Front. Comput. Sci., 2022, 16(5): 165612
https://doi.org/10.1007/s11704-020-0210-y

RESEARCH ARTICLE

Efficient and stable quorum-based log replication and replay for modern cluster-databases

Donghui WANG, Peng CAI (✉), Weining QIAN, Aoying ZHOU

School of Data Science and Engineering, East China Normal University, Shanghai 200062, China

© Higher Education Press 2022

Abstract The modern in-memory database (IMDB) can support highly concurrent on-line transaction processing (OLTP) workloads and generate massive transactional logs per second. Quorum-based replication protocols such as Paxos or Raft have been widely used in the distributed databases to offer higher availability and fault-tolerance. However, it is non-trivial to replicate IMDB because high transaction rate has brought new challenges. First, the leader node in quorum replication should have adaptivity by considering various transaction arrival rates and the processing capability of follower nodes. Second, followers are required to replay logs to catch up the state of the leader in the highly concurrent setting to reduce visibility gap. Third, modern databases are often built with a cluster of commodity machines connected by low configuration networks, in which the network anomalies often happen. In this case, the performance would be significantly affected because the follower node falls into the long-duration exception handling process (e.g., fetch lost logs from the leader). To this end, we build QuorumX, an efficient and stable quorum-based replication framework for IMDB under heavy OLTP workloads. QuorumX combines critical path based batching and pipeline batching to provide an adaptive log propagation scheme to obtain a stable and high performance at various settings. Further, we propose a safe and coordination-free log replay scheme to minimize the visibility gap between the leader and follower IMDBs. We further carefully design the process for the follower node in order to alleviate the influence of the unreliable network on the replication performance. Our evaluation results with the YCSB, TPC-C and a realistic micro-benchmark demonstrate that QuorumX achieves the performance close to asynchronous primary-backup replication and could always provide a stable service with data consistency and a low-level visibility gap.

Keywords log replication, log replay, consensus protocol, high performance, high availability, quorum, unreliable network, packet loss

1 Introduction

Replication is the technique that can be used for a traditional database management system (DBMS) or fast, multi-core scalable in-memory database (IMDB) to support high-availability. In this work, we assume a full database copy is held on a single IMDB node, and each backup node has the full replication. In replicated IMDBs, the execution of a transaction is completely in the primary IMDB. Primary-backup replication is the well-known replication method in the database community. The *asynchronous* (*lazy*) primary-backup replication used in traditional database systems (e.g., MySQL, DB2) trades consistency for performance and availability. In contrast, the *synchronous* (*eager*) primary-backup replication trades performance and availability for consistency.

Today's mission-critical enterprise applications in banking or E-commerce require the back-end database system to provide a stable, high-performance and high-availability service without sacrificing consistency. Compared with primary-backup replication, the quorum-based replication (e.g., Multi-Paxos [1], Raft [2], etc.) can guarantee strong consistency, tolerate up to f out of $2f+1$ fail-stop failures, and achieve better performance than the synchronous primary-backup replication because it only requires the majority of replicas to respond to the leader. The quorum-based replication adopts consensus protocols to take more reasonable trade-offs among performance, availability and consistency, and thus it has been regarded as a practical and efficient replication protocol for large scale datastores [3−5].

Quorum-based replication protocols are the natural choice for replicating IMDB as a highly available and strongly consistent OLTP datastore. However, it is non-trivial to translate the quorum-based replication protocol into a pragmatic implementation for industrial use. The basic principle of various quorum-based protocols is that committing a transaction requires its log to be replicated and flushed on the non-volatile storage on the majority of follower replicas. A transaction may take extremely short time to complete its execution in the leader IMDB. But, committing this transaction may take more time to wait its log replicated to the majority of the followers. As a result, the performance of a replicated IMDB significantly depends on the quorum-based log replication which is influenced by many factors.

To achieve read scalability, the followers need to replay

Received May 19, 2020; accepted November 23, 2020

E-mail: pcai@dase.ecnu.edu.cn

committed logs into its memory table (*memtable*) at a fast speed to keep up with the leader's state, so that it could provide fresh data for online analytical processing (OLAP) services. The classic quorum-based replication needs the leader to send followers the maximal committed log sequence number (MaxComLSN), and then followers can commit and replay these logs with LSN smaller or equal to MaxComLSN. *Replaying logs after receiving the specified MaxComLSN leads to that the committed data on followers are visible at a later time than that on the leader all the time, referred to as visibility gap (VGap)*. Without careful designs, VGap would be larger when the leader IMDB is running under a heavy OLTP workload, and generates transactional logs at a high rate.

Modern database systems are increasingly deployed in a cluster of commodity machines connected by networks with less high-end configurations whereby the network Anomalies frequently crop up, including increased network delays and packet loss. Consensus protocols are aware of the problem of network disconnection between some of the servers and propose to elect a new leader to guarantee availability and safety when the old leader cannot connect with the majority.

Anomalies in the unreliable network also bring problems in log replication and replay. For example, when the packet is lost during replication, the logs on the follower node will be discontinuous. Due to log coherency, Raft logs are not allowed to have any hole on both leader and followers. The follower would stop log persistence and replaying and wait for the leader to resend the missed logs, resulting in blocks of the process of normal log replication. Mulit-Paxos replication relaxes the restriction of log coherency, and allows holes exist. Although the normal replication would not be blocked, log replaying in the follower node is still blocked. Also, the relaxation in Paxos also makes the new elected leader that may not have full logs. The leader should fetch missed logs from other servers and commit them before taking over the system to provide service, resulting in a long unavailable time.

In this paper, we present an efficient and stable quorum-based replication framework, called QuorumX, to optimize log replication and replay for IMDB under highly concurrent OLTP workloads. Main contributions are summarized as follows:

- QuorumX combines critical path based batching and pipeline based batching to adaptively replicate transactional logs, which takes into account various factors including the characteristics of transactional workloads and the processing capability of follower.
- We introduce a fast and coordination-free log replay scheme without waiting for the MaxComLSN, which applies logs to memory ahead of time in parallel to reduce the risk of increased VGap. Thus, the replicas is able to provide a fresher data for OLAP queries in the premise of data consistency.
- We analyze the problem caused by network anomalies and redesign the process in the follower node. First, we relax the constraint of *log coherency* and allows holes to exist in logs as Multi-Paxos did, ensuring the log replication would not be affected by network anomalies. Second, we let the follower fetches the lost logs from the leader proactively so that replaying in QuorumX would not be blocked with ensuring a low unavailability time.
- QuorumX has been implemented in Solar [6], an in-memory NewSQL database system that has been successfully deployed on Bank of Communications, one of the biggest commercial banks in China. Extensive experiments are conducted to evaluate QuorumX under different benchmarks.

The paper is organized as follows. Section 3 describes some preliminaries of quorum-based replication, including the overall architecture and processing flows of both the leader and follower database. Section 4 gives several detailed designs of our adaptively self-tuning batching method. Log replay optimization is described in Section 5. In Section 6, we analyze the influence of network anomalies on both log replication and replaying and then give the solution. Section 7 presents the results of performance evaluation. Finally, in Section 8, we give an overview of related works and Section 9 concludes the paper.

2 Consistency model

In distributed system, the *consistency model* describes how different replicas are kept in sync. As observed by Brewer in his well-known CAP Theorem [7], in a distributed system, among consistency, availability, and partition tolerance, only two out of three are possible. Any replicated systems need to trade off these three aspects. In this section, we first clearly define the related terminologies, and then show the design choice of QuorumX. We adopt the definition of consistency in replicated system from prior work [8–10].

Definition 1 Strong consistency Strong consistency is commonly equated with *linearlizability*, which requires: after a transactions T_i commits, any new transaction T_j following T_i observes the updates of T_i.

Definition 2 Weak consistency Weak consistency relaxes the requirements of strong consistency. Within a weak-consistent database, after a transaction T_i finishes, there is no guarantee that the following transactions can read the updates of T_i. With weak consistency, applications may see different versions of data from different replicas.

As network partition is unavoidable in a distributed system, CAP theorem formally proved that when a system chooses strong consistency, it can not provide high availability. However, database systems like Google Spanner [11] claim to be highly available and strongly consistent, which seems to avoid the CAP theorem. Actually, this is because of the misunderstanding of availability. To clearly understand it, we classify availability into Node-level Availability and Service-level Availability.

Definition 3 Node-level availability Node-level availability means that, when network partition occurs, every replica of the system still could provide services for applications.

Systems like Dynamo [12] and Cassandra [13] choose to guarantee node-level availability, and they can not ensure strong consistency at the same time. Instead, they use *eventual consistency* to provide service when partition happens. We believe that the availability in CAP theorem means node-level availability.

Definition 4 **Service-level availability** Service-level availability requires that, when network partition occurs, as long as the partition degree is not too serious (e.g., a majority of replicas are still in the same partition), the system is still capable to provide service.

Service-level availability does not require that all replicas provide services in the case of failure, but require that some of them are available. Also, the system needs to automatically elect a new master from replicas to take the place of a failed one, ensuring no loss of service. Consensus protocols are always used to achieve service-level availability. Since QuorumX targets to provide service for mission-critical applications, strong consistency is the primarily necessary. Therefore, QuorumX is designed to satisfy *both strong consistency and service-level high availability*.

