Performance Analysis of Key-Value Stores with Consistent Replica Selection Approach

Thazin Nwe(✉), Tin Tin Yee(✉), and Ei Chaw Htoon(✉)

University of Information Technology, Yangon, Myanmar
{thazin.nwe,tintinyee,eichawhtoon}@uit.edu.mm

Abstract. A key-value store is the primary architecture of data centers. Most modern data stores tend to be distributed and to enable the scaling of the replicas and data across multiple instances of commodity hardware. Defining static replica placement mechanisms in different data centers lack the efficiency of the storage system. In the proposed system, dynamic scaling that changes the key/value store with replicas dynamically joining or leaving. To enhance the dynamic scaling of the replicas, the consistent hashing mechanism is enhanced in key-value stores due to the adaptability of node changes. This mechanism performs the eventual consistency services that offer quorum key-value store with increased consistency. According to the ordering of the hash values among the replicas in the ring, it could provide higher system throughput and reduce lower latency cost without using the random of the original consistent hashing method. An experimental result overwhelms the loss of original consistent hashing algorithms entirely and is proper for the distributed key-value store.

Keywords: Key-value store · Dynamic scaling · Consistent hashing · Eventual consistency · Quorum

1 Introduction

Replica and data management has become a challenge to support different performance such as consistency, fault tolerance, and scalability of large scale data centers. Consistency model such as causal consistency, eventual consistency, etc., has been applied broadly in distributed key-value store research [19]. Eventual consistency is weak consistency and returns the latest updated data with final accesses [4]. The probability of inconsistency depends on portions such as delay of the network communication, the workload of the system and the total replicas required in the replication method. For example, the only master to handle block position is applied in the Hadoop Distributed File System and Google File System. The main server optimizes the block placement but it may not have the scalability of the storage. Large block sizes are used in GFS to reduce the amount of metadata. Consistent hashing [3] is a different generally applied method for block placements to process many small data objects. Systems similar to Dynamo [12] or FDS (Flat Datacenter Storage) [13] all implement consistent hashing. One difficulty of consistent hashing is that the hash function defines the position of

P. Sitek et al. (Eds.): ACIIDS 2020, CCIS 1178, pp. 377–387, 2020.
https://doi.org/10.1007/978-981-15-3380-8_33

the per block, and thus there is a problem in anywhere to place each block. In Apache Cassandra, the hash strategy uses with a random arrangement mechanism and defines the location of objects supported the hashing. It efficiently adjusts the load of the data within the system.

This article is constructed as follows. Section 2 performs related work. Section 3 defines the two replica placement strategies in the existing system. Section 4 represents the proposed architecture and algorithms. Section 5 outlines the implementation of the intended methods, and Section VI completes this paper.

2 Related Work

Several existing replica allocation methods for the consistency of distributed systems has been analyzed broadly and compared them in this section.

Suresh et al. presented a C3 algorithm as a replica selection method that chooses the servers according to the rank of the client with the scoring function. It could decrease the latency, but it does not work well in heterogeneous workloads [14]. Therefore, V. Jaiman et al. purposed Heron as a replica reading mechanism with the prediction of requests required important time by sustaining a record of keys according to large values [15]. This mechanism selected the replica that an incoming request more responsive than the separate replicas. Barroso et al. analyzed the problems of latency and presented a collection of tail latency tolerance methods performed in Google's massive scale systems [16]. Anti-entropy [17] method that started with the initiating replicas sending a vector of the latest local versions of all key. It optimized with Merkel or prefix trees to make comparisons faster. Experiments compared with uniform random peer selection with a greedy approach. It is efficient replication, faster visibility, and stronger eventual consistency while maintaining high availability and partition tolerance. J.P. Walters and V. Chaudhary et al. proposed to locate replica at some distant node with randomly rather than storing it near the significant place. The aim is to form replicas by node randomly, subject to specific restrictions [18].

P. Bailis et al. solved a stale read problem for eventual consistency. Probabilistically Bounded Staleness (PBS) is proposed to compute the inconsistency rate. The total replicas, replicas for a read request, replicas for the write request, timeSinceWrite, number versions, and read/ write latency are considered as an evaluation parameter. Although partial quorums were good enough for latency benefits, communicating some replicas for several requests typically decreases the guarantees on consistency [5].

