Dependency Preserved Raft for Transactions

Zihao Zhang[1], Huiqi Hu[1(✉)], Yang Yu[1], Weining Qian[1], and Ke Shu[2]

[1] School of Data Science and Engineering, East China Normal University, Shanghai, China
{zach_zhang,yuyang}@stu.ecnu.edu.cn, {hqhu,wnqian}@dase.ecnu.edu.cn
[2] PingCAP Ltd., Beijing, China
shuke@pingcap.com

Abstract. Modern databases are commonly deployed on multiple commercial machines with quorum-based replication to provide high availability and guarantee strong consistency. A widely adopted consensus protocol is Raft because it is easy to understand and implement. However, Raft's strict serialization limits the concurrency of the system, making it unable to reflect the capability of high concurrent transaction processing brought by new hardware and concurrency control technologies. Upon realizing this, the work targets on improving the parallelism of replication. We propose a variant of Raft protocol named DP-Raft to support parallel replication of database logs so that it can match the speed of transaction execution. Our key contributions are: (1) we define the rules for using log dependencies to commit and apply logs out of order; (2) DP-Raft is proposed for replicating logs in parallel. DP-Raft preserves log dependencies to ensure the safety of parallel replication and with some data structures to reduce the cost of state maintenance; (3) experiments on YCSB benchmark show that our method can improve throughput and reduce latency of transaction processing in database systems than existing Raft-based solutions.

Keywords: Consensus protocol · Log replication · Log dependency

1 Introduction

To solve a variety of fault tolerance problems and ensure the availability of distributed database systems, multiple replicas are required. How to keep consistency among replicas is a challenge [10,18]. The traditional approach adopts primary-backup replication [17] to replicate writes from primary replica node to backup nodes. To achieve strong consistency, the replication process is synchronous, which sacrifices performance. In contrast, to achieve better performance, asynchronous replication is adopted, which does not guarantee consistency between primary and backups. Therefore, primary-backup replication requires a compromise between strong consistency and high performance.

Recently, many transactional systems require both high-performance and strong consistency to build mission-critical applications. A common way is to

Y. Nah et al. (Eds.): DASFAA 2020, LNCS 12112, pp. 228–245, 2020.
https://doi.org/10.1007/978-3-030-59410-7_14

use quorum-based replication to maintain a consistent state machine, which is a database instance in each replica. By storing a series of database logs and executing the logs in sequence, the same state can be ensured among replicas. Quorum-based replication only requires replicating to majority nodes, instead of waiting for all backup nodes' response like in primary-backup replication so that it can achieve high performance. Besides, using consensus protocols can guarantee strong consistency, and tolerate up to $\lceil N/2 \rceil - 1$ failures. Therefore, quorum-based replication makes a good balance on performance and consistency. Consensus protocols such as Paxos [11] and Raft [14] are used to keep the log consistent in quorum-based replication. Paxos is a classical algorithm to ensure distributed consensus, but due to the high complexity, many works devoted to simplifying it so that it is easy to understand and implement. Raft is one of them, it can achieve strong consistency as Paxos, and much easier to be implemented. As a result, Raft is widely used in commercial systems like etcd [1], TiDB [2] and PolarFS [5] in recent years.

Unfortunately, to make Raft is an understandable protocol, there are some restrictions that make Raft highly serializable. Logging in Raft is in a sequential order, which means logs should be acknowledged, committed, and applied in sequence, and log hole is not allowed, this dramatically limits the concurrency of systems. Nowadays, multi-core CPU and large memory are commonly used to improve the performance of database. To make better use of hardware resources, concurrency control strategy is adopted to improve the concurrency of transaction execution. Furthermore, in production systems, logs usually be replicated in a pipeline way [13]. After leader sends a log entry to followers, it can send the next one instead of waiting for the response of the last entry. Moreover, when there are multiple connections between leader and followers, follower may receive out-of-order logs. In other words, some logs may reach follower earlier than those in front of them. All of these strategies are to improve system's concurrency, but with the restriction of Raft, the acknowledgment in follower must in order, which causes the serial commit of transactions. This is even worse when the network is poor. If some connections are blocked, followers cannot receive log entries and reply to leader. As a result, leader may also be stuck when majority followers are blocked, causing the system to be unavailable. So even with high concurrency in transaction execution, it still needs to wait for replicating logs serially.

To improve the performance of log replication with Raft, so that it can match the speed of transaction execution, we need to modify Raft to replicate logs in parallel. Recently, a variant protocol of Raft named *ParallelRaft* is used in PolarFS [5], it can replicate logs out-of-order. It is designed for file system and has no guarantee on the order of completion for concurrent commands. Therefore, it does not satisfy transaction's commit semantics and cannot be used in OLTP database systems directly. What's more, it adds a *look behind buffer* into log entry which contains the LBA (Logical Block Address) modified by the previous N log entries, N is the maximum size of a log hole permitted. With this information, it can check if an entry's write set conflicts with the previous to decide if the entry can be applied out-of-order. If the N is too large, it will greatly increase the size of log entry and waste bandwidth.

In this paper, we propose a new variant called *DP-Raft* (dependency preserved Raft) for transactions to support concurrent log replication which will gain high performance in OLTP systems. We redesign Raft to overcome the shortcomings of low parallelism so that it can replicate logs in parallel. In the meantime, we preserve the dependency of log entries to support commit semantics of transactions and make sure the correctness of log applying. Our contributions can be summarized as follows:

- Based on the analysis of log dependencies, we form the requirements to satisfy transaction commit semantics and apply safely in parallel replication. In short, commit order must follow the order of RAW dependency; and to apply logs to a consistent state machine, WAW dependency must be tracked.
- We design DP-Raft to preserve dependencies between log entries and use dependency information to acknowledge, commit and apply log entries out-of-order, which significantly improves Raft's parallelism.
- Experiments show that our new approach achieves a better performance than Raft and ParallelRaft.

