International Journal of Innovative
Computing, Information and Control
Volume **15**, Number **1**, February **2019**

ICIC International ©2019 ISSN 1349-4198
pp. 79–90

HOTSPOT SENSITIVE DYNAMIC SCALING FOR DISTRIBUTED CACHE SYSTEMS

Bo Zhou[1], Yaqiong Li[1], Yongbo Liu[1], Shouchao Li[1] and Yunkui Song[2,*]

[1]Jiangsu Hoperun Software Company
No. 168, Ruanjian Ave., Yuhuatai Dist., Nanjing 210012, P. R. China

[2]Institute of Software, Chinese Academy of Sciences
No. 4, South Fourth Street, Zhongguancun, Beijing 100190, P. R. China
*Corresponding author: songyk@otcaix.iscas.ac.cn

Received April 2018; revised August 2018

Abstract. *Distributed cache systems have been widely used to improve the applications' performance of accessing data by moving data and applications together in a cluster. However, existing works cannot well rebalance data partitions, and guarantee the data consistency and service availability during expanding cache systems. This paper proposes a hotspot sensitive dynamic scaling approach for distributed cache systems. First, we propose a hotspot sensitive data rebalance method, which is suitable for heterogeneous environment. It considers both memory utilization and network traffic, identifies hotspot data partitions, and then sets a high priority for the cache nodes with light workloads to guarantee the balance of cache nodes. Then, we propose a data accessing method based on two-phase controlled data migration to ensure the data consistency and service availability during scaling up cache systems. Finally, we have implemented a cache framework CacheScale, and conducted a series of experiments to validate the approach. The experimental results demonstrate that our approach can dynamically scale up cache systems, guarantee the data consistency and service availability, and achieve shorter response time.*

Keywords: Distributed cache, Dynamic scaling, Hotspot data partition, Data migration

1. **Introduction.** In cloud computing, traditional database technologies cannot well deal with the performance bottleneck of accessing massive data from a huge number of requests, so distributed cache systems are introduced and widely used in cloud computing platforms. Distributed cache technologies move data and applications in a cluster together to improve the applications' performance of accessing data. Many providers of cloud computing services, e.g., VMware, Microsoft, IBM, and Oracle, have proposed their own distributed cache solutions to guarantee the service level agreement. The cache technologies can be divided into three categories that are local cache, static distributed cache, and dynamic distributed cache. Local cache technologies focus on accessing data with high performance [1]; static distributed cache technologies support clustering and static scalability; dynamic distributed cache technologies support dynamic scalability and fault tolerance. Nowadays, academies have conducted extensive works on dynamic cache technologies. Chiu et al. [2,3] study the dynamic expansion of a distributed caching cluster, and propose a greedy based data migration method, which sets a high priority for the destination node with light weight. This work cannot guarantee the data consistency and service availability and does not consider the impact of heterogeneous nodes and hotspot partitions. Dynamo [4] has implemented a distributed consistency Hash algorithm, which

DOI: 10.24507/ijicic.15.01.79

improves load balancing and supports heterogeneous nodes. [5,6] make similar improvement on data rebalance with distributed hashing algorithm, but they do not consider the impact of hotspot partitions in the real scenario. Pfaffhauser [7] has introduced the dynamic expansion and load balancing method for cloud storage.

Recent related works pay much attention to improving the performance of distributed caching systems in different application scenarios. Xing et al. [11] propose a distributed multi-level storage model with a multiple factors least frequently used algorithm to solve the problems of restricted computation, limited storage and unstable network. Xiong et al. [12] propose a replication strategy for spatiotemporal data based on a distributed caching system, which mines the files with a high popularity from historical user accesses, generates replicas and selects appropriate cache nodes for placement. Jošilo et al. [13] provide a polynomial time solution when the link costs are induced by a potential and propose a 2-approximation algorithm based on the content demands for the general case. Chu et al. [14] partition each cache into slices dedicated to content providers, propose a content-oblivious request routing algorithm to optimize the routing strategy, and then extend the caching model to bandwidth-constrained and minimum-delay scenarios. Ma et al. [15] construct a segment access aware dynamic semantic cache for relational databases and propose a cache access algorithm using cache items with effective lifecycle tag for cache consistency.

