Rethink the Linearizability Constraints of Raft for Distributed Systems

Yangyang Wang, Zikai Wang, Yunpeng Chai, and Xin Wang, *Member, IEEE*

Abstract—With the deployment of modern hardware such as Flash-based SSDs and the high-speed network in distributed systems, the distributed consensus and consistency module (e.g., Raft) is typically the most time-consuming part. The reason lies in that Raft introduces some very strict constraints to ensure the linearizability. Therefore, in this paper, we rethink these constraints in-depth and find that some of them are not necessary, and can be broken to accelerate the performance significantly without breaking the linear consistency for distributed systems. An improved distributed consensus algorithm called *BUC-Raft* (Breaking Unnecessary Constraints of Raft) is proposed in this paper and implemented in an industry-level distributed system. The experimental results suggest that both the write and the read performance can be accelerated significantly by BUC-Raft.

Index Terms—Consensus, distributed system, linearizability, Raft

1 Introduction

In recent years, many enterprise-level distributed systems are coupled with the high-end modern hardware, such as Flash-based Solid State Drives (SSDs) and the high-speed network, to promote the system performance as much as possible to meet the requirements of various Big Data applications. Meanwhile, consensus is a key issue in distributed systems. For example, if multiple servers store account balances, it is necessary to keep the data of these servers consistent, otherwise errors will occur.

However, as one of the most important and the most complex modules in a distributed system, the distributed consensus protocols (e.g., Paxos [1], [2] and Raft [3], [4]) are not easy to be accelerated with hardware upgrades. There are many strict rules and many steps of data processing (e.g., RPCs, logs, data persistence, etc.) to ensure the consensus and the consistency of a distributed system even under all kinds of system fault cases. For modern hardware-based distributed systems, these constraints imposed by the consensus protocols have been a major performance challenge.

Therefore, we have made a thorough and in-depth discussion about the constraints introduced by Raft, which is widely used in many practical distributed systems. We found that not all the constraints are necessary, and we also had the observation that some of the constraints will cause serious performance problems for both processing read and write requests in distributed systems. For example, waiting for the accomplishment of the *apply* operations accounts for a high percentage of write processing, especially under high-speed networks; and waiting for the state machine to catch up with the required data version significantly slows down the read performance. But in distributed systems, some of these limitations can be broken to promote the performance.

In this paper, we propose an enhanced distributed consensus algorithm, i.e., BUC-Raft. In order to accelerate the processing of requests, BUC-Raft has different processing for different types of requests, and BUC-Raft introduces two mechanisms called *Commit Return* (CR) and *Read Acceleration* (RA), respectively, to break the above constraints introduced by Raft. In addition, BUC-Raft introduces two new concepts, i.e., *Semantic-influencing Request* and *Consistent Request*, to ensure the correctness of the request.

We have implemented our proposed BUC-Raft in an enterprise-level distributed system and carried out the evaluations driven by the popular YCSB benchmark.The experimental results indicate that BUC-Raft can boost the system comprehensively, no matter for the read or write performance, and also regardless of the throughput or the latency. For example, compared with Raft, BUC-Raft can achieve 62%, 53.6%, and 18.4% higher throughput for the typical write-only (i.e., *Load*), read/write-balanced (i.e., *Workload A*, 50% writes), and read-dominant (i.e., *Workload B*, 95% reads) scenarios, respectively. When there are semantic-influencing requests in write requests, BUC-Raft can also improve the throughput by 31.9% on average.

The contributions of this paper include: 1) We find that some of the constraints introduced by Raft to ensure the linear consistency based on Paxos are not necessary for distributed systems; some of the constraints have become the bottleneck of performance in modern hardware-based distributed systems.

- *Yangyang Wang, Zikai Wang, and Yunpeng Chai are with the Key Laboratory of Data Engineering and Knowledge Engineering, MOE, and School of Information, Renmin University of China, Beijing 100872, China. E-mail: {yangyangwang, zkwang, ypchai}@ruc.edu.cn.*
- *Xin Wang is with the College of Intelligence and Computing, Tianjin University, Tianjin 300072, China. E-mail: wangx@tju.edu.cn.*

Manuscript received 8 February 2022; revised 24 November 2022; accepted 5 January 2023. Date of publication 9 January 2023; date of current version 6 October 2023.
This work was supported in part by the National Key Research and Development Program of China under Grant 2019YFE0198600, in part by the National Natural Science Foundation of China under Grants 61972402, 61972275, and 61732014.
(Corresponding author: Yunpeng Chai.)
Recommended for acceptance by F. Rusu.
Digital Object Identifier no. 10.1109/TKDE.2023.3235399

1041-4347 © 2023 IEEE. Personal use is permitted, but republication/redistribution requires IEEE permission.
See https://www.ieee.org/publications/rights/index.html for more information.
Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on March 20,2025 at 14:10:55 UTC from IEEE Xplore. Restrictions apply.

2) We propose an enhanced distributed consensus algorithm, i.e., BUC-Raft. BUC-Raft can boost the system performance (including both throughput promotion and latency reduction) through different processing for different types of requests and the two new mechanisms, i.e., *Commit Return* and *Read Acceleration*.

3) We introduce two new concepts, i.e., *Semantic-influencing Request* and *Consistent Request*, to ensure the correctness of the request.

4) BUC-Raft has been implemented in an enterprise-level distributed system and we can observe a great performance advantage of BUC-Raft via a series of YCSB-driven experiments. The experimental results show that BUC-Raft is of good practical value to promote the performance of distributed systems comprehensively.

The rest of the paper is organized as follows. Section 2 introduces the background of Raft, consensus, and linearizability. Then we conduct a detailed review and some discussions about breaking Raft's limitations in Section 3. In Section 4, we elaborate on our proposed BUC-Raft in detail, and the following Section 5 describes in detail the situation of BUC-Raft in KV systems. Section 6 presents the system implementation and the evaluations, accompanied by the related work outlined in Section 7. Finally, we conclude this paper in Section 8.

2 Constraints of Raft to Achieve Linearizability

In this section, we first introduce some basic concepts (i.e., *distributed consensus* and *linearizability*) in Section 2.1, and then present Paxos in Section 2.2. Finally, the additional mechanisms added by Raft to guarantee the linear consistency are elaborated in Section 2.3.

2.1 Consensus and Linearizability

Definition 2.1. (Distributed Consensus) *A fundamental issue in distributed systems is to agree on a certain data value by the multiple replicas in the case of system failures.*

Definition 2.2. (Linearizability) *Linearizability is a strong condition of consistency, which limits what outputs are possible when an object is accessed by multiple processes concurrently. In a linearizable system, although operations may overlap on a shared object, each operation appears to take place instantaneously [5].*

Without a distributed consensus, the distributed system is unreliable. The most widely used distributed consensus algorithms are Paxos [1], [2] proposed by Lamport and its variants [3], [6].

Linearizability is also a very important feature for practical distributed data management or storage systems, in addition to distributed consensus, Linearizability is also known as *Strong Consistency* or *Linear Consistency*. It implies that theoretically, there is actually an implicit global order of all the operations. We can reason the results of a distributed system with linearizability by discovering all or part of the implicit global order of operations and will not find conflicting.

The widely used consensus algorithms in practical distributed systems (e.g., Raft [3], [4]) generally also provide the guarantee of linearizability, i.e., these systems not only preserve the consistent data among all the replicas, but also ensure the system outputs are still correct even when partial system failures occur. In addition, Raft does not handle Byzantine failure cases, and work on Byzantine failures can be found in these references [7], [8], [9].

2.2 Paxos

Paxos [1], [2] is used to make an agreement on a single value among distributed replica nodes. There are three types of roles for the nodes in Paxos, namely *proposer*, *acceptor*, and *learner*. One or more proposers suggest the value; one or more acceptors decide the chosen value together and the decided value will no longer change; the learners then will obtain the chosen value afterwards.

Two stages are included in the Paxos procedure, i.e., *Prepare* and *Accept* to achieve the agreement. The *Prepare* phase finds out about any chosen values and blocks older proposals that have not yet been completed; the *Accept* phase asks acceptors to approve a particular value.

Paxos was strictly proved to be correct in all kinds of partial failure cases, and the other distributed consensus algorithms (e.g., ZAB [6] and Raft [3]) are some variants of Paxos with some modifications.

2.3 Constraints Introduced by Raft

Since Paxos is difficult to fully understand and implement, Ongaro et al. [3] proposed Replicated And Fault Tolerance (Raft), which is easy to be comprehended and realized. Raft has been widely adopted by many practical distributed systems like Etcd [10], CockroachDB [11], TiDB [12], and PolarFS [29] since it was proposed in 2014. Compared with Paxos, Raft has primarily introduced the following three aspects of constraints based on Paxos:

2.3.1 Single Proposer

When the number of *proposers* is greater than one, the first *Prepare* step has to be done by Paxos to coordinate and fix one proposal. In this step, Remote Procedure Calls (RPCs) are required and are time-consuming. Raft reduces the number of *proposers* to one, like ZAB, another variant of Paxos, thereby eliminating the *Prepare* stage to save time. The only *proposer* in Raft often serves as one of the *Acceptors* and is often referred to as *Leader*; the other *Acceptors* are often called *Followers*.

In order to tolerate the node or network failure for a single *proposer*, i.e., *Leader*, Raft contains a strict leader-election mechanism to ensure that only one leader is elected. There are additional RPCs needed for the leader election which is not a frequent operation. Although the only leader usually undertakes the most work and poses a challenge for load balance, the single-leader replica mechanism is simple and efficient, i.e., less RPCs and faster data processing procedures.

2.3.2 Read Support

Paxos is an abstract consensus algorithm and does not consider many requirements of practical distributed systems. For example, Paxos determines only one value for the data with multiple replicas, but it is not responsible for how to read the stored data. Raft is built to fulfill the functional

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on March 20,2025 at 14:10:55 UTC from IEEE Xplore. Restrictions apply.

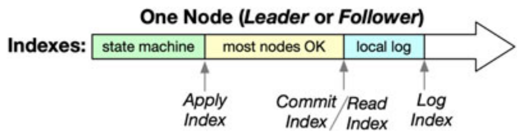


Fig. 1. The indexes used in Raft.

requirements except for the fundamental consensus function inherited from Paxos.

Specifically, Raft splits the data status into two parts, i.e., the *log* and the *state machine*, which is different for diverse applications (e.g., key-value stores, databases, file systems, etc.). The append-only log is efficient for synchronization among the nodes, but it is usually inefficient to read data from it due to lack of order. So the state machine can supply a much faster read speed. For example, key-value stores utilize HashTable, B+Tree, or LSM-Tree to keep the stored key-value pairs sorted and easy to read.

