Measuring the Effect of Node Joining and Leaving in a Hash-based Distributed Storage System

Nam DANG† and Haruo YOKOTA†

† Department of Computer Science, Graduate School of Information Science and Technology
Tokyo Institute of Technology 2–12–1 Oookayama, Meguro–ku, Tokyo 152–8552, Japan

Abstract Distributed storage systems have become the core of many large-scale applications today. Much research has focused on the distribution of data in such systems, with two main approaches of tree-based and hash-based algorithms. Hash-based algorithms tend to provide a good distribution based on the randomness of the hash function while maintaining a good performance if correctly implemented. However, the approach also incurs performance cost when a node joins or leaves the system, especially in the context of high-performance datacenter clusters. This paper investigates such costs with a simulation model and discusses the effect of a node joining and leaving for a variant of hash-based algorithms for datacenter-oriented storage clusters.

Key words hash-based index, distributed storage systems, distributed index, scalable hashing

1. Introduction

Distributed storage systems are a current research focus in computer systems and data engineering today. Such systems provide many attractive features such as scalability, availability, and fault tolerance by harnessing collective capacity of multiple storage nodes. The design of distributed storage systems also comes with a number of difficulties due to the distributed nature; oftentimes a good design is a compromise of scalability, availability, and fault tolerance based on the requirements of the system.

Most storage systems write data to underutilized device. The problem with this approach is that once written, data is rarely, if ever, moved. Even a perfect distribution can become imbalanced because the storage is expanded. Moreover, it is difficult to equally distribute the load across the systems since data is distributed to node gradually over time, which often yields unexpected hotspots.

A robust solution for the above issues is to randomly distribute data among available storage devices. The result is a probabilistically balanced distribution and uniformly mixes of old and new data. In this approach, all device will have similar load, enabling the system to perform well under various workloads [1]. Furthermore, random distribution across a large number of devices offer a high degree of parallelism and better aggregate bandwidth, thus exploiting the distributed nature of the system. However, simple hash-based methods are unable to cope with changes in the number of participating nodes, often incurring a massive reshuffling of data. Another challenge is to place data's replicas on uncorrelated devices to avoid loss of two replicas in case of failure on correlated devices.

This paper discusses the effect of node leaving and joining in hash-based distributed storage systems. We also present an approach to measure node joining and leaving costs in a hash-based distributed storage system designed for datacenter networks. The algorithm in this approach is CRUSH (Controlled Replication Under Scalable Hashing) [2], the core distribution algorithm employed by Ceph [3], targeted for its proven effectiveness in both theoretical and industrial practice. CRUSH shares many similar characteristics with other hash-based distribution algorithms including high parallelism, fast look-up, and randomized distribution of data. The algorithm implements a deterministic hashing function that maps an input value typically an object identifier to a list of devices on which to store object replicas. Only a compact, hierarchical description of the devices comprising the storage cluster and the replica placement policy is required to determine the location of an object's replica; therefore, the approach provides a high level of parallelism by eliminating any shared structure or central communication point and near-random data distribution.

The structure of the paper is organized as follows. The second section presents several notable data distribution approaches. We review the design CRUSH in details in section 3. Section 4 introduces our simulation model for evaluating the costs of node joining and leaving. Section 5 presents the results of our model from simulating a large cluster. Finally,

conclusion and future work are discussed in section 6.

2. Related Work

We present several related works of data distribution for distributed storage systems. Each of them offers a different paradigm and address different requirements in today's storage systems.

2.1 Hadoop Distributed File System (HDFS)

HDFS [4] is a popular distributed file system designed based on Google File System [5] for MapReduce [6]. With one single central namenode server (metadata server), HDFS utilizes several techniques to guarantee a robust and highly scalable storage platform. Files are divided into small blocks of 64MB and the blocks are distributed to data nodes under the supervision of the namenode. Since the namenode handles every operation except data read/write, it also becomes the central point of failure of the system. There has been some research [7] [8] to address this weakness with proposals including mirrored namenode (or secondary namenode) or using BigTable-like architecture [9].

