Elastic Consistent Hashing for Distributed Storage Systems

Wei Xie and Yong Chen
Department of Computer Science, Texas Tech University, Lubbock, TX 79413

***Abstract*—Elastic distributed storage systems have been increasingly studied in recent years because power consumption has become a major problem in data centers. Much progress has been made in improving the agility of resizing small- and large-scale distributed storage systems. However, most of these studies focus on metadata based distributed storage systems. On the other hand, emerging consistent hashing based distributed storage systems are considered to allow better scalability and are highly attractive. We identify challenges in achieving elasticity in consistent hashing based distributed storage. These challenges cannot be easily solved by techniques used in current studies. In this paper, we propose an *elastic consistent hashing* based distributed storage to solve two problems. First, in order to allow a distributed storage to resize quickly, we modify the data placement algorithm using a *primary server design* and achieve an *equal-work* data layout. Second, we propose a *selective data re-integration* technique to reduce the performance impact when resizing a cluster. Our experimental and trace analysis results confirm that our proposed elastic consistent hashing works effectively and allows significantly better elasticity.**

I. Introduction

The growing power consumption is a critical issue for data centers. It adds substantial cost to an organization and not to mention the carbon footprint. *Elastic* distributed storage systems have gained increasing attention in recent years in order to alleviate this problem [1]. Such storage systems attempt to achieve *power-proportionality* by making power consumption proportional to the dynamic system load leveraging the periods with light load.

Most existing studies [2]–[5] consider resizing the storage cluster as the fundamental technique to achieve power proportionality. When the system load decreases, some of the storage servers in the cluster are powered off. When the load increases, these servers are powered back on. To achieve better elasticity by resizing the storage cluster, several techniques have been introduced in recent studies. For instance, a *primary server* design is introduced in Sierra [2] and has been used in several other studies too [3], [5]. It ensures that a full data copy is maintained in a small subset of servers called primary servers. With primary server design, no clean-up work is needed when powering down several storage servers. To achieve better performance, the data layout needs to be modified [3] for performance proportionality along with power proportionality. In addition, since the small number of primary servers limits the write performance, several recent studies [3], [5] propose to dynamically change the number of primary servers to balance the write performance and elasticity.

Existing efforts have demonstrated the importance and effectiveness of provisioning elasticity in distributed storage systems. However, these efforts and methodologies focus on conventional distributed storage systems with metadata servers, for example the Hadoop Distributed File System (HDFS) [6]. Consistent hashing based distributed storage systems are emerging in data centers recently due to significantly better scalability potential and easy manageability [7]–[9]. Compared to distributed storage systems with metadata servers, they do not have a metadata service to manage the mapping from data to storage servers. Instead, they use a deterministic hash function to determine the placement of data items. However, these different characteristics of consistent hashing based distributed storage make it challenging to achieve elasticity and power-proportionality.

In this study, we introduce a series of techniques for enabling the power-proportionality for consistent hashing based distributed storage systems. The design of the elastic consistent hashing is inspired by existing techniques that are proven effective in prior studies. We introduce *primary server data placement* and *equal-work* data layout to make it possible to scale down to very few active servers with small data movement. We propose a selective data re-integration technique to alleviate the performance degradation when a cluster resizes. The proposed techniques are implemented on Sheepdog storage system and evaluated with a storage cluster testbed. We also conduct analysis based on real-world traces and compare our proposed techniques against the original system. We find that the proposed elastic consistent hashing techniques are able to achieve both better elasticity and performance, and saves significantly more machine hours (which means power consumption). The contribution of this study includes:

- To the best of our knowledge, this is the first study to systematically investigate the elasticity in consistent hashing based storage systems.
- We introduce an elastic consistent hashing that not only adapts existing elasticity techniques to consistent hashing but also introduces new selective data re-integration techniques to alleviate the performance impact of data migration.
- We demonstrate the effectiveness of the elastic consistent hashing techniques based on a Sheepdog storage system. Our solutions significantly outperform the original consistent hashing based Sheepdog system.
- We analyze real-world traces and confirm that elastic consistent hashing is able to substantially reduce the power consumption in many cases.

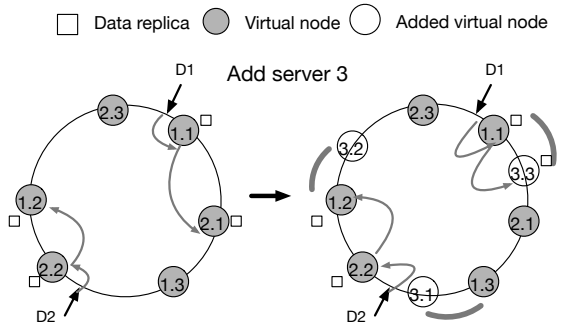

Figure 1: The consistent hashing algorithm

II. Background

A. Consistent Hashing

Consistent hashing was first introduced to manage distributed web cache on the Internet [10]. Since then, it has been widely used as a data placement algorithm for distributed storage due to the fact that it does not depend on metadata servers for managing the mapping between data items (data item is interchangeable to data object or data block in other studies) and storage servers, which otherwise can largely limit the scalability of a distributed storage system. Another benefit of consistent hashing is that it is able to automatically adapt to node additions and removals without completely reorganizing data layout. This makes fail-over handling easy and the storage cluster expansion with minimal intervention. For example, several large-scale distributed storage systems like GlusterFS [11] and Sheepdog [12] use consistent hashing as the data placement algorithm. Consistent hashing uses a *hash ring*, a hypothetical data structure that contains a list of hash values that wraps around at both ends. The ID number of the buckets/nodes (i.e. the storage servers) are hashed and sorted to form the hash ring. Given a key (i.e. the data item), it hashes the key to a position on the hash ring, and continues walking along the ring in a clockwise direction from that position till the first bucket/node (also called a successor of the hashed key) is found, and returns the associated bucket/node for storing/retrieving the data item. In fact, it usually selects multiple buckets/nodes along the ring because redundant copies are usually used in distributed storage systems. In addition, to improve the balance of the distribution, there are usually multiple/many virtual nodes generated for a bucket/node to be placed on the ring. If a virtual node is encountered, the physical storage server associated with that virtual node is selected to place the data item.

We use an example to explain how consistent hashing works (see Figure 1). We assume a cluster with two storage servers (such a small number is used for easier illustration), server number 1 and 2. Each server is associated with three virtual nodes. For example, server 1 has three virtual nodes 1.1, 1.2, and 1.3, where the number before the dot represents the physical server number and the one after the dot represents the index of its virtual nodes. To illustrate how consistent hashing finds the buckets/nodes for a key, a key $D1$ is considered and we place two copies (for redundancy and availability) on the ring. To locate the first bucket/node to place $D1$, $D1$ is hashed to a position on the hash ring, and then walking clockwise from that position, the next virtual node (a successor) on the ring is selected for data placement or retrieval, i.e. the virtual node 1.1 in this example. By continuing to walk along the hash ring, the second replica is placed on virtual node 2.1, or physical server 2. These steps are shown in the left part of Figure 1.