2.3.3 Linear Consistency

Since reading is supported by Raft, and concurrent read/write requests processing are common in practical systems, it is important to ensure that users can always get the correct results (i.e., achieving the linear consistency). In order to support linearizability, Raft introduces these restrictions: 1) The logs have to be processed in sequence; 2) Only when the state machine has contained the latest successfully committed data of the time point that the read request just arrives at the system, the read request can be served; otherwise, it must wait; 3) Only the node with all the committed logs can be elected as the leader. ZAB [6] guarantees linear consistency by treating read requests like write requests, but makes processing read requests very slow.

These mechanisms of Raft ensure the linear consistency, but also slow down the request processing and the system performance. In the following section, we will give an in-depth analysis about the constraints of Raft, to see if some of them can be eliminated or be relaxed to achieve higher performance for distributed systems.

3 Breaking Unnecessary Constraints of Raft

In order to promote the performance of distributed systems by breaking some unnecessary constraints of Raft, in this section, we first introduce the detailed process of Raft in Section 3.1, and then discuss whether all the linearizability constraints of Raft is necessary in Section 3.2.

3.1 Process of Raft

3.1.1 Request Processing of Raft

Raft's processing of requests involves three key operations, i.e., *Append*, *Commit*, and *Apply*, and therefore introduces a series of *indexes* for the management of the request processing to ensure the linearizability [5]. Plotted as Fig. 1, the introduced indexes include *log index*, *commit index*, and *apply index*.

1) *Append*. All requests of client will be sent to the Raft leader. When the leader receives a request, the request will be assigned a unique and increasing version number, such

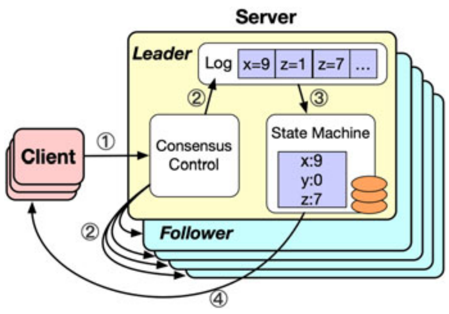


Fig. 2. An example of the request processing of Raft.

as index I. Then the leader *appends* the contents of the request into the local persistent log, at the same time it sends the contents to all the followers. When any of the followers receives the contents from the leader, it will also *append* the content into its local persistent log, and then notify the leader after its appending procedure succeeds. No matter when the leader or the followers have finished local log appending work, their *log index* will be increased to the latest requests' index.

2) *Commit*. By receiving responses from followers, when the leader finds out that more than half of the replicas have successfully appended the request with index I, the request is set to the *committed* status, i.e., the request is assur-ed to be safe in the system as long as no more than half the nodes crash. Accordingly, the *commit index* is updated to I.

3) *Apply*. After the request with index I is committed, the leader and all the followers start to *apply* this log into the state machine. When the applying work is finished on one replica node, the node will update its *apply index* to I. Only after the leader's *apply* operation succeeds, the leader can return the result of the request to the client. According to Raft, the log appending should be performed sequentially following the requests' arrival orders (i.e., their *indexes*); so do the log committing and the data applying.

Example 3.1. Raft's request processing. As Fig. 2 illustrates, a request from a client first comes to the consensus control (CC) module of the leader (i.e., *Step 1*). CC begins the local log-appending of the leader and synchronizes the contents to all the followers in parallel (i.e., *Step 2*). The leader needs to apply the log into the state machine after the log is committed (i.e., *Step 3*). Finally, the leader responds to the clients to accomplish the request (i.e., *Step 4*).

3.1.2 Processing Read-Only Requests More Efficiently

Read-only requests do not change the state machine, so they can bypass the Raft log to get better performance. In order to ensure linearizability [5], Raft introduces the *read index*. When the leader receives a read request, the request's *read index* is set to the current *commit index*. Only when the leader's *apply index* is no less than the *read index*, the leader can execute the read request and return the result to the client. In this case, we can ensure that the client would not get the out-of-date data [4].[1]

1. This optimization is not described in detail in the original paper [3], but is in the doctoral thesis [4] of the author.

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on March 20,2025 at 14:10:55 UTC from IEEE Xplore. Restrictions apply.

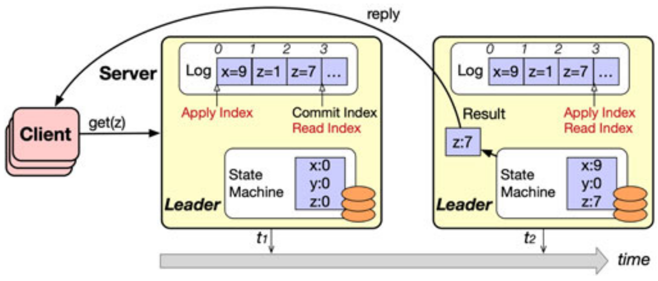


Fig. 3. Processing read-only requests.

Example 3.2. Processing read-only requests. As Fig. 3 shows, we assume the *read index* of *GET(z)* is 3, while the current *apply index* of the leader is only 0 at time t_1. In this case, the leader has to apply the written data with the indexes $1 \sim 3$ into the state machine first before it is allowed to process *GET(z)* (i.e., time t_2), in order to ensure the linear consistency, i.e., getting the right result "$z = 7$," but not "$z = 0$".

3.2 Breaking Constraints of Raft

3.2.1 Breaking Constraints of Request Processing

Except for read-only requests, among all the request processing steps of Raft introduced above, 1) the RPCs among the leader, the followers, and the client, 2) the log appending, and 3) the state machine applying may be the most time-consuming parts. For a system coupled with a high-speed network, the network overhead is low. And the structure of the log is very simple and the log-appending is usually a sequential I/O operation with higher speed. Thus updating the complicated state machine is the key performance bottleneck.

Consequently, we have conducted an evaluation of the percents of the time consumption for all the steps of Raft request processing in a distributed key-value store (see Section 6.2 for more about the experimental environments).

Fig. 4 exhibits the consumed time percentage for Raft request processing when the value size ranges from 1 KB to 1 MB. It is obvious that *Apply* is actually slow and accounts for almost 40% of all the time of request processing when the value size is less than 16 KB, which is common for most applications. *Apply* is slow because the apply operation involves writing the state machine log to disk, and changing the state machine. Along with the increase of value sizes, the proportion of the network transmission time increases compared with the *apply* phase. If the network has a higher speed, the time consumption proportion of the *Apply* phase will be increased further.

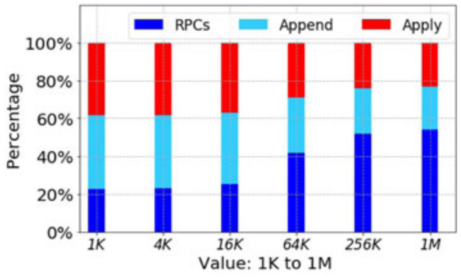


Fig. 4. Time-consuming percent for Raft request processing.

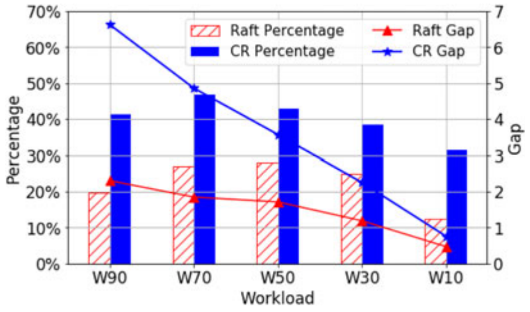


Fig. 5. The gap between the *apply index* and the *read index* and the percentage of the waiting time.

Therefore, if we can remove the applying phase from the shortest path of the Raft request processing, the consumed time will be much saved. For write-only requests, they only need to know whether the write was successful. So theoretically we can respond to the client exactly when the log has been committed (i.e., more than half nodes have stored the log of the written data in persistent storage) to reduce the response time of writing, but not have to wait for the accomplishment of the state machine applying. This mechanism can be called *commit return*, and it can boost the write operation significantly. *Commit return* will not break the correctness and linearizability. For example, in KV stores, even if a *GET* operation for one key immediately follows a *PUT* operation for the same key, the worst case is that we should wait for the finish of this key's applying before the execution of the *GET* request. The *GET* can always get the latest value. In summary, *commit return* can always accelerate the write processing, but it may increase the waiting time of read requests.

3.2.2 Breaking Constraints Under Read-Only Processing

In order to evaluate the gap between the *apply index* and the *read index* and the percentage of the waiting time caused by the gap in the read request time, we have made some experiments to detect the gap and the percentage in a typical distributed KV store driven by YCSB with different write request percents (i.e., 90% ∼ 10%).

From the results exhibited in Fig. 5, we can get the following observations: 1) The percentage of the waiting time is basically above 20%. 2) For a workload with a larger write percentage, the gap is larger because more write requests have not been processed in time when a read request arrives. 3) For the *Commit Return* (CR) strategy discussed above, the gap between the two indexes is much larger than the original Raft, confirming the above conclusion that CR can boost write processing but may slow down the read processing. If we can remove the waiting time caused by the gap between the *apply index* and the *read index*, the read performance will be significantly improved.

4 Design of BUC-Raft

Motivated by the above observations about the unnecessary constraints of Raft in the last section, we propose an improved distributed consensus algorithm based on Raft for distributed systems, i.e., BUC-Raft.

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on March 20,2025 at 14:10:55 UTC from IEEE Xplore. Restrictions apply.

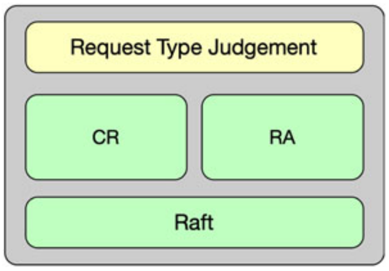


Fig. 6. The architecture of BUC-Raft.

4.1 Overview

In order to break the unnecessary constraints of Raft, BUC-Raft proposed the architecture shown in Fig. 6. The request type judgment determines the request process. For different request types, BUC-Raft has different processing. For write-only requests, follow the *Commit Return* (CR) process, which will be introduced in Section 4.2. For read-only requests, follow the *Read Acceleration* (RA) process, which will be introduced in Section 4.3. For read-write mixed requests, there are two types. The first type does not require specific return results, but only needs to ensure that the request can be executed correctly, such as request *a*++; the second type requires specific return results, such as request *return ++a*. For the first type, follow the *Commit Return* process, and for the second type follow the original Raft process. In fact, for the second type of request, the client can first try to split the request, split into the first type of request + read-only request, such as request *return ++a* can be split into request ++*a* and request *return a*, and then follow the CR process and the RA process. If it cannot be split, send it to the leader to follow the original Raft process.