2.2 Fat-Btree

Fat-Btree [10] is a variant of parallel B-Tree structure widely used in many distributed Database Management Systems (DBMS's). In a Fat-BTree, each Processing Element (PE) has a subtree of the whole B-tree consisting of the root node and all intermediate nodes between the root and the leaf nodes allocated to that PE. As a result, the lower level index nodes with high update frequency have fewer copies compared to upper level index nodes with lower update frequency. This technique reduces synchronization costs among PEs when the Btree structure is modified by data operations or node additions and removals. Nevertheless, there is still significant performance cost due to the strict synchronization of the shared nodes among the PEs, especially when the number of nodes increases.

2.3 Chord

Chord [11] is a protocol for a peer-to-peer distributed hash table that uses a variant of consistent hashing to assign keys to nodes. In Chord, each node requires routing information about $O(\log n)$ nodes, and performs lookups via $O(\log n)$ messages to other nodes. Chord provides good support as node joins or leaves the system, with each event resulting in $O(\log^2 n)$ messages. The algorithm also scales well with the number of nodes, and is able to recover from large numbers of simultaneous node failures and joins, and provide correct answers for lookups even during recovery. One big drawback of Chord in the context of distributed storage systems is its message-based exchange nature, which makes lookups in a datacenter-oriented file system rather expensive compared to HDFS's centralized namenode or Fat-Btree. In fact, Chord is mainly designed for and mostly used only in peer-to-peer distributed systems.

3. The CRUSH Algorithm

CRUSH is the basic distribution algorithm of Ceph, an object-based distributed storage system. It shares many common characteristics with other hash-based distribution approaches such as high parallelism, fast look-up and balanced data distribution. At the same time, similar to other hash-based algorithms, CRUSH also faces with the difficulty of node joining and leaving. The inherent costs of mitigating such changes in the cluster are translated into computational costs and exchanged messages. This paper investigates CRUSH in terms of node joining and leaving costs as an example of hash-based storage systems, and discusses measurements for such costs.

The fundamental of CRUSH is a per-device weight-based distribution of data object that approximates a uniform probabilistic distribution. CRUSH utilizes a hierarchical *cluster map* that represents the hierarchy of the cluster and the weight of individual devices to control the distribution of data. In addition, the distribution is also based on *policy*, which is defined in terms of *placement rules* that specify how many replica targets are selected from the cluster and what restrictions are imposed on replica selection. Given a single integer input value x, CRUSH will output an ordered list $\overrightarrow{R}$ of n distinct storage targets. Relying upon a strong multi-input integer hash function with x as a parameter, the algorithm provides a deterministic and independently calculable mapping scheme using cluster map, placement rules, and x.

3.1 Hierarchical Cluster Map

CRUSH organizes the cluster topology according to *devices* and *buckets*, both of which have numerical identifiers and weight values associated with. A *bucket* may contain any number of devices or other buckets; they can form interior nodes in a hierarchical manner with the storage devices at the leaves. Bucket weights are defined as the sum of the weight of the items they contain. By combining buckets into a hierarchy, the structure as in a data-center cluster can be flexibly represented.

There are four different types of buckets, each with a different selection algorithm to address different data movement patterns resulting from the addition or removal of devices and overall computation complexity.

3.2 Replica Placement

CRUSH is designed to address the placement of data with regards to replication in a heterogeneous cluster. The placement of replicas on storage devices in the hierarchy also has a significant effect on data safety. By projecting the struc-