3 Preliminary

3.1 Architecture

Figure 1 shows the overall architecture of replicating an IMDB. The replicated IMDB cluster contains one primary IMDB as a leader and more than two replica IMDBs as followers. Requests that contain write operations are routed to the leader IMDB, while read-only transactions are performed by the follower nodes. Transactions are concurrently executed on the leader. When a transaction completes all transactional logics and starts to execute the COMMIT statement, the worker threads in the leader generate the transactional logs and appends them to log buffer (at steps 1 and 2 in the left side of Fig. 1). Then this transaction enters the commit phase, waits to be committed (at step 3) and finally responds to the client (at step 6). The single commit thread in the leader sends these transactional logs to all followers and flushes them to local disks (at steps 4 and 5). A transaction can be committed only after the leader receives more than half responses from followers, and the latest committed log sequence number (MaxComLSN) is set to the index of this committed log entry. After that, the leader will asynchronously send the MaxComLSN to followers. Follower replicas then replay committed logs less than the latest received MaxComLSN. It should be noted that the execution worker is multi-threaded. The new arrived transaction requests from clients can be processed in parallel although previous transactions have not been committed. The new arrival transactions cannot be committed until the previous ones have been committed. That means the commit order is sequential.

3.2 Log replication

The follower replica who receives log packets will first parse it into entries with log format and check the integrity, then write it to the non-volatile storages and send a response message to leader. Under a heavy OLTP workload, if followers use a single thread to process received logs in a sequential manner, the replication latency would be unacceptable in practical settings. Pipeline and batching are general methods used to improve the performance of log replication.

- **Pipeline parallelism of log replication in the follower** The basic steps for processing a received log by replica can be divided into three relatively independent stages: parsing logs, flushing logs and sending response to leader. The pipeline of replicating logs in the follower is that: the log receiving thread (e.g., *revLog thd*) gets network packets from the receive queue, parses them to log entries and appends these logs to the *receive buffer*(at steps I in the right side of Fig. 1). After that *revLog thd* pushes the indexes of log entries into the *wait-for reply queue*. At the back-end, the single persistence thread (i.e., *wrtDisk thd*) reads a batch of logs from the receive buffer and flushes them to log files (at step II). When finishing writing a batch of logs, *wrtDisk thd* notifies the *reply thd* to send a response to the leader (at step III). Then *reply thd* pops the indexes of log entries from the *wait-for reply queue* and send an acknowledgment to the leader. The replication latency introduced by follower replicas is hidden through pipelined log processing.
- **Batching logs in the leader** Pipeline and batching are often used together [4,14]. Without batching, the pipeline will be hard to work effectively. Basically, batching several requests into a single instance allows

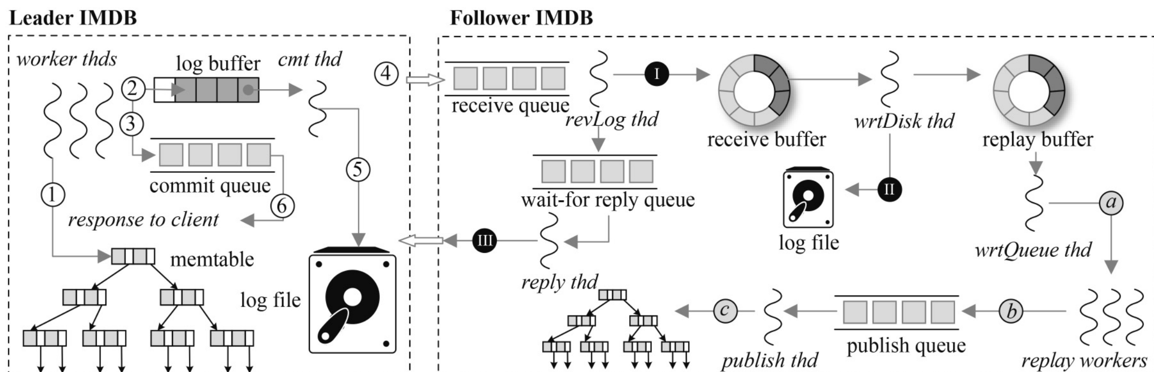

Fig. 1 Overall architecture of log replication and replaying in an IMDB cluster. ○ represents the step of log replication in the leader. ● indicates the process of log replication in the follower. ◉ denotes the flow of log replaying

the overhead to be amortized over per-request. The systems built over quorum-based replication can adopt the batching method to boost the throughput. However, the parameters such as the batch size or the sending interval have greater impact on the performance. Manual configurations for these parameters are proved to be time consuming and can not adapt to different environments. Existing works on automatic batching are limited in the replicated IMDBs. For example, the factor on processing capability of follower has not been fully considered in the batching scheme.

In this work, we investigated several batching methods and found that they were not always effective under the context of replicating a fast IMDB. Quorum-based replication needs an adaptively self-tuning batching mechanism that not only is parameter-free but also considers: 1) the capacity of follower; 2) the workload characteristics (e.g., the arrival rate).

The receive buffer in the follower is an important structure which is responsible for caching the received logs from the leader. The *wrtDisk thd* can flush a batch of buffered logs at one time. We choose to use the ring buffer to implement the receive buffer in order to avoid frequent memory allocations and deallocations. The size of the receive buffer is a key design consideration. If the size is set to a small value, the received but un-flushed logs would be soon overlapped by the new arrival logs. This makes the follower require to pull the covered logs from the leader, which introduces additional latency for the log replication. The basic idea of determining the buffer size is that it should be greater than the rate of log generation on the leader.

3.3 Log replaying

Basically, the *wrtDisk thd* flushes logs into disks and at the same time appends them to the replay buffer waiting for replaying. Log replication and replaying use different buffers in order to avoid that flushed but un-replayed log entries are overlapped by the new arrived logs caused by the speed mismatch between disk write and replaying.

To avoid lagging behind the leader too much, the follower requires a fast mechanism of replaying committed logs. On the back-end, follower IMDBs replay logs to memtables (which is often implemented by B+ Tree or SkipList in IMDB) to provide read-only transaction requests according to the maximal committed log sequence number (MaxComLSN), which is piggybacked on logs to notify the follower the latest committed point.

Under the conventional quorum-based replication schemes, the *wrtQueue thd* fetches log entries with LSN less than MaxComLSN from the replay buffer and pushes them to the *replay workers* (step a). After that, *replay workers* apply logs into local memtable in parallel and put the replayed LSNs in the *publish sorted queue*. A single *publish thd* commits the modifications in the memtable and make them visible in a serial LSN order. In the case of highly concurrent workloads, this principle of relaying logs by follower causes a challenge in visibility gap. In this paper, visibility gap is defined as the time difference between leader and follower for making the same committed data be visible. Real applications such as HTAP often take real-time OLAP analysis over the follower nodes [15], and it is expected that there is a VGap as small as possible between the leader and followers. There are two challenges to minimize VGap in QuorumX.

First, recently proposed solutions to minimize VGap aimed at resolving the problem in the asynchronous primary-backup replication, which can not be applied to the quorum-based replication [16,17]. In the asynchronous primary-backup replication, the follower could replay the received logs immediately without any coordination with leader. However, in the quorum-based replication, it is the leader that notifies followers the consensus decision of transactional logs by sending the current MaxComLSN. After receiving MaxComLSN, follower nodes are agreed to replay logs with LSN not larger than MaxComLSN.

Second, since it is expensive to read logs from disk for replaying, the replicated but un-replayed logs need to reside in the memory for a period of time before being replayed. The structure holding un-replayed logs is the replay buffer. However, in the case where the leader generates logs at a high speed, e.g., SiloR could produce logs at gigabytes-per-second rates [18], or the case that the follower server is under high pressure, causing a slow replaying speed, un-replayed logs resided in the buffer can be soon erased by the new arrivals. The follower still needs to read flushed logs from disk for replaying, and would definitely lag behind the leader and produce larger and larger VGap.

Therefore, in this paper, in order to minimize the VGap and keep up with the state of the leader, we first propose a coordination-free log replaying mechanism for the follower. Second, we devise a remedial action against the case that the un-replayed logs in the replay buffer are covered by the new logs.

4 Adaptively self-tuning batching scheme

Basically, batching several requests into a single instance reduces the amortized per-request overhead, which boosts the throughput of system. The design objectives of batching scheme in QuorumX have three aspects.

First, no parameters are required to be calculated offline and then manfully tune system configurations. Because once the environment settings are changed, these parameters need to be calculated again. It should be totally automatic to cope with the uncertainties without manual intervention.

Second, workloads in real setting are often dynamically changed and have an important effect on the performance of batching scheme. For instance, if the transaction arrival rate becomes low, a batch should be constructed by a small number of log entries.

Last but not least, considering the processing capacity of the follower is essential for adaptively tuning algorithm in quorum-based replication, especially in the case where the whole performance relies on the processing speed of followers under heavy workloads. Follower replicas may be overloaded if log replication with a wrong batch size.

Although many batching methods claimed that could tune with a better performance, but in our development, we found that almost neither of existing algorithms can satisfy all our needs, as shown in Table 1.

Table 1 Features of different batching algorithms

Batching scheme	Parameter-free	Workload-adaptive	Follower-friendly
JPaxos	×	√	×
N Santos [14]	×	√	×
P Romano [19]	×	√	×
AB [20]	√	×	×
TAB [20]	×	√	×
QuorumX	√	√	√

4.1 Batching scheme in QuorumX

Based on the above design objectives, we propose to combine critical-path-based batching (CB) [20] and pipeline-based batching (PB). CB automatically adjusts the batch size according to workload characteristics. PB is complementary to CB by considering the processing capability of follower, which can adaptively tune the frequency of sending logs to avoid followers being overloaded in highly concurrent workloads.