4.2 Commit Return

Recall Section 3.2.1 that the apply operation of the state machine usually takes a relatively long time during the Raft processing. According to Raft, the result can be returned to the client only after the apply operation is completed. However, for write-only requests and the first type of read-write mixed request, they do not require specific return results, but only need to know whether the request can be executed successfully, so we can respond to the client immediately when the request is *committed*.

Because the commit completion requires more than half of the replica nodes have successfully appended the log, the written data is safe in the distributed systems when it is in the *committed* status. Finishing the apply operation is not necessary for writing, and thus it can be performed asynchronously in the background.

In fact, *Commit Return* transfers the applying overhead from the write request processing to the read processing. CR reduces the write latencies but also lengthens the time-consumption of the read request processing. However, its side effects on reading will be solved by the following *Read Acceleration* mechanism (see Section 4.3 for more).

Example 4.1. Commit Return. As Fig. 7 plots, compared with the original procedure of Raft shown in *Example 4.1*, *Commit Return* returns the result to the client in advance when the commit operation is completed, reducing the step count from 4 to 3. The apply operation is performed asynchronously in the background.

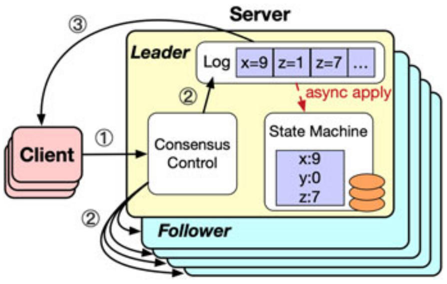


Fig. 7. An example of the commit return mechanism in BUC-Raft.

4.3 Read Acceleration

In Raft, the read operation waits for the apply index to catch up with the read index before performing the read operation; especially when *Commit Return* is implemented, the gap between the apply index and the read index will get larger, resulting in longer waiting time. If we can reduce the waiting time, the read performance will be improved. Therefore, we proposed *Read Acceleration* (RA).

Read Acceleration consists of two parts: *Immediate Read* (IM) and *Relaxed Read* (RR). IM is faster than RR, but IM is not suitable for all situations. For example, when there is a semantic-influencing request and there is no consistent request, an error will occur when using IM. In addition, when there is a semantic-influencing request and there is a consistent request, the cost of IM may increase, and the cost of IM and RR needs to be compared at this time. The semantic-influencing request and the consistent request are introduced in Sections 4.3.3 and 4.3.4, respectively.

The process of *Read Acceleration* is shown in Fig. 8. When the leader receives a read request, it traverses the Raft logs between the apply index and the read index from front to back, and takes out the Raft logs that may be related to the read request. For these related logs, if there is no semantic-influencing request of the read request, go through the *Immediate Read* (IM) process; if it exists, judge whether the read request can be converted into a consistent request. If the conversion is not possible, follow the *Relaxed Read* (RR) process; if it can, then judge the lower cost of IM and RR, and follow the lower cost process.

4.3.1 Immediate Read

The process of *Immediate Read* (IM) is shown in Fig. 9. For the read request, we do not need to wait for the apply index to catch up with the read index, we can execute the read request in the state machine immediately. Then use the

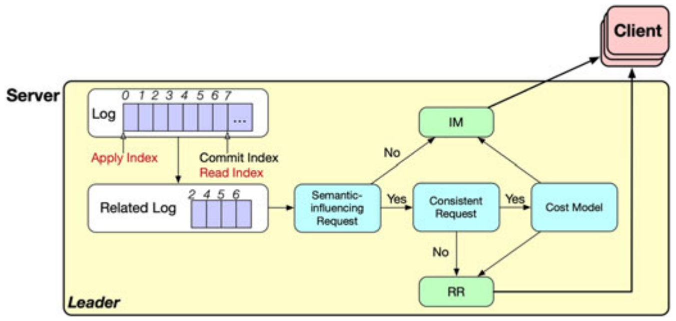


Fig. 8. The process of Read Acceleration in BUC-Raft.

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on March 20,2025 at 14:10:55 UTC from IEEE Xplore. Restrictions apply.

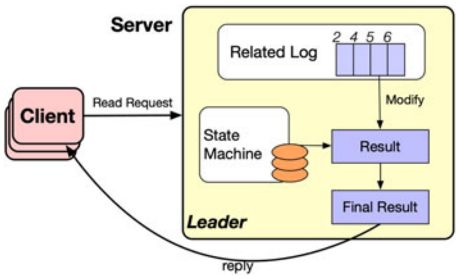


Fig. 9. The process of Immediate Read in BUC-Raft.

above related logs to modify the result of the state machine one by one in the order of log index, and get the final result. For *Immediate Read*, different systems can make some changes according to the read request semantics to obtain better performance. For details, please refer to the KV system in Section 4.

4.3.2 Relaxed Read

We proposed a new index: *relaxed read index*. The relaxed read index is the largest index of the above related logs. Therefore, the Raft logs between the relaxed read index and the read index must be irrelated to the read request, so for the read request, we do not need to wait for the apply index to catch up with the read index, we just need to wait for the apply index to catch up with the relaxed read index. It will reduce the waiting time. Because the above RA has traversed the logs, the relaxed read index can be obtained after the traversal, and the RA will transfer the relaxed read index to *Relaxed Read* (RR).

Example 4.2. Relaxed Read. As Fig. 10 shows, we assume the *relaxed read index* and the *read index* of *GET(x)* are 1 and 3 respectively, while the current *apply index* of the leader is 0 at time t_1. In this case, the leader only needs to wait for 1 Raft log to be applied to the state machine to process *GET(x)* and get the correct result, instead of waiting for 3 Raft logs.

4.3.3 Semantic-Influencing Request

Definition 4.1. (Semantic-influencing Request)For a read request, if the IM process is followed, an incorrect result is generated when there is a request to modify the result of the state machine, then the request is a semantic-influencing request of the read request.

Most distributed systems have semantic-influencing requests. Take four distributed systems as examples: KV systems, file

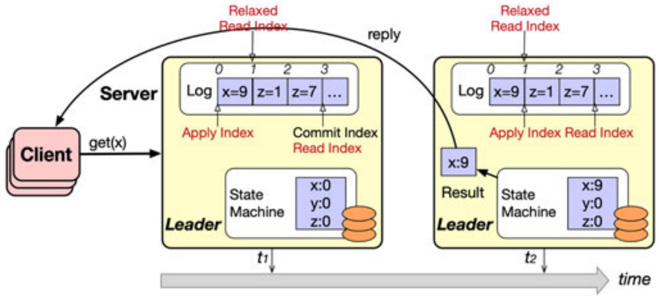


Fig. 10. An example of Relaxed Read in BUC-Raft.

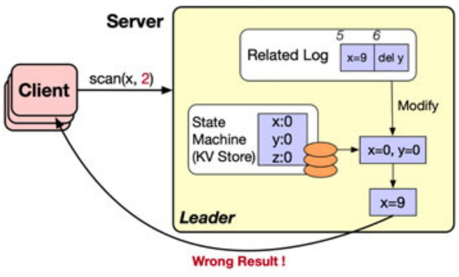


Fig. 11. An example of the semantic-influencing request in the KV system.

systems, relational databases, and graph databases. In the distributed KV system, only *SCAN(startKey, count)* has the semantic-influencing request, and its semantic-influencing request is only *DELETE(key)*. Because *DELETE(key)* modifys the result of the state machine, it may make the number of KV pairs in the final result less than *count*, or make the final result incorrect.

Example 4.3. Semantic-influencing Request in KV. As Fig. 11 shows, for the read request $SCAN(x, 2)$, we assume that the related logs traversed by RA are $\{x : 9, del\ y\}$. If the IM process is followed at this time, the final result will be $\{x : 9\}$, and the number of KV pairs in the final result is less than the required 2. So *DELETE(key)* is the semantic-influencing request of *SCAN(startKey, count)*.

In the distributed file system, *pwrite(fd, buf, len, offset)* is not the semantic-influencing request of *pread(fd, buf, len, offset)*, but *write(fd, buf, len)* is the semantic-influencing request of *pread(fd, buf, len, offset)*, because *write()* does not know where the file offset is when modifying the result of *pread()*. For *read(fd, buf, len)*, it is not the read-only request, because it changes the file offset, it is the second type of read-write mixed request. The above methods are described on the website: *https://linux.die.net/man/2/pwrite*.

In the distributed relational database, *UPDATE* is the semantic-influencing request of *SELECT*. For example, when *UPDATE* contains *a = a + 1* and *SELECT* contains *where a = 100*, an error will occur if the IM process is followed. In the distributed graph database, *DELETE* is the semantic-influencing request of the read request with *LIMIT*, the same as in Example 4.3.

In fact, many distributed relational databases and graph databases convert data into KV storage, and all requests are converted into KV operations. For example, distributed relational databases CockroachDB [11], TiDB [12], YugabyteDB [13] and Kudu, etc., distributed graph databases NebulaGraph [14] and Dgraph, etc., HBase [15] and Bigtable also convert tables into KV on the storage model. Therefore, this type of database does not need to consider its own semantic-influencing request, only the semantic-influencing request of KV.

4.3.4 Consistent Request

Definition 4.2. *(Consistent Request)* When there are semantic-influencing requests of the read request in the related logs, if the read request can be changed so that the IM process is still followed without error, the changed read request is a consistent request.

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on March 20,2025 at 14:10:55 UTC from IEEE Xplore. Restrictions apply.

In the distributed KV system, for *SCAN(startKey, count)*, if there are k DELETE requests in the related logs, the consistent request is *SCAN(startKey, count + k)*, i.e., the consistent request needs to read k more KV pairs.

In the distributed file system, for *pread(fd, buf, len, offset)*, if *write(fd, buf, len)* is in the related logs, the consistent request is *pread(fd, buf, len, offset) + lseek(fd, 0, SEEK_CUR)*, where *lseek(fd, 0, SEEK_CUR)* is to get the current file offset.

In the distributed relational database, suppose the read request is *"select * from T where a > 100,"* if the request *"update T set a = a + 10"* is in the related logs, the consistent request is *"select * from T where a > 90"*. For some complex situations of some systems, if it is difficult to convert a consistent request, it can go directly to the *RR* process.

4.3.5 Cost Model

When there are semantic-influencing requests of the read request in the related logs and the read request can be converted into a consistent request, it is necessary to judge which process of *IM* or *RR* has the lower cost. At this time, it is only need to determine the size of the read more cost ($cost_{read\_more}$) of the consistent request of the *IM* and the waiting cost ($cost_{rr\_wait}$) of the *RR*.

