Access-Load-Aware Dynamic Data Balancing for Cloud Storage Service

Haopeng Chen, Zhenhua Wang, and Yunmeng Ban

School of Software, Shanghai Jiao Tong University
Shanghai, P.R. China
chen-hp@sjtu.edu.cn, aspiration@foxmail.com,
banyunmeng@sjtu.edu.cn

Abstract. Cloud storage is the typical way for storing massive data in Big Data Era. Dynamic data balancing is important for cloud storage since it aims to improve the utilization of computing resource and the performance of data process. However, storage-load-aware data balancing, adopted by almost all existing cloud storage services and systems, is far less effective than access-load-aware one for typical cloud applications with hotspots of data. This paper focuses on the latter and puts forward a mechanism of dynamic data balancing for optimization of resource utilization. The mechanism detects the overloaded and underloaded physical nodes and virtual nodes by monitoring their utilization of resource. Then, it dynamically balances the access load among the nodes by pair, merge, mark, scale up and scale down operations. This mechanism is useful for the applications with hotspots in data. So it is a complementation of storage-load-aware data balancing. The results of experiments on Swift demonstrated the effectiveness of this mechanism.

Keywords: access-load-aware, cloud storage service, data balancing, resource utilization, VM migration.

1 Introduction

With high scalability and reliability, cloud storage has become the dominant technology for both of enterprises and individuals to store their massive structured and unstructured data. According to the report of IDC, the unstructured data has a growth curve substantially greater than that of structured data and its capacity will be shipped to 80EB of storage[1]. To satisfy the growing demand for cloud storage, many providers have offered their public cloud storage services, including Amazon S3[2], Google Cloud Storage[3], Rackspace Cloud Files[4], and so on. Several open source frameworks are also available for building private cloud storage, such as Openstack Swift[5], Eucalyptus Walrus[6] and Nimbu Cumulus[7]. Certainly, we also can store the massive unstructured data with NoSQL database systems, such as Amazon Dynamo[8], Google BigTable[9] and MongoDB[10].

With the diversity of cloud storage services, consumers have to consider over that what an ideal storage service for massive data should be. Firstly, a cloud storage

M. Pathan, G. Wei, and G. Fortino (Eds.): IDCS 2013, LNCS 8223, pp. 307–320, 2013.

service should be high scalable in order to make the extension of its capacity easy and efficient. As a result, it seems more reasonable to build a distributed storage environment with a cluster of storage resources. For example, Google File System (GFS), a scalable distributed file system for large distributed data-intensive applications[11], has been the infrastructure of many cloud storage services. The high scalability means the extension of capacity can be accomplished by simply adding more resources into the storage cluster.

Secondly, a cloud storage service should be able to balance the workload among its storage nodes in order to improve its access performance. For example, in MongoDB[10], the data are grouped into chunks by key ranges, and the chunks are stored on physical nodes, named as shard servers. By specifying the upper bound of the size of a single chunk and the maximal acceptable difference of the numbers of chunks between any two shard servers, it ensures that all the shard servers hold roughly same amount of data. The other example, Swift[5], also provides a hashing mechanism to distribute the data onto its storage nodes. With such a mechanism and a reasonable key generating algorithm, the data can be evenly distributed.

However, we can find that the existing balancing mechanisms are all storage-load-aware. From the view of access performance, unless all the data are accessed with the same possibilities, that is there is not hotspots in data, the effectiveness of storage-load-aware balancing is hardly acceptable. For example, in a news website, the latest news are hotspots since they are accessed much more frequently than other news. In this scenario, the workload of the physical node the latest news stored on is much heavier than other ones though each of these nodes holds almost same amount of data. It is obvious that the heavier the workload of a node is, the lower its performance is. Thus, with a storage-load-aware balancing, the performance of physical nodes are possibly quite different with each other, which is less meaningful for the applications with hotspots in data. Actually, for such applications, access-load-aware data balancing is much more suitable than the storage-load-aware one.

The aim of access-load-aware data balancing is to control the resource utilization of all the physical nodes into a specific range in order to make the performance of data access on each virtual node acceptable. This paper puts forward such a mechanism for optimization of resource utilization. With access-load-aware data balancing, the amount of data stored in a storage node is not forced to be roughly same as other nodes. On the contrary, a node can store much less or more data than other nodes. Moreover, since the storage nodes are virtual nodes hosted on physical nodes, they will be dynamically migrated among physical nodes to ensure the performance of data access.

