ALOHA-KV: High Performance Read-only and Write-only Distributed Transactions

Hua Fan
University of Waterloo, Canada
Waterloo, Canada
h27fan@uwaterloo.ca

Wojciech Golab
University of Waterloo, Canada
Waterloo, Canada
wgolab@uwaterloo.ca

Charles B. Morrey III*
cbmorrey@gmail.com

ABSTRACT

There is a trend in recent database research to pursue coordination avoidance and weaker transaction isolation under a long-standing assumption: concurrent serializable transactions under read-write or write-write conflicts require costly synchronization, and thus may incur a steep price in terms of performance. In particular, distributed transactions, which access multiple data items atomically, are considered inherently costly. They require concurrency control for transaction isolation since both read-write and write-write conflicts are possible, and they rely on distributed commitment protocols to ensure atomicity in the presence of failures. This paper presents serializable read-only and write-only distributed transactions as a counterexample to show that concurrent transactions can be processed in parallel with low-overhead despite conflicts.

Inspired by the slotted ALOHA network protocol, we propose a simpler and leaner protocol for serializable read-only write-only transactions, which uses only one round trip to commit a transaction in the absence of failures irrespective of contention. Our design is centered around an epoch-based concurrency control (ECC) mechanism that minimizes synchronization conflicts and uses a small number of additional messages whose cost is amortized across many transactions. We integrate this protocol into ALOHA-KV, a scalable distributed key-value store for read-only write-only transactions, and demonstrate that the system can process close to 15 million read/write operations per second per server when each transaction batches together thousands of such operations.

CCS CONCEPTS

• **Information systems → Distributed database transactions**; **Distributed data locking**;

KEYWORDS

Epoch-based concurrency control, consistency, distributed transaction, multi-put or multi-get key-value store

*Work done while author was at Hewlett-Packard Labs, Palo Alto.

SoCC '17, September 24–27, 2017, Santa Clara, CA, USA
© 2017 Copyright held by the owner/author(s). Publication rights licensed to Association for Computing Machinery.
ACM ISBN 978-1-4503-5028-0/17/09…$15.00
https://doi.org/10.1145/3127479.3127487

ACM Reference Format:
Hua Fan, Wojciech Golab, and Charles B. Morrey III. 2017. ALOHA-KV: High Performance Read-only and Write-only Distributed Transactions. In *Proceedings of SoCC '17, Santa Clara, CA, USA, September 24–27, 2017,* 12 pages. https://doi.org/10.1145/3127479.3127487

1 INTRODUCTION

To overcome the performance gap between traditional relational databases and web-scale data requirements [12, 33], recent research efforts have explored coordination avoidance in special cases [3, 25], as well as the more general trade-off between transaction isolation and performance under the following assumption: concurrent serializable transactions under read-write or write-write conflicts require costly synchronization, and thus may incur a steep price in terms of performance [3]. However, this assumption ignores the possibility that conflicting writes need not block each other, or violate serializability if read-write conflicts are not present at the same time.

An alternative school of thought favors the continued use of transactions and strong consistency in distributed storage systems [1, 2, 4, 10]. ACID transactions shelter application developers from complex concurrency control mechanisms and from analysis of application-level consistency violations under weak isolation.

Atomic read-only or write-only transactions, although less powerful than general ACID transactions, have been debated intensely in recent research [11, 14, 19, 27]. They are well suited to systems that process reads and writes in batches for efficiency, but also require atomicity for each batch (i.e., two writes within one batch must both succeed or both fail). For example, StoreAll with Express Query [16] is a scalable file metadata system built on top of LazyBase, a distributed storage layer in which clients batch updates together into *self-contained units* (SCUs) [9]. Write-only transactions naturally support the atomic insertion of an SCU containing up to thousands of updates. For instance, atomically moving a set of files from system A to system B, involves deleting metadata in system A and inserting in system B. Another application of such transactions is distributed system automatic reconfiguration [26] to achieve data migration and dynamic replication factor adjustment [22, 29]. Although this paper focuses on processing read or write operations in batches, atomic read-only or write-only transactions can be used to support read-write transactions as discussed in Section 8.

