata module should also be able to provide fast responses for requests about the location of a given key. That is, to find the location of the bucket (`s/d/b`) that holds the key, and its associated interval.

The set of functions that compose the metadata module include the following:

—for key-value pairs:
`get(key)`
The function returns a set of paths `s/d/b` where the key should be stored (or retrieved) and the key interval associated with the bucket. Observe that the metadata module does not store information on individual keys. This function is used by the control module to obtain the location of buckets that should hold the key and, with this information in hand, call the storage model to get the bucket.

—for buckets:
`put_bucket(dirName, idBucket, initialKey, finalKey)`
`drop_bucket(dirName, idBucket)`
`getInterval(dirName, idBucket)`
Function `put_bucket` creates a metadata entry associating a bucket to its key interval. It is invoked

by the control module whenever a `create_bucket` is executed. `drop_bucket` removes a metadata entry, and `getInterval` returns the interval associated with the bucket.

—for directories:
`put_dir(serverName, dirName)`
`drop_dir(serverName, dirName)`
`getServers(dirName):` returns a set of server names
These functions are similar to the ones for buckets. `put_dir` creates a metadata entry associating a directory to a server. It is invoked whenever the control module executes functions `create_dir` or `replicate_dir`. Function `getServers` returns the set of servers that currently hold a copy of the directory.

3.2.3 *Storage module.* The main function of the storage module is to provide ALOCS with a distributed storage system. It is responsible for the interface between the control module and the distributed storage system, and for implementing the ALOCS model components (server, directory, buckets, and key-value pairs) on the physical storage model.

The storage module is also responsible for managing the cache. Thus, whenever a key is requested, it checks whether the corresponding *bucket* is already in cache to prevent possible inter-server communication. The cache is used only for read operations. Write operations are directly forwarded to the underlying storage system. Operations implemented by the Storage Module are similar to the ones in the control module. However, they include as parameters the complete path to find the components: s/d/b for key-value pairs and buckets, and s/d for directories. This information is obtained from the metadata module by the control module before calling functions in the storage module.

—for key-value pairs:
`put_pair(serverName, dirName, idBucket, key, value)`
`rem_pair(serverName, dirName, idBucket, key)`

—for buckets:
`create_bucket(serverName, dirName, idBucket)`
`split_bucket(serverName, dirName, idBucket)`
`get_bucket(serverName, dirName, idBucket)`
`drop_bucket(serverName, dirName, idBucket)`
`isEmpty(serverName, dirName, idBucket)`

—for directories:
`create_dir(serverName, dirName)`
`copy_dir(dirName, serverOri, serverTarget)`
`drop_dir(serverName, dirName)`
`isEmpty(serverName, dirName)`

The definition of standard interfaces for all modules that compose the system does not only allows the development of "pluggable components", but also serves as the basis for the development of additional mechanisms, such as load balancing, which requires replication and removal of directories.

## 4. ALOCS IMPLEMENTATION

We have implemented a first version of ALOCS following the architecture described in the previous section. The control and storage modules were implemented in C, while the metadata module was implemented in Java.The interface between the control and metadata modules was developed in Java Native Interface (JNI). Details about the implementation of the storage module and the metadata module are presented in the following sections.

### 4.1 Implementation of the Storage Module

We have used Ceph Storage Cluster [3] to implement the storage module. Our choice was motivated by the fact that Ceph has the ability to control the file location using the CRUSH algorithm. Another DFS with this property, such as PVFS, could also be used for implementing the storage module [Maeda and Hara 2015].

A Ceph Storage Cluster consists of a set of Object Storage Devices (OSDs). Each OSD corresponds to a single storage device or a combination of devices. In the logical level, files (or objects) stored on Ceph are organized in pools and placement groups. A pool manages a set of placement groups, which in turn store objects. The mapping of pools to physical locations is defined by CRUSH map rules [Weil et al. 2006]. These user defined rules determine the placement and replication strategies for pools. Among other things, a rule can define the location of the primary copy and secondary copies of a pool. A CRUSH map is compiled and kept by Ceph in binary format (the cluster map) in order to be used when pools are created. However, Ceph allows rules to be modified after pools are created, if necessary.