A great advantage of consistent hashing is that the data layout does not change significantly when the number of buckets/nodes (storage servers) changes. When buckets/nodes are added or removed, the corresponding virtual nodes are added or removed from the hash ring. The changes to the virtual nodes on the ring cause a different data placement for the key if the successor changes, which means a new virtual node is located by walking clockwise. However, the number of keys affected is usually small, which makes the adaption to the node membership changes relatively inexpensive. For example, we consider adding a server 3 to the cluster as shown in Figure 1 (right part). Three virtual nodes 3.1, 3.2, and 3.3 are generated and placed on the hash ring. According to consistent hashing, the key $D1$ will find two successor virtual nodes 1.1 and 3.3, instead of 1.1 and 2.1 before. This results that the first copy of $D1$ stays at server 1 but the second copy should be placed on server 3 instead (if $D1$ already exists in the storage cluster, i.e. on server 2, before server 3 added, then $D1$ should be migrated to server 3 when server 3 added). Notice that a large portion of keys and data items will stay at their original locations after server 3 is added. As shown in Figure 1 (right part), the arc outside the hash ring shows the region that a data copy hashed in this range will be placed onto the newly added server. Other than these regions, data copies should still stay at their original locations.

B. Elastic Storage Systems

The elasticity in distributed storage in the context of this study refers to the ability to resize the scale of the storage cluster in order to save power consumption and optimize resource utilization. With elasticity, not only power consumption of a distributed storage would be proportional to the nodes being active, the performance should also be proportionally scaled with the number of active nodes. With a small subset of nodes being active, a distributed storage system could stay in a low-power mode for a period of light workloads, which could be a few hours or even days. Once the workload demand increases, it can be sized-up accordingly to satisfy the new performance demand. With fast resizing, a cluster does not need to over-provision resources all the time to prepare for sudden bursts of load; instead, it could resize the number of servers in the cluster to match how much performance is actually needed from the current load.

The agility of resizing is very important in an elastic distributed storage. With fast resizing, smaller variations in the system load can be utilized to reduce machine hours, thus to decrease power consumption. An elastic storage should also have minor impact on performance. When resizing a cluster, the IO footprint should be as small as possible.

C. Motivation of This Research

We have discussed that consistent hashing is able to adapt to node addition or removal with small data migration. This

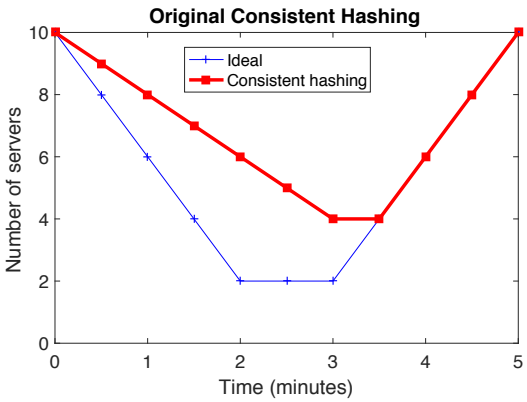

Figure 2: Resizing a consistent hashing based distributed storage system

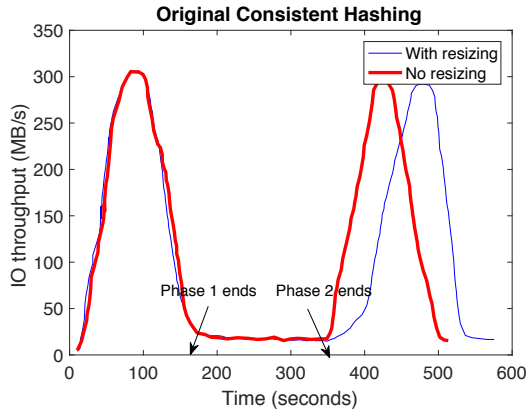

Figure 3: Performance impact of resizing

makes it naturally elastic, but we argue that the current consistent hashing based distributed storage does not achieve adequate elasticity and power-proportionality.

First of all, data objects are distributed across a cluster of servers randomly and uniformly in consistent hashing. Even though with data redundancy, removing more than two servers from the hash ring (powering them off to reduce the power consumption when light load observed) at the same time would still make much of data unavailable. When one server leaves the hash ring, lost data copies have to be re-replicated on the rest servers. Additionally, before the re-replication finishes, the consistent hashing based distributed storage is not able to tolerate another server's departure. In an elastic distributed storage, the agility of resizing a cluster is critical. For instance, sometimes it could require resizing a storage cluster from 1,000 active servers straight to 10. In such cases, the latency of re-replication would make it very difficult to size down the cluster so quickly.

We present our observation with an experiment tested on a 10-server cluster running the consistent hashing based Sheepdog system [12] (the detailed configuration is discussed in Section V-A). The resizing capability of the cluster was tested. In our test, the cluster started with 10 active servers and we attempted to remove/extract 2 servers every 30 seconds (even though Sheepdog is not able to do this) for the first 2 minutes, and then at the 3rd minute, we attempted to add 2 servers back every 30 seconds till all 10 servers are active. In fact, we were not able to make the desired resizing. We had to remove/extract one server at a time and to allow Sheepdog to finish its re-replication operation before removing/extracting the next server. In Figure 2, we can see that Sheepdog lags behind when sizing down the cluster comparing to the desired resizing pattern (shown as "ideal" in the figure), but catches up when sizing up the cluster. We attribute this behavior to the previously stated problem that the data layout in consistent hashing based distributed storage prohibits fast resizing. In the proposed elastic consistent hashing, we use *primary server* and *equal-work data layout* techniques to solve this problem (see Section III-B and III-C).

The second problem with elasticity in consistent hashing based distributed storage is the data migration that occurs when adding servers to a cluster. When servers are added, certain data objects are due to be moved to the added servers, which is part of the adaptability feature of consistent hashing (see Section II-A). This data migration does not affect the ability of resizing significantly because data migration is not a pre-requisite operation for adding servers. However, an elastic distributed storage may resize frequently and introduce considerable data migration that consumes IO bandwidth and degrades users' perceived performance. In order to illustrate the problem, we conducted a test that used a 3-phase benchmark (further introduced in Section V-A). As seen from Figure 3, the first and third phase are IO intensive and the middle phase is much less IO intensive. In the "no resizing" case, no servers were turned off all the time, but the "resizing" case shut down 4 servers when the first phase ended and added them back when the second phase ended. As shown in this figure, we can observe that the system's IO throughput is significantly affected when we added 4 servers back to the cluster (between phase 2 and 3). The throughput degradation was caused by the data migration, which consumed substantial IO bandwidth. There are two ways the data migration overhead can be alleviated. First, consistent hashing assumes that the added servers are empty so that it migrates all the data that are supposed to place on the added servers. In fact, in an elastic distributed storage, data on the servers that are turned down still exist. When they are turned back on, it does not need to migrate these data back because they are still there, unless they are modified in the period when these servers are shut down. Second, the rate of migration operation is not controlled and it substantially reduces the improvement of system's performance that sizing-up a cluster should deliver. To solve these problems, we design a selective data migration (we use data re-integration and data migration interchangeably) policy that reduces the amount of data to migrate and limit the migration rate (see Section III-E).