However, the used Hash algorithm keeps key orders, so the initial data distribution is unbalanced, when the key value is not distributed. Furthermore, the load balancing is decided by coordinating nodes in the system, so it requires many iterations, which introduce significant overhead. There are two major challenges to implement flexible distributed cache systems as follows. First, a distributed cache system rebalances data partitions during dynamically expanding the cache cluster. The data balance is to equally distribute data partitions on each cache node [8]. The redistribution of data partitions is represented by the change of the routing information [9]. Thus, an effective data rebalance method is important to improve the performance of a distributed cache system, which equally distributes workloads to every cache node with a limited migration overhead. In a distributed cache system, different data partitions have different characteristics of accessing data. The hotspot data partitions have much heavy workloads in the distributed cache system. An efficient data rebalance method should distribute hotspot partitions in each cache node equally and consider the performance differences of different servers to adapt to heterogeneous environment. Second, the data in a distributed cache system should be consistent and available, when the cluster is dynamically expanded. Traditional cache systems used for caching pages and objects mainly aim at speeding up data access. The contemporary cache systems store applications' state to ensure the scalability and reliability of applications, so they guarantee the data consistency and service availability. However, most existing distributed cache systems cannot support dynamic expansions, e.g., OSCache Cluster, Memcached, and Terracotta EX. They restart services when scaling up, which often leads to data loss and service unavailability. The dynamic expansion of distributed cache systems needs to migrate data partitions between nodes, which brings the challenges of data consistency and continuous availability.

To address the above issues, this paper proposes a hotspot sensitive dynamic scaling approach for distributed cache systems. We access expected data partitions in the migration, when the data version of the source node is inconsistent with that of the destination node. Furthermore, we avoid extensive overhead caused by migration to guarantee the continuous availability of cache services. Specifically, the contributions of this paper are as follows.

- We identify hotspot data partitions with high resource utilization and set high priority for the cache nodes with light workloads to dynamically rebalance data partitions.
- We propose a data accessing method based on two-phase controlled data migration to guarantee data consistency during expanding cache systems.
- We have implemented a distributed cache framework CacheScale based on the proposed methods, and conducted a series of experiments to validate the methods.

This paper is organized as follows. Section 2 proposes a hotspot sensitive data rebalance method. Section 3 proposes a data accessing method based on two-phase controlled data migration. Section 4 proposes our dynamic scaling framework – CacheScale. Section 5 conducts and discusses a series of experiments. Section 6 summarizes this paper and discusses the future work.

2. **Hotspot Sensitive Data Rebalance.** When the scale of a distributed cache system increases, we redistribute data partitions to rebalance data on cache servers. We propose a data rebalance algorithm of redistributing data partitions to minimize the overall unbalance degree of distributed cache systems in one-way migration, which is an NP-hard optimization problem. Concretely, we first set the memory limit and the network limit to avoid that a cache node interferes with other cache nodes consolidated on the same server. Second, we monitor a cache node's memory utilization and network utilization, and then use their product to present the workload of a cache node. Third, we evaluate the data balance of a cluster, and then online recognize hotspot data partitions. Finally, as a distributed cache system often has a large scale, we use a greedy based method to solve the approximate optimal problem of minimizing the overall unbalance degree of distributed cache systems. The greedy based method migrates the recognized hotspot data partitions to the server with light workloads. Our approach considers heterogeneous nodes with different physical resources, which is suitable for complex cloud computing environment. Furthermore, our approach rebalances and migrates data partitions, which can improve resource utilization and reduce the number of servers. Finally, our approach online perceives workloads and adjusts the cache server cluster, which can well adapt to changing deployment environment. We describe the notations in Table 1 and the algorithm of rebalancing data partitions in Table 2.

Table 1. Notation description

Notation	Description
C	a distributed cache cluster
mem$[C]$	the utilized memory of a cache cluster C
net$[C]$	utilized network traffic of a cache cluster C
memcap$[C]$	memory limit of a cache cluster C
netcap$[C]$	network traffic limit of a cache cluster C
mnrate$[C]$	resource utilization rate of a cache cluster C
sizehot$[C]$	the size of hotspot partitions in a cache cluster C
S_i $(1 \leq i \leq n)$	ith cache server, n is the number of servers in the cache cluster
mem$[S_i]$	the utilized memory of a cache server S_i
net$[S_i]$	utilized network traffic of a cache server S_i
memcap$[S_i]$	memory limit of a cache server S_i
netcap$[S_i]$	network traffic limit of a cache server S_i
mnrate$[S_i]$	resource utilization rate of a cache server S_i
sizehot$[S_i]$	the size of hotspot partitions in a cache server S_i

TABLE 2. Algorithm of rebalancing data partitions

Algorithm 1: Rebalancing data partitions
Input: mem[S_i], net[S_i], memcap[S_i], netcap[S_i] ($1 \leq i \leq n$)
Output: ($S_1, S_2, \ldots, S_n$), *transSet*