The remainder of this paper is structured as follows. Section 2 briefly summaries the related works; Section 3 describes a mechanism of access-load-aware data balancing; Section 4 gives the results of experiment performed on Swift to demonstrate the effectiveness of our mechanism; and conclusion is in Section 5.

2 Related Work

Actually, many storage tools have provided the storage-load-aware data balancing. For example, Dynamo supports to evenly distribute the data to all nodes with an improved consistent-hash algorithm[8]; BigTable organizes the nodes into a server farm

in which the master server monitors the slave servers and performs the data migration[9]; MongoDB provides auto-sharding to dynamically balance the storage-load by data migration[10]; Swift provides a data balancing based on the key range[5]. However, as discussed in section 1, all the existing data balancing are storage load oriented. It is clear that such data balancing is not suitable for the applications with hotspots in data. So many researchers focus on this area.

The research on data balancing concentrates on three aspects. The first one is the discovery of workload exception, since it is the basis of data balancing. In [12], the nodes of a cluster are grouped into small sets and the central nodes of each set are in charge of monitoring other nodes of its set. This way improves the performance of monitoring, but it doesn't precisely locate the hotspots in data. In [13], for each chunk in MongoDB, its access-load is evaluated by the numbers of various operations on it. However, the weights of different operations are assigned by experience which makes them too subjective. In [14], the exceptions of workload can be found on the base on predicted access load. Predicted workload is much better than real-time access load for data balancing, but it needs much historical records to guarantee the precision of prediction, which makes the monitoring costly. In summary, we need an effective way to monitor the real-time access load and find the exceptions of workload.

The second aspect is the algorithm of workload balancing. In [15], the files are divided into zones according to the foreseen workload in order to balance the access load. This way is a static one since the access load is foreseen and the location of a file will not be changed once it has been stored into a zone. In [16], the data is dynamically re-partitioned to facilitate rapid data balancing by a graph theoretic way. It is noticeable that since a re-partition operations is independent of others, it is time costly in some situations because it is possible that too much data needs to be migrated in a re-partition operation. In [17], an automated control for elastic storage is put forward to improve the efficiency of data balancing. The amount and the distance between the source and target nodes of data to be migrated are not taken account into this mechanism. Actually, we need an algorithm to reduce the cost of data balancing, particularly the cost of data migration, since it has negative impact on the data balancing.

The third aspect is the method of data migration. In [18], a cost-aware method of data migration is designed to minimize the interference between virtual machines. But the amount of data to be migrated is not reduced in the method. In [19] and [20], a location-aware method is proposed to save power when performing data migration in large-scale datacenters. Similarly, the amount of data to be migrated is not optimized in this method. Data migration is a bandwidth-intensive task, so it is costly to balance storage load by data migration. Actually, we need to balance the access load but not storage load. Thus, we can realize this aim to migration virtual machines of process nodes but not storage nodes.

In summary, access-load-aware data balancing is necessary for applications with hotspots in data, but there is not a comprehensive solution has been proposed. Actually, we proposed an approach to dynamic workload balancing in [21], which periodically checks the overloaded and underloaded nodes and then the dynamically balances the access load by VM migration. We also improved this approach in [22] by optimizing the VM migration. However, both the approaches we proposed are potentially costly because data migration is inevitable in them. So this paper tries to put forward a feasible solution to improve the performance of data balancing by reducing its cost.

3 A Mechanism of Access-Load-Aware Data Balancing

In this section, we will give a mechanism of access-load-aware data balancing, including data storage model, resource management flow and algorithms.

3.1 Data Storage Model

In order to improve the access performance, the data are divided into subsets according to some rules, such as key range and consistent hashing. Each of such subsets is stored in a "storage node" and managed by a dedicated "process node" which is running on a dedicated VM (virtual machine). The storage nodes are mapped onto physical nodes by some way, such as hashing. Consequently, there are two maps in a storage service: the first one maps the keys to process nodes, the second one maps the process nodes to physical nodes. It is noticeable that both of them are theoretically dynamic mappings, which means a subset can be divided into more subsets or merged with other subsets and a process node can be migrated among physical nodes. The two maps is stored in proxy node, which is the access entrance of storage service and in charge of routing the requests to storage nodes. Proxy node is also running in VM and multiple proxy nodes can be built as a cluster to support large-scale concurrent clients. Such an architecture is shown in Fig.1.