III. Elastic Consistent Hashing Based Distributed Storage

A. General Design Architecture

The general design of elastic consistent hashing based distributed storage is inspired by those existing studies such as SpringFS. It adopts a *primary server* concept by redesigning the original consistent hashing: storage servers are divided

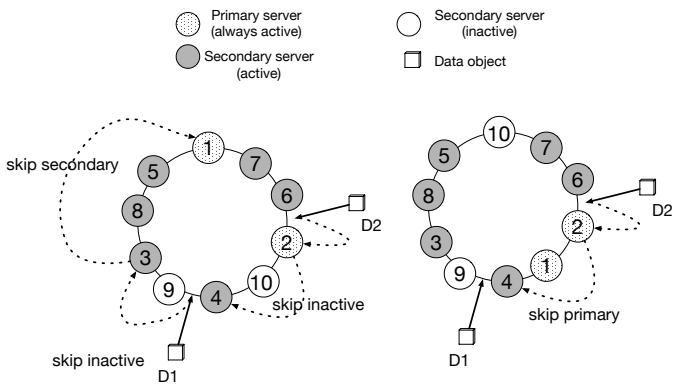

Figure 4: The *primary server* data placement in consistent hashing based distributed storage. Server 1 and 2 are primaries. Two servers (server 9 and 10) are inactive secondary servers. Inactive servers are skipped in the placement. On the left-side figure, $D1$ is placed on server 3 and server 1, and $D2$ is placed on server 2 and 4. On the right-side figure, the location of server 10 and 1 on the ring is switched, which results a different data placement.

into two groups, one group called *primaries* and the other group called *secondaries*. When placing data onto servers, the primaries always contain exactly one copy of the data, and the secondaries contain the rest of replicas. This primary server design ensures that the data will be available (at least one copy) even with only these primary servers being active. The design requires modifying the original data placement algorithm in consistent hashing. We will further discuss the details of the data placement in elastic consistent hashing in Section III-B.

In elastic consistent hashing algorithm, data layout is a critical factor that affects the elasticity. Comparing to the original consistent hashing algorithm, elastic consistent hashing algorithm is based on an equal-work data layout that allows power proportionality and read performance proportionality at the same time. We will discuss more details about this data layout in Section III-C.

In addition, we try to alleviate the performance degradation problem that is observed when sizing up a cluster. By introducing a *selective data re-integration* (data re-integration means the *data migration* when servers are re-integrated to a cluster), an elastic consistent hashing based distribtuted storage is able to achieve better performance in a dynamically resizing environment. We further discuss the design details of *selective data re-integration* in Section III-E.

B. Primary Server Data Placement

Before discussing the primary server data placement, we introduce server ranks in consistent hashing based distributed storage. Comparing to the completely symmetric design in consistent hashing where every server has an equal role in the storage cluster, the elastic consistent hashing ranks the servers in the cluster based on the order that servers would be powered off or on, which is called an expansion-chain in existing literatures such as [3]. With such ranks, the order that servers are turned down or on is fixed. With the ranks of

Algorithm 1 Primary server data placement algorithm.

INPUT: Data object ID (Data_ID), number of servers (n), number of primary servers (p), number of replicas (r);

1: /*First replica*/
2: $server(1)$=$next\_server(hash(Data\_ID))$
3: **for** i from 2 to $r-1$ **do**
4: **if** $\exists j = 1$ to $i-1$ that $isprimary(server(j)) = TRUE$ **then**
5: $server(i)$=$next\_secondary(hash(server(i-1)))$
6: **else**
7: $server(i)$=$next\_server(hash(server(i-1)))$
8: **end if**
9: **end for**
10: /*For last replica*/
11: **if** $\exists j = 1$ to $r-1$ that $isprimary(server(j)) = TRUE$ **then**
12: $server(i)$=$next\_secondary(hash(server(i-1)))$
13: **else**
14: $server(i)$=$next\_primary(hash(server(i-1)))$
15: **end if**

OUTPUT: Servers selected for placing replicas of Data_ID

servers defined, we illustrate the primary server data placement as follows.

For a storage cluster with n servers, p servers are selected as *primary servers* and the rest of servers are *secondary servers*. Primary servers rank from 1 to p. Secondary servers are assigned a rank number starting $p+1$ to n. With primary servers, the data placement becomes different from the original consistent hashing. For placing a data object D onto a n-server cluster with $r-way$ replication, the goal is to place exactly one copy on a primary server and the rest copies on secondary servers.

In order to place exactly one copy of each data item on primary servers, the elastic consistent hashing uses the following way to conduct data placement. First, for the first $r-1$ replicas of a data item, they are placed according to line 2 to line 9 in Algorithm 1. To place the ith replica, it first checks if any of the 1st to $(i-1)$th replicas is already placed on a primary server. If yes, the ith replica should be placed on a secondary server; otherwise it should be placed following the original consistent hashing, i.e. on the next server along the hash ring. For the last replica, however, it needs to ensure that the requirement for placing one copy on a primary server is met. It checks if any of the previous $r-1$ replicas is already assigned to a primary server. If yes, the last replica must be placed on one of secondaries; otherwise there would be two copies on primaries. If not, the last replica is placed on one of primaries (see line 11 to 15 in Algorithm 1).

Notice that there are two cases when a server is skipped in primary server data placement. The first case is skipping a primary server when this data item already has a replica on a primary, and the second case is skipping a secondary server, which only occurs when placing the last replica.

Figure 4 shows how data placement takes place in a sample cluster with 10 servers (in which 2 servers are primaries and 2 servers are inactive secondaries). Note that it shows a simplified hash ring with no virtual nodes. In fact, there should be multiple "virtual primary nodes" or "virtual secondary nodes" for each server. The first copy of the data object $D1$ in this figure is placed on server 3, which is the next server encountered on the hash ring. The second copy is placed on server 1, which is the first primary server next to server 3.

Please note that, in the original consistent hashing algorithm there is no primary server concept, and the second replica of $D1$ would have been placed on server 8. In contrast, for data object $D2$, since the first replica is already placed on a primary server (server 2), the second copy is placed on a secondary server (server 4).