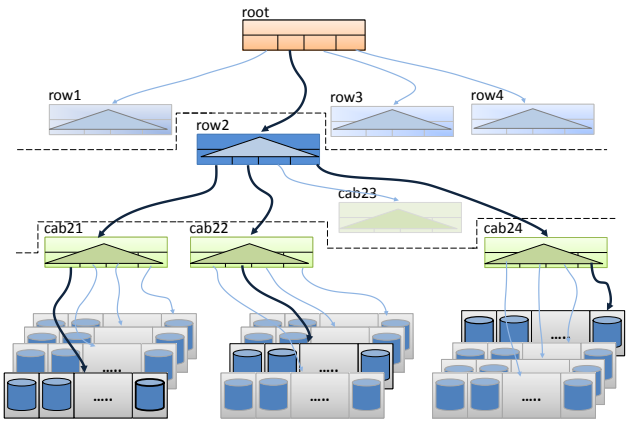

Figure 1 A partial view of a four-level cluster map hierarachy consisting of rows, cabinets, and shelves of disks.

ture of the system into the cluster map, CRUSH can separate storage devices into *failure domains* according to the physical location of the devices. For example, devices on the same shelf belong to the same failure domain since they are more likely to fail together. Although placing replicas on devices in distant locations ensures better data safety, communication costs also increases due to lower bandwidth between nodes across long distance than nodes in vicinity.

CRUSH accommodates a wide range of scenarios in datacenter storage clusters with different data replication strategies and underlying hardware configurations with *placement rules* that allow the system administrator to specify how objects are place with fine granularity. Figure 1 shows a system of four level of hierarchy, the algorithm starts from the root that has pointers to four rows. The placement policy then requires the selection of one row, and within the row it chooses three cabinets. For each cabinet, the placement rule selects a disk to store a replica. In this example, three disks are selected in three separate cabinet of the same row to store three corresponding replicas. This approach is useful to spread replicas across separated failure domains (separate cabinets), but constrain them within some physical boundary (the same row).

3.3 Bucket Types

CRUSH is designed to reconcile two competing goals: efficiency and scalability of the mapping algorithm, and minimal data migration to maintain a balanced distribution of data. In order to achieve these goals, CRUSH utilizes four different types of buckets to represent internal (non-leaf) nodes in the cluster hierarchy: *uniform buckets*, *list buckets*, *tree buckets*, and *straw buckets*. Each bucket type is based on a different data structure and utilizes a different hashing function to pseudo-randomly select nested item during the replica placement process. The performance of each bucket is a trade-off between speed and efficiency of minimizing data movement in case of node addition or remove. A summary of the difference among the four bucket types is presented in Table 1.

Action	Uniform	List	Tree	Straw
Speed	$O(1)$	$O(n)$	$O(\log n)$	$O(n)$
Additions	poor	optimal	good	optimal
Removals	poor	poor	good	optimal

Table 1 Summary of mapping speed and data reorganization efficiency of different bucket types when items are added or removed from a bucket

In this study, we focus on only Tree Bucket and Straw Bucket for their performance and high potential for practical applications. The other two, Uniform and List Bucket, lack adequate performance for a cluster with frequent node additions and removals and are mainly for theoretical discussion.

3.4 Implementation of CRUSH

The implementation of CRUSH is presented in details in [12] as part of RADOS. In addition to storage nodes, the design also includes monitors which provide information about the current state of the cluster and manage node joining, leaving, and failures. Clients access one of the replicated monitors first to retrieve a cluster map before accessing any objects; later updates in cluster map can be retrieved from any storage nodes. Within the storage cluster, changes in cluster map are propagated rather than directly transferred from the monitor to all the nodes. This approach enables a small number of replicated monitors to support a large number of storage nodes in datacenter clusters.

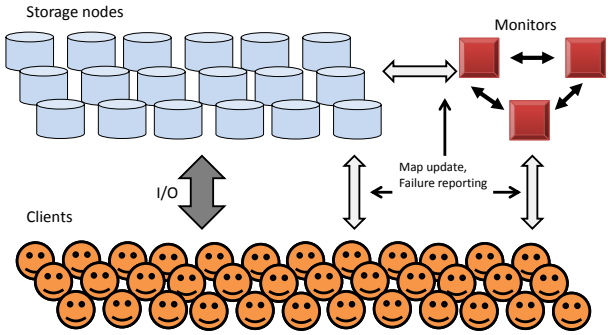

Figure 2 A cluster of thousands of storage nodes. A small number of tightly coupled, replicated monitors collectively manage the cluster map that defines cluster memeberships and placement rules. Each client performns direct IO operations on data objects with storage nodes.