Different from the above studies, the proposed system enhances the consistent hashing algorithm to improve the performance of quorum-based replication of key-value applications. By considering the location and read/write access time of replicas with adjusting the read/write consistency level [8], the proposed algorithms select the appropriate replicas for client access.

3 Background

In this section, the consistent hashing algorithm and Dynamic Snitching which is used in the proposed system is presented.

3.1 Replica Placement Strategy

This is correlated with the replica arrangement strategy for a data property. Cassandra uses the hashing way as placing randomly and decides the situation of objects with the hashing. It efficiently adjusts the amount on the system, nevertheless, it is not sufficient for transactional storage that demands co-located many data items. The method replicates a data object in the data nodes as the replicas (*R1, R2, R3*) which holds the most offers this data object. Although it lowers the transfer in the system, it is not suitable for a transactional store because the relevant data obtained with a transaction might be stored in separate areas.

3.2 Dynamic Snitching

Several Cassandra servers combine into a one-hop distributed hash table (DHT). A client will communicate either replica for the read offer. This replica then is a coordinator and inside retrieves the history of the replica holding the data. Coordinators choose the most reliable replica for the read request handling Dynamic Snitching. With Dynamic Snitching, every Cassandra replica rank and more lasting replicas by a factor of the read latencies to each of its peers, like I/O load information that each replica gives with the cluster into a gossip protocol [20].

3.3 Consistent Hashing

It is a key-value store type distributed data store technology. Servers and data are located on a circular space as a "Ring". Each datum has an ID as a point on the Ring. Each server manages data in the areas on the left side of the virtual nodes on the ring. To prevent data loss, each virtual node replicates its own data to the virtual node on the right. However, it lacks the performance of a heterogeneous environment where different storage devices such as hard disk drives. Various existing storage systems with hashing based distribution have a great role in a distributed system such as the Dynamo, DNS, P2P, Cassandra, Ceph, Sheepdon, and GlusterFS [3].

4 Architecture of Proposed System

Figure 1 presents the design of the proposed key/value store cluster. The proposed cluster model has two components of the read and writes for the client request. The requested data are distributed on different virtual nodes in the cluster as a ring. Every server has multiple virtual nodes. Loads of the servers are more balanced by "the law of massive numbers".

The procedures of the proposed system are mentioned as four steps. In the first step, the hash values of each replica are calculated with the MD5 algorithm. In the second step, the replicas are arranged in a cluster with the descending order of the PoR Algorithm. In the third step, the latency will be computed among various replicas with NR Algorithm. In the last step, the inconsistency rate is computed with CR Algorithm (Fig. 2).

And the proposed algorithms [6, 7] are used to retrieve the updated data associated with the request message. The description of algorithms is mentioned as follows.