As shown in Equation (1), the read more cost ($cost_{read\_more}$) of the consistent request of the *IM* is the cost of executing the consistent request minus the cost of executing the original read request. The waiting cost ($cost_{rr\_wait}$) of the *RR* is the gap between the relaxed read index and the apply index multiplied by the average cost of waiting to apply a log according to Equation (2).

$$cost_{read\_more} = cost_{consistent} - cost_{read} \tag{1}$$

$$cost_{rr\_wait} = (index_{rr} - index_{apply}) * cost_{wait\_apply}. \tag{2}$$

4.3.6 Linearizability of BUC-Raft

BUC-Raft can ensure the linearizability, because the *Append*, *Commit*, and *Apply* of BUC-Raft are also in the order of index, the following two properties [16] are sufficient for linearizability.

P1. Visibility. A read operation sees the effects of all write operations that finished before it started.

P2. Integrity. A read operation will not see the effects of any write operation that had not committed at the time the read finished.

The read operation of BUC-Raft can see the effect of the write operation before the commit index, so P1 is satisfied. The read operation of BUC-Raft will not see the effect of the write operation after the commit index, so it satisfies P2.

4.4 Discussion

When are Constraints Useful? When the read-write mixed request cannot be split and a specific return result is required, the constraint of apply is required at this time, and the *Commit Return* process cannot be followed.

Comparison With Commit Protocols. Compared with commit protocols like (2 PC, EC [17]), BUC-Raft only needs more than half of the nodes to include requests to commit, while 2 PC and EC require all nodes to include. But when more than half of the nodes lose data, the data already committed by BUC-Raft may be lost, so 2 PC and EC are more secure.

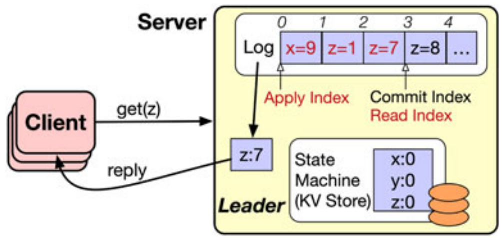


Fig. 12. An example of *GET(z)* processing in BUC-Raft: z is in Raft logs.

5 BUC-Raft in Distributed KV Systems

Most of the state machines of distributed systems using Raft are KV storage, such as Etcd [10], TiKV [18], LogCabin [19], dragonboat, CockroachDB [11], TiDB [12], YugabyteDB [13], Kudu, NebulaGraph [14] and Dgraph, etc. As mentioned above, many distributed relational databases and graph databases convert data into KV storage, and all requests are converted into KV operations. Therefore, this section will describe in detail the situation of BUC-Raft in KV systems. The *Commit Return* of all systems is the same as described in Section 4.2, only *Read Acceleration* is somewhat different, so the following will describe the *Read Acceleration* in the KV system in detail.

5.1 Read Acceleration for KV Systems

In a KV store, the read operations can be divided into three categories: *GET(key)*, *SCAN(startKey, endKey)*, and *SCAN (startKey, count)*, which means to get the KV pairs for the number of *count* sequentially since *startkey*. Because only *SCAN(startKey, count)* has the semantic-influencing request, *GET(key)* and *SCAN(startKey, endKey)* do not need to consider the semantic-influencing request, and go directly to the *Immediate Read* process.

Processing GET in Read Acceleration. For the GET operations, we can read the in-memory Raft logs between the *apply index* and the *read index* from back to front to search the target key, because the later logs contain the newer versions. If the target key is found in these logs, its related value will be returned immediately to the client directly without reading the state machine; if it is not found, we will read the target key from the state machine. Since the Raft's logs are in memory, and only the logs between the apply index and the read index need to be read, the time to read these logs is very small compared to the time to wait for these logs to be applied.

Example 5.1. An Example of Read Acceleration: GET(key) in which the key is in the logs between the apply index and the read index. As Fig. 12 plots, the leader receives a read request *GET(z)* from the client. First, the leader sets the read index of the read request to the current commit index, i.e., 3, and then search the target key z from log 3 to log 1. It is fortunate that the key z is found when reading the log 3. Then, the result $z:7$ is returned to the client immediately; during the whole read process, the state machine is never read.

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on March 20,2025 at 14:10:55 UTC from IEEE Xplore. Restrictions apply.

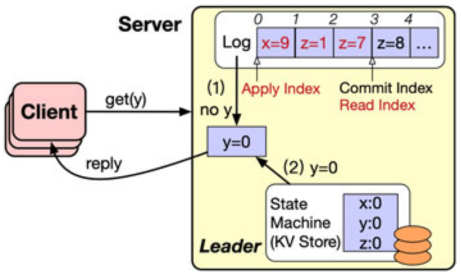


Fig. 13. An example of *GET(y)* processing in BUC-Raft: y is NOT in Raft logs.

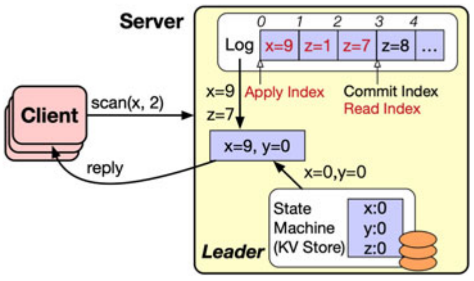


Fig. 15. An example of *SCAN(x, 2)* processing in BUC-Raft: there is no DELETE in the result of reading the Raft logs.

Example 5.2. An Example of Read Acceleration: GET(key) in which the key is NOT in the logs between the apply index and the read index. As shown in Fig. 13, assuming the current commit index is 3, the leader starts to search the in-memory logs from the index 3 to 1 after it receives *Get*(y). Unfortunately, the target key y is not found between the logs 1 and 3. Then, the leader needs to read the key y from the state machine and return the result $y : 0$ to the client.

Processing SCAN(startKey, endKey) in Read Acceleration. For SCAN(startKey, endKey), we also do not need to wait for the apply index to catch up with the read index. However, different from GET, we should perform reading both the in-memory Raft logs between the apply index and the read index from back to front and the state machine at the same time for SCAN. The reason lies in that we cannot make sure that the in-memory logs contain all the target data for range queries. Note that the result of the SCAN operation from the Raft logs is in order. Thus after getting the results from both the in-memory logs and the state machine, we can perform a merge sort for the two parts of results, and the data from logs have higher priority for the same key.

Example 5.3. An Example of Read Acceleration: SCAN(startKey, endKey). A read request of *SCAN*(x, z) is sent to the leader, as Fig. 14 illustrates. Assuming the current commit index is 3, the leader will perform reading from the log 3 to log 1 and the state machine at the same time. The result $\{x : 9, z : 7\}$ is gotten from the logs; the state machine returns $\{x : 0, y : 0, z : 0\}$. Then the leader merges the two parts of the results together. Because the in-memory logs contain relatively newer data, the overlapped keys, i.e., x and z, in the two result sets adopt the result of logs. Therefore, the final read result is $\{x : 9, y : 0, z : 7\}$ and is returned to the client. Note that without reading from the state machine, y will not be put into the read result.

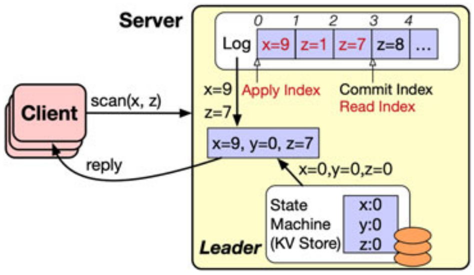


Fig. 14. An example of *SCAN(x, z)* processing in BUC-Raft.

Processing SCAN(startKey, count) in Read Acceleration. For SCAN(startKey, count), we also do not need to wait for the apply index to catch up with the read index. Because *DELETE (key)* is the semantic-influencing request of *SCAN(startKey, count)*, we should first perform reading the in-memory Raft logs between the apply index and the read index from back to front for SCAN. Note that DELETE does not participate in the counting of *count* during the reading process.

After that, if the read result does not contain DELETE, then read the state machine, and finally perform a merge sort for the two parts of the results. If the read result contains DELETE, the number of DELETE in the result is counted as k, and the largest log index in the result is the relaxed read index. Then compare the cost ($cost_{read\_more}$) of reading k more KV pairs for the consistent request with the cost ($cost_{rr\_wait}$) of waiting for the apply index to catch up with the relaxed read index.

If $cost_{read\_more}$ is smaller, use the consistent request *SCAN (startKey, count + k)* to read the state machine, then perform a merge sort for the results from the Raft logs and the state machine, and only *count* KV pairs are taken. If $cost_{rr\_wait}$ is smaller, wait for the apply index to catch up with the relaxed read index, then read the state machine and return the result to the client.

Example 5.4. An Example of Read Acceleration: SCAN(startKey, count) in which there is no DELETE in the result of reading the Raft logs. Plotted as Fig. 15, when the leader receives a read request *SCAN*(x, 2) and the current commit index is 3, the leader reads the in-memory logs from index 3 to 1. The log reading leads to a result set $\{x : 9, z : 7\}$. There is no DELETE in this result, so the leader reads the state machine immediately, and the result of the state machine is $\{x : 0, y : 0\}$. Finally, the two result sets are merged to $\{x : 9, y : 0\}$, because z from the logs is out of the final result sets (i.e., x and y) and $x : 0$ from the state machine is overwritten by $x : 9$ from the logs with higher priority.

Example 5.5. An Example of Read Acceleration: SCAN(startKey, count) in which $cost_{read\_more}$ is smaller. Plotted as Fig. 16, when the leader receives a read request *SCAN*(x, 2) and the current commit index is 3, the leader reads the in-memory logs from index 3 to 1. The log reading leads to a result set $\{x : 9, del\ y, z : 7\}$. There is one DELETE in the result, so the consistent request is *SCAN*(x, 3). Through judgment, the cost ($cost_{read\_more}$) of reading one more KV pair for the consistent request is less than the cost ($cost_{rr\_wait}$) of waiting for the apply index to catch up with the relaxed read

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on March 20,2025 at 14:10:55 UTC from IEEE Xplore. Restrictions apply.

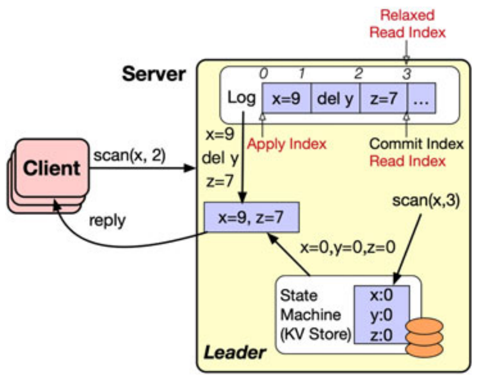


Fig. 16. An example of *SCAN(x, 2)* processing in BUC-Raft: $cost_{read\_more}$ is smaller.

index, so we use $SCAN(x, 3)$ to read the state machine, and the result of the state machine is $\{x:0, y:0, z:0\}$. Finally, the two result sets are merged to $\{x:9, z:7\}$.