While useful in practice, read-only or write-only transactions in distributed storage systems inherit two costly aspects of conventional transactions: they require concurrency control for transaction isolation since both read-write and write-write conflicts remain possible, and they rely on distributed commitment protocols to ensure atomicity in the presence of failures. Many systems either bite the bullet and pay a performance penalty for serializable distributed transactions [2, 10, 28], or sacrifice serializability while providing some alternative form of strong consistency [4, 18]. The transactional solutions rely on costly atomic commitment protocols in the sense that at least two network round trips are required to commit a transaction in the presence of contention. The high contention footprint of such protocols easily becomes a performance bottleneck as contention increases, which triggers additional protocol messages as conflicting transactions resolve their relative serialization order.

In search of a simpler and leaner protocol for read-only/write-only(ro/wo) transactions, we present a system that draws inspiration from the slotted ALOHA network protocol [24]. Specifically, we propose a scheme for supporting serializable multi-partition read-only and write-only transactions by splitting time into read-only and write-only epochs. Our design relies on an *epoch-based concurrency control* (ECC) mechanism that minimizes conflicts between multi-partition transactions while using minimal metadata, thus enabling high throughput. On the other hand, ECC leads to potentially greater latencies and latency variations by forcing transactions to execute in alternating read-only and write-only epochs.

To better understand the performance envelope of ECC, we incorporate it into a scalable distributed key-value store, called ALOHA-KV, and compare it against RAMP [4]. The experimental results show that when transaction size exceeds 6-10 key-value pairs, our system outperforms RAMP in terms of both throughput *and* latency, at the same time providing stronger transaction isolation.

2 SYSTEM MODEL AND ARCHITECTURE

ALOHA-KV is a scalable multiversion storage system that supports serializable read-only and write-only transactions across multiple data partitions. The system is optimized for high throughput. ALOHA-KV is therefore most suitable for applications that tolerate larger latencies and latency variations, although as we show in Section 6, it can be tuned to achieve a balance of latency and throughput that meets or exceeds a best-of-breed system.

Internally, ALOHA-KV uses a distributed transaction protocol that combines epoch-based concurrency control (ECC) for transaction isolation and an atomic commitment mechanism whose amortized complexity is one network round trip per transaction.

2.1 System Model

In memory DB. All data can reside in main memory.
Horizontal partitioning. Each data item, identified by a key, has a single logical copy in its hash partition.
Read-only and write-only transactions. The system exploits the special structure of read-only and write-only transactions to minimize concurrency control overheads.
Multiversioning. Each write operation creates a new version of a data item, identified by a distinct version number. We leverage object versions for concurrency control, but their main purpose is to support historical queries. The version number is therefore a timestamp generated by the system at the beginning of transaction processing. Versions older than a user-specified threshold can be removed by a garbage collector to free memory.
Timestamps and clocks. To generate a unique timestamp, a server combines its unique server ID, a monotonically increasing number, and the time from its local clock. Tight clock synchronization across servers benefits performance but is not required for correctness of ECC. Standard synchronization techniques suffice, such as NTP executed over a low-latency network.
Serializable isolation. Committed transactions are serialized according to their timestamps. Write-only transactions are physically isolated from read-only transactions by our epoch-based concurrency control mechanism, which automatically deals with read-write conflicts. Conflicting write-only transactions are permitted to execute in parallel since they act on different versions of data. Read-only transactions that access old versions of data can be executed during a write epoch without causing conflicts as long as the version accessed precedes the start of the write epoch. Otherwise, such transactions must wait until the next read epoch.

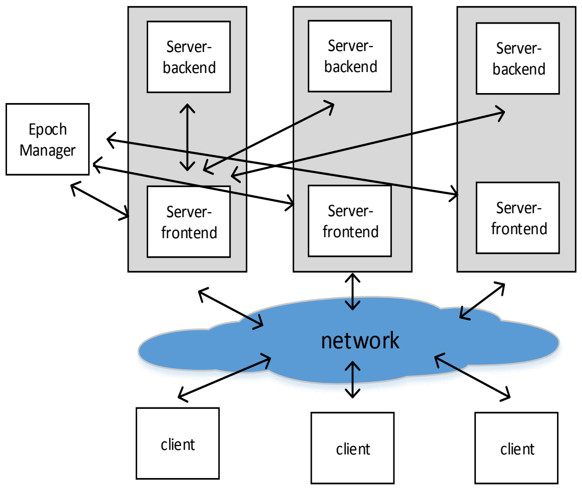

Figure 1: ALOHA-DB architecture.