4. Node Joining and Leaving Costs

A critical element of data distribution in a large storage system is the response to the addition or removal of storage nodes. Hash-based algorithms typically exploit the random nature of hash functions to achieve a uniform distribution of data and workload to avoid asymmetries in data placement. When an individual device fails, the algorithm needs to redistribute the data to maintain such balance. The total data to be remapped is minimized to as much as w_{failed}/W, where W is the total weight of all devices. However, when the cluster hierarchy is modified, as in adding or removing a storage device, the situation is much more complex. The mapping process with hashing often struggle to minimize data movement and can result in additional data movement beyond the theoretical optimum $\frac{\triangle w}{W}$. For example, in CRUSH changes in the cluster map also leads to propagation of the map to all storage nodes, which takes a $O(\log n)$ time for a cluster of n nodes [12]. In worst case scenario, the implementation of CRUSH in RADOS [12] offers a performance of less than 20% of duplicated massages. Clients that access the data in a CRUSH cluster also needs to update their local copy of the cluster map in order to correctly locate data objects in this case.

Another performance cost incurred when a node joins or leaves the cluster hierarchy is from re-calculating the location of every data object. As discussed in 3.3, the randomness of data distribution in has-based algorithms relies on the quality of hashing functions, which can be expensive when there are a large number of objects. Given that in a large-scale storage clusters with thousands of nodes, node removals and additions occur frequently, re-calculation of objects' position each time a change occurs is rather costly and can have significant impact on the overall performance. Some algorithms can avoid calculating the position of every data object, while some other must perform a full check to ensure the correctness of data placement.

This paper discusses the costs of node joining and leaving with respect to two aspects: number of messages exchanged and number of re-computations. We then present our approach to model the costs and evaluate the effect of node joining and leaving with simulation of Tree Bucket and Straw Bucket.

4.1 Exchanged Messages

The number of messages exchanged can be classified into two categories: intra-cluster and client-cluster. Intra-cluster messages are exchanges that occur among the storage devices within the cluster; client-cluster messages are exchanges between clients and the storage nodes in case of map change. Note that clients also require an up-to-date map to correctly locate date in storage nodes.

In a cluster with n storage nodes, it is obvious to see that the minimum number of messages required to update the cluster map on every node is also n. In case of CRUSH, as discussed in [12] the hard upper bound on the number of copies of a single update a node might receive is proportional to μ, the number of objects on the storage nodes. The reason behind this value is that each node has to communicate with the nodes that store the replicas of the object. Simulation results in [12] indicate that the actual number of duplicated messages is at most as much as 20% of μ. The number of intra-cluster messages is therefore bounded and does not vary significantly across different systems with the same number of nodes and objects.

On the other hand, the client also requires the information of an up-to-date cluster map to access data objects. Given there are often more clients than storage nodes, the number of messages between clients and the cluster imposes significant costs on the system. In this paper, we focus on the number of messages exchanged between clients and storage nodes as the indicator for the costs of exchanged messages in the system. In the event that a client with an out-of-date map tries to access the cluster, the client will have to update its map from the incorrectly accessed node, and access the data again to a correct storage node. The number of messages incurred from such events is 2 for an access, which is the number used in our simulation.

4.2 Re-computation Costs

Hash-based distribution algorithms rely on the assumption that the hash function provides a good degree of randomness to achieve a balanced distribution. An object's location is determined by feeding this multi-input hash function a number of parameters, including the object identifier and the cluster map. This approach enables the clients and storage nodes to independently calculate the placement of objects without referring to a central management. At the same time, when the cluster map changes due to node joining or leaving, it is imperative to update the local copy of the map. Given that hashing function can take up 45% of a hash-based algorithm's computation time [2], in a system with a large number of objects, node joining and leaving can lead to considerable performance penalty in terms of delay of access.

