Fast Quorum-Based Log Replication and Replay for Fast Databases

Donghui Wang[1], Peng Cai[1,2](✉), Weining Qian[1], and Aoying Zhou[1]

[1] School of Data Science and Engineering, East China Normal University, Shanghai 200062, People's Republic of China
donghuiwang@stu.ecnu.edu.cn, {pcai,wnqian,ayzhou}@dase.ecnu.edu.cn

[2] Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin 541004, People's Republic of China

Abstract. The modern In-Memory Database (IMDB) can support highly concurrent OLTP workloads and generate massive transactional logs per second. Quorum based replication protocols such as Paxos or Raft have been widely used in distributed databases. However, it's 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. To this end, we built QuorumX, an efficient 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. Our evaluation results with the YCSB and TPC-C benchmarks demonstrate that QuorumX achieves the performance close to asynchronous primary-backup replication without sacrificing the data consistency and availability.

Keywords: Log replication · Log replay · High performance · Quorum

1 Introduction

Replication is the technique used for a traditional 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 database community. The asynchronous primary-backup replication used in traditional database systems [3,4] trades consistency for performance and availability. The synchronous 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 high-performance and high-availability without sacrificing consistency. Compared with primary-backup replication, the quorum-based replication (e.g. Multi-Paxos [9], Raft [19], etc.) can guarantee strong consistency, tolerate up to F out of 2F+1 fail-stop failures, and achieve good performance because it only requires the majority of replicas to response to the leader. The quorum-based replication adopts consensus protocols to take more reasonable trade-off among performance, availability and consistency, and thus it has been regarded as a practical and efficient replication protocol for large scale datastores [14,22,23].

Quorum-based replication protocols are the natural choice for replicating IMDB as a highly available and strongly consistent OLTP datastore. However, it's 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 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 followers. As a result, the performance of replicated IMDBs significantly depends on the quorum-based log replication which is influenced by many factors.

To achieve read scalability, the followers need to replay committed logs at a fast speed to keep up with the leader's state. The classic quorum-based replication needs the leader to send followers the maximal committed log sequence number (MaxComLSN), and then follower 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, referred to as visibility gap (VGap).* Without careful design, VGap would be larger when the leader IMDB is running under a heavy OLTP workload, and generates transactional logs at a high rate.

In this paper, we present an efficient quorum-based replication framework, called as 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.
- QuorumX has been implemented in Solar [10], 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.

2 Preliminary

Overview of Quorum-Based Log Replication. 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. All requests of read/write transactions are routed to the leader IMDB. Transactions are concurrently executed on the leader. When a transaction completes all transactional logics and starts to execute the COMMIT statement, the leader generates its 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 responses to the client (at step 6). The single commit thread in the leader sends these logs to all followers and flush 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. After that, leader will asynchronously send the latest committed log sequence number (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 transactions cannot be committed until the previous ones have been committed. That means the commit order is sequential.

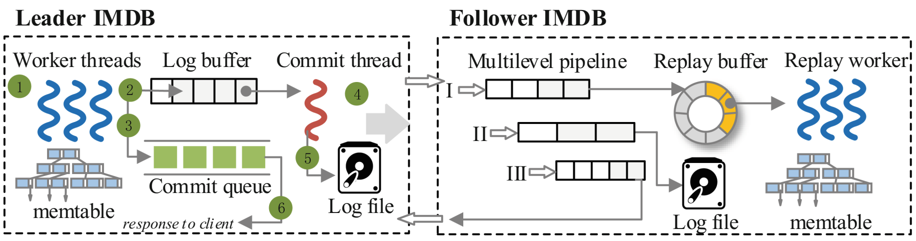


Fig. 1. Overall architecture of replicating an IMDB.