2.2 Architecture

The architecture of ALOHA-KV, illustrated in Figure 1, comprises a collection of server backends (BEs), server frontends (FEs), and an epoch manager (EM).
FE: the transaction coordinator. An FE accepts transaction requests from clients, and acts as a transaction coordinator: it starts transaction execution during the correct epoch, generates a timestamp for each transaction, and communicates with the BEs to determine the outcome of each transaction. Clients may connect to any FE, and each FE may contact any BE where the required data items reside.
EM: decides epochs. The EM communicates with all FEs to control epoch changes, and thus determines when the FEs are able to start executing a given transaction. The durations of read and write

epochs are either determined automatically by the EM, or tuned manually.

BE: the data store. A BE stores the data items in one partition of the database, and serves requests from FEs to read and write these items.

Deployment. The system is optimized for deployment in a private data center with a high-speed network. Although not necessary for correctness, good network performance and predictability (e.g., low jitter) help our system achieve high throughput and low latency. BE/FE pairs can be co-located on the same host, denoted by a gray box in Figure 1. Each system component can be replicated for fault tolerance, as discussed in Section 4.4. Furthermore, the epoch manager can be distributed for scalable performance, as discussed in Section 5.2.

3 EPOCH-BASED CONCURRENCY CONTROL MECHANISM

ALOHA-KV achieves transaction isolation using *Epoch-based Concurrency Control (ECC)*. Conceptually, ECC is the combination of two techniques that maximize parallelism. First, ECC schedules read-only transactions and write-only transactions into disjoint time slots, called read-only and write-only epochs, to eliminate read-write conflicts. Note that ALOHA-KV accepts both types of transactions at all times, and merely delays the start of transaction execution as needed to ensure that each transaction runs during the correct epoch type. For example, a write-only transaction accepted during a read epoch begins executing in the next write epoch. Second, ECC uses multiversioning to resolve write-write conflicts, which allows write-only transactions to proceed in parallel even when their write sets overlap. Both techniques combined ensure that transactions never abort or deadlock due to conflicts, which benefits throughput under contention.

In addition to dealing with conflicts efficiently, ECC minimizes communication overheads by simplifying atomic transaction commitment. Specifically, ECC cannot have "reads-from" dependencies among transactions executing within the same epoch, which means that the effects of a partially committed transaction cannot be observed until the next epoch. This enables one-phase commitment for write-only transactions, with a second phase required only if the transaction must be rolled back, for example due to an abort on failure. Any additional messages needed to orchestrate epoch switches are amortized over a large number of transactions. In contrast, two-phase commit requires both phases even when a transaction commits in the failure-free case.

In this section we describe implementation details pertaining to the epoch switching mechanism.

3.1 States and Invariants

The performance benefits of ECC are contingent on tight synchronization of the epoch status (read vs. write) across FEs. The synchronization mechanism records state information at the EM and FEs and guarantees a number of invariants with respect to this state.

Authorization. An FE can start processing a transaction only if it holds appropriate *authorization*, which is granted by the EM. An authorization comprises the epoch type (read or write), as well as two timestamps indicating a finite *validity period*. Transaction timestamps are always assigned within the validity period. An FE may hold at most one authorization at a time, ensuring exclusion among read-only and write-only transactions.

Epoch duration. From the point of view of an FE, an epoch is the period of time from when an authorization is granted by the EM to when the authorization is revoked, which is always after the end of the validity period.

Timestamp generation. A write-only transaction is assigned a timestamp when it is started by an FE. Recall that the timestamp is also the version number of the transaction. The FE guarantees that the timestamp is within the epoch's validity period to ensure that the serialization order of transactions, which is determined by the timestamps, is consistent with the order of epochs.

Transaction start policy. Write-only transactions can only be started under valid write authorization. Similarly, read-only transactions retrieving the latest data version can only be started under valid read authorization. However, read-only transactions accessing old versions of data (i.e., historical queries) can be started either under valid read authorization, or under valid write authorization if the version accessed precedes the start of the current validity period.

Transaction completion policy. A transaction that begins in one epoch must complete within the same epoch. Once the validity period of an epoch expires, an FE must wait for all pending transactions to complete before acknowledging to the EM that authorization has been revoked.

Total order on epochs. The EM must revoke all authorizations from FEs before granting a new authorization, which is tagged with a monotonically increasing ID. Epochs and their corresponding authorization validity periods are therefore disjoint.