4.3 Simulator

The simulator first builds a hierarchy map with a specified number of devices. This cluster map is used as a physical cluster for the simulation to evaluate data placement. The simulator then generates a large number of objects, each with a unique integer identifier. The objects are then assigned to the storage devices based on the CRUSH distribution algorithm. A large number of clients randomly requesting objects

from the storage clusters are then created. Here, we assume that the clients access objects in a random manner, as object identifiers are typically randomly assigned in an actual system. During the access, the simulator then subsequently changes the hierarchy map by removing or adding a number of devices randomly. Since we assume that intra-cluster message exchanged is necessary to maintain the uniform state throughout the cluster, the simulator only tracks the number of messages introduced by the change, including map updates and failed client requests. The simulator also keeps track of the number of re-calculation performed due to the change in topology. The results are recorded for different cluster sizes, number of clients, and number of objects. The setup of the environment for the implementation is shown in Table 2.

CPU	Intel Xeon E5620 (×2)
CPU Speed	2.40 GHz
Cores	4 (×2) (Hyper-Threading)
Memory	24 GB
OS	Ubuntu 11.10
Programming Language	C/C++

Table 2 Environment Setup for the Simulator

The simulator measures the number of re-calculations required in the event of node joining and leaving with different topology sizes and configurations to evaluate the costs. There are two aspects of this cost that are measured in this report. The first one is computation time, which depends on both the number of computations and the speed of the distribution algorithm. The second one only concerns with the number of computations that are executed to re-evaluate the existing distribution.

In order to simulate exchanged messages, each client is assumed to randomly access a random object from a pool of object identifier. Although this random access pattern may not correlate to one of a real system, it is adequately generic to demonstrate the effect of node joining and leaving. Also, since data objects are distributed randomly, this pattern in the simulation is not too far off from actual access pattern. If the client tries to access a data object with a non-up-to date map, the storage node will sends back a new map, and the client may need to re-access the data according to the new information. The simulator then records the number of exchanged map messages in the system when a node is simulated to join or leave, resulting in topology change.

5. Results

5.1 Computation Costs

Computation cost of a distribution algorithm can be categorized in to two categories: number of computations and computation time. We examine the performance of Tree Bucket and Straw Bucket over thirty millions ojbects with different cluster sizes, ranging from 512 nodes to 2048 nodes. The results are shown in Fig. 4 and Fig. 3.

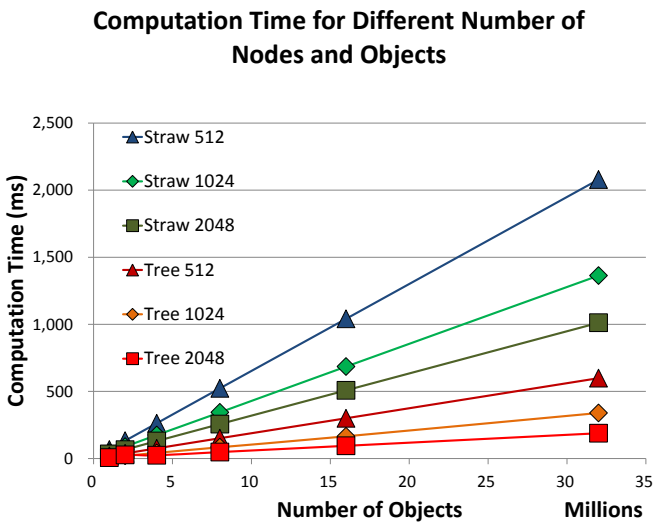

Figure 3 The largest number of computation that a node must perform in case of topology change.

Figure 3 reports the number of computations in different cluster sizes. In this paper, the maximum value of objects that a node in a cluster must re-evalutated in case of topology change is considered as the maximum number of computations per node in the given cluster. Due to the probablistically uniform distribution of both Tree and Straw Bucket algorithms, the maximum number of computations in each case for the same cluster sizes varies litte. Note that with larger clusters, with the same number of objects this cost should decrease since the number of objects per node decreases accordingly.