There is a special case that all secondary servers become inactive (lowest power state) or the number of active secondary servers is less than $r-1$ (r is the number of data replicas). In such case, we treat primary servers as secondary servers temporarily to ensure that the replication level can be still maintained, as long as there are enough active servers (including primary ones) to accommodate r replicas (which is in fact mostly true because there are usually more primary servers than the number of replicas).

C. Equal-Work Data Layout

The primary server design makes sure that all data are still available when resizing to p servers. With a uniform data distribution, the value of p equals to $\frac{n}{r}$. For example, for a 10-server cluster with 2-way replication, half of data should be placed on $p = 5$ primary servers, and the rest is placed on the other 5 secondary servers. This significantly limits the minimum power consumption the elastic consistent hashing is able to accomplish.

To achieve lower minimal power usage and performance proportionality, it is necessary to adopt a different data layout ([3]) across the storage cluster so that the IO balancer is able to achieve optimal proportionality. A promising solution is the equal-work data layout that was first utilized in Rabbit [3] for laying out data in HDFS to achieve elasticity. Inspired by Rabbit, in elastic consistent hashing based distributed storage, we achieve similar data layout by assigning an appropriate number of virtual nodes for each server.

We define an equal-work data layout for a storage cluster with n servers. First, $p = \lceil \frac{n}{e^2} \rceil$ servers are selected as primaries (e is Euler's number; $e \approx 2.7183$). For each primary server, the weight (i.e. the number of virtual nodes) is assigned according to Equation 1, where B is an integer that is large enough for data distribution fairness [13]. For a secondary server ranked i, its weight is defined by Equation 2.

$$v\_primary = \frac{B}{p} \tag{1}$$

$$v\_secondary_i = \frac{B}{i} \tag{2}$$

This elegant weight design achieves the same equal-work data layout as in Rabbit [3] and SpringFS [5].

For example, in a 10-server, 2 primary, and 2-way replication storage cluster depicted in Figure 4 (no virtual nodes in this figure for simplicity, but in fact, there should be multiple virtual nodes for each server on the hash ring), the number of virtual nodes assigned to each server can be calculated with Equation 1 and 2. Each of these two primary servers is assigned with $1000/2 = 500$ virtual nodes ($B = 1000$ in this case). The other servers' virtual node number is $1000/i$, where i is the rank of the server. For example, server 6 is assigned with $1000/6 = 167$ virtual nodes. Note that, in real situations, a much larger B will be chosen for better load balance.

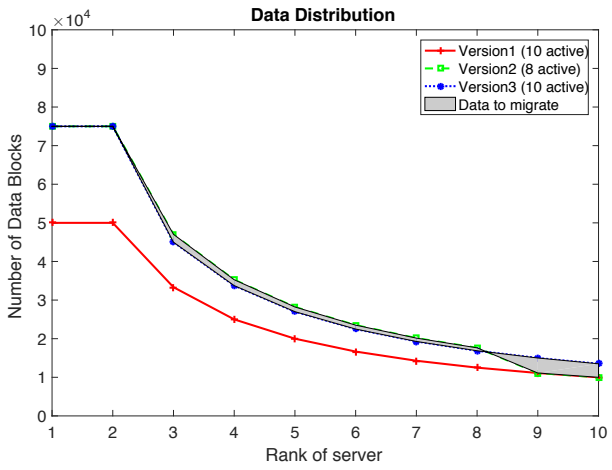

Figure 5: The Equal-Work Data Layout and Data Re-Integration Between Versions

Figure 5 illustrates a visual view of the equal-work data layout (see the red solid line). We do not repeat the mathematical derivation to prove that this equal-work layout achieves the optimal performance proportionality, which has already been done in literature [3]. But note that higher ranked servers always store more data comparing to lower ranked servers.

D. Node Capacity Configuration

The primary server data placement and equal-work data layout deviate from the original consistent hashing's data layout. One of the major differences is that our proposed schemes place largely different amount of data on each storage server, while the original consistent hashing distributes data evenly. The original consistent hashing uses uniform distribution (via assigning an equal number of virtual nodes) so that the utilization of storage is maximized (assuming each storage server has the equal capacity). It means that our proposed schemes may under-utilize the capacity of some servers (and over-utilize some other servers) due to the uneven data distribution.

To mitigate this problem, we provide a simple solution that involves a modified cluster configuration. In this configuration, we set the capacity of each storage server proportional to the assigned virtual nodes according to the equal-work data layout. This would make optimal utilization of the capacity of each server. However, in a large storage cluster with many servers, the number of different capacity configurations needed by the equal-work data layout may be huge, which is difficult to achieve. In our design, we use only a few different capacity configurations (for example 2TB, 1.5TB, 1TB, 750GB, 500GB, and 320GB) and each configuration is assigned to a group of neighboring-ranked servers. This provides an approximate solution to the data distribution imbalance problem.

E. Selective Data Re-Integration

With primary server data placement and equal-work data layout, the elastic consistent hashing is able to achieve a