For ALOCS, a directory is the unit for allocation and replication. Thus, directories are implemented as Ceph pools, created in specific servers (Ceph OSDs). ALOCS *buckets* are implemented as Ceph objects. In order to store a set of key-value pairs in a Ceph object, we have defined a file structure composed of a header followed by a sequence of slots. The header contains the following information: the maximum number of pairs in the bucket, a bitmap to control the slots in the bucket that are empty (value 0) or occupied (value 1), and a sequence of `(key, offset)` pairs. Each slot starts at the `offset` byte in the file, and contains the value associated with the `key`. We limit the size of an object to coincide with the user defined size of a *bucket*, which was set to 64 KBytes in our current implementation.

ALOCS storage module communicates with Ceph using the librados API, which provides access to the Ceph RADOS service. This service is responsible to obtain the cluster map that contains pools location in the distributed file system, and then request objects from one of their associated OSDs.

### 4.2 Implementation of the Metadata Module

We have used Apache Zookeeper[4] to implement the metadata module. Zookeeper is a distributed, open-source coordination service for distributed applications [Junqueira and Reed 2009]. The storage units in Zookeeper are known as znodes. They are organized according to a hierarchical name space, similar to the structure of directories [Skeirik et al. 2013]. The metadata module has been implemented in Java and uses the set of primitives provided by the Java Zookeeper API.The main reason for using Zookeeper is its hierarchical data storage model, similar to the model proposed by ALOCS. The metadata module has two structures in order to provide fast retrieval of buckets that hold a specific key.

—**Physical Structure**: it is responsible for storing information about all storage devices used in the system, all directories and *buckets*. An example of the physical structure is shown in Figure 7.

—**Search structure**: it is an index structure used to find the bucket that contains a searched key. It is an adaptation of the interval tree in which nodes correspond to key ranges [Pal and Pal 2009]. Each node maintains, besides the key interval stored in the bucket, the maximum value of all the intervals in its sub-tree and the complete path to locate the *bucket* (/server/directory/bucket). An example of this tree is shown in Figure 8, where each znode is depicted with the key interval and the associated value in the form `[maxKeyInSubtree minKey maxKey location]`. The system ensures