Fig. 4 discusses the time required for a cluster to re-evalute all the objects. Since objects are distributed in a probablistic uniform manner accross the nodes, the computation time is measured as the time from the start of the computation till the finish of every node in the cluster. However, due to the map propapation process to propagate changes in the cluster, in reality the starting time may vary from systems to systems with different data and cluster configuration. Therefore, to simplify our measure, this paper reports the longest time that some node in the cluster must undertake to re-evalute the distribution of its objects. Such values are representative of the performance of the cluster in terms of computation time. In Fig 4, the difference in terms of performance can be easily observed, with the maximum of computation time going as high as 2 seconds in a Straw-based cluster of size 512 nodes.

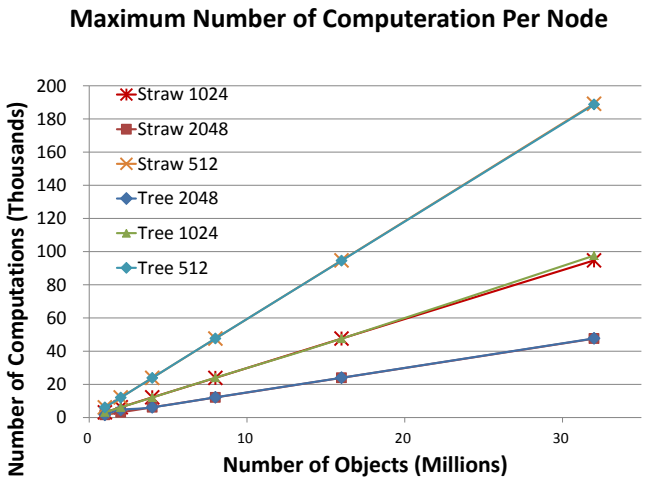

Figure 4 The time required for CRUSH to re-evaluate the positions of every object with different cluster size. Storage nodes are assumed to perform such calculations in parallel; the graph shows the longest time of some given nodes with the largest nubmer of objects.

Both graphs show the effect of cluster sizes and number of objects with regard to node joining and leaving costs. Notice that such costs are without the cost of communication within the cluster itself, which vary depending on the configuration and the amount of data). This computation cost can result in significant delay from the client's side when trying to access a server undergoing distribution evaluation, especially when the amount of data increases at a much faster rate than capacity of storage systems.

5.2 Exchanged Messages

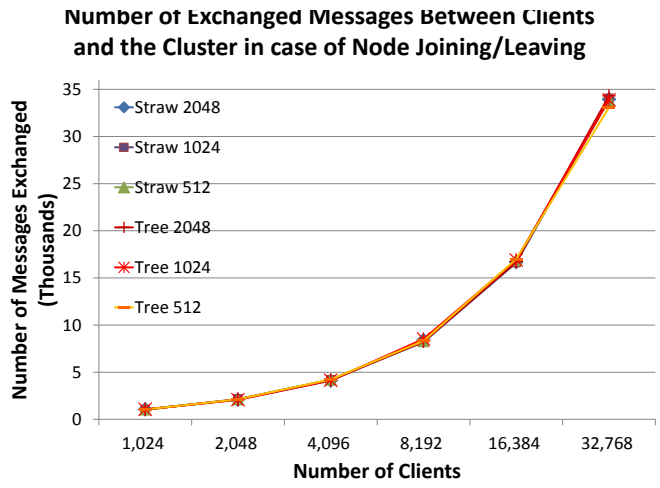

Figure 5 Number of messages exchanged in different topologies with different number of clients. The cost increases linearly with the number of the clients

As previously discussed, our work focuses on the number of messages exchanged between clients and the storage nodes. This cost is shown in Fig. 5 with over thirty thousand clients and different cluster sizes. A message is incurred when a client with an out-of-date cluster map tries to access the storage nodes. Figure 5 shows a near-linear graph of the number of messages exchanged with different cluster sizes. Since the update is done per storage node and client basis, the total number of messages only depend on the number of clients. As the number of clients increases, typically along with the amount of data, this message exchange mechanism may lead to network congestion and performance degration.