1. for each S_i in C
   a) calculate mem[S_i], net[S_i]
   b) mnrate[S_i] = memcap[S_i] $\times$ netcap[S_i]
2. end for
3. mnrate[C] = memcap[C] $\times$ netcap[C]
4. calculate sizehot[C]
5. *inset* = $\{S_i|$mnrate[S_i] >= mnrate[C]$\}$,
6. *outset* = $\{S_i|$mnrate[S_i] < mnrate[C]$\}$
7. while true do
   a) S_f = find_max_mnrate (*outset*)
   b) if ((sizehot[S_f] $-$ 1)/netcap[S_f] $\geq$ size_hot[C]/netcap[C])
      i. P = best_hot(S_f)
   c) else
      i. P = best_other(S_f)
   d) end if
   e) if without P, mnrate[S_f] < mnrate[C]
      i. break
   f) S_t = best_dst(*inset*, P)
   g) if S_t == null
      i. break
   h) end if
   i) add $< S_f, S_t, P >$ to *transSet*
   j) remove P from S_f
   k) add P to S_t
8. end while
9. return ($S_1, S_2, \ldots, S_n$), *transSet*

Table 2 shows the pseudocode of our algorithm, which is described as follows. In the inputs, the mem[S_i] and net[S_i] are derived from the monitoring information of each cache node in period, and memcap[S_i] and netcap[S_i] are based on the physical resources in the startup period. The cache cluster manager executes the method, when it expands the cluster by adding nodes. From the output, we can get the partition-server mapping table recording routing information from ($S_1, S_2, \ldots, S_n$), and get the migration plan with the output *transSet*. The cache cluster manager coordinates cache nodes to update the routing information of partitions and migrate data partitions.

(1) The nodes in C are divided into two sets that are outset and inset, according to the calculated mnrate, and data partitions are migrated from the nodes in *outset* to those in *inset*.

(2) We select a node S_f with the maximum mnrate in *outset* as the node to be migrated.

(3) We select appropriate data partitions to be migrated from S_f.

(4) When the data partition P is migrated from S_f, if mnrate[S_f] < mnrate[C], the execution is jumped to step (7).

(5) We select node S_t from *inset* to store the partition P.

(6) If a suitable S_t exists, we insert $< S_f, S_t, P >$ into the *transSet*, update related changed variables, and then the execution jumps to step (2).

(7) The algorithm returns each cache node and *transSet*.

3. **Migration with Data Consistency and Service Availability.** After migrating data partitions, we delete the migrated data partitions from the source nodes to guarantee data availability during migration. Thus, the same data partitions exist in both the source nodes and the destination nodes during the migration, and then two versions of the same data partition may not be consistent. To address the above issue, we propose a data consistency method. The "Get" operation reading data uses ingoing node priority, which uses the data in the ingoing node. The "Update" operation writing data first updates the outgoing node, and then updates the ingoing nodes to avoid the data loss during updating.

As shown in Figure 1, we propose a data access protocol based on the above consistency strategy. The clients and servers store their own routing information of data partitions, use piggyback confirmation [10] to synchronize partition routing information, and achieve the consistent access of cached data. Processing requests can be divided into two phases. In the first phase, a client sends access requests with routing information to a server. When the server receives the request and detects that the client's routing version is not consistent with its own, the server sends a response to the client with its latest routing information. The client resends a request to the server according to the latest routing information. If the server detects that the routing information of the client is consistent with its own, the server sends a response with the status of the current cache cluster to the client.

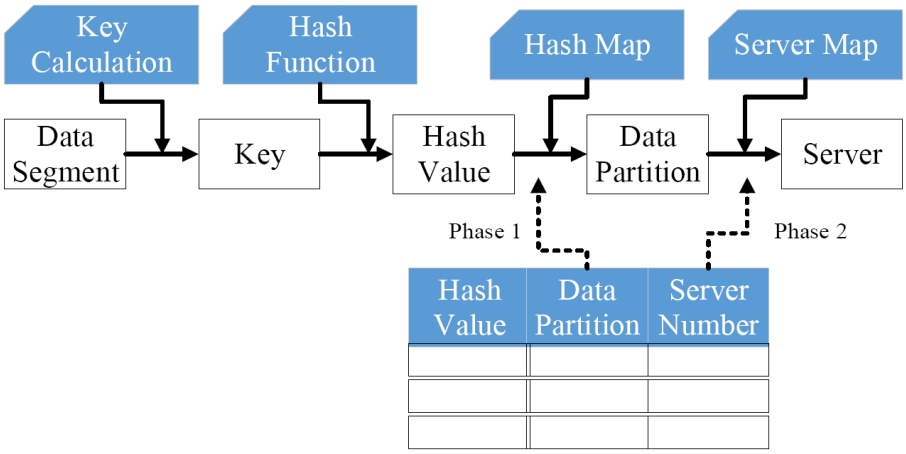


FIGURE 1. Two-phase data access

We take the reading operation as an example to introduce the two-phase procedure of processing requests. Figure 2 shows that three cache servers S_1, S_2 and S_3 are in the distributed cache system. Each server has many data partitions, for example, server S_1 has two partitions: p_1 and p_2. According to the Hash algorithm, the client maps key a to partition p_2 that is stored in S_3, so a is stored in it. The system adds a cache server S_4 to dynamically expand the distributed cache system. According to our data rebalance algorithm, many partitions migrate among cache servers, for example, partition p_2 migrates from S_3 to S_4. Since the migration is not completed, the distributed cache system is in the "Unstable" status. Thus, when requests arrive at cache clients, the system processes these requests as follows.