The paper is organized as follows: Section 2 gives some related works. Section 3 analyzes if logs can be committed and applied with dependencies. In Sect. 4, we describe DP-Raft in detail. Experimental results are presented in Sect. 5. Finally, we conclude our work in Sect. 6.

2 Related Work

Lamport proposed Paxos [11] to guarantee the consistency in distributed systems, which is a general-purpose consensus protocol for asynchronous environments and has been proven to ensure distributed consensus in theory. Paxos has no primary node, so each request is a new Paxos instance, and must go through two phases to be accepted in the majority. To resolve the problem that Paxos unable to serve consecutive requests which is required in practical, many works devoted to Paxos variants. Multi-Paxos [12] remove the proposal phase to reduce the round number for every single request by electing a leader, who can serve requests continuously. As a result, Multi-Paxos is widely used in distributed database systems like Google's Spanner [7] and MegaStore [4], Tencent's PaxosStore [19] and IBM's Spinnaker [16].

Despite this, Paxos is still difficult to understand and implement correctly. Raft [14] is a consensus protocol designed for understandability. It separates the protocol into leader election and log replication. Leader in charges of processing requests and replicating the results to followers. Raft introduces a constraint that the logs must be processed in serial. This constraint makes Raft easy to understand but also limits the performance of concurrent replication. Because of the simplicity of implementation, many systems such as etcd [1], TiDB [2] and PolarFS [5] use Raft to provide high availability.

As mentioned above, the strict serialization of Raft limits systems' concurrency. Many researchers are working on it to improve Raft's performance.

PloarFS [5] proposed ParallelRaft which supports concurrent log replication. But it is designed for file systems, so out-of-order replication is implemented by recording the write block address of the previous N entries in each log entry. Only checking conflict of write is not enough for commit semantics of transaction, and maintain the write set of N entries in each entry is too expensive for memory and network if we want to get higher concurrency by increasing parameter N. TiDB [2] optimizes Raft in another way, it divides data into partitions, and each partition is a single Raft group to replicate. By using multi-group Raft, it balances the workload in multiple servers, which improves the concurrency of the whole system. Vaibhav et al. [3] discussed the issue of consistently reading on followers so that followers can share the workload of leader to avoid leader becoming a performance bottleneck. DARE [15] gave another way to speed up replication with hardware. It uses RDMA to redesign the Raft like protocol to achieve high performance.

3 Parallel Committing and Applying

The purpose of keeping the restriction of log order in Raft is to make sure of the correctness of apply and finally reach a consistent state among different servers. But in a transactional database system, it is not necessary to guarantee this strict constraint. In database systems, we use Raft to replicate the log entry which records the results of a transaction when the transaction enters commit phase. After the transaction log is committed, the commit of transaction is complete. We keep the log order the same as transaction commit order so that replay log we can achieve the same state. But when transactions do not have a dependency, which means that they do not read or write on the same data, they can be executed in any order. Raft can benefit from the out-of-order execution between such non-dependent transactions, the log entries of these transactions also have no dependencies, so there is no need to strictly acknowledge, commit, and apply them in order. But to ensure the correctness of the state, it is still necessary to keep the order between dependent logs. To do this, we need to analyze how to commit and apply in parallel with dependency.

To correctly commit and apply logs to a consistent state, the relative order among log entries of conflicting transactions must comply with the transactions commit order (the commit order of transactions can be generated by any concurrency control mechanism). It indicates that dependency should be tracked during log processing. We will demonstrate how to use log dependency to guarantee the correctness of Raft in parallel. For the sake of illustration, we use T_i to represent a transaction and L_i as the log entry of transaction T_i. For each transaction, we generate one log entry to record the results of the transaction.

Figure 1 shows four dependencies that may exist between two transactions. In each case of dependency, the logs can be acknowledged by followers in an out-of-order way, but whether it can be committed or applied is determined by the type of dependency.

Dependency	Log Order	Can commit in parallel?	Can apply in parallel?
$T_1 \xrightarrow{WAW} T_2$	$L_1 < L_2$	yes	no
$T_3 \xrightarrow{RAW} T_4$	$L_3 < L_4$	no	yes
$T_5 \xrightarrow{WAR} T_6$	$L_5 < L_6$	yes	yes
$T_7 \xrightarrow{Null} T_8$	$L_7 < L_8$	yes	yes

Fig. 1. Four dependencies may exist between transactions. For the correctness of state machine, the order of committing must track RAW dependencies and the order of applying must track WAW dependencies.

Write after write dependency (WAW): For T_1 and T_2 with WAW dependency, it means that T_2 writes the data which has been modified by T_1. In this scenario, the log entries cannot be applied out-of-order. This is because according to the execution order, the results in L_2 must be written after L_1. If we apply L_2 first, then L_1 will overwrite L_2's update results, which causes an incorrect state. If both logs have been committed, simply applying them in order will reach a consistent state, regardless of the order of commit. And if someone commits failed, it can be ignored and the applied state is consistent as if the failed log entry has never existed. So L_1 and L_2 can commit in parallel.

Read after write dependency (RAW): For T_3 and T_4 with RAW dependency, T_4 reads the result that T_3 wrote. In this scenario, L_4 cannot commit before L_3. If L_4 commits first and then L_3 commits failed, T_4 will read a data that never exists, it is called *DirtyRead*, which is not allowed in transaction processing. When both T_3 and T_4 have been committed in sequence, because they do not update the same data, they can be applied in parallel, which will not cause an inconsistent state.