The CB mechanism operates as following: as shown in Fig. 2, after finishing processing transaction logics, each worker thread will enter a global common code fragment, that is *commit_transaction*. The entry code is used for registering the commit queue as the task is inside. Similarly, the exit code de-registers the task and appends it to the sending batch. The intuition behind CB is that multiple tasks should be included in the same batch only if they arrive "close together" to the *sendBatch*(). When implementing CB, we treat the commit queue as a doorway. A batch is complete and sent to followers when the commit queue is empty, since the next task is too far behind to join into the current batch. Compared with batching with a fixed time or a fixed size, CB could adjust sending frequency according to the arrival rate. When the arriving rate is high, CB gathers a lot of close tasks and achieves good throughput. And if the arrival rate is low, CB will not waste a long time for waiting for more tasks. The disadvantage of CB is that when the arrival rate stays constantly high, CB will continue to gather too many tasks without sending a batch in a proper size. We combine PB with CB mechanisms to resolve this issue.

PB takes a full consideration of the pipelined replication scheme in the follower. As described above, the pipelined replication scheme in follower consists of three stages (s_1, s_2, s_3). *It should be noted that an optimal performance can be achieved if the slowest pipeline stage handles tasks all the time and has no idle time.* Taking Fig. 3(a) for example, suppose that s_2 is the most time-consuming stage, and the optimal send interval for a batch should be t_{s2}. Upon this sending rate, every batch could get a smallest replication latency and next batches would not be blocked by the previous ones. Therefore, during the pipeline replication, QuorumX collects the consumed time of each stage by followers for each batch, and embedded them into the response to be sent to the leader. QuorumX requires the time interval of sending two batches should not to be less than it. If logs are sent with an interval larger than that value, the resources cannot be utilized sufficiently. On the contrary, if the sending interval is less than that value, congestion should happen during replication.

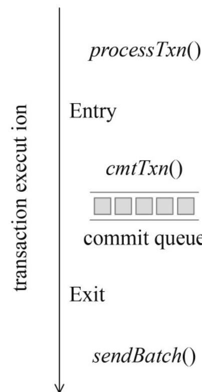

Fig. 2 Critical-path-based batching (CB)

4.2 Discussion

We demonstrate that network latency between the leader and follower has no effect to set sending frequency with Fig. 3(a) and Fig. 3(b). No matter how the network latency changes, the optimal frequency is always restricted by the most time-consuming stage of follower. However, we need to point out that network bandwidth can affect the sending frequency. Under complicated network environment especially the wide area network, bandwidth is often limited and may be occupied by something unknown. Here, log replication is constrained by the limited network bandwidth of the leader. As a result, the sending frequency should be lowered properly. How to automatically adjust the frequency of sending logs over complicated, unreliable networks is still an open question, and we will study this problem in our future work.

4.3 Calculation method of the sending interval

In the PB mechanism, the leader calculates the time interval of sending a log batch by averaging the previous send interval and the feedback time (the time consumed by the slowest stage in the pipeline replication) by all followers. The value is a uniform value, which means that the leader sends logs to all the followers at the same time interval. Certainly, we could set a frequency for each follower according to their slowest pipeline stage, but it would be less space-efficient and time-efficient because we need to store and copy different logs for each follower.

In a realistic cluster, replicas may have different replication performance. For example, servers with HDDs have a much longer disk-write latency than servers that are equipped with

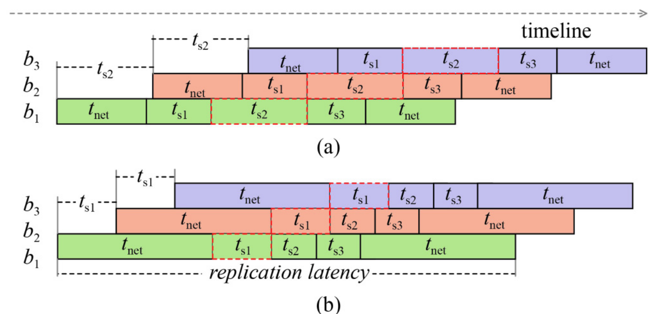

Fig. 3 Pipeline-based batching (PB). (a) S_2 is the most time-consuming part, under low-latency network; (b) S_1 is the most time-consuming part, under high-latency network

SSDs. When there are other tasks running on a follower node, its replication performance would be largely affected. As a result, the replication latency will be prolonged.

In this situation, if the uniform sending interval is still averaged by all follower replicas, it is unappealing since for the slow followers, the frequency may be so high that congestion will very likely happen. As discussed in Section 3, the receive buffer may be overlapped under the congestion. As a result, the slow followers will be increasingly lagging behind the leader, also resulting in a larger and larger VGap that affects the system availability.

Hence, we refine the calculation method for the uniform sending interval according to the following cases.

- When the difference of the maximum index of persisted log entries (e.g., *maxPstLSN*) among the followers is small, the sending interval takes the average;
- When the difference of *maxPstLSN* is greater than a threshold, the sending interval is set to the feedback of the slowest follower. The active reduction of sending frequency by the leader node enables the lagged followers to catch up with the leader as soon as possible.

5 Coordination-free log replay

5.1 Design choices for replay buffer

The replay buffer in follower is responsible for buffering the persisted logs waiting for replaying to the memtable. We have discussed the design of the receive buffer which is used to cache the received logs waiting for persistence. We avoid the case when the un-flushed logs in the receive buffer are overlapped by the new arrivals through adjusting the sending interval in the leader.

The size of the replay buffer is also a significant design consideration. IMDB such as SiloR could generate logs at gigabytes-per-second rates. When the follower is under high workload pressure and the replay speed slows down, the flushed log entries that have not been replayed will be covered by new arrived logs. As a result, the un-replayed log entries must be read from the disks, which would introduce extra disk I/O latency. This causes the risk of cascading latency as more un-replayed logs continue to be covered by newly arrived logs. Finally, it will make the memtable state of the follower nodes never catch up with the leader. The design of replay buffer should guarantee replicated logs are replayed from memory most of the time and avoid re-loading them from HDD/SDD.

In order to provide read services on fresh data by followers, they need to replay flushed logs to memory as fast as possible. However, as discussed above, different from asynchronous replication, the time to replay a log entry is restricted by the quorum-based replication scheme. A follower is only allowed to replay logs with LSN not larger than MaxComLSN for guaranteeing consistency. However, wait-for-replay logs residing in the memory may cause the replay buffer overwhelmed. To this end, we design a coordination-free log replay (CLR) scheme which directly applies the received logs to the memtable without waiting for the MaxComLSN. CLR ensures consistency by separating the replay procedure into two phases. The first phase converts logs into uncommitted cell lists of memtable in parallel, where the applied data are invisible. The second phase sequentially installs them into memtable according their LSNs, where the consistency is guaranteed. *It should be emphasized that the second phase is extremely lightweight without introducing overhead as the installation only contains a few pointer manipulations*.

5.2 Mechanism of coordination-free log replay

Basically, different from transaction execution in the leader, there is ***NO*** rollback when replaying logs in follower. That means all of the logs must be replayed successfully in a serial order. We choose to replicate *value logs* instead of operation logs, which could promise a lock-free replay strategy. When CLR begins to replay a batch of logs, in the first phase, multiple threads (e.g., *replay workers*) works in parallel. *Replay workers* first starts a transaction for each log entry. Then it looks up the memtable to find the node the transaction wants to modify. After that, logs are translated into several uncommitted cell informations in which each cell has a pointer pointing to the actual node in the memtable. Translating will not directly be installed the updates into the memtable and therefore there is no need to acquire any locks. The uncommitted cell informations are stored in the transaction context.

Algorithm1 QuorumX commit algorithm of log replaying

```
/* Commit transactions according to
   log sequence                                        */
Input: MaxComLSN
1  while !thread_.stop() do
      /* Get a transaction from publish
         sorted queue sequentially.                    */
2     log_id = publish_queue.seq_;
3     while true do
4        if log_id > MaxComLSN then
5           wait(wait_time_ms);
6           continue;
7        txn_ctx = publish_queue.get(log_id);
8        if NULL==txn_ctx then
9           wait(wait_time_ms);
10          continue;
11       atomic_cas(&publish_queue.seq_,log_id,
           log_id+1);
12       break;
      /* Install the modifications into
         memtable.                                     */
13    for cell_info in txn_ctx.uc_info do
14       memnode = cell_info→node;
15       exclusive_lock(memnode.rowlock);
16       memnode.value_list.append(cell_info);
17       exclusive_unlock(memnode.rowlock);
```

After completing the above procedures, transaction will be pushed into the *publish sorted queue* of a single publish thread (i.e., *publish thd*) which is responsible for committing

transactions, making their modifications visible. It was the single *publish thd* that ensures the safety and consistency of quorum-based replication. In the second phase of CLR, *publish thd* sequentially pops transaction whose log id is smaller than the MaxComLSN, and does the commit transaction operation. As shown in the Algorithm 1, transactions with log id less than MaxComLSN will be committed and their uncommitted cell informations will be directly append to the value list in the memtable. Locks are necessary in this part, but as we can see, the duration is short (lines 13–17). Here, the *rowlock* is maintained in a per-tuple fashion and co-located with the raw data (e.g., an additional field in the header of a tuple is used to represent its current locking state) which can be acquired and released by atomic operations. Such lock implementation is popular in modern lock-based database systems and proved to be lightweight and efficient [21−23].

The main processing flow of CLR can be processed totally in parallel and only the commit part is done sequentially in order to promise transaction modifications are installed into memtable by the LSN order. CLR immediately replays the received logs without waiting for MaxComLSN. One advantage is to alleviate the risk of reading flushed logs from disk and the memory resources consumed by the replay buffer has a minor risk of being excessive. Besides, since CLR performs replaying ahead of time, the VGap can be minimized compared with the replaying scheme waiting for MaxComLSN.