Alternating epoch types. The EM drives an alternating sequence of read and write epochs, meaning that the epoch type changes each time an authorization is granted.

3.2 Transaction Barriers

For clarity of presentation, the pseudo-code presented later on in Section 4 omits the low-level details of the message passing protocol for epoch switching and instead use a high-level API we call a *transaction barrier*. The API comprises two primitives called *Begin_Barrier* and *Finish_Barrier*. Begin_Barrier checks the FE's current epoch authorization and either admits the transaction if the authorization is valid and has the correct type, or else blocks the transaction until the correct authorization becomes valid. If the transaction is admitted, the count of in-flight transactions is increased by 1. Finish_Barrier retires transactions by updating the count of in-flight transactions, which must reach zero before the revocation of authorization.

3.3 Example

Figure 2 illustrates ECC using two FEs, each executing transactions in three worker threads. The epoch switch procedure works as follows:

(1) The EM grants the FEs write authorization. The start and end of the validity period are indicated by vertical dashed lines in the figure. In practice, the validity period should be

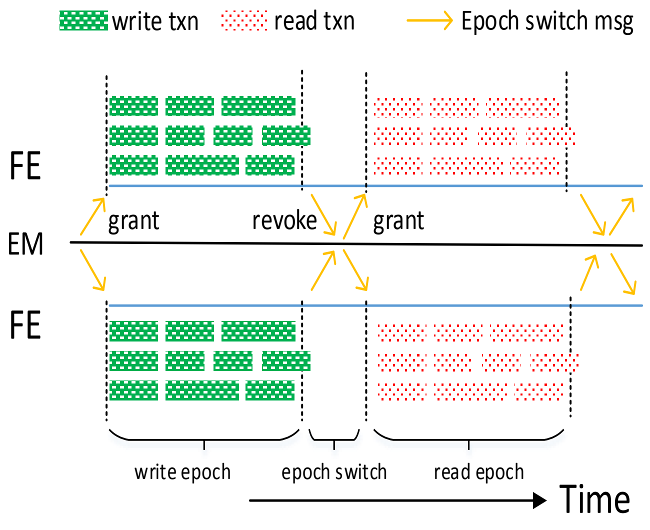

Figure 2: Illustration of epochs.

long enough so that the cost of epoch switching is amortized over many transactions.

(2) In the write epoch, each worker thread executes write-only transactions until the validity period expires. In the meantime, read-only transactions accessing the latest version are buffered but historical queries can be admitted if they access sufficiently old versions.
(3) Some pending transactions may continue to execute past the end of the validity period. All such transactions must be run to completion in the current write epoch before the next epoch begins.
(4) Once all pending transactions are complete, the FE acknowledges to the EM that the write authorization is revoked. The FE now awaits a read authorization.
(5) After the EM receives acknowledgments from all the FEs it grants a read authorization for the next epoch.

4 IMPLEMENTATION

To evaluate the performance envelope of ECC we incorporated it into a key-value storage system called ALOHA-KV. This section discusses the system's implementation details.

4.1 Data Representation in Storage

ALOHA-KV stores key-value pairs in a hash-partitioned distributed table. The values are versioned to support historical queries, as well as to enable multiversion concurrency control for write-only transactions. In other words, each version of a value is represented as a pair of the form $\langle version, value \rangle$. For each key, the versions are organized in a logical list ordered by version, implemented as a linked list of arrays. This data structure is a hybrid of linked list and array to accommodate removing old versions and efficiently accessing recent versions.

Inserting a new version of a key-value pair during a write-only transaction entails adding an entry to the array of the key, keeping the versions in sorted order. Since transactions with different timestamps execute in parallel during a write epoch, it is not always the case that the newly created version is the latest version. In other words, the new version may have already been overwritten by another transaction. Such an outdated version will not be visible to future read-only transactions that access the latest data, but can still be requested by a historical query. This is in contrast to conventional timestamp ordering with Thomas' write rule, where an outdated value need not be written at all. In practice new versions tend to arrive in nearly sorted order since the version numbers are derived from timestamps, and this enables efficient insertion. The system does not allow insertion of versions that predate the start timestamp of the write-only epoch.

Retrieving a value begins with determining the correct data version with respect to the transaction timestamp. This is either the latest version, if the transaction requests the latest data, or the latest version not exceeding a given version number, if the transaction is a historical query.