Write after read dependency (WAR): When T_6 updates the same data that T_5 has read before, there is a WAR dependency between them. In this case, L_5 and L_6 can commit and apply in parallel. Because T_5 has read what it should read, so T_6's update will not affect the result of T_5, they can commit without order. And with the different data they update, they can also be applied in any order. So the WAR dependency need not to track for DP-Raft.

No dependency (Null): If two transactions do not read or write on the same data, as shown in scenario (4), there is no dependency between them, and the log can be committed and applied out-of-order in DP-Raft.

Based on the above analysis, only WAW and RAW dependency of log entries need to be tracked for DP-Raft to commit and apply. When RAW dependency exists, log entries should be committed in sequence, and for WAW dependency, log entries should be applied in order.

4 Dependency Preserved Raft

This section proposes a variant of Raft algorithm for transactions, named as *DP-Raft*, which is independent of concurrency control strategy. It preserves the dependency of transactions in their log entries and uses dependency to decide that whether logs can be committed or applied without order. Because DP-Raft makes the log replication process more parallel, it is suitable for multiple connections between servers and can effectively utilize the network bandwidth. Therefore, it will significantly reduce the latency of log replication.

In the following, we first present related data structures and how to analyze log dependency, then the design of *log replication* and *leader election* algorithms will be introduced.

4.1 Structures of DP-Raft

Log Buffer. DP-Raft uses a ring buffer to store log entries, as shown in Fig. 2, every node has such a log buffer. DP-Raft allows log hole to exist in the log buffer. On followers, log hole means that an entry is empty, but there are non-empty log entries before and after it; on the leader, log holes represent uncommitted entries that exist between committed log entries. Section 3 has mentioned that to ensure the correctness of committing and applying in DP-Raft with log hole, log dependency should be tracked. But if we allow hole log entries to exist without limits, it will cause two problems: (1) According to the strategy that will be introduced in Sect. 4.2, we should not only maintain the write set of hole log entries but also all entries that depend on them. So if we do not limit the range that hole entry exists, the write set will be too large to maintain; (2) Because of the existence of hole entries, we can not use *commitIndex* to represent the status of all entries before this index like Raft. The index of last committed log entry cannot indicate the status of the previous log entries, so we should record all entries' status so that we can know which is committed. This is also difficult to maintain if hole entries appear in a large range.

To solve the problems mentioned above, we introduce a sliding window whose size can be modified. We only allow hole entries in the sliding window, which will limit the range that hole entries may exist. As a result, the maintenance

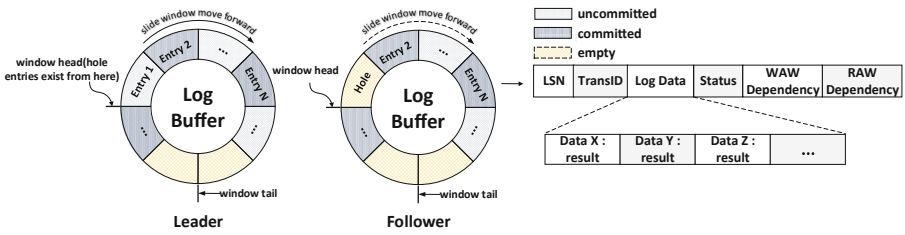

Fig. 2. The structures of log buffer and log entries in DP-Raft. The left one represents the log buffer in leader, and the right one is in follower.

of the write set and entries's status will be simple. Figure 2 shows the window head in leader represents the first uncommitted entry which is also the start point that hole entries can exist in followers. The window head is the same as *commitIndex* in Raft, all entries before window head are committed. The head can move forward when one entry and all the previous are committed. If leader receives acknowledgment of an entry from majority followers, this entry can commit even if someone is uncommitted before it, like Entry 2 in leader's buffer. Reason for Entry 1 not committed may be that more than half followers have not received the replicating request of Entry 1, so the slot of Entry 1 is empty in follower now. But follower receives Entry 2 first, it stores the entry and sends an acknowledgment back to leader, then the slot of Entry 1 will be a log hole. Section 4.3 will explain the replication process in detail.

Comparing to the strategy that allows holes without limitation, we narrow the range of maintaining the write set and entries' status to a sliding window, which significantly reduces the difficulty.

Log Entry. Each log entry has a unique *log sequence number* (*LSN*), which is monotonically increasing and also used as log entry index for DP-Raft. When a transaction is received, leader will process the operations in the transaction and pre-applies the results to memory. Once the transaction enters the commit phase, an LSN will be acquired, and then generates a log entry to reflect the results of the transaction. Log entry format is shown in Fig. 2. We adopt the common *value log* like in [8], so *Log Data* stores the results of the transaction, indicating which data has been modified and its new value. In addition to *Log Data*, a log entry also includes LSN and the transaction ID. *Status* indicates if the entry is committed and applied. Moreover, in order to track log dependency, log entry also stores indexes of WAW and RAW dependent log entries at *WAW* and *RAW Dependency*. With this information, leader can decide if the entry can be committed and followers can decide if it can be applied to the state machine. Compared to ParallelRaft [5] which stores the write set of the previous N entries, storing dependent entries' index will reduce the size of log entries and can satisfy transaction's commit semantics when committing in parallel.