[3] `ceph.com`

[4] `http://zookeeper.apache.org`

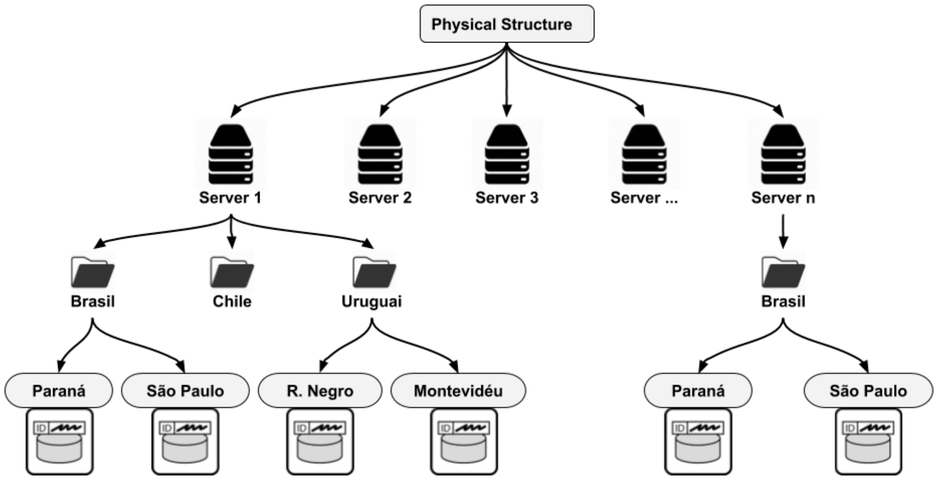


Fig. 7. Physical Structure of the Metadata Module

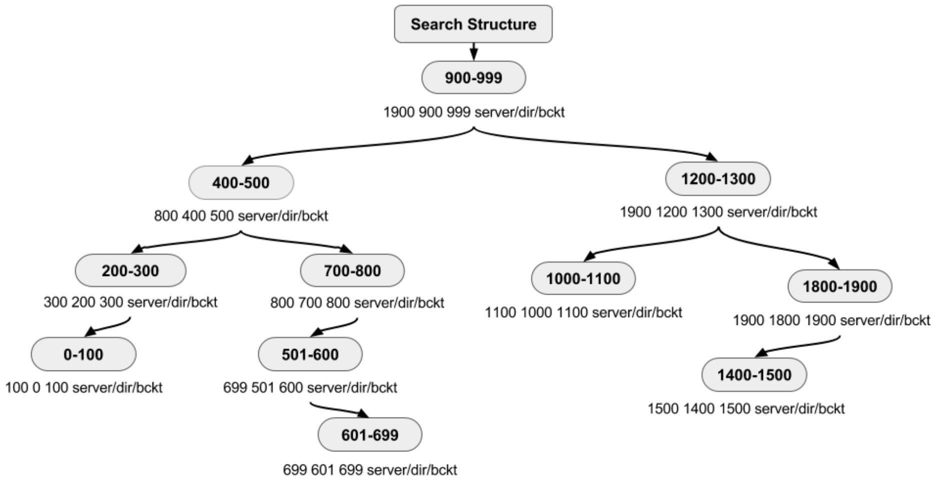

Search Structure
900-999
1900 900 999 server/dir/bckt
400-500
800 400 500 server/dir/bckt
1200-1300
1900 1200 1300 server/dir/bckt
200-300
300 200 300 server/dir/bckt
700-800
800 700 800 server/dir/bckt
1000-1100
1100 1000 1100 server/dir/bckt
1800-1900
1900 1800 1900 server/dir/bckt
0-100
100 0 100 server/dir/bckt
501-600
699 501 600 server/dir/bckt
1400-1500
1500 1400 1500 server/dir/bckt
601-699
699 601 699 server/dir/bckt

Fig. 8. Search structure of the Metadata Module

that there is no overlap between intervals at the time of the *bucket* creation. Thus, the algorithm for searching this tree is identical to searching a binary search tree, considering only the initial value of each interval stored in each node.

There is a direct correspondence of the physical and search structures to Zookeeper directory style structures, composed of znodes. All ALOCS metadata servers are configured to be members of a Zookeeper *ensemble*. As a result, the replication of all the information in both structures is automatically provided by Zookeeper. Next section presents results of experiments conducted to determine the efficiency of the system.

## 5. EXPERIMENTAL STUDY

Experiments were conducted in order to evaluate our implementation of ALOCS. We executed the experiments on a set of 3 virtual machines running Debian 7.0.0, each one with 30 GB of disk and 2 GB of RAM. Two machines were configured as storage servers, and the number of metadata servers ranged from one to three. Both the control and metadata modules run on all metadata servers in other to prevent remote communication for determining the bucket in which a given key is stored. Metadata servers constitute a Zookeeper cluster, while data servers compose a Ceph cluster.

There are some benchmarks for evaluating key-value systems. However, these benchmarks do not address the allocation problem. Thus, we executed the experiments on a synthetic database, in which we were able to define the co-allocation of data items and evaluate its effect on the system performance. On each data server we created a single directory, containing a set of *buckets*. Each *bucket* contains up to 100 key-value pairs. The value associated with each key is a random value of 50 bytes. Although Ceph allows replication of directories, this feature was not used in the experiments. All the values reported in this section correspond to the average of values collected by running each experiment 3 times.

Given that the co-allocation model proposed in this article is based on maintaining metadata that link key-value pairs to its physical storage, this section presents two experiments that aim at determining the cost of metadata maintenance. A third experiment describes the impact of co-allocation data in *buckets*. We have also conducted an experiment to determine the impact of multiple control and metadata servers on the system performance.

**Experiment 1: Cost of writing metadata**
To determine the cost of writing metadata, we executed a sequence of operations to generate *buckets* of distinct intervals in the same directory (operation `createBucket(dirName, idBucket, initialKey, finalKey)`). Each operation requires two metadata write operations: one in the physical structure, and another in the search structure. Furthermore, an object is written in the Ceph DFS, containing only the header, used to keep the information about the key-value pairs to be inserted in the *bucket*. We have considered three scenarios for the experiment: a single metadata server and operations ordered by the initial key value; a single metadata server and operations in a random key order; and three metadata servers with operations in a random key order. In the last scenario each operation requires the physical and search structures to be updated, on all three replicas stored at metadata servers.