Deleting a version from the table occurs in two situations: when old versions are cleaned up by the garbage collector to recover memory, or when a transaction aborts in a write epoch. In general, deleting a version entails removing the corresponding items from the array of the key. In the case of garbage collection, the deleted version must be older than a threshold that is no longer needed by any reads. Garbage collection can be triggered periodically, or when the system is low on memory.

4.2 Transaction Protocol

The transaction protocol executed by the FE and BE is presented in Algorithms 1 and 2, respectively. The epoch switching mechanism and interaction with the EM is represented implicitly in Algorithm 1 using transaction barriers, which were described earlier in Section 3. The entire protocol is implemented in C++ using fbthrift—a popular open-source RPC framework [13].

As presented in Algorithm 1, transaction execution begins with the invocation of the `PutAll`, `GetAllLatest`, or `GetAllHistorical` procedure at an FE, which acts as a coordinator. The FE executes a transaction barrier to ensure that it has appropriate authorization for the given transaction, and then accesses the relevant keys in one or more partitions by invoking *partition requests*—calls to procedures `Put` and `Get` at BEs. `Abort` is called to roll back a write-only transaction.

For a write-only transaction, the FE first generates a unique timestamp *ts* (as described in Section 2) at line 3 of `PutAll`. It then builds a set of data versions at line 5, and distributes these versions to different partitions at line 7 by calling `Put` on different BEs. For simplicity, the pseudo-code shows separate calls to `Put` for each data item (similarly for `Abort` and `Get` later on), but in practice requests destined for the same BE are batched together. Each data version includes the key-value pair, transaction timestamp, epoch ID, as well as the transaction size (used in recovery, see Section 4.4.2). The `PutAll` procedure is regarded as successful if every call to `Put` has succeeded, otherwise if one or more calls fail then the FE invokes `Abort` on each partition involved to roll back the transaction. This is the second round of the atomic multi-write protocol.

A *read-only transaction* is executed similarly, but never needs a second round of messaging to abort. This is because a read-only transaction does not alter the state of a BE, and so there are no actions to roll back on failure. Procedure `GetAllLatest` returns the latest data versions for the given set of keys by calling procedure `Get` on respective partitions with a special timestamp value of ⊥. Procedure `GetAllHistorical` executes a historical query and accepts a user-specified timestamp that defines the desired data version. Internally, the procedure behaves similarly to `GetAllLatest` but uses a transaction barrier only during a write epoch if the requested timestamp exceeds the start of the current validity period.

Procedures `Get`, `Put`, and `Abort` at BEs operate on the multiversion storage *MV* described in Section 4.1 for insertions, lookups, and deletions of key-value pairs. Procedure `Put` inserts versions, `Abort` deletes entries, and `Get` retrieves the latest version as of a given timestamp.

Algorithm 1: Transaction protocol for FE.

```
1  Procedure PutAll(W: set of ⟨key k, value v⟩)
2      Begin_Barrier
3      ts = generate new timestamp
4      eid = current epoch ID
5      $V = \{(w.k, w.v, ts, eid, |W|) \mid w \in W\}$
6      parallel-for $v \in V$ do
7          invoke Put(v) on respective partition
8      if any call to Put fails then
9          parallel-for $v \in V$ do
10             invoke Abort(v) on respective partition
11     Finish_Barrier
12 Procedure GetAllLatest(K: set of keys)
13     Begin_Barrier
14     ts = generate new timestamp
15     Finish_Barrier
16     parallel-for $k \in K$ do
17         invoke Get(k, ts) on respective partition
18     return union of responses from calls to Get
19 Procedure GetAllHistorical(K: set of keys, ts: timestamp)
20     if holding write authorization with validity period starting
         at or before ts then
21         Begin_Barrier
22         Finish_Barrier
23     parallel-for $k \in K$ do
24         invoke Get(k, ts) on respective partition
25     return union of responses from calls to Get
```

4.3 On Straggler Side Effects

A straggler transaction is one that prevents an FE from revoking an authorization for a long time. It may delay the start of the next epoch for all FEs, and further degrade the overall throughput. Stragglers may arise from resource limitations during transaction processing, from long running transactions, or from software/hardware anomalies.