(1) A cache client receives a request for saving a key "a".
(2) The cache client calculates the hash value of a and maps the hash value to partition p_2. It reads the latest routing information, and then stores partition p_2 in the cache server S_4.
(3) Since the distributed cache system is in the "Unstable" status, S_4 sends response with the status.
(4) The cache client detects that the status of the cache server is in the "Unstable" status and reads the last version of the routing information. Thus, the partition p_2 is mapped to cache server S_3.
(5) The cache client sends a request to the cache server S_3, and S_3 responds to the client.
(6) The cache client returns the result to the application using our method of guaranteeing data consistency.

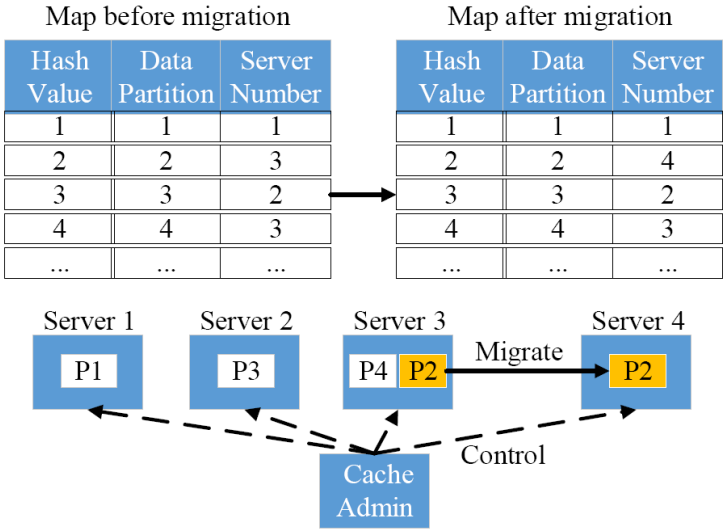


FIGURE 2. Data partition migration procedure

When CacheScale is migrating data, the distributed cache system is in an "Unstable" status. All requests should be completed in two phases, which takes a lot of time, so the data migration should be completed as soon as possible. The data migration occupies much system resources, and the rapid migration may aggravate the resource competition. The cache data is usually migrated from the nodes with heavy workloads to the nodes with light workloads, so data migration has a greater impact on the source nodes. Thus, as shown in Table 3, we designed a controlled data migration algorithm. CacheScale monitors the network overhead of source nodes and increases the wait time by 18% to slow down the data migration, when the network traffic accounted for the proportion of its network bandwidth reaches the predefined threshold. On the other hand, to avoid that the cache cluster is in "Unstable" state for a long time in heavy workloads, we set the data migration to be 8% of the minimum network bandwidth.

4. Dynamic Scaling Framework – CacheScale.

4.1. CacheScale architecture. Figure 3 shows the CacheScale architecture, which consists of cache clients, cache servers, node agents and a master node. In CacheScale, cache servers run independently, cache clients actively request routing, and the cluster administrator centrally manages the framework. A cache client (CC) establishes connections to the cache server and exposes APIs ("GET" and "PUT") to Web applications for accessing data. A cache server (CS) provides efficient key-value data storage interfaces. The cache

TABLE 3. Controlled data migration algorithm

Algorithm 2: Controlled data migration
Input: *partitionServerSet*
Output: migration operations

1. *migrating* = true, *migratedSize* = 0, *timeStart* = current time()
2. foreach $< P, S >$ in *partitionServerSet*
   a) foreach item in P //migrate item to target cache server
      i. migrate(*item*, *targetServer*)
      ii. *migratedSize* += *item.size()*
      iii. if (*migratedSize* > Threshold)
      iv. *timeSpan* = current_time() – *timeStart*
      v. if ((net_usage() > Net_Threshold)&(*migratedSize*/*timespan* > Net_Bandwidth × 0.08))
         1. *sleeptime* = *sleeptime* × 1.18
      vi. end if
      vii. sleep(*sleeptime*)
      viii. *migratedSize* = 0, *timeStart* = current_time()
      ix. end if
   b) end for
3. end for