The execution time (in milliseconds) of sequences containing 5 to 50 operations are shown in Figure 9. The graph shows that sequences with a random key order performs much better than sequences with ordered keys. This is because we do not maintain the metadata search structure balanced. Thus, when the keys are ordered, the height of the metadata search structure has linear growth with the number of operations. As a consequence, the search tree becomes a list. In fact, this is the worst case scenario for any set of `createBucket` operations. For 50 operations the total run time was 4650 milliseconds (93 millisecond per operation). At the end of the execution, the height of the search structure was 50, which explains the higher execution time compared to sequences with random key order insertions.

For random key sequences, the total execution time for 50 operations was 349 milliseconds (7 milliseconds per operation), which is 13 times lower compared to the ordered sequence. The average search structure height reached by the end of the sequence was 6. This shows the huge impact that the height of the tree has on the system performance. Thus, we intend to implement a mechanism to keep the search structure balanced in the future. An important point to notice is that the difference on the execution times for scenarios with a single metadata server and with 3 metadata servers is negligible for sequences containing up to 40 operations. For 50 operations the metadata replication increases the execution time by 34%. This shows that for applications with a heavy load of read operations and small number of buckets creation, it is possible to replicate the metadata in different nodes in order provide scalability and higher throughput.

**Experiment 2: Cost of reading metadata**
In the second experiment, we evaluated the time for reading metadata. Since writing key-value pairs

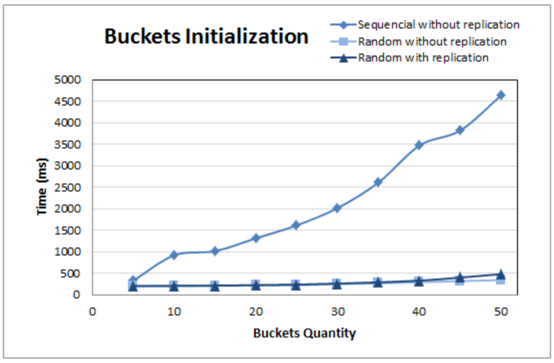


Fig. 9. Metadata writing time

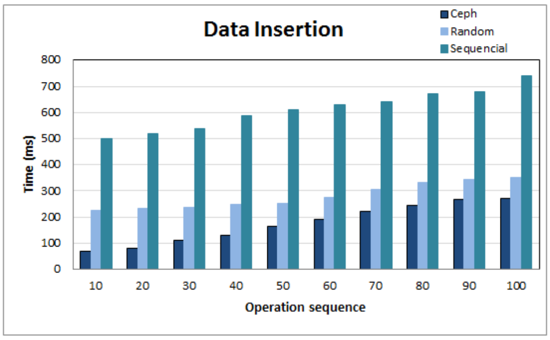


Fig. 10. Data writing time - metadata reading time

requires reading the metadata in order to determine the bucket location, we executed sequences of key-value pair insertions into *buckets* that have previously been created, as reported in Experiment 1. These operations were submitted to a single control module of ALOCS, running on a server that acts also as the system's single metadata server.

Figure 10 shows the total execution time (in milliseconds) of sequences composed of 10 to 100 `put (k, v)` operations. For each operation, the metadata is accessed to obtain the location of the *bucket* which owns the key, and then ALOCS uses the location information for writing the data at the proper server and directory. In Figure 10, the time spent for data writing directly on the DFS (without considering accesses to the metadata) is reported by the bar labelled as *Ceph*. The remaining bars show the execution time considering both the time for writing on the DFS and reading the metadata. The bar labelled as *Sequential* considered that buckets have been generated ordered by their initial key value, while the bar labelled as *Random* considered buckets created in random key order, as reported in Experiment 1. The overhead of reading the metadata is the difference between the first bar and the second (for random key order) and the third bar (for buckets created with ordered keys).

For a sequence of 50 operations, the recording time on the DFS was 163 milliseconds. The total time when the metadata was randomly generated was 253 milliseconds, which results in an average of 5 milliseconds per operation. When the metadata was generated sequentially, the time for accessing the metadata is much higher, with 13 ms per operation. This was expected, given the height of the resulting search structure. Although the cost to access the metadata seems high for small sequences, its *overhead* on the total execution time for larger sequences gets proportionally smaller. To see this, observe that for 10 operations, the time for accessing the metadata increases the time for writing only on the DFS almost four-fold; however, for 100 operations, the increase of the total execution time is only 25% for buckets generated in a random key order.