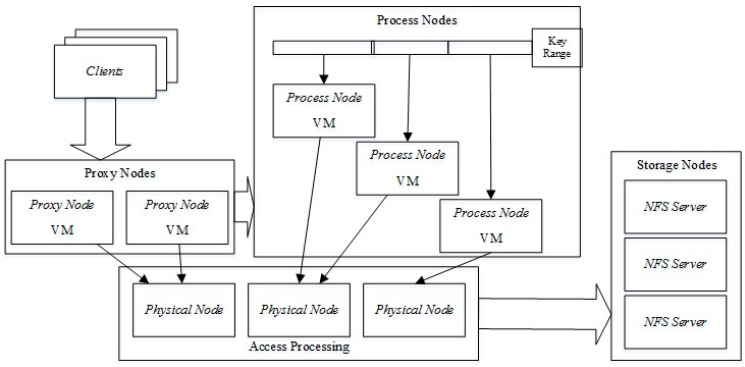

Fig. 1. Architecture of cloud storage service

In Fig.1, each process node processes the access requests to the data it manages. If there are hotspots on a process node, this node will be an overloaded node, and it possibly needs to be migrate to other physical node with more idle resources. If the data are directly stored in process nodes, however, such migration is quite costly since it not only needs to migrate the VM, but also needs to transfer the data, which makes the data balancing non-effective for performance improvement.

Actually, since the data are stored as multiple replicas in cloud storage service, the data can be stored with NFS (Network File System) to avoid the transfer of a large amount of data. With NFS, a process node accesses the NFS servers to fetch and cache the data it manages. When the process node needs to be migrated, only the VM but not the cached data is migrated. After a process node migrated, it needs to fetch the data it manages from the NFS server again, but the fetch is on-demand, so the batch data transfer is not necessary. With such a storage model, the migration of VMs will not be costly any longer and then the data balancing can be accomplished by it.

3.2 Resource Management Flow

In an access load balanced system, the utilization of computing resources of a physical node is roughly same to that of any other physical node. Similarly, the resource utilization of a process node (virtual node) is also roughly equal to that of any other storage node. For a physical node or a process node, if the utilization of any type of resource is too high or too low, we consider it is an overloaded or underloaded node. So we periodically monitor the real-time resource utilization of physical nodes and process nodes and check which nodes are overloaded or underloaded according to the predefined thresholds (upper bounds and lower bounds of resource utilization).

For an overloaded physical node, we should search a paired physical node with more spare resource and choose some process nodes to be migrated to the paired node. After the paired operation, either the overloaded physical node or the paired one is normal. If there is no physical node can be paired, a new physical node is added into the cluster and the process nodes on overloaded physical node can be migrated to it by pair operation.

For an underloaded physical node, we don't process it immediately because it is possible that an overloaded physical node will pair with it later. So we setup a counter, only when it has been underloaded for a specified period, which means there is no overloaded nodes in system, it will search other underloaded node and merge with it. Merging means the physical node with lower resource utilization migrates all the process nodes hosted on it to the physical node with higher resource utilization. After merge operation, the physical node from which the process nodes are migrated out of can be setup as idle.

For an overloaded process node, we allocation more spare resource of its host physical node to it, that is to scale up the process node. If there is no enough spare resource can be allocated to it, we mark the physical node as an overloaded node. Then, the physical node will release more resource by pair operation in order to reallocate them to the overloaded process node.

For an underloaded process node, we simply scale down it by releasing part of its allocated resource. When the allocated resource of a process node is scaled down to the specified lower bound, the scaling down is not executed anymore.

The Fig.2 (a) and (b) respectively shows the resource management flow of physical nodes and process nodes. They are periodically executed and independent with each other.