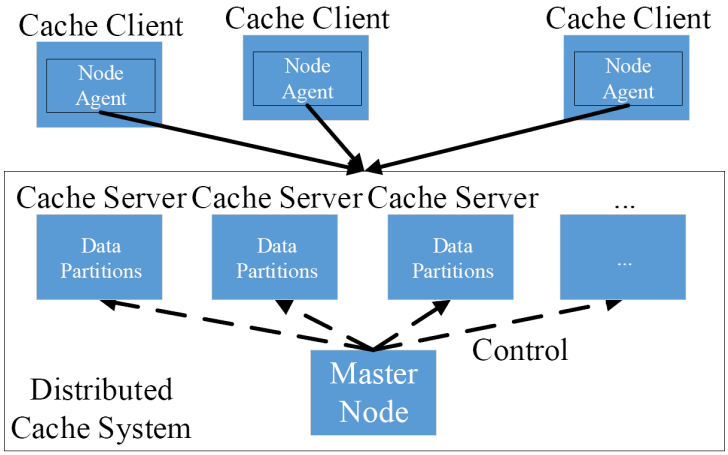


FIGURE 3. CacheScale architecture

server cluster administrator (CA) monitors the distributed cache system, manages cache servers' life cycles, and controls the cache servers' configuration with global configuration interfaces. A node agent deployed on each server deploys, starts, stops, monitors the server, reports the server's resource utilization to the master node in period, and receives and executes control commands from the master node. A master node manages a cache cluster and provides operating interfaces to tenants and the administrator.

(1) Cache Server

The cache server has five components including kernel, configuration, request processor, migration and performance statistics. The kernel allocates, stores and queries cache data; supports concurrent connections with libevent library; schedules caching requests. The configuration component manages routing information and cache groups. The request

processor implements the piggybacking ACKs of a data accessing protocol in the server side. The migration component uses the slicing mechanism to migrate cache data with traffic control. The performance statistics component calculates the cache utilization of a partition.

(2) Cache Client

A cache client has five components including kernel, command processor, configuration manager, request distributor and performance counter. The kernel uses a queue for reading and writing operations to guarantee the sequence of operating data, reads and writes network packets with the Java NIO mechanism, and maps keywords to the target cache nodes. The command processor provides caching invocation interfaces. The configuration manager manages the cache status and routing information. The request distributor implements the piggybacking ACKs of a data accessing protocol in the client side. The performance counter calculates completed tasks number, response time and anomalies, and then sends them to the master.

(3) Node Agent

A node agent has four components including communication manager, cache node manager, resource monitor and status reporter. The communication manager provides JMX and Restful interfaces to the master node and uses a JMX and Restful client to access the master node. The cache node manager manages running cache nodes. The resource monitor collects the resource utilization of the server, cache nodes and the node manager. The status reporter reports the local server's running status to the master node with the heartbeat mechanism.

(4) Master Node

A node agent has three components including cache manager, deployer and resource adjuster. The cache manager manages the cache cluster's physical and logical topology and maintains running status information. The deployer automatically deploys and un-deploys the cache cluster. The resource adjuster uses our proposed approach to dynamically adjust resources. The management console provides graphical user interfaces.

4.2. **Routing strategies.** The routing strategies of distributed cache systems can be categorized as: client-driven, server-driven and load balancing routing. CacheScale with client-driven routing to target nodes requires only one hop, which reduces the network overhead and reduces the response time. Furthermore, the cache server need not forward routing messages, which can improve its performance. CacheScale divides the whole hash space into several partitions with equal sizes, each of which is mapped to a server node. The CacheScale partitions data as follows.

(1) In the initiation period, each cache server S_i sets a memory utilization limit memcap$[S_i]$ and a network traffic limit netcap$[S_i]$, and then defines the workload weight of each cache node as $w_i = \text{memcap}[S_i] \times \text{netcap}[S_i]$, according to the physical configuration of servers.

(2) The whole hash space is divided into $Q >> S$ parts with an equal size, where S is the number of servers in the system.

(3) Each cache node S_i is mapped to T_i data partitions in the hash space: $T_i = w_i \times (Q/\sum w_i)$ $(1 \leq i \leq S)$, where w_i defined in step (1) is the workload weight of a cache node i.

The mapping relationship between partitions and nodes is stored in a mapping table. A hash value is calculated using a hash map function with an input key. A partition identifier from 1 to Q is mapped using a quadratic hash mapping function with the input hash value. The cache node storing the partition is queried from the partition-server mapping table with the input partition identity. The computation complexity of

locating cache nodes with two-phase hash mapping is $O(1)$. Our rebalance method for dynamic expansion changes the mapping relationship between partitions and servers. The computation complexity of synchronizing routing information by clients is $O(Q)$, where Q is the partition number.

5. Evaluation.

5.1. **Experimental environment.** We have implemented the dynamic scaling framework – CacheScale, and validated the effectiveness with a series of experiments. A workload generator uses LoadRunner to generate workloads, and a server deployed with a web server Tomcat and a web application BookStore to process the customers' requests, and the database is MySQL with a CacheScale cache cluster. The CacheScale is transparently embedded in the application as a secondary cache for Hibernate. Figure 4 shows the experimental network topology.