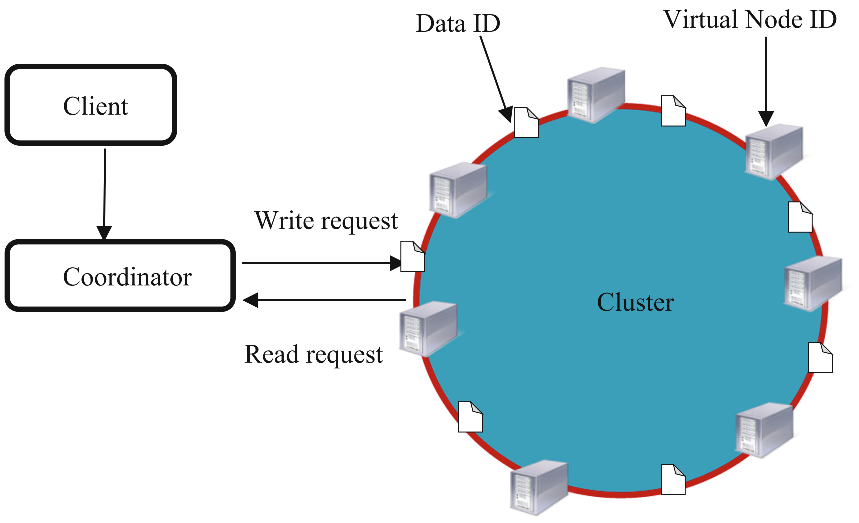

Fig. 1. The architecture of key-value store cluster

A write request is transferred to the PoR algorithm; the client picks the nearest node depends on the distance between neighboring cluster members in the ring. That nearest node is defined as a coordinator node and delivers the request to the adjacent data node. Each data node as a replica in the cluster architecture has a specific ID with the hashing an IP address to get value. Each ID is defined to a scale of a hash function as an output that is set as a circular area such as a ring. The hash values of replicas are sorted into the storage cluster. A key is defined on each data and also placed to the ring and moves clockwise around the ring from that position until finding the first node ID.

PoR Algorithm

```
Input: YCSB workloads, Data nodes
Begin
  Value = hash ($DataNode_1$_ID)
  for j ← 1 to N-1
    if hash ($DataNode_j$_ID) > Value then
        $Replica_j$ = $DataNode_j$
        Value = hash ($DataNode_j$)
    else Value= $DataNode_j$
        Ring. Append ($Replica_j$, data, timestamp, version)
    end if
  end for
End
Output: Replicas selected for data
```

This first node is responsible for the requested data. Thus, if the total request messages are larger than that of servers composing the cluster, the data keys are randomly chosen. Each node is needed to hold the equivalent amount of data to enhance the performance.

```
NR Algorithm
Input: Replicas in Ring
Begin
NearestReplica= $DataNode_j$
Value= hash (requestedID)
for j from 1◄─ N do
    if the hash (DataNode) > Value then
        Sort the replicas in descending order of hash values
        $ClosestReplica_j$.add (j, $DataNode_i$)
        Value= DataNodej
        Compute the distance of DataNodej
    else
        Value= hash (DataNodej)
    end
end for
End
Output: The nearest replicas
```

In the architecture of a read request (NR Algorithm), a client assigns a read request to one of the replicas in the ring to get the value. And then the node sends these requests to other replicas. The coordinator chooses the nearest replicas of the sorted replica lists with ascending order until finding the first node ID.

```
CR Algorithm
Input: The closest replicas
Begin
for each $NearestReplica_j$
    while (consistentReplica <= RCL)
        if (inconsistencyRate<=maxStaleReadRate)
            then consistentReplica++
            $NearestReplica_j$. add (j, $NearestReplica_i$)
        end if
end for
return $ConsistentReplica_j$
End
Output: The number of replicas of consistency
```

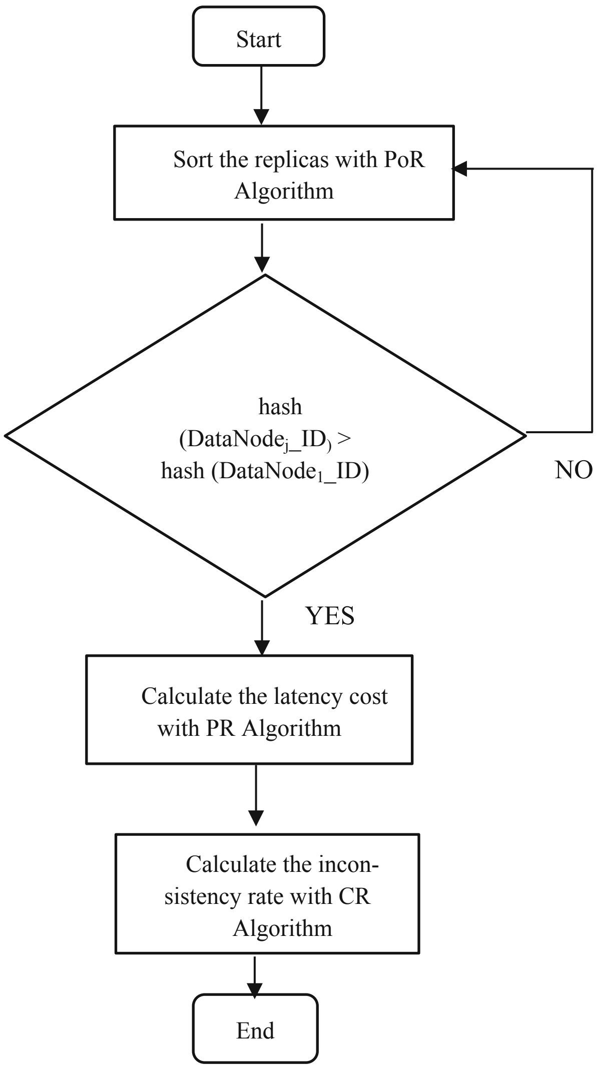
Start
Sort the replicas with PoR Algorithm
hash (DataNode$_j$_ID) > hash (DataNode$_1$_ID)
NO
YES
Calculate the latency cost with PR Algorithm
Calculate the inconsistency rate with CR Algorithm
End