4.2 Dependency Analysis

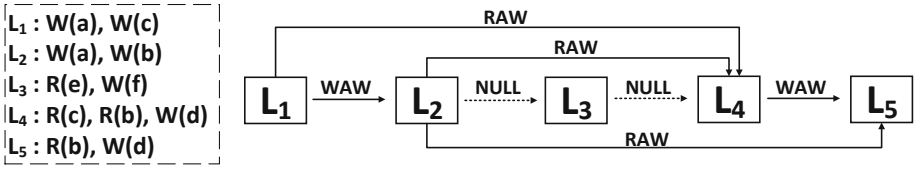

Fig. 3. Transaction dependency graph.

KuaFu [9] has a similar idea about enabling logs to be applied concurrently to reduce the gap between primary and backups by constructing a dependency

graph of logs, which will introduce overheads for database systems. While DP-Raft simply analyzes dependency and records dependency in each entry instead of maintaining a dependency graph. Here is an example of log dependency in Fig. 3. The read and write sets of five entries are shown on the left side. Because L_1 and L_2 both write data a, so there is a WAW dependency between L_1 and L_2. For L_1, L_2 and L_4, L_4 read data b and c, which are updated by L_1 and L_2, so L_4 has a RAW dependency with L_1 and L_2. Other log entries' dependencies are also analyzed in the same way.

Once a transaction enters the commit phase, leader will generate a log entry and add it into a slot in log buffer (once an entry is added into log, it will never be overwritten, so DP-Raft guarantees the *Leader Append Only* property). During this phase, log dependency is analyzed. DP-Raft stores the write set of hole entries and entries depend on them in the sliding window. So when L_5 enters commit phase, the write sets of L_1, L_2 and L_4 are stored. Because L_5 reads data b, we find b was last modified by L_2, so it has a RAW dependency with L_2. And d in L_5's write set was updated by L_4 recently, so it connects with L_4 on WAW dependency. If the read or write set of L_5 is overlapping with an entry which is ahead window head, we will ignore it, because the entry is committed so out-of-order commit and apply will not happen. It should be noted that for an entry that does not depend on both hole entries and those who have dependencies with holes, the write set of the entry is no need to be recorded, like L_3.

4.3 Log Replication

After leader adds an entry into its local buffer, it can replicate the entry to followers asynchronously. Log replication in DP-Raft is quite different from Raft [14]. When leader of Raft replicates log entries, the *commitIndex* will be sent together to inform followers of the current commit status of entries. But DP-Raft does not have *commitIndex*, it needs to store the status of all entries in the sliding window. To simplify the cumbersome maintenance, DP-Raft introduces a *commitFlag* which is a bitmap, wherein each bit of the flag represents the status of an entry in the sliding window, with 1 for committed, and 0 for uncommitted. When a replicating request is sent, *commitFlag* and *windowHead* are attached to the request to replace *commitIndex* of Raft. The entire process of log replication will be divided into three parts.

Followers Acknowledge in Parallel. In Raft, when a follower receives a request to append entries, it must ensure the consistency of the log in leader and itself by checking if the last entry is the same. Only when the checking is successful, it can accept the new entries. If the previous entry is missing, follower needs to wait until the previous one is acknowledged. The checking ensures the *Log Matching* property, which is one of the properties to guarantee the safety of Raft. But in DP-Raft, this constraint is removed, it is no need to wait even if some holes exist before. When follower receives new entries from leader, it just adds them into its log buffer based on the entry's LSN and sends an acknowledgment back. In DP-Raft, follower can acknowledge entries in parallel, thus

Algorithm 1: Commit and Apply in Parallel

```
   Input: the entry to be committed or applied
   /* leader commits in parallel                                        */
 1 Function ParallelCommit()
 2     if entry.replicated_count ≥ majority then
 3         for dependent_entry ∈ entry.RAWDependency do
 4             if dependent_entry.status == commited or applied then
 5                 can_commit ← true;
 6             else
 7                 can_commit ← false;
 8                 break;
 9         if can_commit then
10             entry.status ← committed;
11             new_flag ← commitFlag — 1<<(entry.LSN - windowHead);
12             atomic-cas(&commitFlag, commitFlag, new_flag);
   /* followers apply in parallel                                       */
13 Function ParallelApply()
14     if entry.status == committed then
15         for dependent_entry ∈ entry.WAWDependency do
16             if dependent_entry.status == applied then
17                 can_apply ← true;
18             else
19                 can_apply ← false;
20                 break;
21         if can_apply then
22             DoApply();
23             entry.status ← applied;
```

reducing the extra waiting time even when the network is jittery and entries are received out-of-order, so the latency of log replication is significantly reduced.

Even without checking when the previous entry is missing, the *Log Matching* property is still ensured by DP-Raft. If two entries have the same index and term, the two entries and all entries before their index are considered as the same. So Raft ensures this property by comparing the *prevTerm* and *pervIndex* in request to follower's local last entry when acknowledges the replicated entries from leader. In contrast to this, DP-Raft relaxes this constraint. It skips the check phase and saves the entry directly if the previous one is missing, but the previous entry's information is also stored. When the previous entry arrives later, it will be compared with the previous information stored before to see if matches. If not, the entry located behind will be discarded. As a result, follower cannot have different entries from leader eventually, so this property is ensured.

Leader Commits in Parallel. After leader sends an append entries request to followers, it waits for responses until majority acknowledgments are received. After this, leader can commit the entry and the transaction associated, then replies to client to inform the result of the transaction. The sequence restriction of Raft asks leader to commit entries in order, but in DP-Raft, entries can be committed in parallel. Section 3 has mentioned that parallel commit should obey the sequence of RAW dependency, so DP-Raft must ensure all the RAW dependent entries are committed, so that DP-Raft can commit the entry, as lines 3–8 in Algorithm 1 show. Checks on dependent entries enable DP-Raft to satisfy transaction commit semantics, which is different from ParallelRaft [5] that do not ensure the commit order between dependent entries. DP-Raft maintains a *commitFlag* to represent the status of entries in the sliding window, so leader will update *commitFlag* to set the bit of the commit entry to 1 (line 11) after committing it. To be mentioned, we use atomic *CAS* operation to modify *commitFlag*. Because the flag is a high contention resource when commit in parallel, so the latch-free *CAS* operation will reduce the conflict on the mutex which used to protect *commitFlag*. Next time leader appends entries or sends a heartbeat, *commitFlag* and *windowHead* will be attached to inform followers of which entry has been committed so that followers can apply it into their state machine.