6. Conclustion and Future Work

In this paper, we have presented an evaluation of the cost of node joining and leaving in a hash-based distributed storage system, specifically, CRUSH. The result shows a significant increase in computation cost of node addition or removal when the amount of data increases. Moreover, as the amount of data increases, the number of clients tend to increase accordingly, leading to higher costs in terms of messages exchanged in the system. Along with the cost of inter-cluster communication, the two categories of costs discussed in this paper can lead to considerable performance degration in datacenter-oriented storage clusters.

In the future, we plan to apply similar measurements to different hashing algorithms and evaluate the trade-off between performance costs and node joining or leaving. We also plan to propose an approach that can significantly reduce such costs while maintaining the favorable characteristics of hash-based approaches. The approach can be improved by incorporating real-world access pattern to better simulate the effect of exchanging messages in a system. Finally, the measurements described in this paper can be used as the metrics for evaluating the performance of other datacenter-oriented distribution algorithms in many other scenarios.

7. Acknowledgements

This work is partly supported by Grants-in-Aid for Scientific Research from Japan Science and Technology Agency (A) (#22240005)

References

[1] Jose Renato Santos, Richard R. Muntz, and Berthier Ribeiro-Neto, "Comparing random data allocation and data striping in multimedia servers", *ACM SIGMETRICS Performance Evaluation Review*, vol. 28, no. 1, pp. 44–55, June 2000.

[2] Sage A Weil, Scott A Brandt, and Ethan L Miller, "CRUSH : Controlled , Scalable , Decentralized Placement of Replicated Data", in *SC '06 Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, Tampa, Florida, 2006, ACM.

[3] Sage A Weil, Scott A Brandt, Ethan L Miller, and Darrell D E Long, "Ceph : A Scalable , High-Performance Distributed File System", in *Proceedings of the 7th symposium on Operating systems design and implementation (OSDI '06)*, Seattle, Washington, 2006, pp. 307–320, USENIX Association.

[4] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler, "The Hadoop Distributed File System", in *Proceedings of the 2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*. May 2010, pp. 1–10, IEEE Computer Society.

[5] Sanjay Ghemawat, Howard Gobioff, and Shun-tak ST Leung, "The Google file system", *ACM SIGOPS Operating Systems Review*, vol. 37, no. 5, pp. 29–43, Dec. 2003.

[6] Jeffrey Dean and Sanjay Ghemawat, "MapReduce: simplified data processing on large clusters", *Communications of the ACM*, vol. 51, no. 1, pp. 107–113.

[7] Feng Wang, Jie Qiu, Jie Yang, Bo Dong, Xinhui Li, and Ying Li, "Hadoop high availability through metadata replication", in *Proceedings of the first international workshop on Cloud data management (CloudDB '09)*, New York, New York, USA, 2009, pp. 37–44, ACM Press.

[8] Anup Suresh Talwalkar, *HadoopT - Breaking the Scalability Limits of Hadoop*, PhD thesis, Rochester Institute of Technology, 2011.

[9] Fay A Y Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber, "Bigtable : A Distributed Storage System for Structured Data", *ACM Trans. Comput. Syst.*, vol. 26, no. 2, pp. 4:1—-4:26, 2008.

[10] Haruto Yokota, Yasuhiko Kanemasa, and Jun Miyazaki, "Fat-Btree: An Update-Conscious Parallel Directory Structure", in *Proceed of International Conference on Data Engineering, ICDE*, 1999, pp. 448–457.

[11] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications", *SIGCOMM Comput. Commun. Rev.*, vol. 31, no. 4, pp. 149–160, Aug. 2001.

[12] Sage A Weil, Andrew W Leung, Scott A Brandt, and Carlos Maltzahn, "RADOS : A Scalable , Reliable Storage Service for Petabyte-scale Storage Clusters", *PDSW '07 Proceedings of the 2nd international workshop on Petascale data storage*, pp. 35–44, 2007.