Fig. 2. Proposed system flow diagram

5 Evaluation

PoR Algorithm and NR Algorithm, and CR Algorithm are executed within the Apache Cassandra cluster 3.11.4. Apache Cassandra was designed to have the advantage of multiprocessor/multi-core machines and to work across many of these machines in various data centers. Especially it has been given to achieve the well supporting of the massive load. It consistently can show fast throughput for writes per second on basic commodity computers, whether physical hardware or virtual machines. It does a gossip protocol that implements each node to get the status of all the other replicas in which the nodes are not available. When the gossip shows the coordinator that the node has obtained, it gives the lost data. The Yahoo Cloud Serving Benchmark (YCSB) [1], a standardized benchmark, is applied to estimate the execution of these algorithms. YCSB generates different workloads such as read-heavy, read-only, and update-heavy. Seven servers are managed that provide an Intel ® Core™ i7-6700 CPU @3.40 GHz 3.41 GHz. The quorum consistency level is considered to achieve strong consistency.

5.1 Experimental Result

The set of measures proves the latency cost, the throughput, the probability of inconsistency rate, and the replicas to retrieve the consistent transaction.

Figure 3 indicates the 99th percentile latency among different workloads. The effects of latency are analyzed with varying different workloads. Each measurement includes ten million operations of the workload. In particular, the 99th percentile latencies are recorded wherein the difference between various workloads are up to five times.

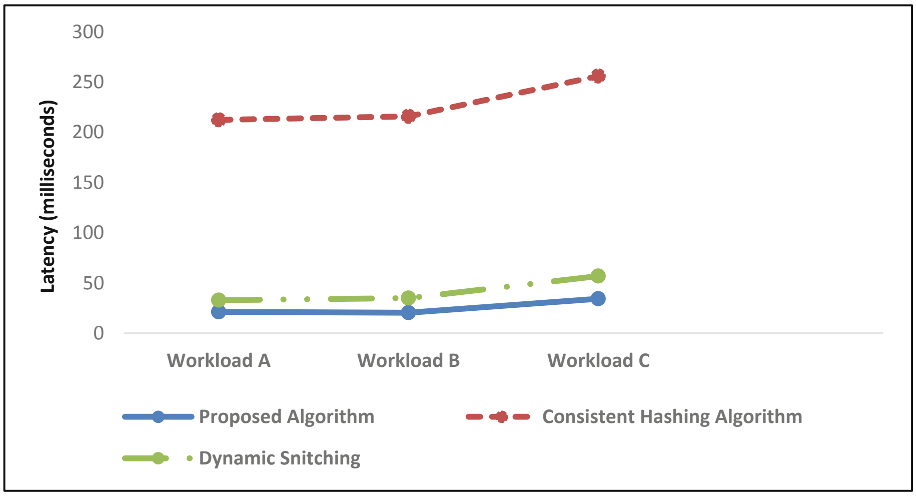

Fig. 3. Latency cost of PoR Algorithm and NR Algorithm

According to Fig. 3, the latency of the PoR Algorithm and NR Algorithm is lower than a consistent hashing algorithm in every different workload. As a reason, when a node joins a cluster with the consistent hashing method, it must perform to locate replicas randomly. To compare the consistent hashing algorithm and proposed an algorithm, the average distance between the node issuing a request and the replica serving it is computed. This increases the replicas, which makes it impossible to reduce the access time because it has to contact the replicas in the long distance. Thus the average distance is less than that allowed by the consistent hashing algorithm according to the descending order of hash values and total round trip time. This proposed replica placement strategy has an impact on the latency cost as well as in reducing resource cost. The proposed PoR

Algorithm and NR Algorithm are a broadly applied algorithm and a suitable method to the distributed key-value stores for load balancing and high reliability.