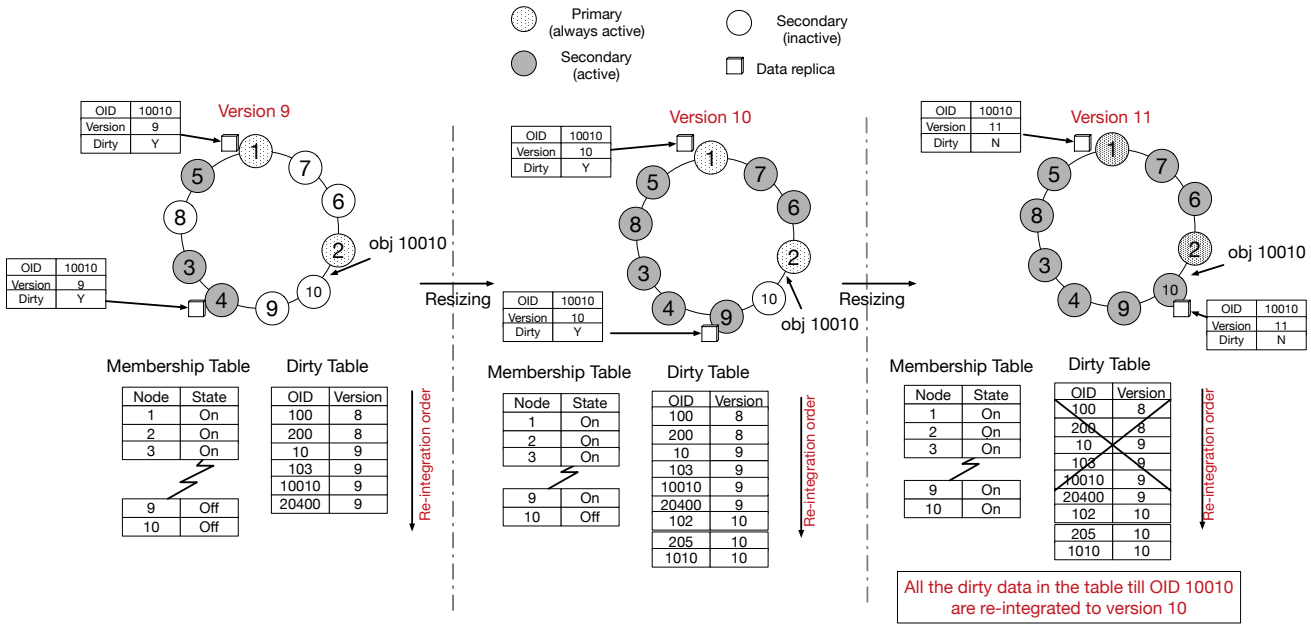

Figure 6: Dirty data (offloaded data) tracking and selective data re-integration

low minimal power and performance state while maintaining good power/performance proportionality at the resizing granularity of one server. However, the performance degradation associated with resizing can not be solved by these designs. In elastic consistent hashing based distributed storage, *write availability offloading* is used to improve elasticity but it causes data migration overhead when resizing an elastic cluster. *Write availability offloading* occurs when not all servers are active and it offloads data writes to an inactive server to the next eligible server so that the replication level can still be maintained. When data are offloaded this way, the offloaded data replicas need to be migrated to the desired servers when those servers are active again. This operation is called *re-integration* because it migrates data to re-integrated servers and recovers the equal-work data layout. For example, in Figure 5, the cluster undergoes three versions (version is defined later, but we can consider it as a state of the cluster for now) that it sizes down from 10 active servers to 8 active servers, and sizes up back to 10 active servers again. We assume that 50,000 data objects are written in version 2, which somehow distorts the curve of data layout because the last two servers are inactive. In version 3, the data layout should recover and the amount of data shown in shaded area should be re-integrated.

The experiment in Section II-C shows that consistent hashing based storage is not able to identify offloaded data but instead over-migrates all the data based on changed data layout. It is also shown that data re-integration consumes considerable IO bandwidth for an extended period and deteriorates the performance scaling significantly. These two problems attribute to the performance degradation when resizing a cluster. To alleviate the impact of data re-integration, we introduce techniques to track offloaded data and limit the rate of data migration.

1) Cluster Membership Versioning: To enable locating data in a cluster that resizes frequently, it is necessary to maintain versions of the cluster being in different resizing states. Most of consistent hashing based distributed storage systems, as far as we know, includes membership version as an essential component. For example, Ceph, GluterFS, and Sheepdog, use version (sometimes called *epoch* in these systems) to ensure the data consistency when node membership changes. We define a data structure *membership table* to keep the state of each server in the cluster in a certain version, where a server in the cluster can be either on or off. Each version is associated with a *membership table* structure to describe the state of each server in that version. With versions of a cluster maintained, it is able to identify where data replicas are written in a historical version, no matter how many versions have passed. Given a data object ID, as long as the last version it is written is known, it is able to accurately find the servers that contain the latest replicas of the data.

2) Tracking Dirty Data: We also introduce a *dirty table* to facilitate selective data re-integration, in which only "dirty data" are re-integrated. "Dirty data" refers to the data written in a cluster in a certain version that is not in full-power state (all servers are active). When placing replicas of these data, some of the replicas may be offloaded from inactive servers to other active servers. When the cluster is in a new version that contains more active servers, these "dirty" data might need to be re-integrated to those servers that they are offloaded from. A data object in elastic consistent hashing is considered dirty until it has been re-integrated to a full-power version, in which replicas will not skip any inactive server during placement.

Each entry of the *dirty table* contains the *data object ID* (OID) that is a universal identifier of a data object, and a *version number* that specifies the version this data object is lastly written. In addition, a version number and a dirty bit are also added to the header of data objects (actually the Sheepdog distributed storage we used for evaluation already includes a version in the header). These two entries allow the data re-

Algorithm 2 Selective data re-integration.

INPUT: Current version (Curr_Ver), number of servers (n), number of replicas (r), running status either RUNNING or PAUSED (state);

```
1:  while isempty_dirty_table()=FALSE and state=RUNNING do
2:      if Curr_Ver > Last_Ver then
3:          restart_dirty_entry()
4:      end if
5:      (OID, Ver)=fetch_dirty_entry()
6:      if num_ser(Curr_Ver) > num_ser(Ver) then
7:          to_ser[1...r]=locate_ser(OID, Curr_Ver)
8:          from_ser[1...r]=locate_ser(OID, Ver)
9:          migrate(from_ser, to_ser)
10:         Last_Ver = Curr_Ver
11:         if num_ser(Curr_Ver)==n then
12:             remove_dirty_entry(OID)
13:         end if
14:     end if
15: end while
```

integration component to identify the newest version of data so that when a data object is written in multiple versions, the data re-integration component is able to identify the latest data version and avoids stale data. Whenever a dirty data is written, the corresponding OID and version are inserted into the dirty table. The dirty table is managed as a FIFO list, which means that the oldest entries in the table are fetched before newer ones. The dirty table is maintained in a distributed key-value store across the storage servers to balance the storage usage and the lookup load. A logging component tracks data object written in the distributed storage and inserts key-value pairs according to the data object ID and current cluster version.

3) Selective Data Re-integration: In this subsection, we describe how to perform selective data re-integration utilizing the versioning and dirty table data structures described above. The selective data re-integration process is described in Algorithm 2. First, it checks if the current version differs from the version that last data re-integration takes place. If yes, it means that the cluster is in a new version that the data re-integration should restart from the first entry in the dirty table, forgeting the current progress (line 2 in Algorithm 2). Then it fetches an entry from *dirty table*, starting from older versions to newer version (line 3 in Algorithm 2). Please note that the function $fetch\_dirty\_entry$ always fetches data in the dirty table in a certain order (version ascending and OID ascending if the version is the same). The fetched OID and version are used to locate the data to be re-integrated. Note that re-integration only occurs when the current version has more active servers comparing to the version of the data (see line 4 in Algorithm 2). The version number can be used to find the corresponding *membership table* to determine the cluster membership at that certain version. According to the cluster membership, the location of existing data replicas can be calculated and the servers that they should be migrated to can be inferred as well (see line 5 to 6 in Algorithm 2). Once the location of the candidate data object is determined, the *data re-integration component* is able to migrate it to the desired servers in accordance with the current version (line 7 in Algorithm 2). After a data object is re-integrated, it further checks if the current version is a full-power version (all servers are active), and if yes, it removes all the entries in the dirty table (line 8 in Algorithm 2).