**Experiment 3: Impact of data co-allocation**

This experiment evaluated the impact of data co-allocation on the time to retrieve key-value pairs. To do so, we generated sequences of `get (k)` operations with distinct percentage of co-allocation of the required keys: 0%, 50% and 100%. More specifically, 0% co-allocation means that every key requested in the sequence is stored in a distinct bucket. For 50% co-allocation, half of the keys is co-allocated with another key in the sequence (50% of co-allocation). In the 100% co-allocation, for every bucket retrieved to process a `get(k)` operation, we also included in the sequence operations to get all the remaining keys in the same bucket. The sequences had sizes ranging from 10 to 100. Figure 11 presents the results.

As expected, the time for processing the sequence of operations with 100% of co-allocation is much smaller than the others. This is due to the fact that a *bucket* is the transmission unit of ALOCS. Thereof, only the first access to a key-value pair of the *bucket* requires communication with a remote

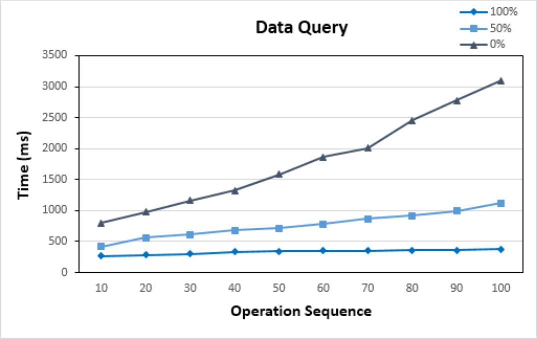


Fig. 11. Data reading Time

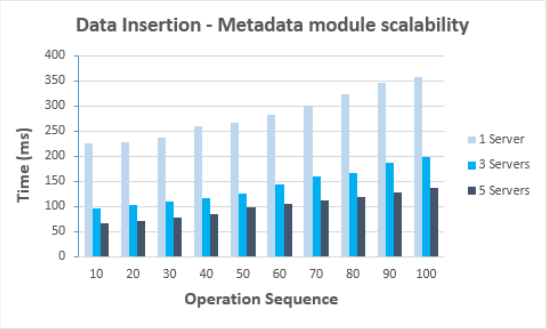


Fig. 12. Data inserting with different number of metadata servers

server. Subsequent readings of pairs in the same *bucket* are executed locally from the system cache. For 10 queries, the average time to process the operations was 269 milliseconds for 100% co-allocation. This time is increased by 55% with 50% co-allocation and 197.7% with 0% co-allocation.

For sequences of 100 operations, with 100% of co-allocation the average execution time was 377 milliseconds. This time increased to 1117 milliseconds for 50% co-allocation, which is 196% greater than the time with 100% of co-allocation, and to 3101 milliseconds for 0% co-allocation (722% higher). These results show the huge impact that data co-allocation has on the data retrieval time.

It is worth mentioning that traditional key-value repositories based on DHTs do not support the concept of *buckets*. Thus, each `get (k)` operation can potentially require an inter-server communication. This is not the case only when the pair is stored in the same server as the one requesting the data. Moreover, since the allocation of data into servers cannot be controlled by the user application, there is no way to force data placement on the server that access them most frequently.

The concept of *buckets* as communication units brings key-value repositories closer to traditional disk page accesses, but in a distributed setting. Clustering commonly accessed data items on disk pages has long been an important technique to optimize DBMS performance. Moreover, query plans are constructed to minimize the number of page transfers between disk and memory. The results in this experiment show that the same applies in a distributed setting. That is, minimizing the number of communication among servers is essential for optimizing the overall system performance. The similarity between page/data items in traditional DBMSs and buckets/pairs in ALOCS may allow traditional query optimization techniques to be applied, and highlights the importance of using a repository that supports data allocation control as a DBMS back-end.