Figure 3 designates the read latency components of Cassandra crossed several workloads when using the proposed algorithms matched to Dynamic Snitching (DS). Regardless of the workload used, the proposed algorithms improve the latency across all the considered metrics, namely, 99th percentile latencies. With the read-heavy workload, the 99th percentile latency is 21 ms with the proposed algorithms, whereas, with DS, it is 33 ms. In the update heavy and read-only scenarios, the proposed algorithms improve the same difference by a factor of 20.4 ms and 30.4 ms. The proposed algorithms also improve the latency by between 12 ms and 15 ms, and 23 ms across all scenarios.

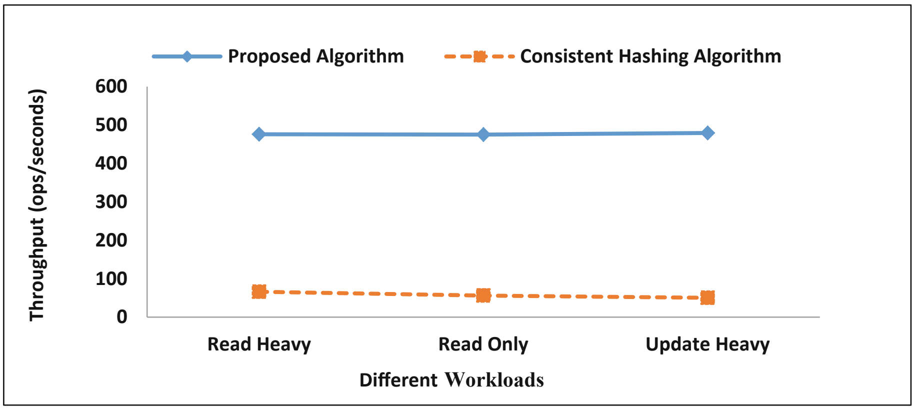

Fig. 4. Throughput of PoR Algorithm and NR Algorithm

In Fig. 4, the client uses a replica election approach to track the requests for a quorum of replicas. A request created at a client has an identical distribution delivered to other replicas. In particular, we recorded throughput between the proposed algorithm and the consistent hashing algorithm is up to five times. The proposed algorithm offers more useful the possible system space, appearing in an improvement in throughput across these workloads. In particular, the proposed algorithm enhances the throughput by roughly 75% across the considered workloads. This proposed replica placement strategy has an impact on the throughput in key-value data stores as the workload varies dynamically.

In the experiment of Fig. 5, 500 million 1 KB size records created by YCSB running in separate VMs are inserted, and the PoR Algorithm and NR Algorithm are analyzed for the impact of strong consistency for various workloads [9].

The read and update heavy workloads in distinct widespread over a description of Cassandra deployments [10, 11]. The staleness rate depends on increasing the consistency level or the read and write requests. The raised to read and write consistency level has a great possibility of an inconsistency rate because the further expecting time is required to obtain the read/write request. Nevertheless, the result proves that the inconsistency rate is not dramatically raising even the number of replicas and replicas for the read and write requests is large.