Followers Apply in Parallel. After follower receives *commitFlag* and *windowHead*, it will check the status of log entries. For entries before *windowHead*, which indicates that they are committed, if some of them are still missing, follower has to notify leader to send the missing entries again so that log holes in follower can be filled. For entries in the sliding window, follower will update status based on *commitFlag*. If an entry is set to committed, it can be applied. Same as acknowledge and commit, apply also need to do in strict order for the correctness of state machine in Raft. As for DP-Raft, follower apply entries in parallel based on the dependency information in each entry. Lines 14–20 in Algorithm 1 show that, before applying an entry, the WAW dependent log entries need to be checked if they are applied. If so, DP-Raft can apply it; otherwise, it needs to wait for the dependent entries to apply.

Log consistency is ensured by *Log Matching* property, so if we want to make sure *State Machine Safety*, the order to apply logs also needs to be consistent. As described in Sect. 3, only the entries without WAW dependency can be applied in parallel, so *DP-Raft* can make sure the correctness of state machine by preserving dependency and applying write conflict entries according to the WAW dependency order.

4.4 Leader Election

Raft's leader is a strong leader who responsible for generating log entries and replicating them to followers, thus its log is the newest and must contain the complete committed log entries, which is called *Leader Completeness*. This property is easy to be ensured in Raft because with no log holes, the one who has the

most up-to-date log will be elected as leader, which means the new leader has the complete logs. But the new leader in DP-Raft may lose some committed entries because of log holes. Therefore, an extra merge stage can make sure that the new leader will find all missing entries from other servers and become a complete leader. DP-Raft obeys the similar rules of Raft to vote, but when a candidate receives majority votes, it still unable to provide service until all holes in its log are filled. So DP-Raft introduces a new state called *pre-leader*, after pre-leader merges logs to fill its log holes, it can transform into a real leader.

Request Votes. When leader election is triggered, follower may transform into candidate and sends a voting request. In the request, the term and index of the lastest log entry will be included. Different from Raft, because the merge stage needs to merge the entries from the first uncommitted in pre-leader to the latest, so the first uncommitted log entry's index of candidate will also be attached to the voting request in DP-Raft. When the voting request is received, DP-Raft has two constraints to vote: (1) Follower first checks the term and index of the last log entry to see if the candidate's log is at least the same new as itself. If the candidate has a newer log, it grants vote; if candidate falls behind, it will refuse to vote; also if they are the same new, follower will use constraint (2) to see the complete status of candidate's log. (2) If the first uncommitted index of candidate is smaller than the local first uncommitted entry's index, it replies false; otherwise, it replies true to vote. By the first constraint, logs of the elected per-leader are at least as up-to-date as logs in majority servers, and only one pre-leader could be elected, which satisfies the *Election Safety* property. Besides this, the second constraint makes sure the number of log entries to be merged is as small as possible.

Merge Log Entries. According to the aforementioned, the pre-leader can transform into leader after the merge stage. In order to merge, followers need to send log entries between the first uncommitted index of pre-leader and the lastest

Is missing in pre-leader?	Is the entry committed?	Other conditions	Decision	Example in Fig.5.
No	Yes	(1) local entry committed	use the local entry	Entry 5
		(2) local entry uncommitted, but committed on other node	merge the committed entry to local log	
	No	(3) empty on all other nodes	use the local entry	Entry 7
		(4) different entry on other nodes	choose the entry with the highest term	Entry 9
Yes	Yes	(5) committed on other node	merge the committed entry to local log	Entry 4
	No	(6) empty on all other nodes	set the entry to NULL	Entry 6
		(7)different entry on other nodes	choose the entry with the highest term	Entry 8

Fig. 4. Seven conditions and their decisions during merging.

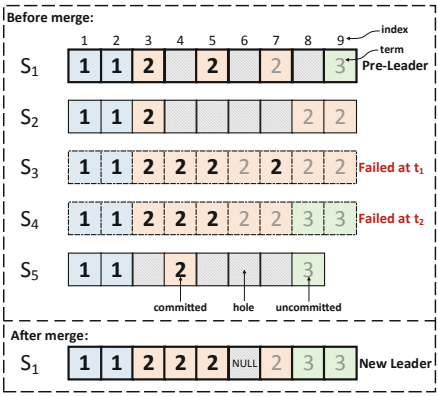

Fig. 5. An example of merging. S_3 and S_4 failed, S_1 is pre-leader now.

index in local to pre-leader. Imagining a follower was far behind leader but received the newest log entry before leader failed. If this follower starts an election, it is likely to be elected as the pre-leader because it has the newest entry. In this scenario, the number of entries to be sent for merging is too large, which wastes network bandwidth and increase the merging time. So the second constraint of voting ensures the range to be merged is small, so that the merging is vary fast.