5.3 Discussion

Nevertheless, there are additional demands on fault handing introduced by our proposed replay strategy. Suppose such a scenario in Fig. 4, five replicas (R_1—R_5) form a cluster and R_1 is the initial leader. Before crashed, R_1 has generated five log entries and committed four log entries. Log five has flushed to disk and entered into the first phase of CLR in R_2 while the other three followers have not received log five. According to the election algorithm, R_4 is elected as the new leader. It generates a different log five and replicates it, and there is a growing problem that R_2 has began to replay a log five from the old leader. Although the modification has not been installed in the memtable, the transaction context with log five still reside in the memory (dirty contents). If R_2 begins to replay another log five, there may be some checksum errors. If similar situations arise when we do not adopt CLR, there are no dirty contents in R_2's memory, R_2 only needs to rewrite log five to its disk.

Based on above description and discussion, when introducing CLR, we also refine the fault handing algorithm. More concretely, when role change happens, each node will firstly perform replay-revoking operation before actually getting into working. CLR ensures that dirty contents can be easily erased since it neither modify any structures storing data nor hold any locks. The commit thread pops all tasks from its commit queue, cleaning uncommitted cell informations and ending these transactions.

5.4 Asynchronous loading mechanism

Although CLR could provide a fast log replaying for quorum-based replication, the risk of reading flushed logs from disk for replaying has not completely been avoided. When there are other loads running on the follower and the resource thrashing problem is heavy, it would makes the replay speed greatly slowed down. Therefore, in order to avoid the risk of reading disks entirely, we propose an asynchronous loading mechanism (ALM) to ensure that all un-replayed logs are obtained from the memory all the time.

Figure 5 illustrates the processing model of ALM. We introduce an additional thread called *asyncLoad thd*, which is responsible for loading log entries from log files to the replay buffer asynchronously. Before trying to append logs to the replay buffer, the *wrtDisk thd* first checks whether the un-replayed logs have occupied the entire replay buffer. It can be completed by comparing the size of the replay buffer and the difference between *tail* and *head*. If so, *wrtDisk thd* does not perform this append operation and wake up the *asyncLoad thd*. The *asyncLoad thd* loads a batch of log entries from the disk and appends them to the replay buffer in the background. When *wrtDisk thd* finds that the LSN difference between the first un-replayed log entry and the appending log entry is small, which means that the follower has caught up with the leader, it notifies the *asyncLoad thd* stop, and reverts to the normal replaying process.

6 QuorumX in an unreliable network

Although QuorumX is able to obtain a nice performance and low VGap under the normal cases, it is fragile to the network jitter. In this section, we first discuss that how the network anomaly causes the performance of QuorumX unstable. Then, we give our solution to avoid this. Finally, we refine the related recovery mechanism.

6.1 Problem analysis

Anomalies in an unreliable network, including package loss and increased network delay, are detrimental for the performance of Raft-replication due to its *log coherency*. As illu-

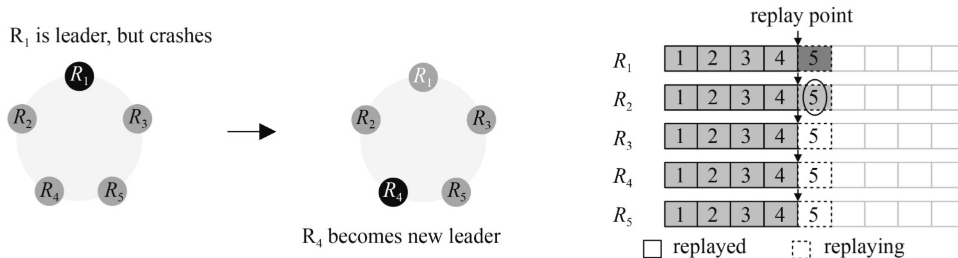

Fig. 4 An example illustrating a fault caused by CLR

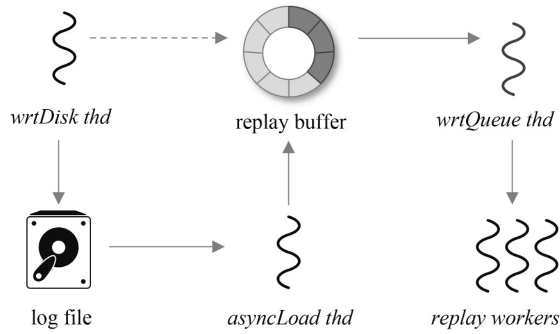

Fig. 5 The model of asynchronous loading mechanism

strated in Fig. 6(a), logs do not allow any holes in both leaders and followers. In other words, log entries are committed by leaders, and applied to all followers in a serial order.

Suppose that a log packet sent to a follower is lost, in Raft, the leader will resent the log packet if it does not receive the acknowledgment from the follower. During this time, the follower can not process subsequent logs until the previous lost logs are replicated: for example, in Fig. 6(a), Node 3 did not receive the log entry with the index 4 and blocked. Although it can receive the following log entries, they can not be processed. It gets worse when the log packet sent to Node 2 is also lost (e.g., log entry with the index 6). It results in an increasing waiting time of the transaction in the commit phase. There is no doubt that the increase of transaction blocking time has a significant impact on system performance. The advantage of Raft-replication is that when the leader crashes, Raft can elect the node with the most data as the new leader. The new leader could take charge immediately since its memtable contains all consistent data.

Multi-Paxos performs steadily when network jitter happens because it allows log holes in the follower. It can be seen from Fig. 6(b) that, the follower can flush logs to disks even it does not receive the previous logs. Although the replication process would not be affected, log replaying can not proceed when the log is incomplete. When a new leader is elected, Multi-Paxos must fetch the missing logs from other servers and replay all logs before taking over the system and providing service. This makes the system suffer from a long time of unavailability.

6.2 Process in the follower

In QuorumX, we combine the characteristics of Raft and Multi-Paxos. First, we relax the constraint of *log coherency*, thus the follower could replicate logs and reply to the leader out of the serial order, which avoids the influence of network jitter on normal replication process. We use the single publish thread to make sure that logs are committed in a serial order. Second, we fill in the missing logs in runtime, ensuring that log replaying would not be blocked. Because the leader IMDB is under great pressure, instead of letting the leader resend the lost log to the follower, we choose to let the follower find the log holes and take the initiative to pull the logs from the leader to fill the holes.

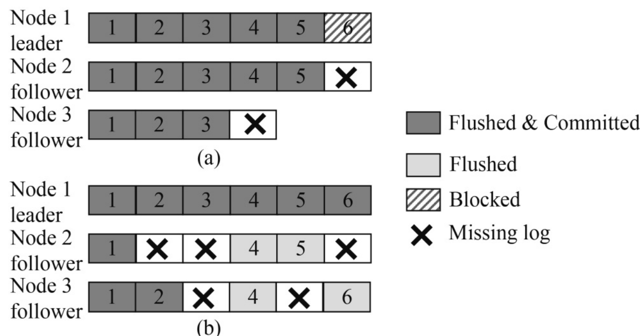

Fig. 6 An example of 3-way replication's logs in Raft and Multi-Paxos. (a) Raft replication; (b) Multi-Paxos replication

Let us review the processing of the follower. **Log replication**: the *revLog thd* parses the received log packet and appends it to the receive buffer. The *wrtDisk thd* fetches logs from the receive buffer and flushes them into HDD/SSD even when they are discontinuous. When the log is persisted, the *reply thd* is responsible for sending an acknowledgment to the leader. **Log replaying**: the *wrtDisk thd* sends flushed logs to the queues of *replay workers*. The *replay workers* apply logs into memtable in parallel and push the replayed LSNs into the sorted publish queue. The single *publish thd* pops LSNs from the publish queue according to MaxComLSN sent by the leader and makes the replayed log visible in the serial order.

When the log packet is lost, threads in log replication and replaying phase can not perceive this since logs can be flushed and replayed out of the order. The *publish thd* can sense this when it finds that the commit queue does not contain a LSN and the commit process has been blocked by that LSN for a long time. It means that the log entry with this LSN is missing. In this case, a dedicated thread in QuorumX is triggered to actively fetch the lost logs from the leader and write it into the receive buffer.

6.3 Recovery mechanism

The correctness of replaying log out of order can be ensured since finally, they are committed in a serial order. Flushing log out of order requires us to make some modifications to the recovery process. Suppose a case in Fig. 7, when Node 1 is the leader, it generates log entry with the index 4, and only replicates the log in Node 3 (log 4 is not committed). Then Node 1 crashed, Node 2 is elected to be the new leader, and also generates a log entry with the index 4 (log 4). Now in Node 3, there has two log entries with the index 4, and clearly log 4 produced by Node 1 is the dirty log. Therefore, when recovering a crashed server, a pre-scan over its logs is required. If there are several log entries with the same index, the log with latest term value (produced by the newest leader)

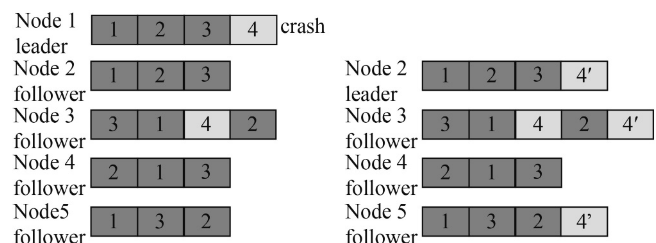

Fig. 7 An example of 5-way replication illustrating the problem caused by flushing logs out of order

is valid and will be retained. After that the recovery process is continued by replaying all log entries into the memtable.