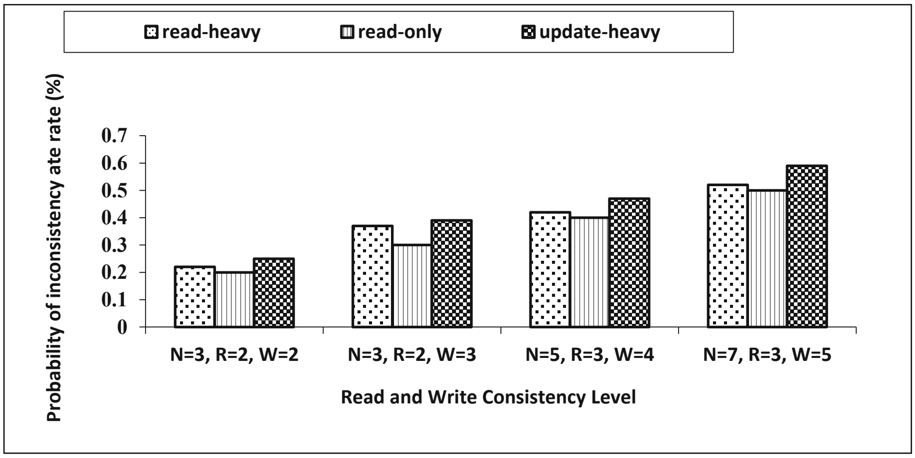

Fig. 5. Probability of inconsistency rate on read and write consistency level

From Fig. 6, CR Algorithm takes the fewest number of replicas for consistent data of each request related to a consistent hashing algorithm for quorum replication. NR Algorithm gets the closest replicas. Thus, the system does not require to obtain additional replicas for the choosing of consistent replicas.

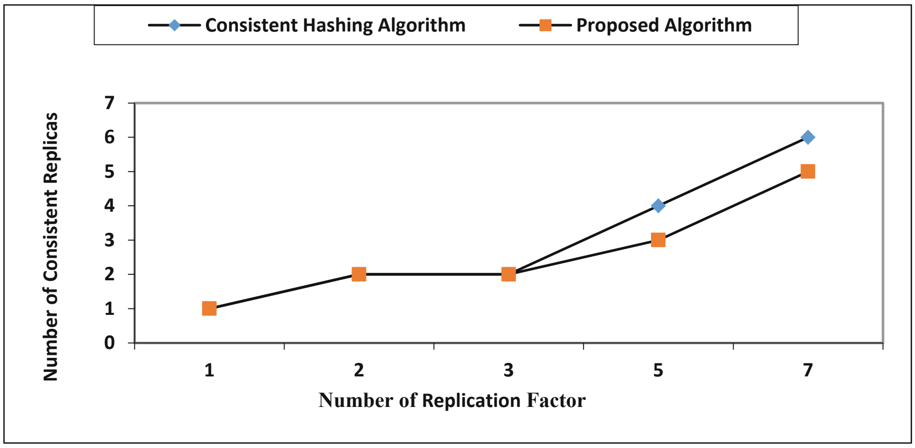

Fig. 6. Consistent replicas of CR Algorithm

The total consistent replicas of the CR algorithm are smaller than the consistent hashing algorithm when the replication factor is five and seven. Each node is needed to hold the equivalent amount of data to enhance the performance. When nodes randomly join the ring, each node is responsible for at most. This proposed idea can be reduced by holding each node run replicas in a cluster model.

6 Conclusion

In this research, we tend to address the issues of the static replication and intended the PoR Algorithm, NR Algorithm, and CR Algorithm for selecting the replica of consistency data in distributed key-value stores. The intended algorithms offer strong consistency which proposes the execution of the key-value stores with replica selection. The result is enhanced the overall achievement of the key-value store for a large-scale variance of different workloads. These algorithms compute the latency cost, throughput, inconsistency rate and choose the appropriate consistency level to take the least number of consistent replicas for increasing the read and write execution time in real-time. The tests prove that the purposed algorithms are adapted to define the minimum number of consistent replicas of key-value stores with the quorum replication.

As future work, YCSB workload D and workload E will be applied in the performance of the PoR Algorithm, NR Algorithm, and CR Algorithm. And, the throughput of the PoR Algorithm and NR Algorithm will be compared with the Dynamic Snitching algorithm.

Acknowledgement. I would also like to express my special thanks to Professor Junya Nakamura, Information and Media Center, the Toyohashi University, Japan for his valuable suggestions and guidelines.