After pre-leader receives log entries from majority servers, it starts to merge. There are several conditions of an entry as shown in Fig. 4. For a committed entry, it at least exists on one server if the system is still available. This means the committed entries can always be found on someone. In scenario (1), if an entry on pre-leader is not empty, and the entry is committed, it just use the local entry; but if a committed entry is uncommitted or missing on pre-leader (scenario (2) and (5)), the commited entry can be found at least on some other server, then pre-leader merges the committed entry to its own log. For an uncommitted entry, if pre-leader holds it, but it is missing on all other servers, pre-leader should preserve it. Because the entry may has been replicated to majority servers and committed, but has not notified the followers to commit, then some servers failed include leader. Now the pre-leader is the only one who has this entry and survived, so this entry cannot be abandoned, as shown in scenario (3). But if one entry is missing on both pre-leader and other servers, which means it cannot be committed, so we can ignore it and set the entry as NULL (scenario (6)). In the last two scenarios, (4) and (7), if an entry is uncommitted and is different among servers, pre-leader will choose the one with the highest term.

To understand how to merge more clearly, we give an example in Fig. 5. S_1-S_5 represent different servers, and entries with bold number and normal number represent committed and uncommitted entries respectively. At first S_3 was leader in term2, but it failed at t_1, then S_4 got votes from S_1 and S_5 to be the leader, and replicated entry 8 and 9 to others. At t_2, S_4 failed, S_1 started leader election and be voted by S_2 and S_5, so S_1 is the pre-leader now. After got entries between 4 and 9 from other servers, S_1 starts the merge stage. For entry 4, missing in its local log, but is committed on S_5, so S_1 merges S_5's entry 4 to its local log; entry 5 is stored and committed in local, it will be skipped; entry 6 is missing on all three servers, it is set to NULL and ignored; entry 7 is only stored in pre-leader, it will be preserved; for entry 8 and 9, different servers hold different entries, so two green entries of term 3 will be chosen. After the merge stage, logs in pre-leader is shown at the bottom of Fig. 5, then S_1 can provide service.

5 Evaluation

We implement DP-Raft in a database prototype with about 8300 lines Golang codes to verify the efficiency of our method. We adopt *Optimistic Concurrency Control* (OCC) as our concurrency control mechanism. The experimental setup and the benchmark used in the evaluation are given below.

Cluster Platform. We deploy the prototype on 7 machines, each of them is equipped with a *2-socket Intel Xeon Silver 4100 @ 2.10 GHz processor* (a total of 16 physical cores), and *192 GB DRAM*, connected by a *10 GB switch*.

Competitors. We compare DP-Raft to the standard Raft [14] and ParallelRaft [5]. Because ParallelRaft is designed for file system and is not suitable for transactional database systems, so we modify it by recording both the read and write key of a transaction in its log entry to replace the logical block address when used in file systems. When an entry enters the commit phase, it will check to see if the read and write set are overlapping with other entries, so that to support commit semantics of transaction when commit in parallel.

Benchmark. We adopt YCSB [6] to evaluate three methods. The schema contains a single table with each record is 100 bytes. The table is initialized to consist of 10 million records. To verify the performance of DP-Raft with RAW and WAW dependency, we modify the workload with one read and one write operation in each transaction, so that each log entry may have RAW and WAW dependencies with other entries.

5.1 Replication Performance

We first measure the throughput and latency under the YCSB workload. All methods have three replicas and we varying the client number to see the replication performance.

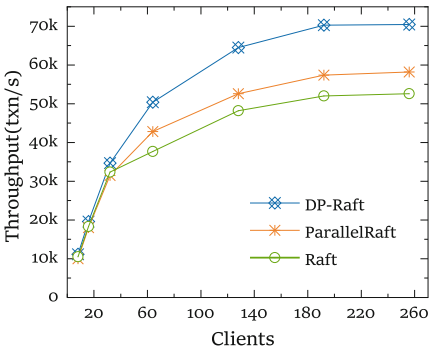

Fig. 6. Throughput of YCSB.

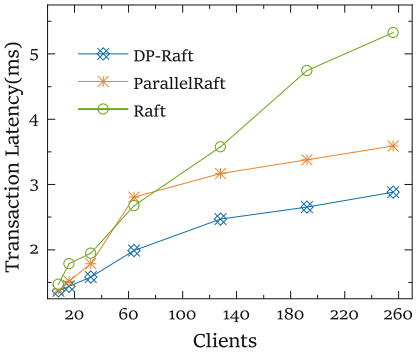

Fig. 7. Latency of YCSB.

The experiment results are shown in Fig. 6 and Fig. 7. In the beginning, all three methods grow quickly, but when the number of clients exceeds 32, the trend of Raft becomes slow because it must replicate logs one by one. Due to the parallel replication, DP-Raft and ParallelRaft will not wait for the result of previous entries and can utilize the bandwidth to replicate logs, therefore their throughput is higher than Raft. But because ParallelRaft needs to maintain the

status of all entries from the first uncommitted, and need to compare both read and write set when committing an entry, which consumes a lot of CPU time and weighs on its throughput. On the contrary, DP-Raft only needs to check if the RAW dependent entries are committed and uses a *commitFlag* to record entries' status, so the low overhead brought by these inspires the throughput of DP-Raft, which is 1.34× than Raft and 1.21× than ParallelRaft. As for latency, Raft's serial replication causes a high latency because later entries have to wait. DP-Raft can replicate entries concurrently, so the latency gap between DP-Raft and Raft almost reaches to 50%. Although ParallelRaft can also replicate in parallel, due to the high overhead in commit checking and status maintenance, its latency is still 20% higher than DP-Raft.