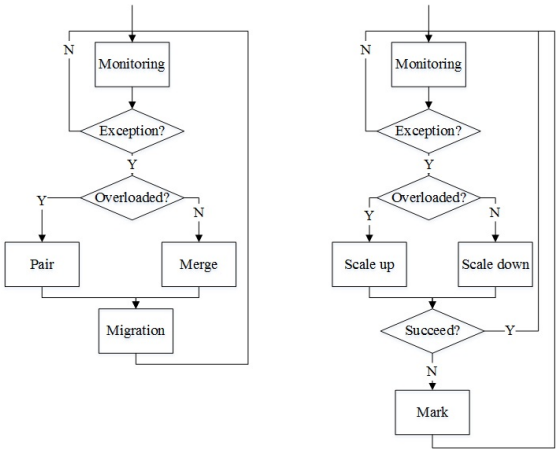

Fig. 2. Resource management flow of (a) physical nodes & (b) process nodes

3.3 Resource Management Algorithms

In the resource management flow, the core is the algorithms for detecting the overloaded and underloaded nodes, finding pair node and merging underloaded nodes.

To describe the algorithms, we define the necessary parameters in Table 1.

Table 1. Parameters for Resource Management

Parameters	Description
U_p	The monitored utilization vector of node p
L_p	The load index of node p
U_{cu}	The upper bound of utilization of CPU
U_{mu}	The upper bound of utilization of memory
U_{bu}	The upper bound of utilization of bandwidth
U_{cl}	The lower bound of utilization of CPU
U_{ml}	The lower bound of utilization of memory
U_{bl}	The lower bound of utilization of memory
w_c	The weight of utilization of CPU
w_m	The weight of utilization of memory
w_b	The weight of utilization of bandwidth
t	The threshold of underloaded utilization of resource
I_m	The interval between two successive monitoring operation
I_r	The interval between two successive exception detection

We used the resource utilization to evaluate the access load of physical nodes and process nodes. Thus, access load balancing is accomplished by controlling the resource utilization into an acceptable range. So we predefine the upper bound and lower bound of the utilization of each type of resources, including CPU, memory and bandwidth. Then we compare the monitored utilization with their upper bounds and lower bounds to detect the overloaded and underloaded nodes.

Actually, the resource utilization of a node p is a vector, since each node has many types of resource, such as CPU, memory, bandwidth, disk and so on. For simplification, we only focus the former three types of resource. So we denote the monitored resource utilization of node p as U_p:

$$U_p = <U_{cp}, U_{mp}, U_{bp}> \quad (1)$$

The utilization is monitored at the interval of I_m. But it is noticeable that the resource utilization is frequently changed in practice. For smoothing the utilization, U_p is the average value of the raw data monitored during I_r.

If any component of U_p is greater than the upper bound of corresponding utilization of resource, such as U_{cp} is greater than U_{cu}, we consider node p is overloaded. However, the process of determining a node whether is underloaded is a little more complex. We calculate the weighted average value of all components of Up as load index L_p, as shown in (2). If L_p is less than t, a predefined threshold, then we consider node p is underloaded. This design makes the responses to overloaded nodes quite rapid and the ones to underloaded nodes prudent. After all, it is the overloaded nodes but not underloaded nodes that are the bottleneck of access performance.

$$L_p = w_c*U_{cp} + w_m*U_{mp} + w_b*U_{bp} \quad (2)$$

The algorithm for determining whether a node is overloaded or underloaded is named as "Exception Detection" shown in Table 2. This algorithm is executed periodically, the interval of between two successive exception detection is I_r, a configurable parameter.

Table 2. Algorithm for Exception Detection

Algorithm: Exception Detection	
Input:	U_p, L_p
Output:	*result:enum<NORMAL, OVERLOADED, UNDERLOADED>*
1.	**foreach** U_{ip} **in** U_p($i \in \{c, m, b\}$)
2.	**if** $U_{ip} > U_{iu}$ **then**
3.	result = *OVERLOADED*
4.	**else if** $L_p < t$
5.	result = *UNDERLOADED*
6.	**else** result = *NORMAL*
7.	**endif**
8.	**endif**
9.	**endfor**
10.	**return** result

For the overloaded and underloaded process nodes, we try to scale up and down them. To avoid generating resource fragments, we scale up or down the process nodes by adding or removing certain number of unit resource. The grain of unit resource is

dependent on the configuration of physical nodes. For example, a 4-core physical node can define a quarter of its resource as a unit. Suppose an overloaded process node has been allocated with N_p units of resource, we will scale up it by adding N_a units of resource:

$$N_a = \lceil N_p * Max\left(\frac{U_{ip} - U_{iu}}{U_{iu}}\right) \rceil, \ where\ i = c,m,p \tag{3}$$

Here, the maximum of the difference between the monitored utilization and the upper bound of utilization of each type of resource reflects the extent of overloaded. So we use it to determine how many units should be added to the overloaded process node. If the physical node has no enough idle resource to be allocated to the process node, it will be marked as an overloaded physical node.