7 Evaluation

In this section, we evaluate the performance of QuorumX for mainly answering the following questions:

- The first question is whether QuorumX could support a high performance replication for fast IMDB, and how much additional performance is sacrificed by QuorumX through comparing it with the asynchronous primary-backup replication and the single replica without replication.
- Another question is that whether QuorumX can be self-tuning to workloads. We evaluate its performance under different concurrency by comparing batching methods include TAB [20] and JPaxos. Since the calculation of offline models in [14,19] requires a lot of additional parameters which are difficult to collect, we did not implement them in QuorumX. Also we evaluate our self-tuning algorithm under the cluster consisting of servers with different configurations to test whether the low-configuration followers in QuorumX will be lagged behind other servers.
- The third question is that how much VGap can be reduced by the CLR of QuorumX in contrast with asynchronous primary-backup replication. Besides, CLR replays logs without waiting for MaxComLSN in order to avoid reading logs from disk and thus reduces the VGap. We also measured how much VGap could be reduced by CLR even if QuorumX replays logs after receiving the MaxComLSN.
- The next question is that whether the log replaying in QuorumX always fetches log from the memory instead of the disk when there has other loads in the follower.
- The final question is that whether QuorumX can obtain a stable performance under an unreliable network environment. We test both the system throughput and the VGap by simulating the network anomalies.

We have implemented QuorumX in Solar [6], an open-source, scalable in-memory database system. We implement QumrumX by adding or modifying 39282 lines of C++ code on the original base. Therefore, Solar is a completely functional and high available in-memory database system. It has also been deployed on Bank of Communications (BOC), one of the biggest commercial banks in China. The default cluster consists of three replicas and the leader has the full-copy of data. We also evaluate performance of different number of replicas. Each server is equipped with two 2.3GHz 20-core E5-2640 processors, 504GB DRAM, and connected by a 10 Gigabit Ethernet.

In the following experiments, we used three benchmarks to test the performance of QuorumX.

- YCSB: The scheme of YCSB contains a single table (*usertable*) which has one primary key (INT64) and 9 columns (VARCHAR). The usertable is initialized to consist of 10 million records. A transaction in YCSB is simple and only includes one read/write operation. The record is accessed according to an uniform distribution.
- TPC-C: We use a standard TPC-C workload and populated 200 warehouses in the database by default. The transaction parameters are generated according to the TPC-C specification.
- Micro-benchmark: We build a write-intensive micro-benchmark. Instead of sending the leader IMDB transaction requests coded by SQL statements, this micro-benchmark directly issues raw write operations to the leader. Therefore, the leader IMDB is running under extremely high-concurrent, write-intensive workloads. By default, the micro-benchmark contains 10GB data modifications.

7.1 Replication performance

We firstly measure the throughput and latency under the YCSB workload with 100% write operations and the complicated TPC-C workload. The comparing methods include QuorumX with three replicas one of which servers as the leader (abbr., QuorumX), asynchronous primary-backup replication (abbr., AsynR) with three replicas and a single replica without replication (abbr., NR).

Experimental results of YCSB are shown in Figs. 8 and 9. We can observe that the throughput trend of all replication scheme is increasing firstly and then remaining at a high level.

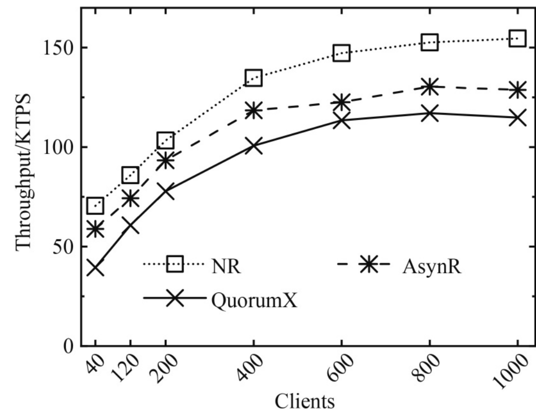

Fig. 8 Throughput of YCSB

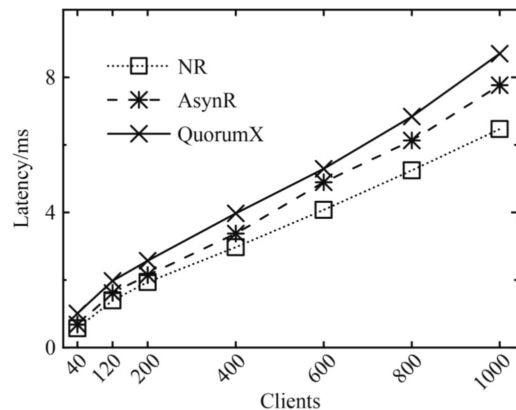

Fig. 9 Latency of YCSB

In general, QuorumX sacrifices about 11% performance compared with AsynR and 26% compared with NR to provide data consistency and high availability. As for latency, QuorumX produces about 0.6 more milliseconds than AsynR and 1.1 ms than NR in average. Figures 10 and 11 illustrate the performance under the TPC-C workload. We find that the throughput gap among QuorumX and AsynR and NR reaches to 2% and 8% respectively, which is smaller than that in YCSB. The reason is that a transaction in TPC-C contains more read/write operations than that in YCSB so the leader takes more time to execute a TPC-C transaction. As a result, the percentage of replication latency is relatively small in the whole transaction latency.

We also evaluate QuorumX under YCSB benchmark with different write/read ratios. In Fig. 12, we vary the percentage of read operations from 0% to 100%. Results show that with the increasing of the percentage of read operations, the performance gap between QuorumX and AsynR is getting smaller and smaller. This is expected since log replication is mainly performed for write operations. When there are less writes, log replication has a smaller impact on the transaction throughput. Where there are 80% read operations in the workload, QuorumX only sacrifices 2% throughput than AsynR. When there are a great number of read requests, NR performs worse than AsynR and QuorumX. This is because read requests are all routed to a single node, as well as write requests. NR lacks read scalability as it has no followers to provide service for read requests.

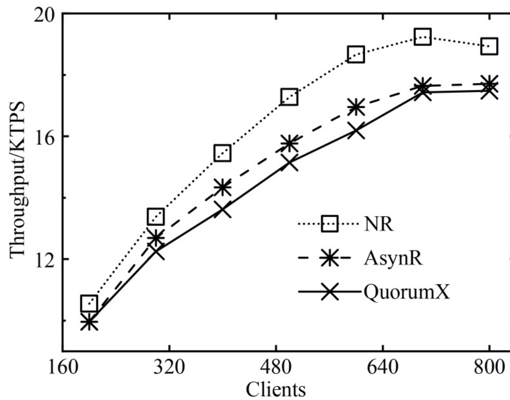

Fig. 10 Throughput of TPC-C

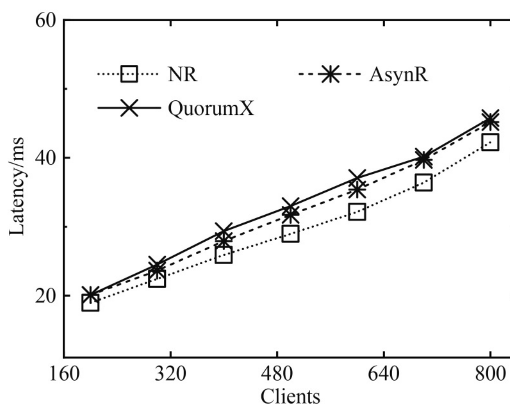

Fig. 11 Latency of TPC-C

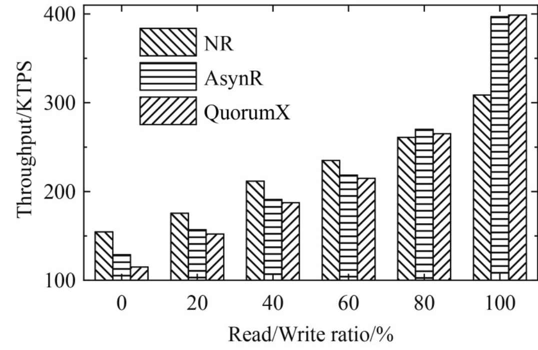

Fig. 12 Impact of read/write ratios

7.2 Availability and consistency

In this section, we present the experimental results under several abnormal scenarios to investigate the availability and consistency of QuorumX. To evaluate the availability of a system, the unable-to-provide-service time is one of the recognized indicators. Therefore, we manually kill the leader while the system is running, and calculate the time interval between the leader's collapse and the system being able to provide services again. Such operation is repeatedly executed 15 times, and Fig. 13 shows the system unavailable time. Results indicate that QuorumX could sense the missing of the leader and re-elect a new leader quickly to take over the system when the leader crashes. And the unavailable time remains about ten seconds, which is mainly used to wait the lease expired (the lease time is nine seconds).

Strong data consistency means that the system will not lose any committed data under any circumstances. We empirical studied the consistency of QuorumX by constructing various abnormal scenarios, e.g., killing the leader, partitioning the network, killing more than half of the severs, etc. Detecting whether a system satisfy data consistency under anomalies can be performed from several aspects. For example, we add transactional cumulative checksum for each row. Replaying logs needs to verify the checksum value. Verification failure means some transaction modifications on the row lost. Also, we could compare the log files of the leader and the followers

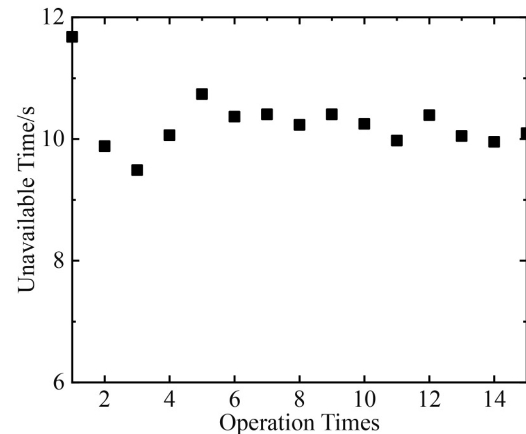

Fig. 13 Unavailable time

by using log reader tools. For insert operations, we could select the total count of rows from the follower and compare it with the number of successful inserted rows to find whether the committed data is lost or not. QuorumX passes massive abnormal tests and thus could be trustingly deployed in real-world applications.