5.2 Skewed Workload Results

Because DP-Raft's commit needs to obey the order of RAW dependencies, it is obvious that skewed data accesses will impact on the performance. So we vary the skewness θ of the Zipfian distribution to see how the performance changes. We experiment by using 256 clients and the same workload as in Sect. 5.1 under three replicas. Figure 8 gives the result that when $\theta < 0.6$, the performance is stable, but when $\theta > 0.7$, all three methods' throughput decrease sharply. This is because, with θ increasing, some records become "hot" and are frequently accessed, so the chance of a transaction conflicting with others is significantly increased. To be mentioned, the performance is influenced by both concurrency control mechanism and log replication, and because Raft does not check the conflict of log entries, so we display the normalized performance of DP-Raft and ParallelRaft against to Raft in Fig. 9 to see the influence of tracking dependency when commit. We can observe that before 0.8, DP-Raft achieves better performance compared to Raft, and when θ is 0.99, the throughput of DP-Raft is only half of Raft. The reason is that if θ is large, an entry is likely to have RAW dependencies with others which leads to a high probability of being blocked to wait for other entries to be committed. This is even worse for ParallelRaft

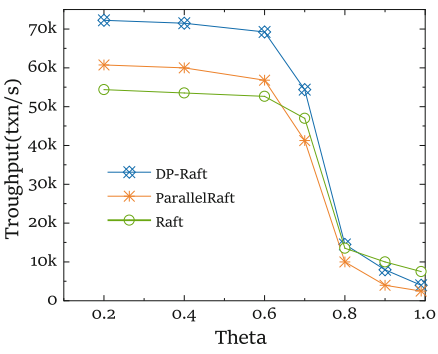

Fig. 8. Skewed workload.

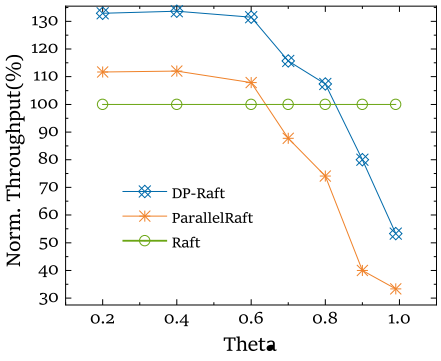

Fig. 9. Skewed workload (normalized).

because it should check if the read and write set are overlapping between entries, which means more kinds of dependencies should be checked, so it is more likely to be blocked. As a result, its throughput is only 30% of Raft.

5.3 The Number of Replicas

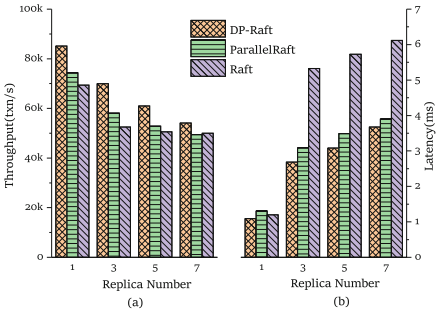

Fig. 10. Performance over replicas number.

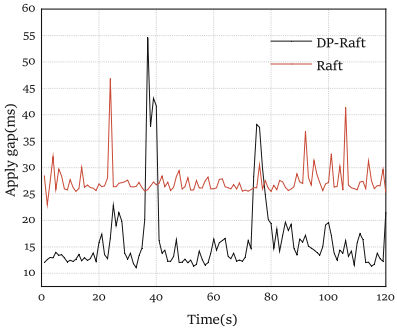

Fig. 11. Apply gap in 120 s.

We further explore the impact of the number of replicas on performance to verify the scalability of three methods. We measure the throughput and latency under different replica numbers with 256 clients and the θ is fixed to 0.5. Results in Fig. 10(a) show that the performance decreases significantly when the replica number changes from one to three. This is because transaction's commit under three-replica needs to go through the network, so the latency is obviously increased, as Fig. 10(b) shows. When the replica number changes to 5 and 7, Raft's performance remains stable, while DP-Raft and ParallelRaft have declined, but DP-Raft still outperforms Raft. The reason for the performance degradation is that with more replicas, DP-Raft and ParallelRaft need to replicate to more servers concurrently and process their results, so the efficiency of replication is reduced. In Fig. 10(b), we can see the latency of Raft is the highest. Therefore, with 3 or 5 replicas that are commonly adopted in practical, DP-Raft has the highest throughput and the lowest latency than ParallelRaft and Raft.

5.4 Apply Gap

Recall from Sect. 4.3, DP-Raft enables apply in parallel, so we use multiple threads to apply logs, which will narrow the gap between the state of leader and followers. We define t_l and t_f as the time of leader and follower apply the log entry of a transaction. Once a server applies an entry to its state machine, the results of the transaction are visible. So the gap between the same visible state of leader and follower is donated as $t_f - t_l$. We measure the apply gap in

120 s as shown in Fig. 11. The result shows that DP-Raft achieves a 2× lower apply gap than Raft, which means followers in DP-Raft can reach the up-to-date state in a short time after leader modified its state. We also notice that there are several jitters in DP-Raft's apply gap, by our analysis, it is because some hole entries on "hot" records block the later entries to apply. Once the hole entries are filled, the apply gap will return to a normal range. Therefore, apply in parallel can minimize the visible gap between leader and followers, which can benefit the read operations on followers to see the most up-to-date state as soon as possible.