Similarly, for an underloaded node, we will scale down it by removing N_r units of resource:

$$N_r = \lfloor N_p * \frac{t - L_p}{t} \rfloor \tag{4}$$

Here, the difference between L_p and t denotes the extent of underloaded, which determines how many units should be removed from the underloaded process node. When N_p is 1, even the process node is underloaded, it will not be scaled down yet, since we wouldn't want the resource fragments. However, it is still possible that there are resource fragments inside units if the grain of unit is coarse. Consequently, the grain of unit should be defined carefully.

If a physical node is marked as an overloaded one, it will execute pair operation to find a suitable peer and migrate some process node(s) into it. In the pair operation, the first task is to determine which process nodes should be migrated out. The aim is to ensure that each component of U_p of the physical node is below its upper bound and the number of process nodes to be migrated is minimized in order to improve the efficiency of migration. For the U_{ip} of physical node in (3), we sort the process nodes by their own U_{ip} in ascending order, and choose the one(s) whose U_{ip} or sum of U_{ip} is closest to and greater than the difference between U_{ip} and U_{ic} of the physical node as the target(s) to be migrated.

The second task of pair operation is to find a physical node which can host all the targets chosen in first task and no component of its U_p will be beyond U_{ic} of physical node after migration. Although it possibly exists multiple candidate peers among a large cluster, the pair operation stops when it finds a suitable peer so that its efficiency is guaranteed. If there is no candidate peer is found, a new physical node will be introduced into the cluster for hosting the target process nodes. The algorithm of pair operation is shown in Table 3.

Table 3. Algorithm of Pair Operation

Algorithm: Pair Operation

Input: *P_o: an overloaded physical node*

Output: *targets: set of process nodes, peer: paired physical node*

```
1.  targets = Ø
2.  peer = null
3.  max_different = 0
4.  max_comp = null
5.
6.  foreach U_ip in U_p of P_o
7.     if (U_ip - U_iu) > max_different then
8.           max_different = U_ip - U_iu
9.           max_comp = U_ip
10. endfor
11.
12. sort(V, max_comp)
13. // sort all the process nodes V hosted on P_o
14. // by max_comp in ascending order
15.
16. if max(U_ip of p in V) > max_different
17.    foreach p_i in V
18.          if U_ip > max_different
19.                targets ←p_i
20.                break
21.          endif
22.    endfor
23. else
24.       foreach p_i in V    // traverse V in reverse order
25.          targets ←p_i
26.          if sum(U_ip of p in targets) > max_different)
27.                break
28.          endif
29.       endfor
30. endif
31.
32. foreach p_i in C
33. // here, C is the set of all available physical nodes
34.       if (U_ip + sum(U_ip of p in targets)) < U_iu
35.          peer = p_i
36.          break
37.       endif
38. endfor
39. if (peer = null)
40.       peer = new PhysicalNode()
41. endif
42.
43. To migrate targets from P_o to peer
44.
45. return targets and peer
```

If a physical node is marked as an underloaded one, it will not be merged with other nodes immediately. Instead, it will wait for a specified period to give the overloaded nodes a chance to pair with it. If the waiting period is expired, which means there is no overloaded node at present, an underloaded node will merge with another underloaded one. For the pair of underloaded nodes, the one with lighter load will

migrate all of its hosted process nodes to the one with heavier load so that the migration can be efficiently accomplished. After merging, one node is passivated and the other one is remained. Similar to pair operation, no component of U_p of the remained node is beyond U_{ic} of physical node after merging. As a result, merge operation will use the same method as pair operation to find the paired peer.

If an underloaded node cannot find a paired underloaded peer, which means that it is the only underloaded node in the whole system, it will remain underloaded status and no process node needs to be migrated.

The access-load-aware dynamic data balancing we proposed is implemented by the combination of all the above algorithms.

4 Experiments

4.1 Setup and Configuration

We built an experimental environment with 8 physical machines in which 6 machines are used to build a Swift system and 2 machines are setup as clients to generate access load. In the 6 machines of Swift system, 2 of them are setup as an NFS to store the data and support the VM migration, while 4 of them are setup as process servers to host virtual nodes.