Example 5.6. An Example of Read Acceleration: SCAN(startKey, count) in which $cost_{rr\_wait}$ is smaller. Plotted as Fig. 17, when the leader receives a read request $SCAN(x, 2)$ and the current commit index is 3, the leader reads the in-memory logs from index 3 to 1. The log reading leads to a result set $\{x:9, del\ y\}$. There is one DELETE in the result, so the consistent request is $SCAN(x, 3)$. Through judgment, the cost ($cost_{read\_more}$) of reading one more KV pair for the consistent request is greater than the cost ($cost_{rr\_wait}$) of waiting for the apply index to catch up with the relaxed read index, so we wait for the apply index to catch up with the relaxed read index. After waiting, read the state machine, the result of the state machine is $\{x:9, z:0\}$ and returns to the client.

5.2 Algorithm Design

The following Algorithm 1 exhibits the pseudo-codes of the two key components of BUC-Raft in distributed KV systems, i.e., *Commit Return* and *Read Acceleration*.

Commit Return. When a write request arrives at the leader, the leader first performs the same actions as the original Raft, i.e., appending local log and meanwhile synchronizing it to all the followers. When more than half the replica nodes including the leader itself have notified the leader that the log persistence is accomplished, i.e., the commit index is changed, the function *commitIndexChange* in Algorithm 1 will be called. At this time, the leader will directly respond to the client's write request whose indexes are between the last commit index and the commit index, not waiting for the applying operations anymore, then the last commit index is set to the commit index, as shown in Lines 1 ~ 6.

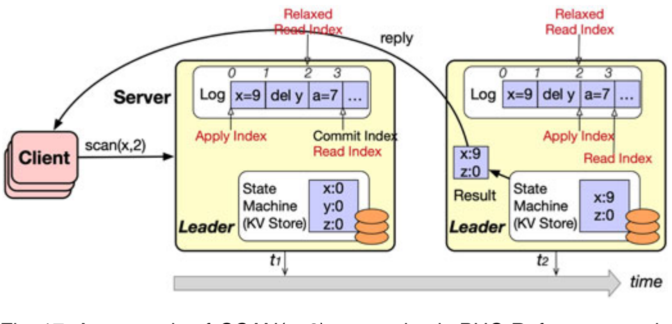


Fig. 17. An example of *SCAN(x, 2)* processing in BUC-Raft: $cost_{rr\_wait}$ is smaller.

Algorithm 1. BUC-Raft in KV

```
1:  function $commitIndexChanged$:
2:      $index \leftarrow lastCommitIndex + 1$;
3:      while $index <= commitIndex$ do
4:          $response(raftLog[index])$;
5:          $index++$;
6:      $lastCommitIndex \leftarrow commitIndex$;
7:  function $handleReadreq$:
8:      $readIndex \leftarrow commitIndex$;
9:      if $req.type = Get$ then
10:         $index \leftarrow readIndex$;
11:         while $index > applyIndex$ do
12:             if $raftLog[index].key = req.key$ then
13:                 return$(req.key, raftLog[index].value)$;
14:             $index--$;
15:         return $KV.handleRead(req)$;
16:     else if $req.type = ScanA$ then
17:         do in parallel
18:             $r1 \leftarrow logRead(req, readIndex)$;
19:             $r2 \leftarrow KV.handleRead(req)$;
20:         return $merge(r1, r2)$;
21:     else if $req.type = ScanB$ then
22:         $r1 \leftarrow logRead(req, readIndex)$;
23:         if $r1.deleteNum = 0$ then
24:             $r2 \leftarrow KV.handleRead(req)$;
25:             return $merge(r1, r2, req.count)$;
26:         else
27:             $deleteNum \leftarrow r1.deleteNum$;
28:             $relaxedReadIndex \leftarrow r1.maxLogIndex()$;
29:             if $r1.deleteNum * costRead < (relaxedReadIndex - applyIndex) * costApply$ then
30:                 $count \leftarrow req.count$;
31:                 $req.setCount(count + deleteNum)$;
32:                 $r2 \leftarrow KV.handleRead(req)$;
33:                 return $merge(r1, r2, count)$;
34:             else
35:                 $waitForApplyIndex(relaxedReadIndex)$;
36:                 return $KV.handleRead(req)$;
```

Read Acceleration. The same as Raft, when the leader starts to process a read request, it will first set the read index of the read request the same as the current commit index, as shown in Lines 7 ~ 8.

GET. If the read request is GET, the leader traverses the in-memory logs between the apply index and the read index from back to front. When the target key is found in the logs, the key-value pair retrieved from the log is directly returned to the client without reading the state machine, as Lines 9 ~ 14 indicate. If there is no target key in the logs, the leader will call the function *KV.handleRead* to read from the state machine (i.e., the KV engine) and return the result to the client (i.e., Line 15 in the pseudo-code).

SCAN(startKey, endKey). If the read request is *SCAN(startKey, endKey)* (i.e., *ScanA*), the leader needs to read the Raft logs and the KV engine in parallel. After both the read results are gotten, the two result sets are merged, and the merged result is returned to the client, as shown in Lines 16 ~ 20.

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on March 20,2025 at 14:10:55 UTC from IEEE Xplore. Restrictions apply.

SCAN(startKey, count). When the read request is *SCAN (startKey, count)* (i.e., *ScanB*), the leader needs to read the Raft logs first, as shown in Lines 21 $\sim$ 22. Note that DELETE does not participate in the counting of *count* during the above reading process.

If the read result does not contain DELETE, then read the KV engine, and merge the two result sets and return the merged result to the client, as shown in Lines 23 $\sim$ 25. Otherwise, compare the cost ($cost_{read\_more}$) of reading *deleteNum* more KV pairs for the consistent request with the cost ($cost_{rr\_wait}$) of waiting for the apply index to catch up with the relaxed read index, as shown in Lines 26 $\sim$ 29. Where $deleteNum$ is the number of DELETE in the result, $relaxedReadIndex$ is the largest log index in the result, $costRead$ is the cost of reading one more KV pair, and $costApply$ is the average cost of waiting to apply a log.

If the former is smaller, adding $deleteNum$ to the $count$ of the read request is the consistent request, then use the consistent request to read the KV engine, and merge the two result sets into *count* KV pairs and return the merged result to the client. Otherwise, wait for the apply index to catch up with the relaxed read index, then read the KV engine and return the result to the client.

6 Implementation and Evaluation

In this section, we will introduce the implementation of BUC-Raft in Section 6.1, and then present the experimental setup in Section 6.2. The detailed evaluation results are in Sections 6.3, 6.4, 6.5, 6.6, 6.7, and 6.8.

6.1 System Implementation

We have implemented our proposed BUC-Raft in TiKV [18], which is a popular enterprise-level open-source distributed key-value storage system written in Rust. TiKV contains more than 100 K LOC of *Rust*, and it is already one of the largest open-source projects in the *Rust* community. TiKV utilizes and has implemented Raft to achieve the consensus and the consistency among its distributed KV storage nodes, which employ RocksDB, a very popular LSM-tree KV store, as the key-value engine (i.e., the state machine) on each node. TiKV relies on another system called Placement Driver (PD) [20] to manage the KV data distribution; in fact, both TiKV and PD are part of the open-source distributed relational database system, i.e., TiDB [12].

Our implementation work mainly includes the newly added or modified metadata to support our proposed BUC-Raft, the modifications on the write request processing procedure, the read request processes covering both the GET and SCAN operations, and some necessary statistical functions. The source code is available at https://github.com/vliulan/BUC-Raft.

6.2 Experimental Setup

The experiments were performed in a cluster consisted of five high I/O nodes; each node is coupled with Linux Centos 7.6.1810, 8 GB DRAM, and an enterprise-level 200 GB NVMe Solid State Drive (SSD). Limited by the environment, the intranet bandwidth among the nodes is 1.5 Gbps. Three of the nodes serve as TiKV nodes, one as PD, and the last one runs the benchmark tool, i.e., Go-YCSB [21].

TABLE 1
YCSB Workloads Used in the Evaluation

Workload	Write Type	Query Type	Category
Load	Insert	/	Insert Only
A	Update	Point Query	50%write 50%read
B	Update	Point Query	5%write 95%read
C	/	Point Query	Read Only
D	Insert	Point Query	5%write 95%read
E	Insert	Range Query	5%write 95%scan
F	RMW	Point Query	50%write 50%read
W90	Update	Point Query	90%write 10%read
W70	Update	Point Query	70%write 30%read
W50	Update	Point Query	50%write 50%read
W30	Update	Point Query	30%write 70%read
W10	Update	Point Query	10%write 90%read
RO	/	Point Query	Read Only
S10	Update	Range Query	90%write 10%scan
S30	Update	Range Query	70%write 30%scan
S50	Update	Range Query	50%write 50%scan
S70	Update	Range Query	30%write 70%scan
S90	Update	Range Query	10%write 90%scan
SO	/	Range Query	Scan Only

Benchmark. Go-YCSB is a *Go* language version of the widely used YCSB benchmark for evaluating key-value systems. In the experiments, each key-value pair contains a 16-B key and a 1-KB value by default, and the data has 3 replicas in TiKV. The keys obey the Zipf distribution by default. Besides the classical workloads of YCSB (i.e., $A \sim F$) that simulate realistic scenarios, the experimental workloads also include the ones constructed with different ratios of random writes mixed with random reads (i.e., GET or SCAN).

All the workloads used are listed in Table 1, including an insert-only (*Load*) workload, a read-only (*RO*) workload, the mixed workloads consisting of 90%, 70%, 50%, 30%, or 10% of writes and point queries (i.e., GET) operations (i.e., *W90* $\sim$ *W10*). And there are similar workloads mixed with writes and different ratios of range queries (i.e., SCAN) operations (i.e., *S10* $\sim$ *S90* and *SO* for SCAN-only). Note that RMW in Table 1 means read-modify-write; the operation is to read a key first, and then to update the key-value pair.

6.3 Overall Results

In this part, we measure the performance of Raft, CR and BUC-Raft in the six classical workloads of YCSB, i.e., $A \sim F$. We first randomly insert 100 million KV pairs with a total of 100 GB data into the cluster, and then perform the workloads *A*, *B*, *C*, *D*, *E*, and *F*. Each one of the workloads includes 10 million requests.

Throughput. As Fig. 18a plots, BUC-Raft can achieve a significant throughput improvement under the workloads except for the read-only one (i.e., workload *C*), in which the performance of different algorithms is almost the same because there is no writing, logging, and the changes of the state machine in this case. Compared with Raft, the throughput of BUC-Raft is 62% higher for *Load*, 53.6% higher for *A* (i.e., 50% writes), 18.4% higher for *B* (i.e., 95% reads), 18.8% higher for *D* (i.e., 95% reads), 2.8% higher for

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on March 20,2025 at 14:10:55 UTC from IEEE Xplore. Restrictions apply.