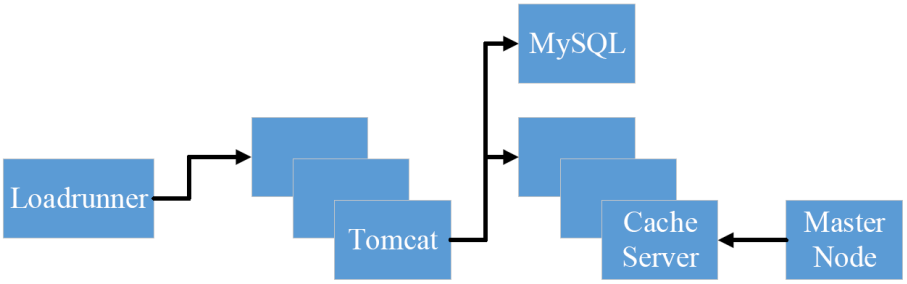


FIGURE 4. Experimental environment

5.2. **Dynamic cluster expansion.** When the distributed application starts, the distributed cache cluster includes two cache servers. After a testing period, we dynamically add two new cache nodes to expand the cache cluster. We use LoadRunner to simulate 100 concurrent customers, each transaction includes 20 requests, each request has a thinking time that is 300ms, and then we calculate the average transaction response time.

Figure 5 shows the experimental results. In the initial period, we pre-heat the BookStore application by continuously inserting data into the distributed cache cluster. The average transaction response time is falling over time. When the memory of the distributed cache cluster is fully utilized, the average transaction response time keeps stable

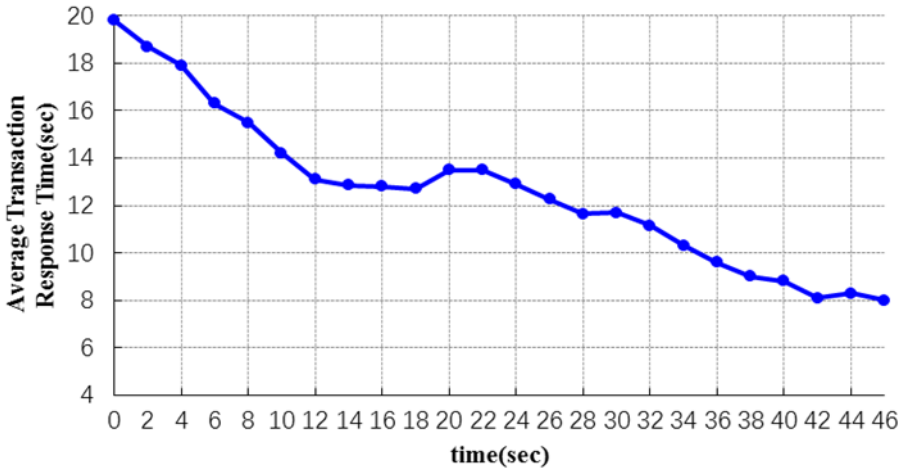


FIGURE 5. Dynamic cache cluster expansion experiment

about 12.9 seconds. At that time, we add two additional nodes in the distributed cache cluster. Since the data migration brings additional workloads to the system, the average transaction response time is slightly up about 0.6 seconds, and the cache cluster provides continuous service during the dynamic expansion period. When the expansion is completed, the average transaction response time decreases to be stable about 8.1 seconds. The experimental result demonstrates that CacheScale can realize dynamic expansion to improve the performance of an application, which guarantees the data consistency and service availability.

5.3. **Hotspot partition-based data rebalance.** We have implemented a static weighted data rebalance algorithm (SWR), which sets a weight for each node, but does not consider dynamic memory utilization and network traffic. We compare our hotspot partition-based data rebalance algorithm (HPR) with SWR with a series of experiments. In the initialization period, the distributed cache cluster consists of two cache servers. After a testing period, we dynamically add two new cache servers to expand the distributed cache cluster. We use LoadRunner to simulate 100 concurrent customers, each transaction consists of 50 requests, and each request has a thinking time that is 100ms, and then calculate the average transaction response time with SWR and DWR, respectively.

Figure 6 shows the experimental results of comparing SWR and HPR. In the initialization period, SWR and HPR have similar performance. When two new cache nodes are added dynamically, the average transaction response time rises, and then it gradually falls to a stable state. Since HPR considers both the memory utilization and network traffic and online identifies hotspot partitions, it has better balancing effect and lower average transaction response time. We also compare the imbalance of SWR and HPR to generate data partitions. The imbalance degree of SWR is 0.63%, and that of HPR is 0.31%, so HPR improves the imbalance of about 18.50%.