The follower replica who receives logs will first parse it into entries with log format and check the integrity, then write it to 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 in a follower replica.** 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 processing logs in follower is that: the parsing thread gets network packets, parses them to log entries and appends these logs to the *replay buffer* (at steps

I in the right side of Fig. 1). At the back-end, the single persistence thread reads logs from the replay buffer and flushes them to log files (at step II). When finishing writing a batch of logs, the persistence thread notifies the reply thread to send a response to the leader (at step III). 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 [22,25]. Without batching, the pipeline will be hard to work effectively. Basically, batching several requests into a single instance allows 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 batch size have greater impacts on the performance of batching method. The manual configurations for these parameters are proved to be time consuming and can not adapt to different settings. Existing works on automatic batching are limited in replicated IMDBs. For example, the factor on processing capability of follower has not been fully considered in the log replication.

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 is not only parameter-free but also considers: (1) the capacity of follower; and (2) the workload characteristics (e.g. the arrival rate).

Log Replay. To avoid the follower lagging behind the leader too much, 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. The maximal committed log sequence number (MaxComLSN) is piggybacked on logs to notify the follower the latest committed point. Conventional quorum replication schemes only allows logs with LSN less than MaxComLSN to be replayed. 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 committed data be visible. Real Applications such as HTAP often take real-time OLAP analysis over follower nodes [20], and it's expected that there is a as small as possible VGap between leader and followers.

Recently proposed solutions to VGap aim at resolving the problem in the asynchronous primary-backup replication, which can not be applied to the quorum-based replication [16,21]. In the asynchronous primary-backup replication, follower could replay the received logs immediately without any coordination with leader. However, in the quorum-based replication, it is leader that notifies followers the consensus decision of committing transactions by sending the current MaxComLSN. After receiving MaxComLSN, follower nodes are agreed to replay logs with LSN no larger than MaxComLSN. Since it's expensive to read logs from disk for replay, the replicated and uncommitted 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 [27], un-replayed logs in the buffer can be soon erased by the new arrivals if the size of replay buffer is insufficient. Followers still needs to read flushed logs from disk for replay, and would definitely lag behind the leader and produces larger and larger VGap. Therefore, to achieve read scalability for IMDB replicated by quorum based protocols, VGap of a follower should be minimized in order to keep up with the state of leader.

3 Adaptively Self-tuning Batching Scheme

The design objectives of batching scheme 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 uncertainty 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 transactional logs. Last but not least, considering the processing capacity of follower is essential for adaptively tuning algorithm, especially in the case where the whole performance relies on the processing speed of followers under heavy workloads. Follower replicas may be overloaded if logs are replicated with a wrong batch size (Table 1).

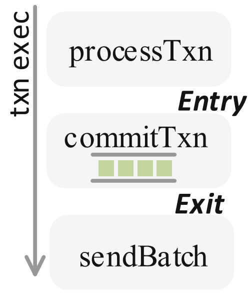
Table 1. Features of batching algorithms.

Batching scheme	Parameter-free	Workload-adaptive	Replica-friendly
JPaxos [13]	×	✓	×
Nuno Santos [25]	×	✓	×
Paolo Romano [24]	×	✓	×
AB [11]	✓	×	×
TAB [11]	×	✓	×
QumrumX	✓	✓	✓

3.1 Batching Scheme

Based on the above design objectives, we propose to combine critical-path-based batching (CB) [11] 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



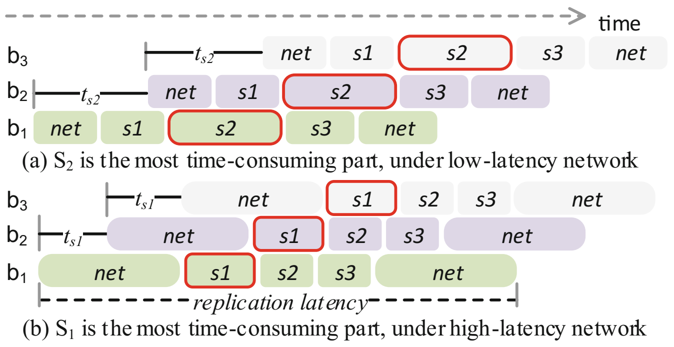

time
b3
b2
b1
t_{s2}
net
s1
s2
s3
(a) S2 is the most time-consuming part, under low-latency network
t_{s1}
replication latency
(b) S1 is the most time-consuming part, under high-latency network

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

Fig. 3. Pipeline-based batching (PB).

fragment, that is `commitTxn`. The entry code is used for registering the commit queue as the task is inside. Similarly, the exit code deregisters 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 follower 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 follower. As described above, 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. As a result, 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 a 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.