7.3 The ability of adaptive self-tuning

We first demonstrate how the batch size impact transaction throughput under the YCSB workload with a fixed number of clients (1000 clients). Figure 14 presents the throughput over different batch sizes. We find that both small and large batch sizes would impair the performance. Obviously, the worst performance occurs when the batch only contains a single log entry. The maximal throughput in this experiment can reach to 120K/s when the batch size is set to 256. In this case, we find the follower can process about 500 batches per second. When the batch size is far beyond 256, the throughput decreases because the transactional logs take more time to construct a big-size batch in the leader, and the log processing pipelines in the follower might be idle. This implies the significance of pipeline-based batching in QuorumX.

To compare the performance of self-tuning batching scheme of QuorumX with other batching algorithms, we implemented TAB (which adopts critical-path-based batching) and JPaxos (which need manually set the parameter of batch size) to evaluate their effectiveness under various number of concurrent clients. JPaxos is configured to two *batchsize* values: 32 and 256 respectively, referred to as JPaxos-32 and JPaxos-256. Experiments are run over YCSB workloads with 100% write requests.

Figure 15 illustrates the experimental results on different client concurrency. It is clear that QuorumX performs best under all concurrency. We can observe that the performance of TAB is close to that of QuorumX when the concurrency is low. However, as the number of clients increases, TAB could not achieve good performance. Recall from Section 4, under a light workload, critical path based batching works well. But, under a highly concurrent workload, the throughput of the system would be determined by the slowest stage in the pipelined processing on followers. In this case, the pipeline batching mechanism in QuorumX can adaptively tune the interval of sending logs.

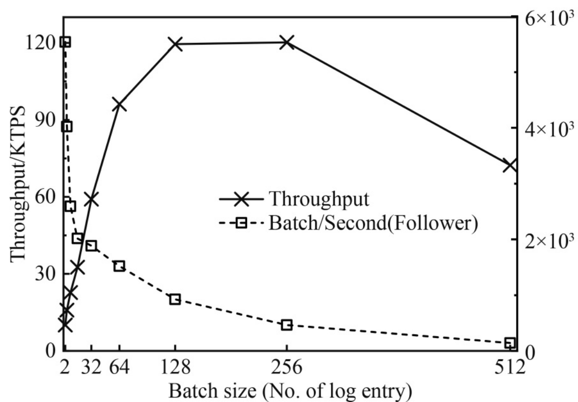

Fig. 14 Performance of different batch size

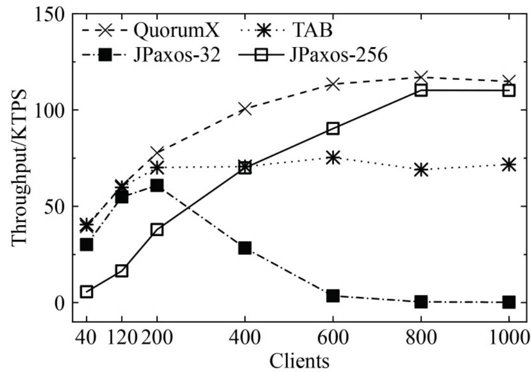

Fig. 15 Performance of different tuning algorithms

The trend of JPaxos-32 increases firstly and could stay at a similar throughput to QuorumX, but decreases sharply when the number of client exceeds 25. This is because, when the client number is small, the arrival rate of transactions is slow, and waiting 32 requests to generate a batch is relatively reasonable. However, when the arrival rate rises, sending batches with size of 32 exceeds the processing capacity of followers. Follower cannot process as many as batches produced by JPaxos-32 in time and these received batches would be blocked. So there is a sudden drop of the performance. On the contrary, JPaxos-256 performs badly when the client concurrency is low and gradually close to QuorumX with the increasing of the number of client. It is clear that, sending batches with size of 256 is too slowly for followers when the arrival rate is low. The leader wastes too much time on waiting for enough requests. Under the high concurrency, collecting 256 requests for a batch becomes easier, and the sending frequency can match the processing capacity of follower.

7.4 Impact of the buffer size

The size of the receive buffer size plays a key role in log replication of QuorumX. When the buffer is small and log entries are generated at a fast speed, it is highly likely that the un-flushed logs in the buffer are overwritten, causing catastrophic decline of log replication performance. This set of experiment investigates the impact of the receive buffer size on the throughout under different frequencies of log generating. We vary the number of worker threads running the micro-benchmark in this scenario, e.g., 10, 20 and 40. The more worker threads executing the micro-benchmark task, the faster the log generation. Results in Fig. 16 indicate that when there are 20 worker threads and the buffer size is smaller than 128MB, the performance decreases sharply. While at 40 worker threads and the buffer size is set to less than 256MB, the throughput starts decreases. This is expected since when the speed of log generation is faster, the system will produce more logs and the possibility of overwritten is higher. This implies that the buffer size must be set according to the rate of log generation.

7.5 VGap results

We measure the VGap between the leader and followers to explore the effectiveness of CLR under a continued, write-

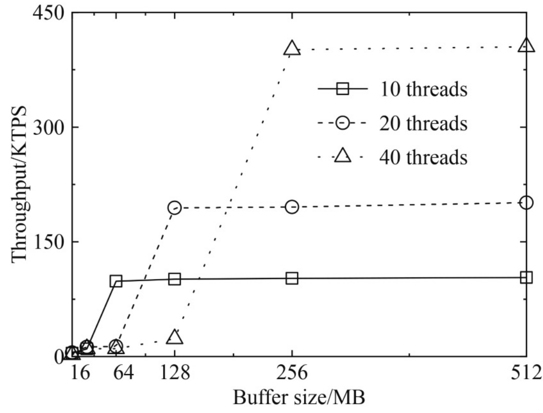

Fig. 16 Performance over various buffer sizes

intensive micro-benchmark. Assuming that the leader l and the follower f commit the same transaction at physical time t_l and t_f, we use the value $t_f - t_l$ to donate the VGap between the same visible state of leader l and the follower f. We compare VGap of three methods: QuorumX, QuorumX without CLR and AsynR.

Figure 17 shows the VGap results over 60 seconds.The number of client is fixed to 800. Results shows that QuorumX could gain the lowest and most stable VGap among three methods. The VGap of AsynR exceeds 200 ms, which suggests that follower in AsynR lags far behind the leader. And the VGap of QuorumX without CLR remains about 100 ms at beginning, but it suddenly increases sharply at time 45. By our analysis, the replica may perform disk-read operations for getting logs to replay, and the trace log also proved that. QuorumX with CLR has a stable VGap and most of it is under 60 ms. Using CLR could achieve a 3.3x lower VGap than AyncR and 1.67x than not using CLR. Therefore, in the case of heavy workload, reading from follower under QuorumX with CLR could get a fresher and more stable state.

7.6 Stability of QuorumX

In this sets of experiments, we evaluate the stability of QuorumX from the following three aspects:

- The maximum log entry difference. We use this indicator to examine whether all servers in the QuorumX cluster could keep the number of logs close especially when the cluster is consisted of servers with different replication performances. Therefore, this experiment mainly tests the stability and effectiveness of our self-tuning algorithm and our refined calculation method for sending interval (Section 7.6.1).
- The rate of log commits. This indicator reflects the stability of our log replaying mechanism, especially can tell us whether there is a phenomenon of reading disk when replaying logs. This is used to investigate the effectiveness of the asynchronous loading mechanism (Section 7.6.2).
- The performance of QuorumX under the unreliable network. Through simulating packet loss and increased network latency, we observe the replication performance changing of QuorumX compared with the classical Raft-replication (Section 7.6.3).

7.6.1 The maximum log entry difference

We denote our refined calculation method for sending interval as RCM. Figure 18 shows the result of the maximum flushed log entries difference between two servers in the Quorum-Replication cluster with RCM and without RCM under the micro-benchmark. The experiment lasts 15 seconds. In order to simulate the servers with different configurations, in Section 5, we manually prolonged the latency of writing disk in one of a follower. During the experiment, we print the LSN of latest flushed logs of all servers every second, and calculate the maximum log difference.

In the first 5 seconds, the log entry difference between two servers in the cluster remains 0. Once the latency of disk writing becomes longer, log difference becomes larger. Since the un-flushed logs in the receive buffer is soon overlapped by new logs, the log difference in QuorumX without RCM gets larger and larger. By taking the slow follower into consideration, RCM calculates a more decent sending interval for the leader. Thus QuorumX with RCM could obtain a stable maximum log difference, which means that the log state of any two servers in the cluster would not be too different.

7.6.2 The rate of log commits

This experiment investigate the effectiveness of our asynchro-

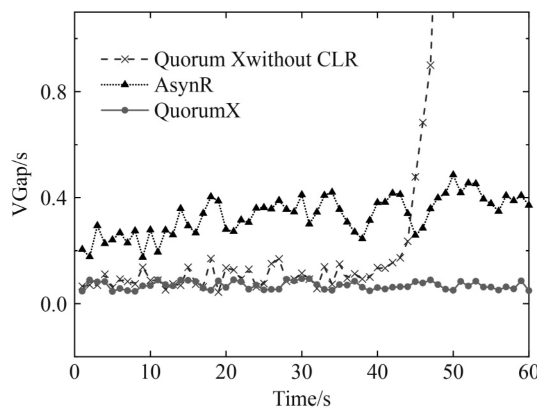

Fig. 17 VGap changing under the write-heavy Micro-Benchmark

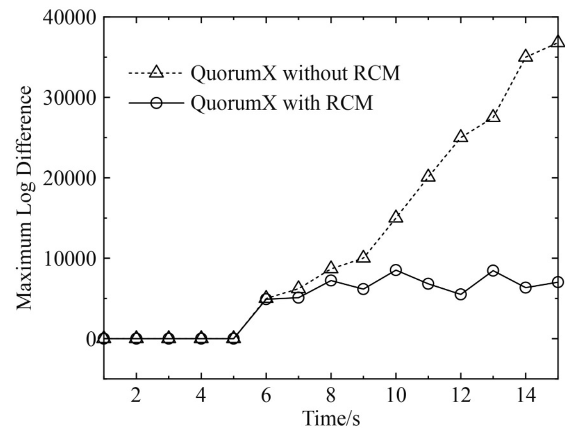

Fig. 18 The maximum log entries difference between two followers

nous loading mechanism to avoid the risk of reading un-replayed logs from disks. Before the experiment, we modified the code to make the *wrtQueue thd* fetch logs from the replay buffer slower. Therefore, the replay buffer is full at the beginning of the experiment. The experiment ran for 30 seconds under the YCSB workloads. We calculate the indicator according to the commit rate of the *publish thd* in the follower. Result can be found in Fig. 19.