On the process servers, we run 17 XenServer[23] VMs to host virtual nodes of Swift, including 2 proxy nodes and 15 process nodes. The locations of these virtual nodes of Swift are not fixed, since they could be migrated among physical nodes. On the client machines, we run 10 XenServer VMs to simulate the customers of Swift and generate access load. All the VMs are installed Ubuntu Server 12.04 as operating systems.

All the machines of Swift are interconnected through LAN, and the client machines access the Swift system through a switch. The configuration of these machines are listed in Table 4.

Each of the VMs is initially provisioned with 1 core, 512MB memory and 25GB disk. All the VMs of Swift has their own preconfigured IPs while the VMs running on client machines are assigned IPs dynamically.

Table 4. Configuration of Physical Machines

Category	Instances	CPU	Memory(GB)	Disk(GB)
Process Server	XenServer1	Intel i5 3.30G Hz	4	500
	XenServer2			
	XenServer3			
	XenServer4			
NFS Server	NFS1	Intel i3 3.30G Hz	4	500
	NFS2			
Client	Client1	Intel i3 3.30G Hz	4	500
	Client2			

Considering all the data and their replicas are stored in NFS servers and each process node can cache at most 25GB data, we generated about 300GB data and stored them into the Swift system. Then, Pylot[24], a free open source tool for testing

performance and scalability of web services, is used to generate access load through the cluster of the 10 VMs hosted on client machines.

To simulate the hotspot, Pylot randomly selected several small sets of the data and generated high access load to these sets. In the testing scripts, the maximum of the number of concurrent clients is 100, the ramp up time, that is the time to generate the maximal number of concurrent clients, is 1200 seconds, the interval between two successive access requests sent by the same client is 0 second, that is the agents continuously send requests without any interval, and the duration of testing with the maximal number of concurrent clients is 6000 seconds.

Table 5 shows the configuration of various parameters for resource management of physical machines and VMs used in the experiments.

In Table 5, we assigned rather higher weight to the utilization of memory because we found that in general the data access in Swift is a memory-intensive operation. Meanwhile, we set the upper bound of the utilization of CPU lower than other ones since it has a more significant impact on the access performance. In other implementation of cloud storage services, this configuration is probably not so reasonable and the parameters can be reconfigured with suitable values.

Table 5. Configuration of Parameters for Resource Management of Physical Machine and VM

Parameters	Values for Physical Machine	Values for VM
U_{cu}	0.6	0.7
U_{mu}	0.95	0.85
U_{bu}	0.95	0.95
U_{cl}, U_{ml} & U_{bl}	0.3	0.3
w_c	0.2	0.2
w_m	0.7	0.6
w_b	0.1	0.3
t	0.7	0.8
I_m	5 seconds	1 mins
I_r	15 seconds	5 mins

4.2 Result and Analysis

The result we recorded when the experiment was performed including the following two parts: the first part recorded the utilization of CPU, memory and bandwidth of physical machines and the number of VMs hosted in each physical machines during the experiment, as shown in Fig.3; the second part recorded the utilization of CPU and memory and the allocated memory of all 15 VMs of storage nodes during the experiment.

In Fig.3, we can find that the utilization of CPU, memory and bandwidth of all the 4 physical nodes has never been out of the range specified by the upper bound and lower bound. The sum of the numbers of VMs running on the 4 physical nodes is always 15.

For XenServer1, there are 2 VMs running on it at the beginning and its resource utilization is lower than that of other nodes, so some VMs are migrated from other nodes to it later. After migration, the VM needs to rebuild its cache by reading the

data from NFS which results in the increase of utilization of bandwidth. After the number of VMs reaches to 5, the resource utilization is stable, so there is not any more VM migration occurs on XenServer1.