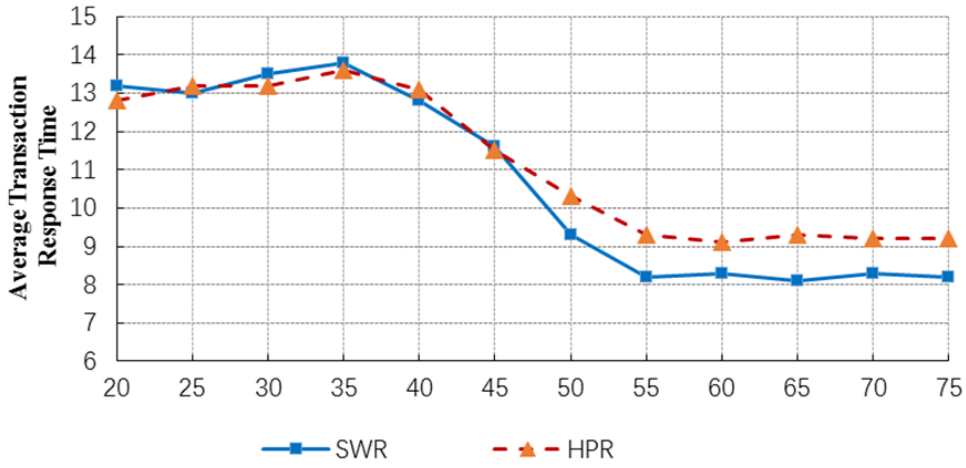
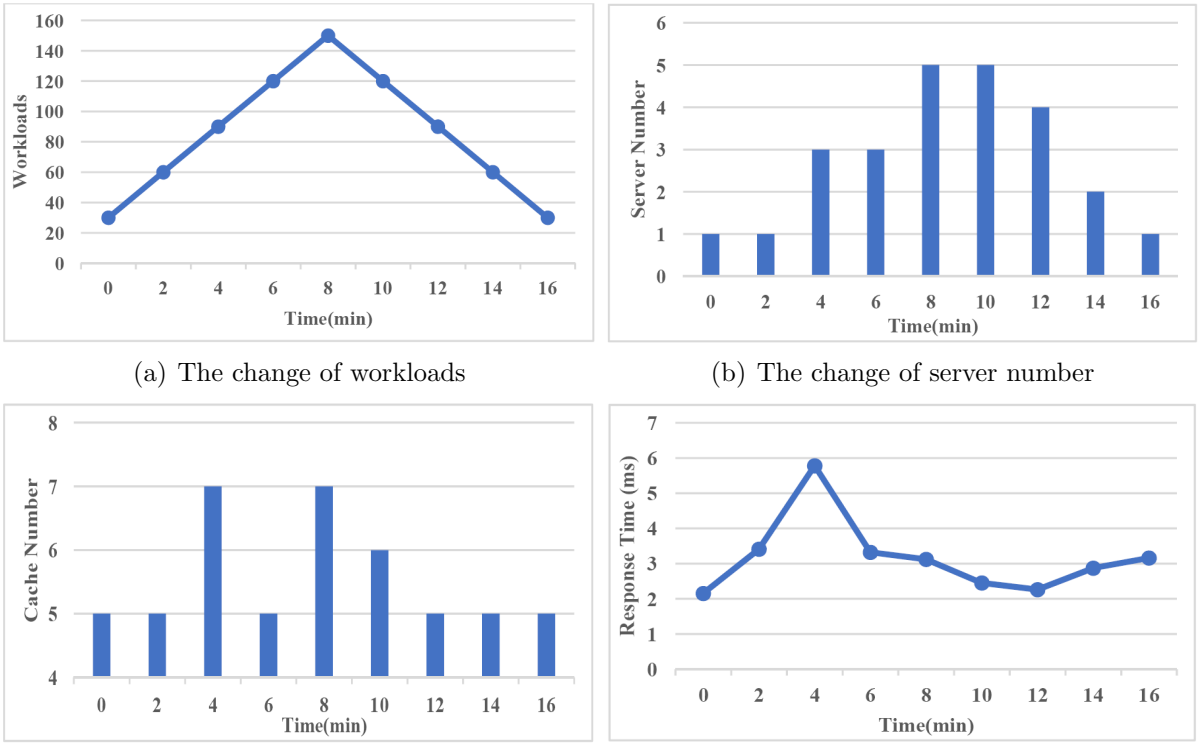


FIGURE 6. Performance comparison

The above two experimental results showed that CacheScale reduces the response time of applications and provides the data consistency and service availability during the expansion period. Furthermore, our hotspot partition-based method considering runtime resource utilization has lower response time.

5.4. **Adaptive adjustment of a cache cluster.** We deploy a caching cluster with three cache nodes in the experimental environment, and use a load generator to simulate concurrent users, where the data size of each item is 4KB. We simulate dynamic workloads



(a) The change of workloads

(b) The change of server number

(c) The change of cache node number

(d) The change of response time

FIGURE 7. Adaptive adjustment of a cache cluster

as shown in Figure 7(a), where the number of concurrent users starts with zero, steadily increases to 150, and then drops to zero. The experimental results are described as follows.