**Experiment 4: Impact of multiple metadata servers** The metadata module is an important component of the ALOCS system. Using a single metadata server makes it a single point of failure. Thus, it is advisable to configure the system with multiple metadata servers. The effect of such a configuration on the time to create *buckets* have already been reported in Experiment 1. In this new experiment, we evaluated the effect of multiple metadata servers on the execution time of key-value pairs insertions in *buckets* that were previously created. This is similar to Experiment 2. However, as opposed to the previous experiment, we considered configurations with multiple control *and* metadata servers. In such a configurations `put(k, v)` operations can be distributed among control servers in order to achieve higher throughput.

We considered configurations with 1, 3, and 5 control/metadata servers. The odd number of metadata servers is recommended to guarantee the majority of the quorum, as required by write operations on Zookeeper. In Zookeeper, an even number of peers is supported, but it is normally not used because an even sized cluster requires, proportionally, more peers to form a quorum than an odd sized cluster.

For example, a cluster with 4 peers requires 3 to form a quorum, while a cluster with 5 also requires 3 to form a quorum. Thus, a cluster of 5 allows 2 peers to fail, and thus is more fault tolerant than a cluster of 4, which allows only 1 faulty peer.

Sequence sizes ranged from 10 to 100 operations were executed. The operations were uniformly distributed among the control servers. The results are shown in Figure 12. As can be observed, the execution time with a single control/metadata server is higher than the other 2 cases (3 and 5 servers). The average insertion time of 10 key pairs is approximately 225 ms when there is only one metadata server, 91 ms for 3 metadata servers and 67 ms for 5. The average insertion time of 100 key pairs is approximately 358 ms when there is only one metadata server, 183 ms for 3 metadata servers and 137 ms for 5. These results show that by adding 2 control/metadata servers, the throughput of the system more than doubles compared to a single server. Similarly, with 4 additional control/metadata servers, the throughput triples compared to a single server. We can conclude that for systems with heavy loads of read operations, adding control/metadata servers is an effective approach towards scalability.

## 6. CONCLUSION

This article presents ALOCS, a distributed key-value repository with data location control. ALOCS is based on a hierarchical storage model and stores metadata in order to manage data locality. It allows the application to control data allocation on a particular server in the distributed repository. The implementation of the proposed solution uses the distributed file system Ceph for data storage and Zookeeper for metadata management.

Current distributed storage solutions have little or no control over the allocation of information on servers. Some implement uniform distribution mechanisms or distribute data based on the lexicographical order of keys. With the model proposed in this article, it is possible to optimize repositories distributed globally, bringing data closer to user applications that use them more frequently. The goal is to minimize the number of inter-server communication and thus the incidence of distributed transactions. Our experiments show that the overhead for maintaining the metadata is low. Thus, ALOCS is a viable solution to be used as the back-end storage for DBMSs in order to cluster data that are frequently used together.

As future work, we intend to conduct experiments on a large-scale environment to assess the impact of a large number of storage and metadata servers on the performance of update and read operations. In addition to replication, the optimization of the search structure adopted by the metadata module implementation with Zookeeper will also be treated. The main idea is to implement a mechanism to keep the search structure balanced. Finally, we intend to add and analyze a cache refresh policy, and implement the storage and metadata modules using different data management systems.

## REFERENCES

Arnaut, D. E., Schroeder, R., and Hara, C. S. Phoenix: A relational storage component for the cloud. In *IEEE Cloud*. Washington, USA, pp. 684–691, 2011.

Bungama, P. A., de Oliveira, W. M., Sousa, F. R. S., and Hara, C. S. Um repositório chave-valor com controle de localidade. In *Anais do XXXI Simpósio Brasileiro de Banco de Dados (SBBD)*. Salvador, Brazil, pp. 88–99, 2016.

Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. Pnuts: Yahoo!'s hosted data serving platform. *Proc. VLDB Endow.* 1 (2): 1277–1288, 2008.

Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J. J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., Hsieh, W., Kanthak, S., Kogan, E., Li, H., Lloyd, A., Melnik, S., Mwaura, D., Nagle, D., Quinlan, S., Rao, R., Rolig, L., Saito, Y., Szymaniak, M., Taylor, C., Wang, R., and Woodford, D. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems (TOCS)* 31 (3): 8–22, 2013.

de S. Rodrigues, C. A., de Almeida, J. F., Braganholo, V., and Mattoso, M. Consulta a bases XML distribuídas em P2P. In *SBBD - Sessão de Demos*. Fortaleza, Brazil, pp. 21–26, 2009.