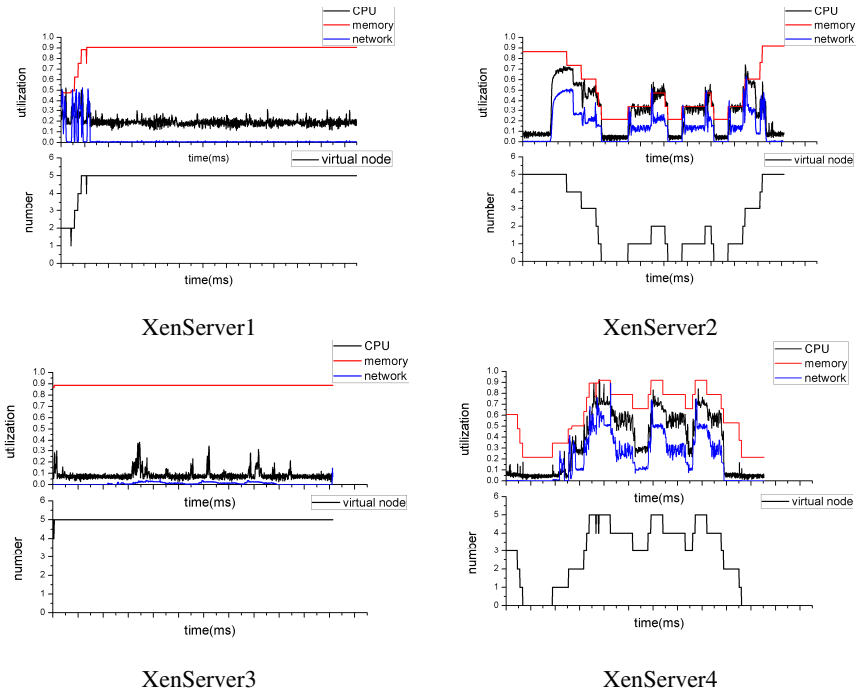

Fig. 3. The Utilization of CPU, memory and network and the numbers of VMs of 4 storage servers

For XenServer2, when the utilization of CPU reaches to the upper bound U_{mu}, the VM migration is executed repeatedly until it becomes lower than U_{mu}. When the utilization of memory is lower than the lower bound U_{mc}, the VM migration is executed again in order to empty the XenServer2 and then passivate it as idle status. When some VM needs to be migrated to an idle server, XenServer2 is activated and hosts the VM.

For XenServer3, its utilization of CPU, memory and bandwidth is rather stable and in the acceptable ranges. Moreover, the utilization of memory is always too high to host any new VM. So the number of VMs running on XenServer3 is unchanged.

The situation of XenServer4 is quite same than XenServer2: VMs are frequently migrated into and out of XenServer4 to ensure the resource utilization is in the acceptable ranges.

We can find that the utilization of CPU and memory of all the 15 VMs has never been out of the range specified by the upper bound and lower bound either. If the utilization of memory of any VM reaches to the upper bound U_{mu}, the VM will be provisioned with more memory. There are 5 VMs got more memory during the experiment.

The Fig.4 shows the number of active physical nodes and the average CPU utilization of all the active physical nodes. We can find that the utilization of CPU is dynamically controlled in an acceptable range to maintain the balance of access load and the number of physical nodes is always the minimum required to guarantee the access performance.

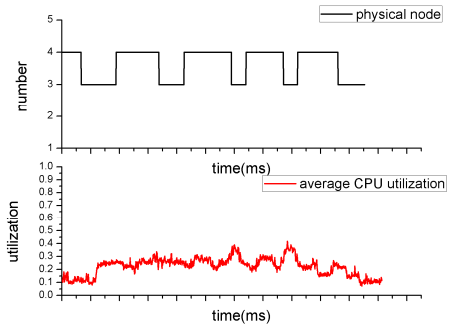

Fig. 4. The number of active physical nodes and the average CPU utilization of all the active physical nodes

In summary, no matter the physical nodes or VMs, their utilization of CPU, memory and bandwidth is always in acceptable ranges, which demonstrates that the access load is indeed dynamically balanced.

5 Conclusion

Aiming to the typical cloud applications with hotspots inside their massive data, we put forward a mechanism of dynamic data balancing by which the access load but not storage load is balanced and therefore the resource utilization is optimized and the performance of accessing data is improved. The mechanism detects the overloaded and underloaded physical nodes and virtual nodes by monitoring their utilization of resource. Then, it dynamically balances the access load among the nodes by pair, merge, mark, scale up and scale down operations. This mechanism is a complementation of storage-load-aware data balancing. At present, our mechanism is only applied in Swift. Although Swift is a popular object storage system widely adopted by many cloud storage services, there are many other storage systems in use. In future, we will apply the mechanism onto other systems to demonstrate its ubiquitous effectiveness.