Since the *wrtQueue thd* in QuorumX without ALM is required to read covered logs from the disk for replaying when the replay buffer is full, the commit latency is significantly increased. As a result, the commit rate in the follower remains low (<1000). ALM ensures that the un-replayed logs would not be covered the new logs. Thus all replaying logs can always be fetched from the memory instead of the disk. Even when the replay buffer is full filled by various reasons, the commit rate would not be affected.

7.6.3 Performance under unreliable network

We investigate the impact of network delay and package loss on different replication schemes and observe their stability in this experiment. We use the traffic control (TC) tool in Linux to simulate the network anomalies. Figure 20 shows the performance comparison of QuorumX and Raft-replication by setting different extra network latency among the servers in the cluster. With the increasing extra network delay, the throughput of both QuorumX and Raft-replication drops. Due to the serial replication in Raft, its throughput declines faster than QuorumX. Results of the impact of packet loss are illustrated in Fig. 21. Raft-replication is very sensitive about packet loss. The reason is that the follower must suspends normal replication process and repairs the log holes at first. Moreover, the leader must wait for the log packet timeout before sending a new packet. During this period, the follower does not process any new log packets. QuorumX allows log holes exist, writes and replays logs out of the serial order, and fills the lost logs asynchronously, minimizing the impact of packet loss on log replication. Therefore, QuorumX performs much better than the classical Raft-replication under the unreliable network environment.

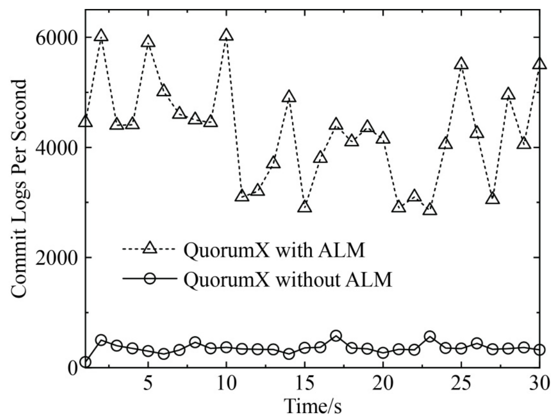

Fig. 19 Rate of log commits in the follower

7.7 Number of replicas

To investigate the scalability of QuorumX, we evaluate the performance over different number of replicas under two YCSB workloads of different write/read ratios: 100/0 and 50/50. Experimental results are shown in Fig. 22. The number of clients is fixed to 125. We can see that under workload with 100% writes, the performance of QuorumX decreased most significantly when the number of replicas is changed from one to three, dropped about 26%. This is because transaction processed under three-replica cluster has obviously longer latency than under single server. When the number of replicas keeps increasing, the throughput decline is not intense, performance under five replicas only decreases 9% than three replicas. This is acceptable since logs have to be replicated to more replicas. Under the workload with 50/50 write/read ratio, the performance decline is even less obvious. As more replicas could provide scalable read service, we can see that with the number of replicas increase, the performance could achieve a sustainable growth. After all, QuorumX has a good scalability with more replicas.

We also compare the scalability of QuorumX with AsynR and NR in this experiment. As NR does not replicate data into replicas, its performance does not scale with the number of replicas. The performance trend of AsynR is similar to that of QuorumX, both of which could scale under read/write workload and decrease under write-only workload. As the result in Section 8 shows, under workload with 50/50

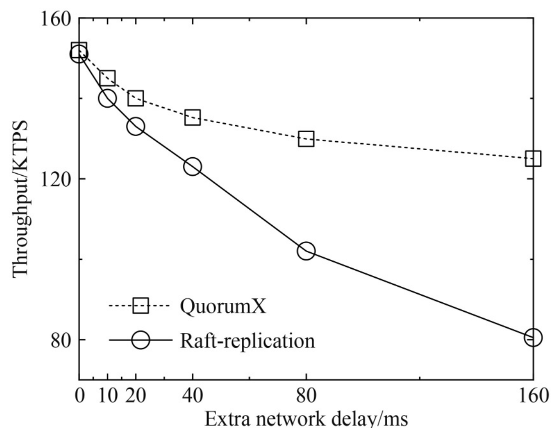

Fig. 20 Impact of network delay

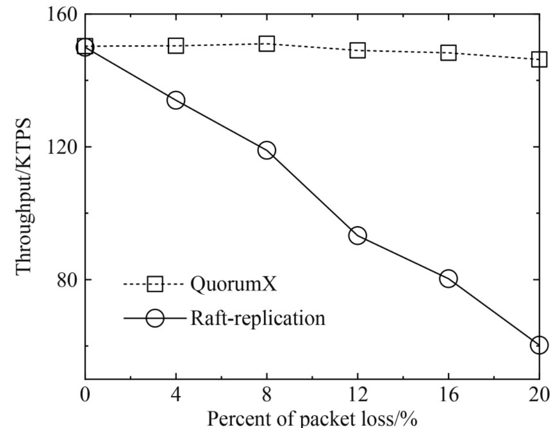

Fig. 21 Impact of packet loss

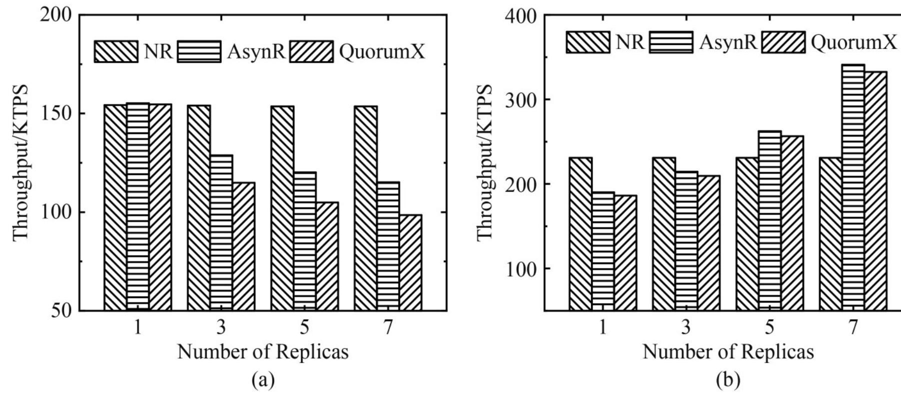

Fig. 22 Throughput over the number of replicas. (a) Write-only YCSB; (b) Read/write YCSB

write/read requests, QuorumX sacrifices less performance than that under workload with 100% write operations.

8 Related work

Replication is an important research topic across database and distributed system communities for decades [24,25]. In this section, we review relevant works mainly on two widely used replication schemes, i.e., primary-backup replication and quorum based replication.

Asynchronous primary-backup replication [26], proposed by Michael Stonebraker in 1979, has been implemented in many traditional database systems. In most typical deployment scenarios, asynchronous primary-backup replication is used to transfer recovery logs from a master database to a standby database. The standby database is usually set up for fault tolerance, and not required to provide the query on the latest data. The performance of log replication and replay have not received much attentions in the last several decades. Recently, the researchers [16,27] suggest that serial log replay in the primary-backup replication can cause the state of replica is far behind that of the primary with modern hardware and under heavy workloads. KuaFu [27] constructs a dependency graph based on tracking write-write dependency in transactional logs, and it enables logs to be replayed concurrently. The dependency tracking method works well for traditional databases under normal workloads, and it might introduce overheads for IMDB under highly-concurrent workloads. [16] proposed a parallel log replay scheme for SAP HANA to speed up log replay in the scenario where logs are replicated from an OLTP node to an OLAP node. Qin et al. [17] proposed to add the transactional write-set into its log in SQL statement formats, which can reduce the logging traffics. Log replay in classical quorum-based replication has different logics to primary-backup replication. Followers using quorum-based replication cannot replay received logs to memtable immediately, and they need to wait for the maximum committed log sequence number from the primary. Due to this difference, these works that optimize log replay for primary-backup replication can not be directly applied to the quorum-based replication.

Despite the low transaction latency, the asynchronous primary-backup replication cannot guarantee high availability and causes data loss when the primary is crashed. PacificA resolves these problems by requiring the primary to commit transactions only after receiving persistence responses from all replicas. The introduced synchronous replication latency depends on the slowest server in all replicas. Kafka reduces replication latency by maintaining a set of in-sync replicas (ISR) in the primary. Here ISR indicates the set of replicas that keep the same states with the primary. A write request is committed until all replicas in ISR reply. Kafka uses the high watermark (HW) to mark the offset of the last committed logs. The replicas in ISR need to keep the same HW with the primary. When the offset of a replica is less than HW, it would be removed from ISR. Through ISR, Kafka can reduce negative impact on performance caused by the network dithering.