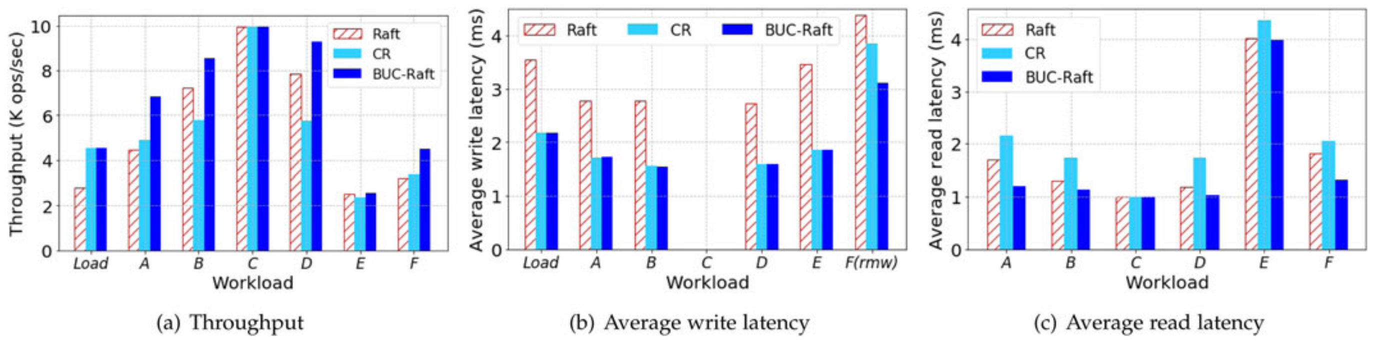


Fig. 18. Throughput and latency comparisons among Raft, Commit Return, and BUC-Raft under **YCSB** workloads.

E (i.e., 95% scans), and 39.6% higher for *F* (i.e., 50 writes). Note that *E* has much more than 95% read operations practically and is very close to a read-only workload, because one SCAN operation reads many key-value pairs once. We also evaluate the strategy of *Commit Return* (CR) only. CR achieves higher throughput than Raft under write-heavy or balanced workloads, i.e., 62% higher for *Load*, 9.6% for *A*, 5.6% for *F*. But when the read percent is very high, CR's performance is lower than the original Raft, e.g., 19.9% lower for *B*, 28.5% lower for *D*, and 6% lower for *E*. The reason lies in that CR transfers the applying overhead from the write processing to the read processing, thus the accumulated additional read latency becomes higher than the reduced write latency in read-dominant applications. However, coupled with *Read Acceleration*, which can eliminate the side effects of CR, BUC-Raft could also achieve a 18.4% higher throughput compared with Raft in a 95%-read scenario.

Latency. Figs. 18b and 18c give the results of the average write and read latency, respectively. For all the other workloads except for *F*, BUC-Raft and CR achieve very close write latency and reduce the average write latency by 38.4% $\sim$ 46.5% compared with Raft. In the workload *F*, the write operation is the type of read-modify-write (RMW), i.e., reading a key first and then updating the key-value pair. The performance of CR is much lower than BUC-Raft because the write operations have to wait for read operations which is processed slower for CR. Thus BUC-Raft reduces the latency by 29% compared to Raft, and CR reduces it by only 13% compared with Raft.

For the average read latency shown in Fig. 18c, BUC-Raft can reduce it in all the other workloads except *C* compared with Raft. i.e., 29.4% lower for *A*, 12.4% for *B*, 12.5% for *D*, 0.8% for *E* and 27% for *F*. Because BUC-Raft eliminates Raft's waiting time for applying the state machine before reading. In addition, the read performance of CR is lower than that of Raft and BUC-Raft. Therefore, BUC-Raft can achieve comprehensive and significant performance improvement including the throughput, the write and the read latency in most situations.

6.4 Impacts of Read and Write Ratios

First, we initialized the key-value store by randomly inserting 100 million KV pairs with a total of about 100 GB data. Then we performed the workloads *W90*, *W70*, *W50*, *W30*, *W10*, *RO*, *S10*, *S30*, *S50*, *S70*, *S90*, and *SO*. All these workloads include 10 million requests each. Because the write requests used here belong to the type of *update*, the total data size is always maintained as about 100 GB.

6.4.1 Point Queries

Throughput. As Fig. 19a plots, BUC-Raft achieves significantly higher throughput than Raft in all workloads except the read-only one (i.e., *RO*). In all the other cases, the throughput improvement of BUC-Raft over Raft is 62% for *Load*, 61.4% for *W90*, 59.5% for *W70*, 51.3% for *W50*, 41.7% for *W30*, and 20.4% for *W10*. Because BUC-Raft improves the write performance more than reading (see the write and the read latency in Figs. 19b and 19c), the more writes a workload has, the more improvement of BUC-Raft can be achieved compared with Raft.

CR also gets a higher performance under the write-heavy or balanced cases (e.g., *Load*, *W90*, *W70*, and *W50*), and has a worse performance under read-heavy ones (e.g., *W10*).

Latency. Fig. 19b exhibits the average write latency of Raft, CR, and BUC-Raft. The average write latency of BUC-

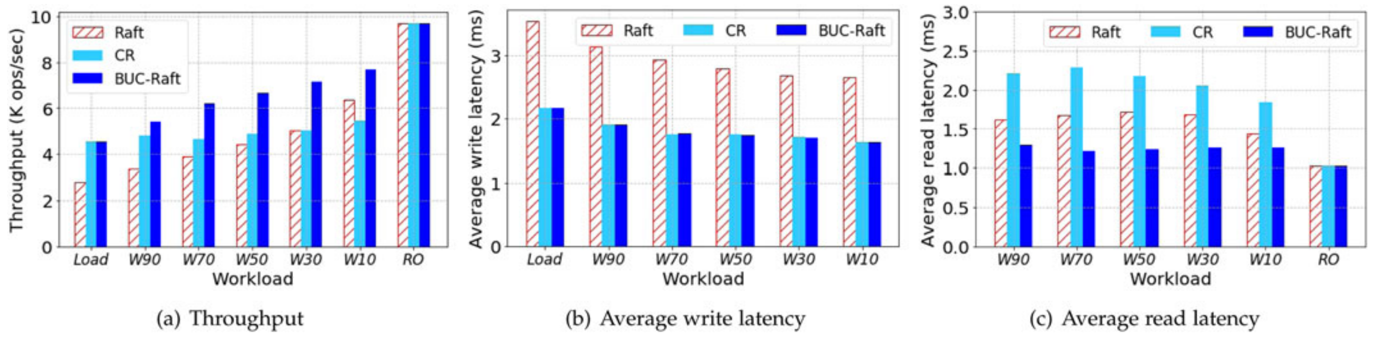


Fig. 19. Throughput and latency comparisons among Raft, Commit Return, and BUC-Raft for **point queries**.

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on March 20,2025 at 14:10:55 UTC from IEEE Xplore. Restrictions apply.

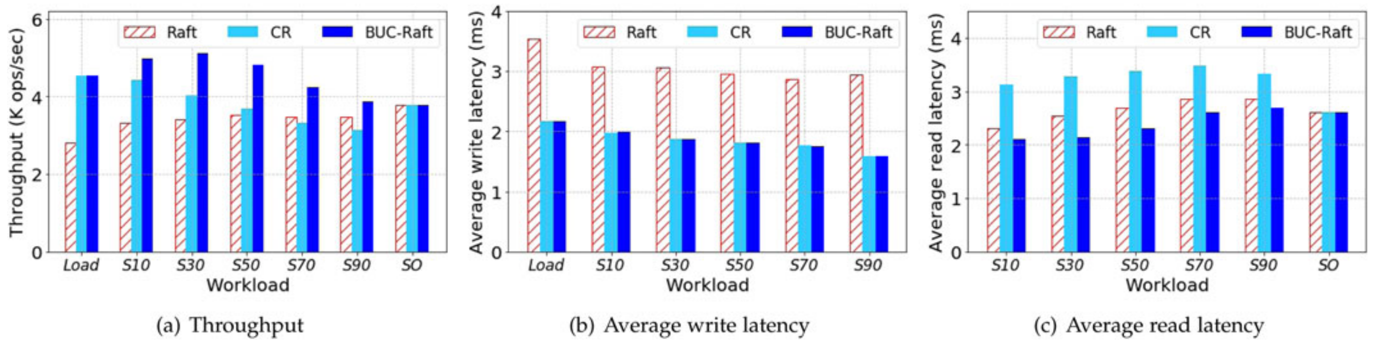


Fig. 20. Throughput and latency comparisons among Raft, Commit Return, and BUC-Raft for **range queries**.

Raft and CR is very close to each other, because the write process of BUC-Raft is the same as that of CR. They reduce the average write latency by 36.4% $\sim$ 39.9% compared with Raft.

Fig. 19c illustrates the average read latency. BUC-Raft achieves a lower average read latency than Raft in all workloads except for *RO*, i.e., 19.6% lower for *W90*, 27.1% for *W70*, 28% for *W50*, 24.9% for *W30* and 12.6% for *W10*. Although the gap between the apply index and the read index is larger for workloads with a larger write percent, the state machine of BUC-Raft processes more writes at this time, which will affect read performance. Therefore, the reduction in average read latency is not monotonous, and BUC-Raft achieves the highest performance improvement under the read/write-balanced case (i.e., 28% for *W50*). In addition, the read performance of CR is lower than that of Raft and BUC-Raft.

For point queries, BUC-Raft may only need to get the target data from the in-memory logs and skip reading from the state machine; this is a possibility for performance acceleration. Thus we have counted the proportion of read requests that did not read the state machine, and the proportion was as low as 0.2% $\sim$ 0.6%. This indicates that BUC-Raft's read performance improvement is mainly from eliminating the waiting time for applying the state machine before reading.

6.4.2 Range Queries

Throughput. The results of range queries are basically similar to those of point queries. Shown as Fig. 20a, the performance improvement of BUC-Raft is 11% $\sim$ 62% compared with Raft for the workloads *Load* and *S10* $\sim$ *S90*. The performance under *SO* is no difference among Raft, CR, and BUC-Raft because the processing procedures of them are the same under the read-only case. The improvement is a bit lower than the previous point query, because the range query needs to read multiple KV pairs within one request and thus the practical read percentage is higher than the above point query workloads.

Latency. The average write latency results are shown in Fig. 20b. Compared with Raft, BUC-Raft and CR reduce the average write latency by 36.3% $\sim$ 46%, accelerating the write request processing significantly. Fig. 20c exhibits the average read latency. The read performance of CR is also lower than that of Raft for SCAN. And BUC-Raft can reduce the read latency by 5.8% $\sim$ 16.2% compared with Raft except for *SO*.