3.2 Discussion

We demonstrate that network latency between leader and follower has no effect to set sending frequency with Fig. 3(a) and (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 some unknown applications. 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 Coordination-Free Log Replay

4.1 Design Choices for Replay Buffer

The replay buffer in follower is an important structure which is responsible for caching the received logs from the leader. The persistence thread can flush a batch of buffered logs at one time, and the replay thread can directly replay the buffered log to keep in sync with the primary. 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.

The size of the replay buffer is a key design consideration. IMDB such as SiloR could generate logs at gigabytes-per-second rates. Caching all logs in the replay buffer leads to excessive memory consumption. If the size is set to a small value, the buffered and non-replayed logs would be covered by the new arrivals under heavy workload. Reading the received yet covered logs from disk for log replay would introduce extra disk I/O latency. This causes the risk of cascading latency as more non-replayed logs continue to be covered by newly arrived logs. Finally, it will make the follower nodes never catch up with the leader. The basic idea of determining the buffer size is that it should be greater than the rate of log generation on the leader.

In order to provide read services on fresh data by followers, they need to replay received logs to memory as soon 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's should be emphasized that the second phase is extremely lightweight without introducing overhead as the installation only contains a few pointer manipulations.*

4.2 Mechanism of Coordination-Free Log Replay

Basically, different from transaction execution in the leader, there has **NO** rollback when replaying logs in follower. That means all of the logs must be replayed successfully in 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 (replay workers) works in parallel. Replay worker first starts a transaction for each log entry. Then it looks up the memtable to find the node that 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 won't directly installed modifications into the memtable and therefore has no need to acquire any locks. The uncommitted cell informations are stored in the transaction context.