First, the working server number changes with workloads as shown in Figure 7(b). Initially, all the cache nodes are in the same server. As workloads increase, the server reaches to a performance bottleneck. The number of working servers reaches to five, and then each cache node has its own server. As workloads decrease, working servers become idle, so cache nodes are consolidated in the same server.

Second, the cache node number changes with workloads as shown in Figure 7(c). The cache node number is five in the initialization and increases during the period of consolidating cache nodes in a server or distributing cache nodes to many servers. The workflow of migrating cache nodes is creating a cache node in a new server, migrating data and shutting down the original cache node.

Third, the response time changes with workloads as shown in Figure 7(d). Although workloads fluctuate significantly, the dynamic adjustment of working servers guarantees that the performance is stable.

6. **Conclusion and Future Work.** We propose a hotspot sensitive dynamic scaling approach for distributed cache systems. To rebalance data partitions, we identify hotspot data partitions with high resource utilization, and then set high priority for the cache nodes with light workloads. To guarantee data consistency during expanding cache systems, we propose a data accessing method based on two-phase controlled data migration. We have implemented a distributed cache framework CacheScale based on the proposed methods, and conducted a series of experiments to validate the methods.

Our current work adjusts distributed cache systems by monitoring workloads, which cannot timely adapt to the changes of workloads. In the future work, we plan to predict

workloads for migrating data partitions in advance. Furthermore, our current work only can scale up the cache cluster, when workloads surge. In the future work, we plan to improve our approach by supporting to scale down distributed cache clusters to save resources, when workloads decrease.

Acknowledgment. This work is supported by the Nanjing high-end talent team introduction project in China under Grant 10072090 and National Natural Science Fund of China under Grant 61602454. The authors also gratefully acknowledge the helpful comments and suggestions of the reviewers, which have improved the manuscript.

REFERENCES

[1] L. Ding, J. Wang, Y. Sheng and L. Wang, An efficient ranking-matched caching strategy for information centric networking, *International Journal of Innovative Computing, Information and Control*, vol.14, no.2, pp.519-535, 2018.

[2] D. Chiu, A. Shetty and G. Agrawal, Elastic cloud caches for accelerating service-oriented computations, *Proc. of the ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2010.

[3] D. Chiu and G. Agrawal, Auspice: Automatic service planning in cloud/grid environments, *Computer Communications and Networks*, pp.93-121, 2011.

[4] G. Decandia, D. Hastorun, M. Jampani et al., Dynamo: Amazon's highly available key-value store, *ACM SIGOPS Operating Systems Review*, vol.41, no.6, pp.205-220, 2007.

[5] D. Karger, E. Lehman, T. Leighton, et al., Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the World Wide Web, *Proc. of the 29th Annual ACM Symposium on Theory of Computing*, NY, USA, pp.654-663, 1997.

[6] I. Stoica and R. Morris, Chord: A scalable peer-to-peer lookup protocol for Internet applications, *IEEE/ACM Trans. Networking*, vol.11, no.1, pp.17-32, 2003.

[7] F. Pfaffhauser, *Scaling a Cloud Storage System Autonomously*, Master Thesis, ETH Zuerich, 2010.

[8] Z. Wei, H. Zeng and W. Zhou, A data balance algorithm based on content sampling histogram in MapReduce, *International Journal of Innovative Computing, Information and Control*, vol.14, no.2, pp.603-614, 2018.

[9] Y. Yang, J. Ruan, J. Li, B. Liu and X. Kong, Route choice behavior model with guidance information based on fuzzy analytic hierarchy process, *International Journal of Innovative Computing, Information and Control*, vol.14, no.1, pp.363-370, 2018.

[10] B. Krishnamurthy and C. E. Wills, Piggyback server invalidation for proxy cache coherency, *Proc. of the WWW Conference*, pp.185-194, 1998.

[11] J. Xing, H. Dai and Z. Yu, A distributed multi-level model with dynamic replacement for the storage of smart edge computing, *Journal of Systems Architecture*, vol.83, pp.1-11, 2018.

[12] L. Xiong, L. Yang, Y. Tao, J. Xu and L. Zhao, Replication strategy for spatiotemporal data based on distributed caching system, *Sensors*, vol.18, no.1, pp.1-14, 2018.

[13] S. Jošilo, V. Pacifici and G. Dán, Distributed algorithms for content placement in hierarchical cache networks, *Computer Networks*, vol.125, pp.160-171, 2017.

[14] W. Chu, M. Dehghan, J. C. S. Lui, D. Towsley and Z.-L. Zhang, Joint cache resource allocation and request routing for in-network caching services, *Computer Networks*, vol.131, pp.1-14, 2018.

[15] K. Ma, B. Yang, Z. Yang and Z. Yu, Segment access-aware dynamic semantic cache in cloud computing environment, *Journal of Parallel and Distributed Computing*, vol.110, pp.42-51, 2017.