With the selective data re-integration algorithm described above, only the offloaded data are migrated when servers are re-integrated. It reduces the amount of data migration needed in such situations.

Figure 6 illustrates an example of a cluster in three versions. In each version, we show the membership table and dirty table in that state. In version 9, only server 1 to 5 are active, which means that all the data written in this version are dirty. We assume that 4 data objects, 10, 103, 10010, and 20400 are written in this version. The dirty data entries are inserted in the dirty table along with entries inserted at version 8. Taking data object 10010 for example, it is placed on server 4 and 1 according to the placement algorithm defined in Section III-B. When the cluster is resized to version 10 by turning on 4 servers, a new version with more active servers appears so that data re-integration starts. It follows the predefined order to migrate data. For example, one replica of object 10010 is migrated from server 4 to 9 in this case. It is noted because version 10 is not a full-power version, the entries in the dirty table are not removed after re-integration is finished. In version 11, however, all the servers are active so that all the data recorded in the dirty table should be re-integrated to this version. In the state shown in the figure, all the dirty data until 10010 have been re-integrated and the corresponding entries are removed from the table. One copy of object 10010 is also migrated from server 9 to server 10.

IV. Implementation

The elastic consistent hashing design and techniques are evaluated based on Sheepdog distributed storage system. Sheepdog is an object-based storage [14], and each data object is distributed to a cluster of storage servers via the consistent hashing algorithm. When the cluster membership changes (servers may leave and join a Sheepdog cluster), Sheepdog automatically recovers data layout based on the consistent hashing algorithm. The recovery feature of Sheepdog is mainly utilized for tolerating failures or expanding the cluster size, but not for elasticity. In our implementation, the servers in the cluster never leave the cluster when they are turned down, but are in the inactive state. The core data placement algorithm in Sheepdog is modified to execute the primary server data placement described in Section III-B (Source code [1], and the weight of each server is set based on the equal-work data layout described in Section III-C.

For tracking dirty data, we use Redis [15], an in-memory key-value store, for managing the dirty table. The dirty table is managed using the LIST data type in Redis. Each dirty data entry is inserted using RPUSH command. During re-integration, if an entry in the dirty table is used for migrating its corresponding data replicas but is not removed from the table afterward (when current version is not a full-power version), a LRANGE command is used to fetch the $(OID, version)$ pair from the store. Otherwise, if the entry should be removed after migration, a LPOP command is used instead to remove the $(OID, version)$ pair from the store. The selective data re-integration component uses the algorithm described in Section III-E.

[1] http://discl.cs.ttu.edu/gitlab/xiewei/ElasticConsistentHashing

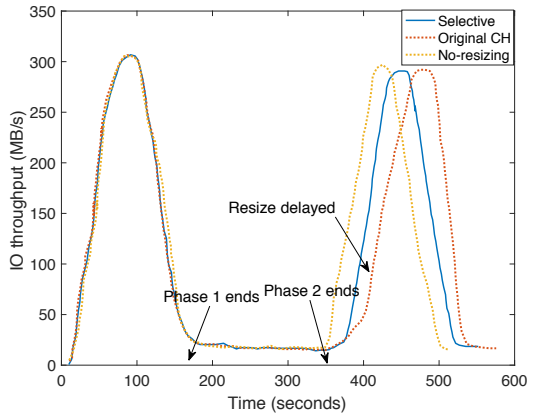

Figure 7: Evaluating the performance of resizing with 3-phase workload.

V. Evaluation and Analysis

A. A Cluster Testbed Evaluation with 3-Phase Workload

We have set up a cluster testbed based on our modified Sheepdog system and used a 3-phase workload to test how the elastic consistent hashing performs when resizing the cluster size and compares it against the case without resizing. The modified Sheepdog system was deployed on a 10-node storage cluster. Each of these 10 servers is equipped with 2 Intel(R) Xeon(R) CPU E5-2450, 32GB RAM, 500GB hard disk drive, and connected via the 10 Gigabit Ethernet. It is noted that we do not use different capacity configuration for different nodes due to the limited hardware availability. Because we do not test our system to the extreme capacity, such configuration does not cause the problem of node being full. Modified sheepdog processes ran on these 10 servers, providing an aggregated block-level store. It used a 2-way data replication, and data object size is 4MB. A 100GB space was partitioned from this store and assigned to a KVM-QEMU virtual machine as a virtual disk. The KVM-QEMU virtual machine ran on a separate client server with the same configuration as storage servers, configured with 2 virtual CPUs and 16GB of virtual RAM.

We generated synthetic benchmarks using Filebench to mimic the 3-phase benchmark used in a previous work [5]. In the first phase, we used Filebench to sequentially write 2GB of data to 7 files, with a total of 14GB data. The second phase benchmark, however, is much less IO intensive. We used rate attribute in Filebench to limit the IO event issued per second so that the I/O throughput in the middle phase was 20MB/s. In this phase, there were totally 4.2GB data read and 8.4GB data written. The last phase is similar to the first phase, except that the write ratio was 20%. The configurations were set to resemble the 3-phase workload used in SpringFS [5]. The first phase of the workload ran in full-power mode with all 10 servers turned on. At the end of the first phase, 4 servers were turned down until the end of the second phase. All the servers were back on in the third phase benchmark.

In our test, we evaluated three cases (see Figure 7): a system without resizing ("no resizing" in the figure), original consistent hashing ("original CH" in the figure), and consistent hashing with selective data re-integration ("selective" in the figure). Note that primary server and data layout are not considered here because they do not have an effect on the performance. From Figure 7, we can see that the I/O throughput in selective data re-integration is substantially faster comparing to the original consistent hashing algorithm when phase 2 workload ends. It indicates that our selective re-integration technique successfully reduces the data that need to be migrated and thus allow more I/O throughput for the testing workload.

The evaluation result illustrates the performance degradation due to data re-integration we have discussed in the paper. Even though there is little difference in the peak IO throughput in the three cases, the "delayed" IO throughput increase has a big impact on the elasticity of the system. When IO throughput is needed (look at where phase 2 ends in the figure), the data re-integration work makes the original consistent hashing based store struggle to deliver the performance. In a real elastic system, it would need to power on much more servers to satisfy the IO throughput requirement at this period. With improvement on the "delayed" IO throughput, it requires powering on less servers to make up the IO bandwidth consumed by data re-integration. This is how our evaluation result translates to saving machine hours.