Algorithm 1. QuorumX commit algorithm of replaying

```
   /* Commit transactions according to log sequence */
   Input: $MaxComLSN$
1  while !$thread\_.stop()$ do
       /* Get a transaction from commit queue sequentially. */
2      $log\_id = commit\_queue.seq\_$;
3      while $true$ do
4          if $log\_id > MaxComLSN$ then
5              wait($wait\_time\_ms$);
6              continue;
7          $txn\_ctx = commit\_queue$.get($log\_id$);
8          if $NULL == txn\_ctx$ then
9              wait($wait\_time\_ms$);
10             continue;
11         __sync_bool_compare_and_swap($\&commit\_queue.seq\_, log\_id, log\_id + 1$);
12         break;
       /* Install the modification into memtable. */
13     for $cell\_info$ in $txn\_ctx.uc\_info$ do
14         $memnode = cell\_info \rightarrow node$;
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 commit queue of a single commit thread. It was the single commit thread that ensures the safety and consistency of quorum-based replication. In the second phase of CLR, commit thread 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).

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 avoid 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 scheme waiting for MaxComLSN.

4.3 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 haven't received log five. According to election algorithm, R_4 is elected as the new leader. It generates a different log five and replicates it, and there is 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 don't adopt CLR, there are no dirty contents in R_2's memory, R_2 only needs to rewrite log five to its disk.

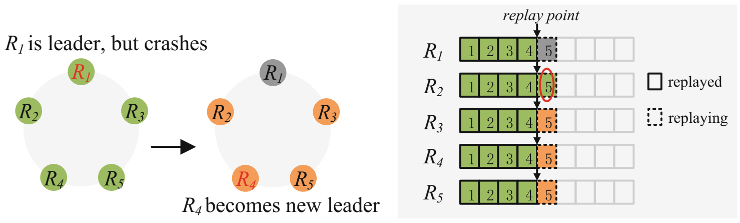


Fig. 4. An example illustrates a fault caused by CLR.

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 structure that stores data nor hold any locks. The commit thread pops all tasks from its commit queue, cleaning uncommitted cell informations and ending these transactions.

5 Evaluation

In this section, we evaluate the performance of QuorumX for 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 AB [11] and JPaxos [13]. Since the calculation of offline models in [24,25] requires a lot of additional parameters which are difficult to collect, we didn't implement them in QumrunX.
- The final 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.

Experiment Setup. We have implemented QuorumX in Solar [10], an open-source, scalable IMDB. We implement QumrumX by adding or modifying 31282 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, 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 2.3* GHz *20-core E5-2640 processors*, *504* GB DRAM, and connected by a *10 Gigabit Ethernet*.

5.1 Workloads

In the following experiments, we use three benchmarks that allows us to measure how QuorumX performs in specific aspects.

YCSB. The Yahoo! Cloud Serving Benchmark (YCSB) [6] is designed to evaluate large-scale Internet applications. The scheme 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. This benchmark models a warehouse ordering processing which simulates an industry OLTP application. 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. As a fully functional database, Solar requires to interact with clients, interpret SQL statements and translate them into physical execution plans, so it could not achieve a similar performance like Silo. Therefore, we build a write-intensive micro-benchmark, which originates from a realistic bank application used for importing massive data into databases everyday. Instead of sending the leader IMDB transaction requests coded by SQL statements, this micro-benchmark directly issues raw write operations to the leader. As a result, the micro-benchmark makes leader IMDB is running under extremely high-concurrent, write-intensive workloads. By default, the micro-benchmark contains 10 GB data modifications, which could produce gigabyte of logs per second.