Algorithm 2: Transaction Protocol for BE.

```
   Data: MV: multiversioning storage described in section 4.1
1  Procedure Put(v: ⟨key, value, ts, eid, txnsize⟩)
2      return $MV[key].insert(ts, v)$
3  Procedure Abort(v: ⟨key, value, ts, eid, txnsize⟩)
4      $MV[key].remove(ts)$
5      return
6  Procedure Get(k: key, ts: timestamp)
7      return $MV[k].get(ts)$
```

In the absence of anomalies, long delayed stragglers are unlikely to occur in our system for the following reasons. *First*, the number of in-flight transactions (preventing authorization revocation) decreases to zero after the epoch's finish timestamp is reached. As a result, in the course of an epoch switch, the contention among in-flight transactions tends to zero. *Second*, our system only handles one-shot transactions and is able to process them quickly by reading and writing in-memory data within epochs. In our experiments, we did not observe long-running stragglers even for transactions including thousands of keys.

Another observation is that slow read-only transactions will not block revoking authorization and delay starting of the next write epoch. This is because the read-only transaction only accesses historical versions after it gets a timestamp at line 14 of Algorithm 1 and then releases the barrier at line 15.

4.4 Fault Tolerance

ALOHA-KV relies on main memory storage for performance, and therefore depends crucially on appropriate fault tolerance mechanisms to protect against data loss and maintain system availability in the event of a server failure. In this section we explain how the strategies of replication, logging, and checkpointing to persistent storage are applied to different components of the system.

4.4.1 Replication. **BE replication.** Replication is essential for BE servers, which store key-value pairs. During write epochs, the FE writes primary BEs as well as backup BEs. Note that in ECC, write-only transactions can achieve both concurrency control and replication in one round trip amortized if there are no aborts. In comparison, traditional 2PL/2PC with primary-backup requires two rounds for 2PC plus an extra round for replication. During read epochs, load balancing is possible between primary and backup servers because they hold exactly the same data.

Fast epoch switch at FE/EM failure. Failures of FE servers do not lead to loss of data, but may stall the ECC mechanism entirely. This is because the EM must receive acknowledgments of authorization revocation from the previous epoch before granting authorization for the next epoch. The backup FE is therefore charged with completing all pending transactions and reporting back to the EM if the primary FE fails during a write epoch. This requires that the primary forwards each transaction to the backup at the beginning of transaction execution, and confirms to the backup after execution. If the primary FE fails, the backup acts temporarily as the coordinator, simply aborting all in-flight transactions in the interest of a fast epoch switch. Once a BE is contacted by the backup

FE, it deems the primary is failed and rejects any further requests from the primary FE within the epoch. This mechanism is similar to [5, 15] and deals with the anomaly where both primary and backup believe they are the primary. As clients continue to send requests to FEs, the failed FE is not allowed to participate in future epochs. In the course of an epoch switch the FEs become nearly idle and so aggressive failure detection (e.g., based on a sub-second heartbeat) can be used.

Similarly, failure of the EM can lead to loss of synchronization among FEs, for example with some servers starting a new epoch while others await authorization. When the primary EM fails, the backup first polls the FEs to determine how many of them have not yet acknowledged authorization revocation for the most recent epoch. When the outstanding acknowledgments are received, the backup EM may resume ordinary execution by granting the next authorization. EMs use a handover mechanism and aggressive failure detection similarly to FEs.

4.4.2 Logging and Checkpointing. To protect data against a system-wide failure, such as loss of power to an entire rack of servers or data center, ALOHA-KV persists data by logging operations to secondary storage. Specifically, the BEs log all the requests they receive from FEs during write epochs. To avoid a performance bottleneck, our design opts for an asynchronous form of logging whereby worker threads append log entries to an in-memory buffer that is flushed periodically by a dedicated thread. We ensure that all data are flushed to disk before the next epoch starts, thus enabling the following guarantees: (1) a transaction executed in a failure-free write epoch is never lost; (2) a write-only transaction whose effects are observed by a read-only transaction is never lost; and (3) transaction recovery is atomic: either all or none of the operations in a transaction are recovered.

A checkpoint is a persistent snapshot of the whole database. As a multiversion system, ALOHA-KV can create a consistent checkpoint simply by dumping the versions of all keys before a given timestamp. Our design creates checkpoints only at an epoch finish timestamp for easy recovery.