References

1. Structured vs. Unstructured Data, `http://www.robertprimmer.com/blog/structured-vs-unstructured.html`
2. Amazon, Amazon S3, `http://aws.amazon.com/s3`
3. Google Cloud Storage, `http://www.google.com/enterprise/cloud`

4. Cloud Files, Cloud CDN, and Unlimited Online Storage, http://www.rackspace.com/cloud/public/files/
5. Openstack, http://www.openstack.org
6. Eucalyptus, http://www.eucalyptus.com
7. Nimbus, http://www.nimbusproject.org
8. DeCanadia, G., Hastorun, D., Jampani, M., et al.: Dynamo: Amazon's Highly Available Key-value Store. In: 21st ACM SIGOPS Symposium on Operating Systems Principles, pp. 205–220. ACM Press, New York (2007)
9. Chang, F., Dean, J., Ghemawat, S., et al.: Bigtable: A Distributed Storage System for Structured Data. J. ACM Transaction on Comput. Syst. 26, 1–26 (2008)
10. MongoDB, http://www.mongodb.org/
11. Ghemawat, S., Gobioff, H., Leung, S.: The Google File System. In: 19th ACM SIGOPS Symposium on Operating Systems Principles, pp. 29–43. ACM Press, New York (2003)
12. Deng, Y., Lau, R.: Heat Diffusion Based Dynamic Load Balancing for Distributed Virtual Environments. In: 17th ACM Symposium on Virtual Reality Software and Technology, pp. 203–210. ACM Press, New York (2010)
13. Liu, Y., Wan, Y., Jin, Y.: Research on The Improvement of MongoDB Auto-Sharding in Cloud Environment. In: 7th International Conference on Computer Science & Education, Melbourne, VIC, Australia, pp. 851–854 (2012)
14. Pearce, O., Gambliny, T., Supinskiy, B., et al.: Quantifying the Effectiveness of Load Balance Algorithms. In: 26th ACM International Conference on Supercomputing, pp. 185–194. ACM Press, New York (2012)
15. Zhu, Y., Yu, Y., Wang, W., et al.: A Balanced Allocation Strategy for File Assignment in Parallel I/O Systems. In: 5th IEEE International Conference on Networking, Architecture and Storage, pp. 257–266. IEEE Press, New York (2010)
16. Bui, T.N., Deng, X., Zrncic, C.M.: An Improved Ant-Based Algorithm for the Degree-Constrained Minimum Spanning Tree Problem. J. IEEE Transactions on Evolutionary Computation 16, 266–278 (2012)
17. Lim, H.C., Babu, S., Chase, J.S.: Automated control for elastic storage. In: 7th International Conference on Autonomic Computing, Washington, DC, USA, pp. 1–10 (2010)
18. Qin, X., Zhang, W., Wang, W., et al.: Towards a Cost-Aware Data Migration Approach for Key-Value Stores. In: 2012 IEEE International Conference on Cluster Computing, pp. 551–556. IEEE Press, New York (2012)
19. Liu, Z., Lin, M., Wierman, A., et al.: Greening Geographical Load Balancing. In: Liu, Z., Lin, M., Wierman, A., et al. (eds.) 2011 ACM SIGMETRICS Joint International Conference on Measurement and Modeling of Computer Systems, pp. 233–244. ACM Press, New York (2011)
20. Lin, M., Wierman, A., Andrew, L.L.H., et al.: Dynamic Right-sizing for Power-proportional Data Centers. In: 2011 IEEE INFOCOM, pp. 1098–1106. IEEE Press, New York (2011)
21. Zhang, C., Chen, H., Gao, S.: ALARM: Autonomic Load-Aware Resource Management for P2P Key-value Stores in Cloud. In: 9th IEEE International Conference on Dependable, Autonomic and Secure Computing, pp. 404–410. IEEE Press, New York (2011)
22. Ban, Y., Chen, H., Wang, Z.: EALARM: An Enhanced Autonomic Load-Aware Resource Management. In: 7th IEEE International Symposium on Service-Oriented System Engineering, pp. 150–155. IEEE Press, New York (2013)
23. XenServer, http://www.citrix.com/products/xenserver/resources-and-support.html
24. Pylot, http://www.pylot.org/