5.2 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).

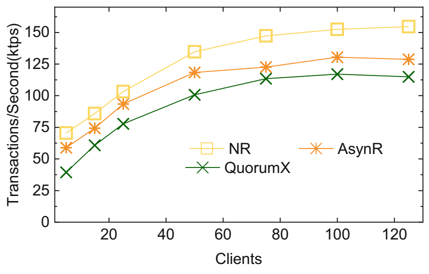


Fig. 5. Throughput of YCSB.

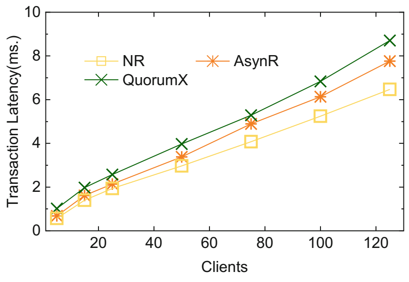


Fig. 6. Latency of YCSB.

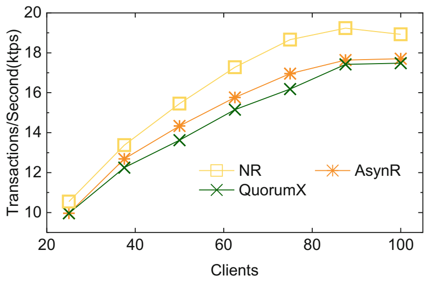


Fig. 7. Throughput of TPC-C.

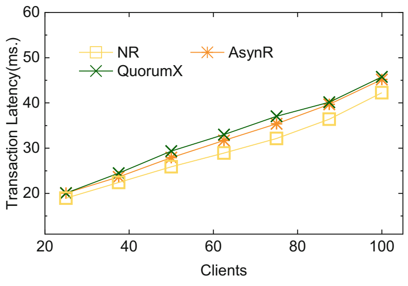


Fig. 8. Latency of TPC-C.

Experimental results of YCSB are shown in Figs. 5 and 6. We can observe that the throughput trend of all replication scheme is increasing firstly and then remaining at a high level. 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 7 and 8 illustrates 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.

5.3 The Ability of Adaptive Self-tuning

To compare the performance of self-tuning batching scheme of QuorumX with other batching algorithms, we implemented a parameter-free method—AB, which adopts critical-path-based batching. We choose AB instead of TAB since the batching method of AB is totally parameter-free. Besides, JPaxos, which needs manually set the parameter of batch size, is also compared with QuorumX 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 9 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 AB is close to that of QuorumX when the concurrency is low. However, as the number of clients increases, AB could not achieve good performance. Recall from Sect. 3, 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.

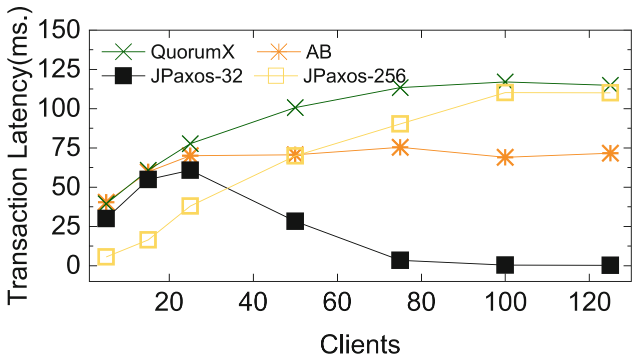


Fig. 9. Throughput under different client concurrency.

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.

5.4 VGap Results

We measure the VGap between the leader and followers to explore the effectiveness of CLR under a continued, write-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 10 shows the VGap results over 60 s. 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

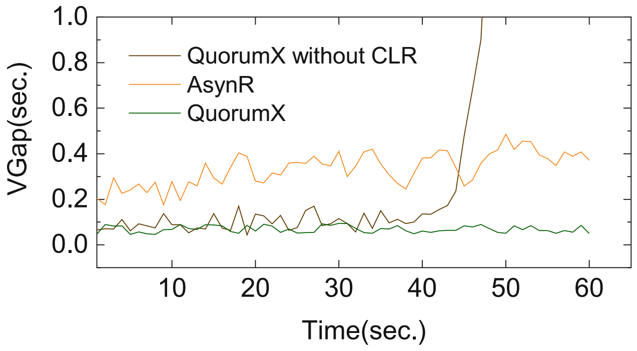


Fig. 10. VGap under write-heavy load.

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.

5.5 The 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. 11. The number of clients is fixed to 125. We can see that under workload with 100% writes, the performance 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.

6 Related Work

Replication is an important research topic across database and distributed system communities for decades [15,18]. In this section, we review relevant works mainly on two widely used replication schemes, i.e. primary-backup replication and quorum based replication.

Primary-Backup Replication. Asynchronous primary-backup replication [26], proposed by Michael Stonebraker in 1979, has been implemented in

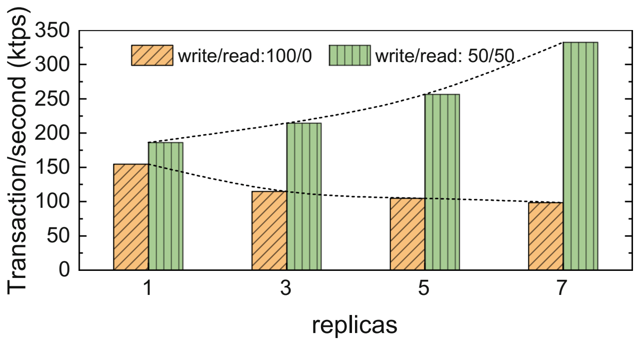


Fig. 11. Throughput over replicas number.

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 [12,16] 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 [12] 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. [21] 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 MaxComLSN 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 [17] 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 [5] 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.

Quorum-Based Replication. 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 [9]. Paxos variants such as Multi-Paxos used by Spanner [7] are designed to improve the performance. Raft [19] 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 [2] 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 [8], 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 [13,25]. Nuno Santos et al. [25] provide an analytical model to determine batch size and the pipeline size through gathering a lot of parameters, like bandwidth and the application properties. [13] 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.

7 Conclusion and Future Work

In this paper, we built QuorumX, an efficient 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. 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 MaxComLSN. 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.