Recovery logs are organized into a collection of files, with one file per BE per write epoch. Each log file is capped with a special record at the end of a write epoch to indicate that it is complete. After a failure, the recovery procedure first reloads a checkpoint and all completed log files. Logs from the most recent epoch are then sent to a recovery coordinator, which checks for each transaction whether the number of operations present in the logs matches the size of the write set, which is recorded in the partition requests (see line 5 in Algorithm 1). For atomicity, any transactions having a full complement of operations are replayed, and the remaining transactions found in incomplete logs are aborted.

5 THEORETICAL ANALYSIS

The performance envelope of ALOHA-KV is substantially different from other systems because ECC schedules transactions into epochs. This section discusses the factors affecting the latency and throughput of the system. We define throughput as the number of key-value pair operations per second, which in our experiments equals the number of transactions executed per second times the (fixed) transaction size. We also sketch out a proof of serializability.

5.1 System Throughput

The ECC protocol periodically switches between write epochs and read epochs. Consider a unit of execution containing one write epoch followed by one read epoch. Let P denote the overall throughput in the unit, P_W the throughput of writes in the unit, P_w the throughput of writes in the write epoch, P_R and P_r denote analogous quantities for reads. Let t_w, t_s, t_r denote the duration of a write epoch, the epoch switch time, and the duration of a read epoch, respectively. Let n_w denote the number of write operations executed in the write epoch. The overall write throughput is

$$P_W = \frac{n_w}{t_w + 2t_s + t_r} \qquad (1)$$

which can be rewritten as

$$P_W = P_w \times \frac{t_w}{(t_w + 2t_s + t_r)} \qquad (2)$$

since $P_w = n_w/t_w$. Using a similar equation for P_r, total throughput can be expressed as follows:

$$P = P_w \times \frac{t_w}{(t_w + 2t_s + t_r)} + P_r \times \frac{t_r}{(t_w + 2t_s + t_r)} \qquad (3)$$

Equation 3 suggests the following strategies to maximize throughput: increase P_w and P_r, decrease the epoch switch time t_s, and increase the epoch durations t_w and t_r.

For the first strategy, the main factors affecting P_w and P_r are as follows: (1) *Transaction size.* Larger transactions benefit from better network and processing efficiency due to batching, but they are more likely to overrun the validity period of an epoch, which complicates epoch switching. (2) *Number of servers.* On one hand, adding servers increases I/O and processing capacity. On the other hand, it increases the number of partitions and hence causes transactions to be processed by BEs in smaller pieces, which counters the benefits of batch processing.

For the second strategy, we observe that epoch switch time t_s roughly amounts to the sum of one round of communication between the EM and FEs, and the time required to complete any pending transactions. In particular, t_s is highly dependent on the slowest FE to acknowledge authorization revocation. We propose the following strategies to reduce t_s: (1) *Minimize EM-FE communication latency.* For example, use a separate execution path for epoch switch messages versus transaction messages, such as using a dedicated and prioritized thread for epoch switch. (2) *Handle stragglers individually.* Some servers may be consistently slower than others due to differences in hardware configuration or network connectivity, in which case they can be authorized by the EM for shorter epoch durations.

On clock skew. The clock skew between any FE and the EM cannot violate the serializability guarantee of ECC, because an FE must assign a transaction timestamp within the validity period given by the EM. In other words, the timestamp lies in the intersection of the FE's and EM's interpretations of the validity period according to their local clocks. Thus, if an FE has a large clock skew relative to the EM, the period during which FE can generate timestamps and process transactions may be small.

5.2 Epoch Switch Scalability

The transaction barrier is one of the main factors affecting system performance, as explained in Section 5.1, and therefore we consider two design alternatives: a centralized approach and a tree-structured hierarchical approach. The remainder of this section will develop performance models for both alternatives.

In the centralized approach, the epoch switch phase entails both message passing over the network and message processing to count how many FEs have reached the barrier. For simplicity, we assume the round trip network latency between hosts is a constant L_n, and that processing a message from one host also takes a constant time L_p. Assuming that all FEs finish pending transactions at the same time, the latency L of the barrier can be written as

$$L = L_n + L_p \times n \tag{4}$$

where n denotes the number of FEs.

In the hierarchical approach, let d denote the degree of the tree. In this model, the tree height is $\lceil \log_d n \rceil$, denoting the number of network hops from an FE to the EM in the tree, and each non-leaf node is expected to process d messages from child nodes. Thus, L in this model is

$$L = (L_n + d \times L_p) \times \lceil \log_d n \rceil \tag{5}$$

Note that equation 4 is a special case of 5 when $d = n$.