Gantz, J. and Reinsel, D. The digital universe in 2020: Big data, bigger digital shadows, and biggest growth in the far east. *IDC iView: IDC Analyze the future* 2007 (2012): 1–16, 2012.

Ghemawat, S., Gobioff, H., and Leung, S.-T. The google file system. In *Association for Computing Machinery (ACM) SIGOPS operating systems review.* Vol. 37. ACM, pp. 29–43, 2003.

Junqueira, F. P. and Reed, B. C. The life and times of a zookeeper. In *Proceedings of the 28th Association for Computing Machinery (ACM) symposium on Principles of distributed computing.* Calgary, Canada, pp. 4–4, 2009.

Maeda, R. M. and Hara, C. S. PVFS-Store: um repositório chave-valor com garantia de localidade. In *Anais do Workshop de Teses e Dissertações em Banco de Dados (WTDBD).* Curitiba, Brazil, pp. 66 – 73, 2015.

Ousterhout, J. K., Da Costa, H., Harrison, D., Kunze, J. A., Kupfer, M., and Thompson, J. G. *A trace-driven analysis of the UNIX 4.2 BSD file system.* Vol. 19, 1985.

Paiva, J. and Rodrigues, L. On Data Placement in Distributed Systems. *ACM SIGOPS Operating Systems Review* 49 (1): 126–130, 2015.

Paiva, J., Ruivo, P., Romano, P., and Rodrigues, L. AutoPlacer: scalable self-tuning data placement in distributed key-value stores. *Association for Computing Machinery (ACM) Transactions on Autonomous and Adaptive Systems* 9 (4): 19:1–19:30, 2014.

Pal, A. and Pal, M. Interval tree and its applications. *Advanced Modeling and Optimization* 11 (3): 211–224, 2009.

Rani, L. S., Sudhakar, K., and Kumar, S. V. Distributed file systems: A survey. *International Journal of Computer Science & Information Technologies* 5 (3), 2014.

Ribas, E. A., Uba, R., Reinaldo, A. P., de Campos Jr, A., Arnaut, D., and Hara, C. Layering a DBMS on a DHT-based storage engine. *Journal of Information and Data Management* 2 (1): 59–66, 2011.

Ross, R. B., Thakur, R., et al. PVFS: a parallel file system for linux clusters. In *Proceedings of the 4th annual Linux showcase and conference.* Houston, USA, pp. 391–430, 2000.

Schroeder, R. and Hara, C. S. Partitioning templates for RDF. In *Proc. of the 19th East-European Conference on Advances in Databases and Information Systems (ADBIS).* Poitiers, France, 2015.

Schütt, T., Schintke, F., and Reinefeld, A. Scalaris: reliable transactional P2P key/value store. In *Proceedings of the 7th Association for Computing Machinery (ACM) SIGPLAN workshop on ERLANG.* Victoria, Canada, pp. 41–48, 2008.

Shvachko, K. V. HDFS scalability: The limits to growth. *Login* 35 (2): 6–16, 2010.

Skeirik, S., Bobba, R. B., and Meseguer, J. Formal analysis of fault-tolerant group key management using ZooKeeper. In *IEEE CCGrid.* Delft, Netherlands, pp. 636–641, 2013.

Tudorica, B. G. and Bucur, C. A comparison between several nosql databases with comments and notes. In *Proceedings of the 10th Roedunet International Conference (RoEduNet).* Iasi, Romania, pp. 1–5, 2011.

Weil, S. A., Brandt, S. A., Miller, E. L., Long, D. D., and Maltzahn, C. Ceph: A scalable, high-performance distributed file system. In *Proceedings of the 7th symposium on Operating systems design and implementation.* Seattle, USA, pp. 307–320, 2006.

Weil, S. A., Brandt, S. A., Miller, E. L., and Maltzahn, C. Crush: Controlled, scalable, decentralized placement of replicated data. In *Proceedings of the 2006 Association for Computing Machinery (ACM)/IEEE conference on Supercomputing.* Tampa, USA, pp. 122, 2006.

Yin, S. and Kaynak, O. Big data for modern industry: Challenges and trends [point of view]. *Proceedings of the IEEE* 103 (2): 143–146, 2015.