B. Policy Analysis with Real-world Traces

The evaluation in Section V-A tests the performance variation when the distributed storage is resized. It is a small-scale test due to the limited resources available and evaluated using synthetic benchmark. In this Section, we analyze real-world traces from large-scale distributed storage and demonstrate the benefit of the elasticity behavior of the elastic consistent hashing and the selective data re-integration. We used two Cloudera traces (there are totally 5 of these traces but we do not have enough page space to show all of them) from Cloudera's enterprise customers' Hadoop deployments running various e-commerce and telecommunication applications, which is initially discussed in [16]. The characteristics of these two traces are described in Table I. These traces contain the I/O load on the storage cluster over a long period of time (several days to a month). The ideal number of servers for each time period is proportional to the data size processed. However, according to our observation, real system is not able to achieve such ideal scaling. Scaling down in the original consistent hashing store may require *d*elay time for migrating data. Scaling up in both original and our modified consistent hashing store may also require processing *e*xtra IOs for data reintegration, which increases the number of servers needed. We calculate the delay time and extra IOs according to the trace data and deduce the number of servers needed for three cases: "Original CH", "Primary+full", and 'Primary+selective". "Original CH" is the original consistent hashing with uniform data layout that no primary server design nor selective re-integration is used, in which much clean up work must be done before resizing. "Primary+full" uses the elastic consistent hashing with primary server and equal-work data layout, but does not turn on the selective data re-integration component. "Primary+selective" also uses the elastic consistent hashing and with selective data re-integration component turned on.

In both of these two figures (Figure 8 and Figure 9),

Table I: The specification of the real-world traces

Trace	Machines	Length	Bytes processed
CC-a	<100	1 month	69TB
CC-b	300	9 days	473TB

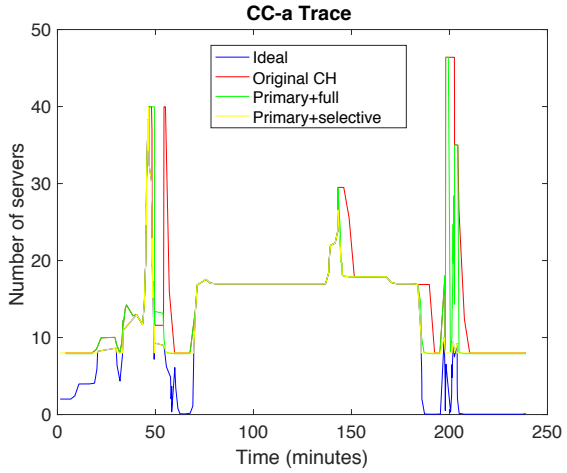

Figure 8: CC-a Trace. "Ideal" case is fully based on the profiling of the IO throughput in the trace. The original consistent hashing ("original CH") needs clean-up work before resizing thus exhibit delays in resizing. Primary server placement "primary" does not require clean-up work when resizing and introduce small IO performance impact. It resizes near ideally except that it is not able to size down further until there are only primary servers.

it is clear to see that "primary+selective" outperforms "primary+full" and "original CH" significantly, especially when sizing down the cluster quickly. "Primary+selective" performs very close to the "ideal" case except that it is not able to work below the minimum number of servers, which is defined by the equal-work data layout.

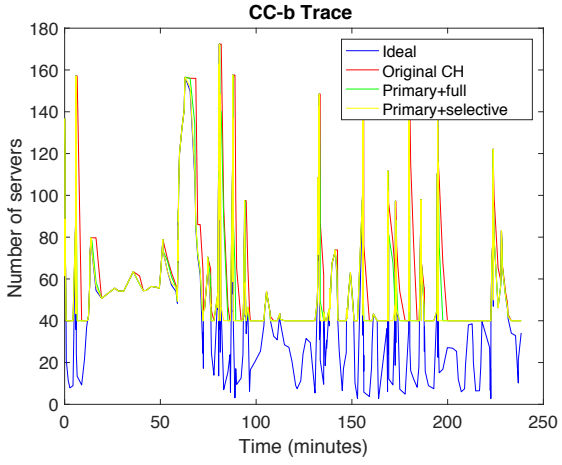

Figure 9: CC-b Trace

Table II: Relative machine hour usage relative to the ideal case

Trace	Original CH	Primary+full	Primary+selective
CC-a	1.32	1.24	1.21
CC-b	1.51	1.37	1.33

In addition, "primary+full" still provides significant improvement comparing to the "original CH", which means that the primary server design and the equal-work data layout attribute to a portion of the better elasticity. For CC-a traces, we find that "primary+full" saves 6.3% machines hours and "primary+selective" saves 8.5% machines hours comparing to the "original CH" case. Comparing to the "Ideal" case, "primary+selective" only has 1.21 times more machine hour usage. For CC-b traces,"primary+full" saves 9.3% machines hours and "primary+selective" saves 12.1% machines hours comparing to the "original CH" case. Comparing to the "Ideal" case, "primary+selective" only has 1.33 times more machine hour usage. (see Table II). These results indicate that our primary server data placement achieves good resizing agility no matter whether the data re-integration technique is used or not. The selective data re-integration also manages to save a little machine hour. We believe that this is because in some situations the IO load from full data re-integration could prevent the cluster from sizing down for some period. However, this only occurs at extreme situations where the cluster resizes abruptly and data re-integration work has no time to catch up. In addition, we can see that CC-a trace has significantly higher resizing frequency. It explains why our techniques are able to achieve more percentage of improvement comparing to the original consistent hashing.

VI. Related Work

Some recent studies [2]–[5] tried to address the elasticity problem by providing new data layout and some other techniques in distributed storage systems. The most notable technique of achieving elasticity is called "primary servers" used in Rabbit [3], Sierra [2], and SpringFS [5]. *Rabbit* proposes an *equal-work* data layout to achieve a much smaller system size (around 10% of cluster size) and optimal performance proportionality. These studies inspired us to accomplish a similar design in consistent hashing based distributed storage to achieve better elasticity.

Even though these studies provide systematic solutions for elastic distributed storage, the techniques discussed in them are not able to be adapted in consistent hashing based storage trivially. In this research, we not only adapt some of their techniques into consistent hashing based storage and also propose new techniques to enable selective data re-integration policy to improve performance and elasticity.

A recent study GreenCHT [17] targets similar problem as we do. In GreenCHT, servers are partitioned into different tiers that each tier of servers can be turned down together to save power. GreenCHT is also integrated with a power mode scheduler to dynamically adjust power mode of the system. Comparing to GreenCHT, our elastic consistent hashing is able to achieve finer granularity of resizing with one server as the smallest resizing unit. In addition, we have systematically discussed how primary server, equal-work data layout, and

selective data re-integration work in consistent hashing based storage that GreenCHT lacks.