References

1. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: Proceedings of the 1st ACM Symposium on Cloud computing, New York, pp. 143–154 (2010)
2. Malkhi, D., Reiter, M.K.: Byzantine quorum systems. In: Proceedings of the 29th Annual ACM Symposium on the Theory of Computing, STOC 1997, pp. 569–578, May 1997
3. Karger, D., Lehman, E., Leighton, T., Panigraphy, R., Levine, M., Lewin, D.: Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the world wide web. STOC 1997 Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pp. 654–663 (1997)
4. Diogo, M., Cabral, B., Bernardino, J.: Consistency models of NoSQL databases. Future Internet **11** (2019). https://doi.org/10.3390/fi11020043
5. Bailis, P., Venkataraman, S., Franklin, M.J., Hellerstein, J.M., Stoica, I.: Probabilistically bounded staleness for practical partial quorums. J. Proc. VLDB Endowment **5**(8), 776–787 (2012). https://doi.org/10.14778/2212351.2212359
6. Nwe, T., Nakamura, J., Yee, T.T., Phyu, M.P., Htoon, E.C.: Automatic adjustment of read consistency level of distributed key-value storage by a replica selection approach. In: The 1st International Conference on Advanced Information Technologies, Yangon, Myanmar, pp. 151–156, November 2017
7. Nwe, T., Yee, T.T., Htoon, E.C.: Improving read/write performance for key-value storage system by automatic adjustment of consistency level. In: The 33rd International Technical Conference on Circuits/Systems, Computers and Communications, Bangkok, Thailand, pp. 357–360, July 2018
8. Nwe, T., Nakamura, J., Yee, T.T., Htoon, E.C.: Automatic adjustment of consistency level by predicting staleness rate for distributed key-value storage system. In: Proceedings of the 2nd International Conference on Advanced Information Technologies, Yangon, Myanmar, pp. 27–32, November 2018

9. Nwe, T., Nakamura, J., Yee, T.T., Htoon, E.C.: A consistent replica selection approach for distributed key-value storage. In: Proceedings of 2019 International Conference on Advanced Information Technologies, Yangon, Myanmar, pp. 114–119, November 2019
10. Rahman, M.R., Golab, W., AuYoung, A., Keeton, K., Wylie, J.J.: Toward a principled framework for benchmarking consistency. In: Proceedings of the Eighth USENIX Conference on Hot Topics in System Dependability, ser. HotDep 2012. USENIX Association, Berkeley, p. 8 (2012)
11. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
12. DeCandia, G., et al.: Dynamo: amazon's highly available keyvalue store. In: ACM SIGOPS Operating Systems Review, vol. 41, pp. 205–220. ACM (2007)
13. Nightingale, E.B., Elson, J., Fan, J., Hofmann, O.S., Howell, J., Suzue, Y.: Flat datacenter storage. In: OSDI, pp. 1–15 (2012)
14. Suresh, L., Canini, M., Schmid, S., Feldmann, A.: C3: cutting tail latency in cloud data stores via adaptive replica selection. In: 12th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2015, 4–6 May 2015, Oakland, CA, USA (2015)
15. Jaiman, V., Mokhtar, S.B., Quema, V., Chen, L.Y., Riviere, E.: He'ron: taming tail latencies in key-value stores under heterogeneous workloads. In: 2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS), pp. 191–200 (2018)
16. Dean, J., Barroso, L.A.: The tail at scale. Commun. ACM **56**(2), 74–80 (2013)
17. Bengfort, B., Xirogiannopoulos, K., Keleher, P.: Anti-entropy bandits for geo-replicated consistency. In: 2018 IEEE 38th International Conference on Distributed Computing Systems (2018)
18. Walters, J.P., Chaudhary, V.: Replication-based fault tolerance for MPI applications. IEEE Trans. Parallel Distrib. Syst. **20**(7), 997–1010 (2009)
19. Davoudian, A., Chen, L., Liu, M.: A survey on NoSQL stores. ACM Comput. Surv. (CSUR) **51**, 1–43 (2018)
20. Williams, B.: Dynamic snitching in Cassandra: past, present, and future (2012). http://www.datastax.com/dev/blog/dynamic-snitching-in-cassandra-past-present-and-future