6.5 Impacts of Semantic-Influencing Request

In this part, we evaluate the impact of the proportion of semantic-influencing requests on performance. We changed the percentage of DELETE operations for write requests in workload *S50* from 0% to 100%, and then measure the performance of Raft and BUC-Raft for these different percentages. In each experiment, we first randomly insert 100 million KV pairs with a total of 100 GB data into the cluster, and then perform the workload *S50* of different DELETE operation ratios in the write request. Each one of the workloads includes 10 million requests.

Throughput. As Fig. 21 plots, BUC-Raft achieves higher throughput than Raft for all DELETE operation ratios in the write request, i.e., 36.8% higher for *0%*, 36% higher for *10%*, 34.1% higher for *30%*, 32.1% higher for *50%*, 30% higher for *70%*, 27.7% higher for *90%*, and 26.5% higher for *100%*. With the increase in the proportion of DELETE operations in write requests, BUC-Raft's improved performance compared to Raft is declining, because the improved read

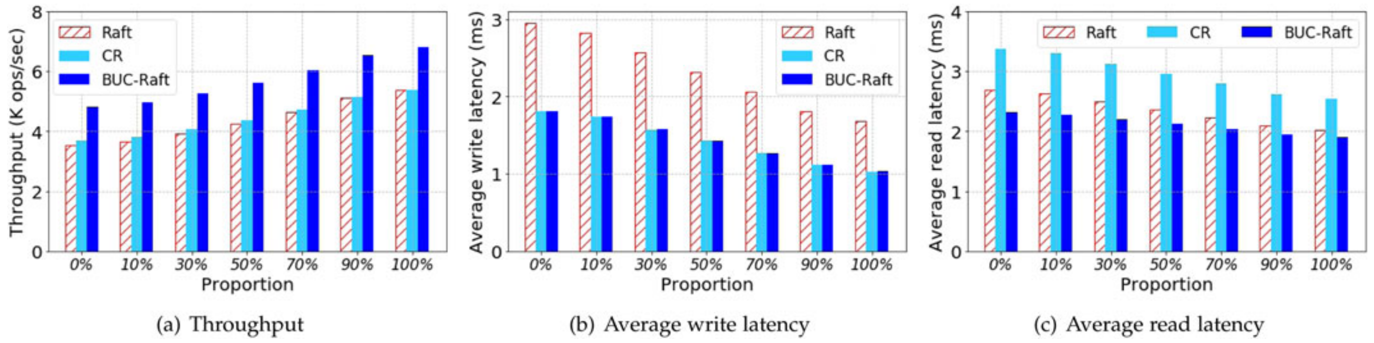


Fig. 21. Throughput and latency comparisons among Raft, Commit Return, and BUC-Raft for different DELETE operation ratios in the write request.

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on March 20,2025 at 14:10:55 UTC from IEEE Xplore. Restrictions apply.

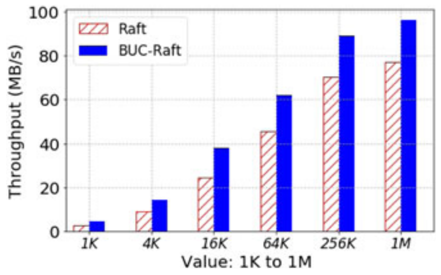


Fig. 22. Throughput for different value sizes.

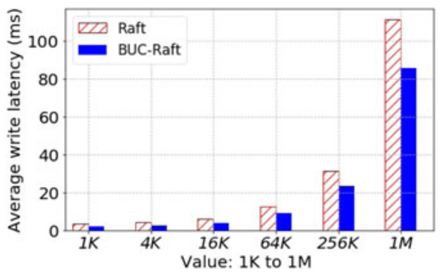


Fig. 23. Average write latency for different value sizes.

performance is declining (see the read latency in Fig. 21c). But even if the write requests are all DELETE operations, BUC-Raft can achieve significantly higher performance (ie, more than 26.5%).

CR also gets a higher performance but very small.

Latency. Figs. 21b and 21c give the results of the average write and read latency, respectively. For all workloads, BUC-Raft can reduce the average write latency by 37.7% $\sim$ 39.1% compared with Raft, and BUC-Raft can also reduce the average read latency compared with Raft. i.e., 14.2% lower for *0%*, 13.4% lower for *10%*, 11.8% lower for *30%*, 10.2% lower for *50%*, 8.7% lower for *70%*, 7.1% lower for *90%*, and 6.3% lower for *100%*. With the increase in the proportion of DELETE operations in write requests, BUC-Raft's reduced read latency compared to Raft is decreasing, because the cost of *Read Acceleration* is rising, but even if the write requests are all DELETE operations, BUC-Raft's average read latency is also lower than Raft.

6.6 Impacts of Key-Value Pair Size

In this part, we measure the performance of Raft and BUC-Raft for different value sizes ranging from 1 K to 1 M. In each experiment, we randomly insert 100 GB data into the cluster.

Throughput. As Fig. 22 plots, BUC-Raft achieves higher throughput than Raft, i.e., 62% higher for *1 K*, 60.2% higher for *4 K*, 54.6% higher for *16 K*, 36.5% higher for *64 K*, 26.5% higher for *256 K*, and 24.7% higher for *1 M*. For larger value size settings, the network transmission time accounts for a larger proportion of the total processing time, so the acceleration of BUC-Raft is more significant for small values and upgrading the network helps to boost more for larger values in distributed KV stores.

Latency. As Fig. 23 shows, BUC-Raft achieves lower average read latency than Raft, i.e., 38.5% lower for *1 K*, 38.3% lower for *4 K*, 37.3% lower for *16 K*, 29.1% lower for *64 K*, 24.1% lower for *256 K*, and 23.1% lower for *1 M*. The write boost of BUC-Raft is relatively stable under different value size settings.

6.7 In-Memory Database Case

In this part, we evaluate BUC-Raft under the in-memory database environment, which is simulated by keeping the entire data set in memory. We first randomly insert 100,000 KV pairs (i.e., about 100 MB) into the cluster, and then perform the workloads *A*, *B*, *C*, *D*, *E*, and *F*; each one contains 100,000 requests. Because we set the size of RocksDB's in-memory Memtable to 128 MB, all the data will be maintained in memory (i.e., an in-memory state machine).

Throughput. Similar to the previous experimental results, among the workloads $A \sim F$, BUC-Raft can achieve a significant throughput improvement under the workloads except for the read-only workload *C*. As Fig. 24a plots, compared with Raft, the throughput of BUC-Raft is 58.1% higher for *Load*, 59.1% higher for *A*, 25.7% higher for *B*, 22.9% higher for *D*, 3.6% higher for *E*, and 54.3% higher for *F*.

First, the throughput improvement of BUC-Raft in *Load* is only a little lower than the previous disk state machine case. The reason lies in that when we write data into the in-memory state machine, we also need to write the data to the disk log for persistence and perform sorting, and the performance of the NVMe SSD we used is particularly good. So the write performance difference between the two types of state machines is not large.

Second, for other workloads, BUC-Raft's throughput improvement is higher than before, because BUC-Raft leads to a much larger read latency decrease than the previous results (see the read latency results and explanations below).

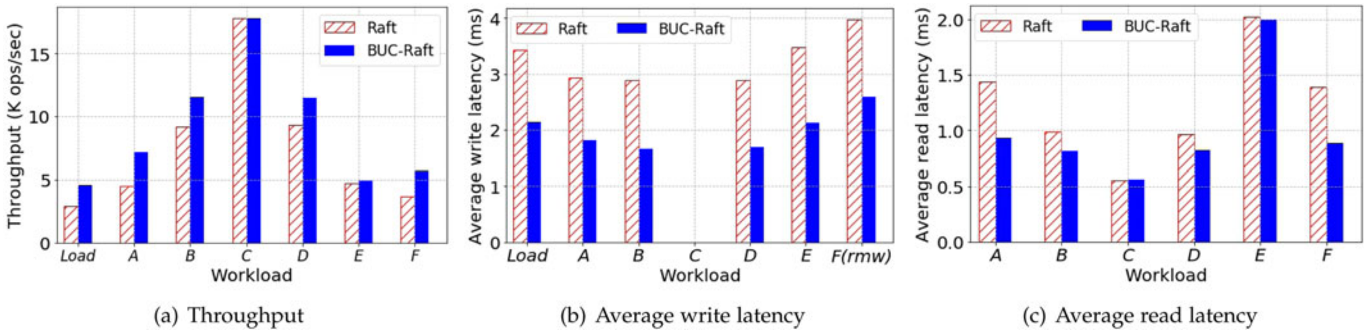


Fig. 24. Throughput and latency comparisons between Raft and BUC-Raft for **in-memory database** cases.

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on March 20,2025 at 14:10:55 UTC from IEEE Xplore. Restrictions apply.

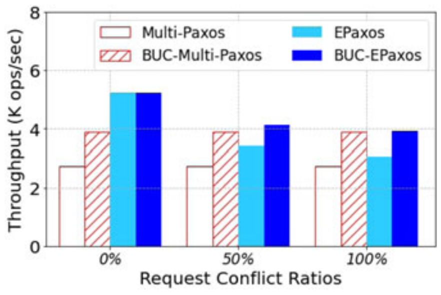


Fig. 25. Combined with other consensus algorithms.

Latency. Fig. 24b and 24c give the results of the average write and read latency, respectively. Compared with Raft, BUC-Raft reduces the average write latency by 37.4% ~ 42.6%. For the average read latency, BUC-Raft can reduce it in all the other workloads except C compared with Raft by 25.3%, 17.2%, 14.8%, 1.3%, and 36.2% for the workloads A, B, D, E, and F, respectively. The read latency has dropped more than the disk state machine case. Since reading from the memory is much faster than SSDs, the time of waiting for the apply index to catch up with the read index accounts for a larger proportion. Therefore, the read latency reduction of BUC-Raft is more significant.

6.8 Combined With Other Consensus Algorithms

The idea of BUC-Raft can be combined with other consensus algorithms to replace their read request processing, please refer to Section 7 for details. We combine BUC-Raft with Multi-Paxos and EPaxos and compare their performance under different request conflict ratios.

We conduct experiments with read-write ratios of 50% each, and the experimental results are shown in Fig. 25. In the absence of conflicting requests, BUC-EPaoxs have no improvement over EPaxos. But at 50% and 100% request conflict ratios, BUC-EPaoxs improves the throughput of EPaxos by 20.9% and 29.1%, respectively. Under three different request conflict ratios, BUC-Multi-Paxos has a stable improvement over Multi-Paxos.

7 Related Work

Existing related work about the performance optimization of distributed consensus algorithms (e.g., Paxos, Raft, etc.) can be divided into the following two categories: *reducing latency* and *reducing leader load*.