5.5 Leader Election Time

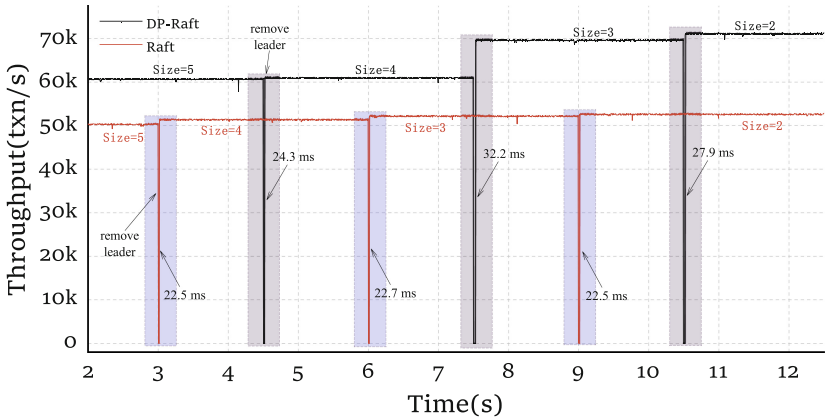

Fig. 12. Leader election in 12 s.

As mentioned in Sect. 4.4, to ensure the *Leader Completeness* property, pre-leader needs to merge followers' log and its own log to fill all the missing entries. An additional merge stage will increase the leader election time. To measure the time cost of the merge stage, we shut down leader three times in 12 s and record the time taken by each election, the result is shown in Fig. 12. Election time contains heartbeat timeout (20 ms by default) and the time to elect leader. There are five replicas initially. At 4.5 s, leader is shut down and trigger a leader election, the first election of DP-Raft takes 24.3 ms and merges 32 entries. Then leader is shut down again at 7.5 s, this time uses 32.2 ms to elect a new leader with 256 entries to be merged. And the third election at 10.5 s merges 128 entries and takes 27.9 ms. The reason for the difference in election time is that each election merges a different number of entries. Same as DP-Raft, we shut down leader three times for Raft at 3 s, 6 s, and 9 s. Each of the elections takes 22.5, 22.7 and 22.5 ms separately. Because Raft does not need to merge, so its election time is stable. We can observe even in the bad case as the second election of DP-Raft, it takes 10 ms more than Raft, which is acceptable because leader election

is not frequent. What's more, we notice that for DP-Raft, when the size of consensus group is 5 and 4, the throughput is almost the same because both of them need to replicate to at least 3 servers. But when size is reduced to 3 at 7.5 s, the throughput rapidly increases because it only needs to replicate to 2 servers. As for Raft, the throughput remains stable regardless of the group size, which is the same as the result in Sect. 5.3.

6 Conclusion

In this paper, we target on how to break the strict serialization of Raft to replicate database logs concurrently. We propose a new variant of Raft protocol named DP-Raft for transactional database systems, which use log dependency to ensure the safety of out-of-order replication. In short, DP-Raft satisfies commit semantics of transaction by tracking RAW dependency and committing logs in RAW dependent entries' order, and to ensure the safety of the state, DP-Raft applies entries based on the order of WAW dependent entries. Experimental results demonstrate that DP-Raft can significantly improve the throughput of transaction processing and reduce transaction latency.

Acknowledgments. This work is partially supported by National Key R&D Program of China (2018YFB1003404), National Science Foundation of China under grant number 61672232, and Youth Program of National Science Foundation of China under grant number 61702189.

References

1. etcd. https://etcd.io/
2. Tidb. https://pingcap.com/
3. Arora, V., et al.: Leader or majority: Why have one when you can have both? improving read scalability in raft-like consensus protocols. In: HotCloud (2017)
4. Baker, J., Bond, C., Corbett, J.C., et al.: Megastore: providing scalable, highly available storage for interactive services. In: CIDR, pp. 223–234 (2011)
5. Cao, W., et al.: PolarFS: an ultra-low latency and failure resilient distributed file system for shared storage cloud database. PVLDB **11**(12), 1849–1862 (2018)
6. Cooper, B.F., Silberstein, A., Tam, E., et al.: Benchmarking cloud serving systems with YCSB. In: SoCC, pp. 143–154 (2010)
7. Corbett, J.C., Dean, J., Epstein, M., et al.: Spanner: Google's globally distributed database. ACM Trans. Comput. Syst. **31**(3), 8:1–8:22 (2013)
8. Guo, J., Chu, J., Cai, P., et al.: Low-overhead paxos replication. Data Sci. Eng. **2**(2), 169–177 (2017)
9. Hong, C., Zhou, D., Yang, M., et al.: KuaFu: closing the parallelism gap in database replication. In: ICDE, pp. 1186–1195 (2013)
10. Kemme, B., Alonso, G.: Don't be lazy, be consistent: Postgres-r, A new way to implement database replication. In: VLDB, pp. 134–143 (2000)
11. Lamport, L.: The part-time parliament. ACM Trans. Comput. Syst. **16**(2), 133–169 (1998)
12. Lamport, L., et al.: Paxos made simple. ACM SIGACT News **32**(4), 18–25 (2001)

13. Ongaro, D.: Consensus: Bridging theory and practice. Ph.D. thesis, Stanford University (2014)
14. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: 2014 USENIX Annual Technical Conference, USENIX ATC, pp. 305–319 (2014)
15. Poke, M., Hoefler, T.: DARE: high-performance state machine replication on RDMA networks. In: HPDC, pp. 107–118 (2015)
16. Rao, J., Shekita, E.J., Tata, S.: Using paxos to build a scalable, consistent, and highly available datastore. PVLDB **4**(4), 243–254 (2011)
17. Stonebraker, M.: Concurrency control and consistency of multiple copies of data in distributed INGRES. IEEE Trans. Software Eng. **5**(3), 188–194 (1979)
18. Wiesmann, M., Schiper, A., Pedone, F., et al.: Database replication techniques: a three parameter classification. In: SRDS, pp. 206–215 (2000)
19. Zheng, J., Lin, Q., Xu, J., et al.: Paxosstore: high-availability storage made practical in WeChat. PVLDB **10**(12), 1730–1741 (2017)