There are also numerous studies focusing on the determining how to resize resource provisioning in cloud infrastructure. AutoScale proposes a dynamic capacity management approach to maintain the right amount of spare capacity to handle bursts in requests. Lim et al. [1] designed an elasticity-aware resource management controller to resize a cluster according to demands with short delays. Elastisizer [18] tries to compute an optimal cluster size for a specific MapReduce job with job profiling and modeling. The SCADS director framework [19] uses a model-predictive control (MPC) framework to make cluster sizing decisions based on the current workload state, current data layout, and predicted SLO violation. AGILE [20] predicts medium-term resource demand to add servers ahead of time in order to avoid the latency of resizing. Our elastic consistent hashing based distributed storage solves the problem of how to resize a distributed storage quickly, without affecting the performance much, but does not discuss the problem of how to make resizing decision based on workload demands. These previous studies can be integrated with elastic consistent hashing to provide a more attractive solution.

Other than achieving elasticity, it is also critical to improve the performance of distributed data store. Our previous work proposed distributed hybrid storage for consistent hashing based data store [13], [21]. These work attempt to achieve a unified distributed data store to combine distributed SSD and HDD devices [22] and provide balance between performance and storage utilization. Liu et al. [23] uses a result reusing technique to reduce data movement in scientific analytic applications. These studies are orthogonal to our elastic storage research to improve the distributed data store.

VII. Concluding Remarks

In this study, we carefully examine the elasticity in consistent hashing based storage systems and find that even though consistent hashing is designed to be easily adaptable to server addition and removal, it is not elastic. To enable an elastic consistent hashing based storage, we propose three techniques including primary server, equal-work data layout, and selective data re-integration. We present how primary server and equal-work data layout are designed and how they avoid clean-up work when sizing down a cluster and achieves low minimum power consumption. A selective data re-integration technique is introduced to reduce data migration work and alleviate the performance degradation problem when resizing a cluster. Our proof-of-concept implementation and testing have confirmed the effectiveness of the elastic consistent hashing in terms of the ability of agile resizing and low performance impact.

Metadata based distributed storage systems have encountered scalability bottleneck in recent years due to the demand for very large-scale storage systems. Consistent hashing based distributed storage systems are considered an option to address this problem. Meantime, the elasticity has become increasingly critical for a large-scale storage system. Our study attempts to lay the foundation of allowing better elasticity in this type of distributed storage system. We will continue to work on these issues and try to solve other problems in this area, such as a resizing policy based on workload profiling and prediction.

As a future work, we consider the overhead of managing dirty data table in the key-value store, which introduces memory footprint and latency to manage such data structure. We have not carefully evaluated the overhead yet but we believe the performance of state-of-the-art key-value store is able to make the overhead minor.

Acknowledgment

This research is supported in part by the National Science Foundation under grant CNS-1162488, CNS-1338078, and CCF-1409946.

References

[1] H. C. Lim, S. Babu, and J. S. Chase, "Automated Control for Elastic Storage," ser. ICAC '10, 2010.

[2] E. Thereska, A. Donnelly, and D. Narayanan, "Sierra: Practical Power-proportionality for Data Center Storage," ser. EuroSys '11, 2011.

[3] H. Amur, J. Cipar, V. Gupta, G. R. Ganger, M. A. Kozuch, and K. Schwan, ser. SoCC '10, 2010.

[4] D. Narayanan, A. Donnelly, E. Thereska, S. Elnikety, and A. Rowstron, "Everest: Scaling Down Peak Loads Through I/O Off-loading," ser. OSDI'08, 2008.

[5] L. Xu, J. Cipar, E. Krevat, A. Tumanov, N. Gupta, M. A. Kozuch, and G. R. Ganger, "SpringFS: bridging agility and performance in elastic distributed storage," in *Proceedings of the 12th USENIX Conference on File and Storage Technologies (FAST 14)*, 2014.

[6] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The Hadoop Distributed File System," in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, ser. MSST '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 1–10.

[7] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area Cooperative Storage with CFS," ser. SOSP '01, 2001.

[8] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A Scalable, High-performance Distributed File System," in *Proceedings of the 7th Symposium on Operating Systems Design and Implementation*, ser. OSDI '06. Berkeley, CA, USA: USENIX Association, 2006, pp. 307–320.

[9] A. Lakshman and P. Malik, "Cassandra: A Decentralized Structured Storage System," *SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 35–40, Apr. 2010.

[10] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent Hashing and Random Trees: Distributed Caching Protocols for Relieving Hot Spots on the World Wide Web," in *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, ser. STOC '97, 1997.

[11] A. Davies and A. Orsaria, "Scale out with GlusterFS," *Linux J.*, 2013.

[12] "Sheepdog project," https://sheepdog.github.io/sheepdog/, accessed: 2015-07-08.

[13] W. Xie, J. Zhou, M. Reyes, J. Noble, and Y. Chen, "Two-mode data distribution scheme for heterogeneous storage in data centers," in *Big Data (Big Data), 2015 IEEE International Conference on*. IEEE, 2015, pp. 327–332.

[14] M. Mesnier, G. R. Ganger, and E. Riedel, "Object-based storage," *IEEE Communications Magazine*, 2003.

[15] J. L. Carlson, *Redis in Action*. Manning Publications Co., 2013.

[16] Y. Chen, S. Alspaugh, and R. Katz, "Interactive analytical processing in big data systems: A cross-industry study of mapreduce workloads," *Proceedings of the VLDB Endowment*, 2012.

[17] N. Zhao, J. Wan, J. Wang, and C. Xie, "GreenCHT: A power-proportional replication scheme for consistent hashing based key value storage systems," in *2015 31st Symposium on Mass Storage Systems and Technologies (MSST)*, 2015.

[18] H. Herodotou, F. Dong, and S. Babu, "No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics," in *Proceedings of the 2Nd ACM Symposium on Cloud Computing*, ser. SOCC '11, 2011.

[19] B. Trushkowsky, P. Bodík, A. Fox, M. J. Franklin, M. I. Jordan, and D. A. Patterson, "The SCADS Director: Scaling a Distributed Storage System Under Stringent Performance Requirements," in *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, ser. FAST'11.

[20] H. Nguyen, Z. Shen, X. Gu, S. Subbiah, and J. Wilkes, "Agile: Elastic distributed resource scaling for infrastructure-as-a-service," in *ICAC 13*, 2013.

[21] J. Zhou, W. Xie, J. Noble, K. Echo, and Y. Chen, "SUORA: A Scalable and Uniform Data Distribution Algorithm for Heterogeneous Storage Systems," in *Proceedings of the 11th IEEE International Conference on Networking, Architecture, and Storage*, 2016.

[22] W. Xie, Y. Chen, and P. C. Roth, "ASA-FTL: An Adaptive Separation Aware Flash Translation Layer for Solid State Drives," *Parallel Computing*, 2016.

[23] J. Liu, S. Byna, and Y. Chen, "Segmented Analysis for Reducing Data Movement." in *In the Proc. of the IEEE International Conference on Big Data,(Bigdata'13)*, 2013.