Replication based on consensus protocols is referred to as quorum-based replication, which is also called as state machine replication in the community of distributed system. Paxos based replication ensures all replicas to execute operations in their state machines with the same order [1]. Paxos variants such as Multi-Paxos used by Spanner [11] are designed to improve the performance. Raft [2] is a consensus algorithm proposed in recent years. One of its design goals is more understandable than Paxos. For this reason, Raft separates log replication from the consensus protocol. Many systems such as AliSQL[1)] and etcd adopt Raft to provide high availability. However, these systems use Paxos or Raft to replicate meta data, where replication performance is not a serious problem. Spanner as a geo-distributed database system supports distributed transactions, and each partitioned database node is not designed to handle highly concurrent OLTP workloads. AliSQL only uses Raft to elect leader in the occurrence of system failures. Etcd is a distributed, reliable key-value store that uses the Raft for log replication. Similar to Zookeeper [28], these kinds of datastore are designed to provide high availability for meta data management and are not suitable for highly concurrent OLTP workloads.

There are a few works on tuning replication performance of Paxos with batching and pipeline [14]. Nuno Santos et al. [14] provide an analytical model to determine batch size and the

1) https://github.com/alibaba/AliSQL

pipeline size through gathering a lot of parameters, like bandwidth and the application properties. JPaxos proposed to generate batches and instances according to three input parameters: the maximum number of instances that can be executed in parallel, the maximum batch size, and the batch timeout. These parameters need to be calculated offline and set manually which can not adapt to various environments.

Our previous work [29] presents the idea of the self-tuning batching scheme in log replication and the basic algorithm of coordination-free log replay to make the process faster. This work extensively investigates how to make the quorum-based replication and replay architecture more practical and stable in real-world application environments.

9 Conclusion

In this paper, we built QuorumX, an efficient and stable quorum-based replication framework for replicating fast IMDB. We propose an adaptive batching scheme which could self-tuning sending frequency and could adapt to both light and heavy workloads and clusters with different configuration servers. In order to produce a minimal and stable visibility gap between leader and follower, we design a fast and coordinate-free log replay mechanism to replay logs without waiting for maximum committed log sequence number of the leader. We also refine the process of the follower in QuorumX under the unreliable network environments, ensuring the performance of log replication and replaying would not be significantly affected by network anomalies. Experimental results show that QuorumX supports strong data consistency and high availability by sacrificing only 8%–25% performance than single IMDB replica and has a 2%–11% decline than asynchronous primary-backup replication. The batching scheme always performs better than existing methods. Also, the visibility gap produced by QuorumX can reach to a low level. We also present the stability of QuorumX under various settings and exceptions.

Acknowledgements This work was partially supported by National Key R&D Program of China (2018YFB1003404), NSFC (Grant Nos. 61972149, 61977026), and ECNU Academic Innovation Promotion Program for Excellent Doctoral Students.

References

1. Chandra T D, Griesemer R, Redstone J. Paxos made live: an engineering perspective. In: Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing. 2007, 398–407
2. Ongaro D, Ousterhout J. In search of an understandable consensus algorithm. In: Proceedings of 2014 USENIX Annual Technical Conference. 2014, 305–319
3. van Renesse R, Altinbuken D. Paxos made moderately complex. ACM Computing Surveys, 2015, 47(3): 42
4. Rao J, Shekita E J, Tata S. Using paxos to build a scalable, consistent, and highly available datastore. Proceedings of the VLDB Endowment, 2011, 4(4): 243–254
5. Zheng J, Lin Q, Xu J, Wei C, Zeng C, Yang P, Zhang Y. PaxosStore: high-availability storage made practical in WeChat. Proceedings of the VLDB Endowment, 2017, 10(12): 1730–1741
6. Zhu T, Zhao Z, Li F, Qian W, Zhou A, Xie D, Stutsman R, Li H, Hu H. Solar: towards a shared-everything database on distributed log-structured storage. In: Proceedings of 2018 USENIX Conference on Usenix Annual Technical Conference. 2018, 795–807
7. Gilbert S, Lynch N. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. ACM SIGACT News, 2002, 33(2): 51–59
8. Breitbart Y, Garcia-Molina H, Silberschatz A. Overview of multidatabase transaction management. The VLDB Journal, 1992, 1(2): 181–239
9. Daudjee K, Salem K. Lazy database replication with ordering guarantees. In: Proceedings of the 20th International Conference on Data Engineering. 2004, 424–435
10. Elnikety S, Pedone F, Zwaenepoel W. Database replication using generalized snapshot isolation. In: Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems. 2005, 73–84
11. Corbett J C, Dean J, Epstein M, Fikes A, Frost C, et al. Spanner: Google's globally-distributed database. In: Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation. 2012, 251–264
12. DeCandia G, Hastorun D, Jampani M, Kakulapati G, Lakshman A, Pilchin A, Sivasubramanian S, Vosshall P, Vogels W. Dynamo: amazon's highly available key-value store. In: Proceedings of the 21st ACM SIGOPS Symposium on Operating Systems Principles. 2007, 205–220
13. Lakshman A, Malik P. Cassandra: a decentralized structured storage system. ACM SIGOPS Operating Systems Review, 2010, 44(2): 35–40
14. Santos N, Schiper A. Tuning paxos for high-throughput with batching and pipelining. In: Proceedings of the 13th International Conference on Distributed Computing and Networking. 2012, 153–167
15. Özcan F, Tian Y, Tözün P. Hybrid transactional/analytical processing: a survey. In: Proceedings of the 2017 ACM International Conference on Management of Data. 2017, 1771−1775
16. Lee J, Moon S, Kim K, Kim D H, Cha S K, Han W S. Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. Proceedings of the VLDB Endowment, 2017, 10(12): 1598–1609
17. Qin D, Brown A D, Goel A. Scalable replay-based replication for fast databases. Proceedings of the VLDB Endowment, 2017, 10(13): 2025–2036
18. Zheng W, Tu S, Kohler E, Liskov B. Fast databases with fast durability and recovery through multicore parallelism. In: Proceedings of the 11th USENIX Symposium on Operating Systems Design and Implementation. 2014, 465–477
19. Romano P, Leonetti M. Self-tuning batching in total order broadcast protocols via analytical modelling and reinforcement learning. In: Proceedings of 2012 International Conference on Computing, Networking and Communications. 2012, 786–792
20. Friedman R, Hadad E. Adaptive batching for replicated servers. In: Proceedings of the 2006 25th IEEE Symposium on Reliable Distributed Systems. 2006, 311–320
21. Yu X, Bezerra G, Pavlo A, Devadas S, Stonebraker M. Staring into the abyss: an evaluation of concurrency control with one thousand cores. Proceedings of the VLDB Endowment, 2014, 8(3): 209–220
22. Wang T, Kimura H. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. Proceedings of the VLDB Endowment, 2016, 10(2): 49–60
23. Ren K, Thomson A, Abadi D J. Lightweight locking for main memory database systems. Proceedings of the VLDB Endowment, 2012, 6(2): 145–156
24. Kemme B, Alonso G. Don't be lazy, be consistent: Postgres-R, a new way to implement database replication. In: Proceedings of the 26th International Conference on Very Large Data Bases. 2000, 134–143
25. Wiesmann M, Pedone F, Schiper A, Kemme B, Alonso G. Database replication techniques: a three parameter classification. In: Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems. 2000, 206–215
26. Stonebraker M. Concurrency control and consistency of multiple copies of data in distributed INGRES. IEEE Transactions on Software Engineering, 1979, SE-5(3): 188–194
27. Hong C, Zhou D, Yang M, Kuo C, Zhang L, Zhou L. KuaFu: closing the parallelism gap in database replication. In: Proceedings of the 2013

IEEE 29th International Conference on Data Engineering. 2013, 1186-1195

28. Hunt P, Konar M, Junqueira F P, Reed B. ZooKeeper: wait-free coordination for internet-scale systems. In: Proceedings of 2010 USENIX Annual Technical Conference. 2010
29. Wang D, Cai P, Qian W, Zhou A. Fast quorum-based log replication and replay for fast databases. In: Proceedings of the 24th International Conference on Database Systems for Advanced Applications. 2019, 209–226

Donghui Wang is a PhD candidate in School of Data Science and Engineering from East China Normal University (ECNU), China. She received her bachelor's degree in computer science and technology from Zhejiang Normal University, China in 2016. Her research interests include high performance transaction processing in database management systems and high availability in distributed systems.

Peng Cai is a researcher in the School of Data Science and Engineering at East China Normal University (ECNU), China. He received his PhD degree in computer science and technology from ECNU, China in 2011. He joined ECNU in 2015, prior to which Peng worked for the IBM China Research Lab and Baidu. His work has been published in various leading conferences, such as ICDE, SIGIR and ACL. His main research interests include in-memory transaction processing and building adaptive systems using machine learning techniques.

Weining Qian is a professor and Dean of the School of Data Science and Engineering, East China Normal University, China. He received his MS and PhD degrees in computer science from Fudan University, China in 2001 and 2004, respectively. He is now serving as a standing committee member of Database Technology Committee of China Computer Federation, and committee member of ACM SIGMOD China Chapter. His research interests include scalable transaction processing, benchmarking big data systems, and management and analysis of massive datasets.

Aoying Zhou, a professor, Vice President of East China Normal University, China. He got his master's and bachelor's degrees in computer science from Sichuan University, China in 1988 and 1985 respectively, and he won his PhD degree from Fudan University in 1993. He is the winner of the National Science Fund for Distinguished Young Scholars supported by National Natural Science Foundation of China (NSFC). He is a CCF Fellow and the Vice Director of Database Technology Committee of CCF. He served Vice PC Chair of ICDE'2009, ICDE'2012, PC Co-chair of VLDB'2014. His research interests include Web data management, data management for data-intensive computing, in-memory cluster computing and distributed transaction processing and benchmarking for big data.