7.1 Reducing Latency

Traditional leader-based consensus protocols (e.g., Multi-Paxos [2] or Raft [3]) require 2 RTTs to complete operations: 1 RTT between the client and the leader, and another RTT for the leader to replicate operations to other replicas. In Fast Paxos [22], the client directly sends the request to all replicas, then the leader waits for the majority of replicas to agree to the request order, and finally the leader responds to the client, with a total of 1.5 RTT. Generalized Paxos [23] extends Fast Paxos by allowing non-conflicting requests to execute in a different order, also 1.5 RTT. Speculative Paxos [24] and Network-Ordered Paxos [25] achieve 1 RTT latency by using a special network to guarantee the order in which requests arrive at replicas. CURP [26] separates durability and ordering by adding witnesses to achieve 1 RTT latency.

Flexible Paxos [27] uses different quorum sizes to perform operations, and lower quorum sizes mean lower latency. EPaxos [28] and ParallelRaft [29] achieve parallel execution of non-conflicting requests. APUS [30] combines RDMA to reduce network transmission latency in consensus protocols. Tapir [31] combines transactions and consensus to commit transactions in 1 RTT.

Summary. All of the above algorithms must wait for the completion of the previous conflicting requests for a read request. For BUC-Raft, in most cases, it can directly execute read requests without waiting for conflicting requests (i.e., related requests). At the same time, the read request processing of BUC-Raft only needs 1 RRT. Therefore, the idea of BUC-Raft can be combined with these consensus algorithms to replace their read request processing.

7.2 Reducing Leader Load

PigPaxos [32] transfers the load of the leader to the relay nodes, and rotates the relay nodes so that they do not become hotspots. HovercRaft [33] uses in-network accelerators (e.g., a P4 ASIC) to convert leader-to-multipoint interactions into point-to-point interactions, and the leader can entrust the follower to reply to the client, reducing the load of the leader replying to all clients. ChainPaxos [34] adopts chain replication to reduce the transmission load of the leader. MongoDB [35] allows replicas to pull data from the follower.

EPaxos [28], Atlas [36] and Hermes [37] are leaderless consensus protocols, the client can send requests to any replica, and any replica can commit the request and respond to the client. Obiden [38], Copilot [39] and Caesar [40] are multi-leader consensus protocols, they provide two or more leaders to reduce the load of a single leader.

Summary. The above algorithms mainly optimize request transmission, so the idea of BUC-Raft can be combined with these consensus algorithms to replace their read request processing.

8 Conclusion

Consensus and consistency assurance is one of the core components of distributed systems, and also becomes the bottleneck of performance for modern hardware-based distributed systems. However, some of the strict constraints that affect performance much imposed by Raft, one of the most commonly used consensus algorithms, are found not to be necessary for distributed systems. We propose an improved consensus algorithm entitled *BUC-Raft* to substantially boost distributed systems by breaking some of Raft's limitations on requests processing. BUC-Raft is very productive, with great potential to be widely adopted for accelerating distributed systems.

References

[1] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, 1998.
[2] L. Lamport, "Paxos made simple," *ACM SIGACT News*, vol. 32, no. 4, pp. 18–25, 2001.
[3] D. Ongaro and J. Ousterhout, "In search of an understandable consensus algorithm," in *Proc. USENIX Annu. Tech. Conf.*, 2014, pp. 305–319.

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on March 20,2025 at 14:10:55 UTC from IEEE Xplore. Restrictions apply.

[4] D. Ongaro, "Consensus: Bridging theory and practice," PhD thesis, Stanford University, Stanford, CA, USA, 2014.

[5] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects," *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, 1990.

[6] F. P. Junqueira, B. C. Reed, and M. Serafini, "Zab: High-performance broadcast for primary-backup systems," in *Proc. IEEE/IFIP 41st Int. Conf. Dependable Syst. Netw.*, 2011, pp. 245–256.

[7] S. Gupta et al., "ResilientDB: Global scale resilient blockchain fabric," *Proc. VLDB Endowment*, vol. 13, no. 6, pp. 868–883, 2020.

[8] S. Gupta, J. Hellings, and M. Sadoghi, "RCC: Resilient concurrent consensus for high-throughput secure transaction processing," in *Proc. IEEE 37th Int. Conf. Data Eng.*, 2021, pp. 1392–1403.

[9] S. Gupta, J. Hellings, and M. Sadoghi, "Fault-tolerant distributed transactions on blockchain," *Synth. Lectures Data Manage.*, vol. 16, no. 1, pp. 1–268, 2021.

[10] ETCD, 2022. [Online]. Available: https://github.com/etcd-io/etcd

[11] CockroachDB, 2022. [Online]. Available: https://github.com/cockroachdb/cockroach

[12] TiDB, 2022. [Online]. Available: https://github.com/pingcap/tidb

[13] YugabyteDB, 2022. [Online]. Available: https://github.com/yugabyte/yugabyte-db

[14] NebulaGraph, 2022. [Online]. Available: https://github.com/vesoft-inc/nebula

[15] HBase, 2022. [Online]. Available: https://hbase.apache.org/

[16] H. Zhu et al., "Harmonia: Near-linear scalability for replicated storage with in-network conflict detection," *Proc. VLDB Endowment*, vol. 13, no. 3, pp. 376–389, 2019.

[17] S. Gupta and M. Sadoghi, "EasyCommit: A non-blocking two-phase commit protocol," in *Proc. 21st Int. Conf. Extending Database Technol.*, 2018, pp. 157–168.

[18] TiKV, 2022. [Online]. Available: https://github.com/tikv/tikv

[19] Logcabin, 2022. [Online]. Available: https://github.com/logcabin/logcabin

[20] PD, 2022. [Online]. Available: https://github.com/pingcap/pd

[21] go-ycsb, 2022. [Online]. Available: https://github.com/pingcap/go-ycsb

[22] L. Lamport, "Fast paxos," *Distrib. Comput.*, vol. 19, no. 2, pp. 79–103, 2006.

[23] L. Lamport, "Generalized consensus and Paxos," Microsoft Res., Tech. Rep. MSR-TR-2005-33, 2005.

[24] D. R. K. Ports et al., "Designing distributed systems using approximate synchrony in data center networks," in *Proc. 12th USENIX Symp. Netw. Syst. Des. Implementation*, 2015, pp. 43–57.

[25] J. Li et al., "Just say NO to paxos overhead: Replacing consensus with network ordering," in *Proc. 12th USENIX Symp. Operating Syst. Des. Implementation*, 2016, pp. 467–483.

[26] S. J. Park and J. Ousterhout, "Exploiting commutativity for practical fast replication," in *Proc. 16th USENIX Symp. Netw. Syst. Des. Implementation*, 2019, pp. 47–64.

[27] H. Howard, D. Malkhi, and A. Spiegelman, "Flexible paxos: Quorum intersection revisited," 2016, *arXiv:1608.06696*.

[28] I. Moraru, D. G. Andersen, and M. Kaminsky, "There is more consensus in egalitarian parliaments," in *Proc. 24th ACM Symp. Operating Syst. Princ.*, 2013, pp. 358–372.

[29] W. Cao et al., "PolarFS: An ultra-low latency and failure resilient distributed file system for shared storage cloud database," *Proc. VLDB Endowment*, vol. 11, no. 12, pp. 1849–1862, 2018.

[30] C. Wang et al., "APUS: Fast and scalable paxos on RDMA," in *Proc. Symp. Cloud Comput.*, 2017, pp. 94–107.

[31] I. Zhang et al., "Building consistent transactions with inconsistent replication," *ACM Trans. Comput. Syst.*, vol. 35, no. 4, pp. 1–37, 2018.

[32] A. Charapko, A. Ailijiang, and M. Demirbas, "PigPaxos: Devouring the communication bottlenecks in distributed consensus," in *Proc. Int. Conf. Manage. Data*, 2021, pp. 235–247.

[33] M. Kogias and E. Bugnion, "HovercRaft: Achieving scalability and fault-tolerance for microsecond-scale datacenter services," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–17.

[34] P. Fouto, N. Preguiça, and J. Leitão, "High throughput replication with integrated membership management," in *Proc. USENIX Annu. Tech. Conf.*, 2022, pp. 575–592.

[35] S. Zhou and S. Mu, "Fault-tolerant replication with pull-based consensus in MongoDB," in *Proc. 18th USENIX Symp. Netw. Syst. Des. Implementation*, 2021, pp. 687–703.

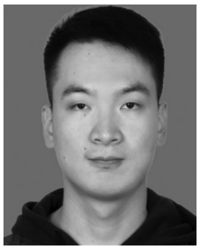
[36] V. Enes et al., "State-machine replication for planet-scale systems," in *Proc. 15th Eur. Conf. Comput. Syst.*, 2020, pp. 1–15.

[37] A. Katsarakis et al., "Hermes: A fast, fault-tolerant and linearizable replication protocol," in *Proc. 25th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2020, pp. 201–217.

[38] J. Sorensen, A. Xiao, and D. Allender, "Dual-leader master election for distributed systems (Obiden)," 2015. [Online]. Available: https://www.cse.scu.edu/~m1wang/projects/Distributed_dual Leaders_15s.pdf

[39] K. Ngo, S. Sen, and W. Lloyd, "Tolerating slowdowns in replicated state machines using copilots," in *Proc. 14th USENIX Symp. Operating Syst. Des. Implementation*, 2020, pp. 583–598.

[40] B. Arun et al., "Speeding up consensus by chasing fast decisions," in *Proc. IEEE/IFIP 47th Annu. Int. Conf. Dependable Syst. Netw.*, 2017, pp. 49–60.



Yangyang Wang received the BS degree in computer science from the Renmin University of China, in 2017. He is currently working toward the PhD degree with the School of Information, Renmin University of China. His research interests include distributed storage systems and consensus algorithms.

Zikai Wang received the BS degree from Data Science and Big Data Technologies, Renmin University of China, in 2021. He is now working toward the master's degree with the School of Information, Renmin University of China. His research interests include distributed databases, distributed transactions and consensus algorithms.

Yunpeng Chai received the BE and PhD degrees in computer science and technology from Tsinghua University, in 2004 and 2009, respectively. He is a professor with the School of Information, Renmin University of China. His research interests include key-value storage systems, the emerging storage devices, and cloud resource allocation.

Xin Wang (Member, IEEE) received the BE and PhD degrees in computer science and technology from Nankai University, in 2004 and 2009, respectively. He is currently a professor with the College of Intelligence and Computing, Tianjin University. His research interests include knowledge graph data management, graph databases, and Big Data distributed processing. He is a member of the IEEE Computer Society.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** www.computer.org/csdl.

Authorized licensed use limited to: INDIAN INSTITUTE OF TECHNOLOGY KHARAGPUR. Downloaded on March 20,2025 at 14:10:55 UTC from IEEE Xplore. Restrictions apply.