QuorumX is designed for fast IMDBs without harsh assumptions, so it is also applicable to NoSQL systems. In our future work, we will erect a more general, pluggable quorum-based replication framework that not only provides high replication performance for these high-throughput systems but also takes many complicated factors like network bandwidth into consideration.

Acknowledgement. This work is partially supported by National Key R&D Program of China (2018YFB1003404), NSFC under grant numbers 61432006, and Guangxi Key Laboratory of Trusted Software (kx201602). We thank anonymous reviewers for their very helpful comments.

References

1. AliSQL. https://github.com/alibaba/AliSQL
2. etcd. https://coreos.com/etcd/
3. IBM DB2. https://www.ibm.com
4. Oracle Corporation and/or its affiliates. MySQL Cluster (2017)
5. W. contributors. Apache kafka (2018). https://en.wikipedia.org/w/index.php?title=Apache_Kafka&oldid=831864654
6. Cooper, B.F., Silberstein, A., Tam, E., Ramakrishnan, R., Sears, R.: Benchmarking cloud serving systems with YCSB. In: SoCC (2010)
7. Corbett, J.C., Dean, J., Epstein, M., Fikes, A., et al.: Spanner: Google's globally distributed database. ACM Trans. Comput. Syst. **31**(3), 8:1–8:22 (2013)
8. Hunt, P., et al.: ZooKeeper: wait-free coordination for internet-scale systems. In: USENIX ATC (2010)
9. Chandra, T.D., et al.: Paxos made live: an engineering perspective. In: PODC (2007)
10. Zhu, T., et al.: Towards a shared-everything database on distributed log-structured storage. In: ATC (2018)
11. Friedman, R., Hadad, E.: Adaptive batching for replicated servers. In: 25th IEEE Symposium on Reliable Distributed Systems, pp. 311–320 (2006)
12. Hong, C., Zhou, D., Yang, M., Kuo, C., Zhang, L., Zhou, L.: KuaFu: closing the parallelism gap in database replication. In: ICDE (2013)
13. Kończak, J., de Sousa Santos, N.F., et al.: JPaxos: state machine replication based on the Paxos protocol. Technical report (2011)
14. Zheng, J., et al.: PaxosStore: high-availability storage made practical in WeChat. PVLDB **10**(12), 1730–1741 (2017)
15. 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)
16. Lee, J., Moon, S., et al.: Parallel replication across formats in SAP HANA for scaling out mixed OLTP/OLAP workloads. PVLDB **10**, 1598–1609 (2017)
17. Lin, W., Yang, M., Zhang, L., Zhou, L.: PacificA: replication in log-based distributed storage systems (2008)
18. Wiesmann, M., Pedone, F., et al.: Database replication techniques: a three parameter classification. In: SRDS, pp. 206–215 (2000)
19. Ongaro, D., Ousterhout, J.K.: In search of an understandable consensus algorithm. In: ATC, pp. 305–319 (2014)
20. Özcan, F., Tian, Y., Tözün, P.: Hybrid transactional/analytical processing: a survey. In: SIGMOD Conference, pp. 1771–1775. ACM (2017)
21. Qin, D., Goel, A., Brown, A.D.: Scalable replay-based replication for fast databases. PVLDB **10**(13), 2025–2036 (2017)

22. Rao, J., Shekita, E.J., Tata, S.: Using paxos to build a scalable, consistent, and highly available datastore. PVLDB **4**, 243–254 (2011)
23. Liu, Y.A., Chand, S., Stoller, S.D.: Moderately complex Paxos made simple: high-level specification of distributed algorithm. CoRR abs/1704.00082 (2017)
24. Romano, P., Leonetti, M.: Self-tuning batching in total order broadcast protocols via analytical modelling and reinforcement learning. In: ICNC, pp. 786–792 (2012)
25. Santos, N., Schiper, A.: Tuning paxos for high-throughput with batching and pipelining. In: Bononi, L., Datta, A.K., Devismes, S., Misra, A. (eds.) ICDCN 2012. LNCS, vol. 7129, pp. 153–167. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-25959-3_11
26. Stonebraker, M.: Concurrency control and consistency of multiple copies of data in distributed INGRES. IEEE Trans. Softw. Eng. **5**(3), 188–194 (1979)
27. Zheng, W., Tu, S., et al.: Fast databases with fast durability and recovery through multicore parallelism. In: USENIX OSDI (2014)