Finally, if FEs finish pending transaction processing at different times, and the slowest FE takes L_f time to finish pending transactions, the latency can be bounded as

$$L \leq L_f + (L_n + d \times L_p) \times \lceil \log_d n \rceil \tag{6}$$

which is tight when the slowest FE is a leaf node in the tree topology.

Formula 6 suggests that when network latency is low and message processing is expensive, a smaller d and larger tree height yield lower epoch switch latency L. Otherwise a larger d is preferred, which in the extreme case $d = n$ makes the hierarchical and centralized alternatives equivalent. Asymptotically, a hierarchical barrier has better complexity in terms of n ($O(\log n)$ in formula 5) than centralized ($O(n)$ in formula 4), but experimentally (see Section 6.4.2) the centralized approach yields very fast epoch switch times even with hundreds of FE nodes.

5.3 Analysis of Serializability

Informally, serializability is the illusion that transactions are processed in a serial order [8, 21]. We now sketch a proof that ALOHA-KV satisfies this property.

Lemma 5.1. *Every history of committed transactions generated by ECC is serializable.*

Proof sketch. Given a history H of committed transactions, assign a timestamp to each transaction as follows: for a write-only or read-only transaction use the timestamp assigned by the FE at line 3 and at line 14 of Algorithm 1, respectively; for a historical read-only transaction use the timestamp passed by the client to the `GetAllHistorical` procedure. Now arrange the transactions into a serial history S in increasing order of their timestamp, breaking ties arbitrarily for read-only transactions. It follows easily that H and S have the same committed transactions and operations. Furthermore, each read-only transaction obtains the highest version of a key that does not exceed its own timestamp, and this version is created by a unique write-only transaction. As a result, H and S have the same "reads-from" relationships, and moreover S is one-copy serial. This implies that H is serializable. □

6 EVALUATION

The experiments presented in this section demonstrate that ALOHA-KV performs favorably compared to RAMP [4] in terms of both throughput and latency for large transaction size (>6 for RAMP-S, >10 for RAMP-F). Micro-benchmark results show that although the latency of transactions in ALOHA-KV grows linearly with the epoch duration, short epochs (tens of ms) are sufficient to attain throughput levels close to the limit of the performance envelope. ALOHA-KV is able to process around 230M operations per second for large transactions using fifteen servers. We also show that a single EM can orchestrate epoch switches in a timely manner even when controlling hundreds of FEs.

6.1 Experimental Setup

The experiments were deployed in Amazon EC2, using c3.8xlarge virtual machine instances in a single availability zone. Clocks across hosts were synchronized using NTP to a double-digit microsecond clock offset (usually $< 20\mu s$).

The experiments use the following default settings: five client hosts and five server hosts running a co-located BE/FE pair (different processes), with one server host running the EM. Data items comprise eight-byte keys and values. We vary the transaction size, defined as the number of key/value pairs accessed per transaction. Keys are drawn uniformly at random from a space of 1M elements. The ratio of write-only to read-only transactions is 1:1 to target both read-intensive and write-intensive workloads. ALOHA-KV uses alternating read epochs and write epochs with an epoch duration of 20ms. Fault tolerance (see Section 4.4) is disabled by default.

For comparison, we implemented a *baseline* system that represents an upper bound for performance with respect to the chosen implementation language and RPC framework. In the baseline, FEs and BEs process transactions without any concurrency control or atomic commitment protocol, and there is no epoch switching. We also execute RAMP on the same infrastructure with the same workload settings. Experimental measurements focus on average latency and aggregate throughput of operations (transaction throughput times transaction size), because we are more interested to see the performance trade-offs of batching operations into transactions. However, we present both the throughput of operations and of transactions, both of which are used in the RAMP paper [4]. Each data point represents the average of three runs, and is plotted with error bars indicating the min and max measurements. In many cases *the error bars are imperceptibly small.*

6.2 ALOHA-KV vs. RAMP Results

First, we compare ALOHA-DB with the recently published transaction protocol RAMP, which supports weaker-than-serializable distributed ro/wo transactions [4]. We do not include 2PC/2PL in our experiments because RAMP exhibits significant performance gains over these techniques. Published RAMP experiments consider only small transaction size (default 4, maximum 128), whereas we

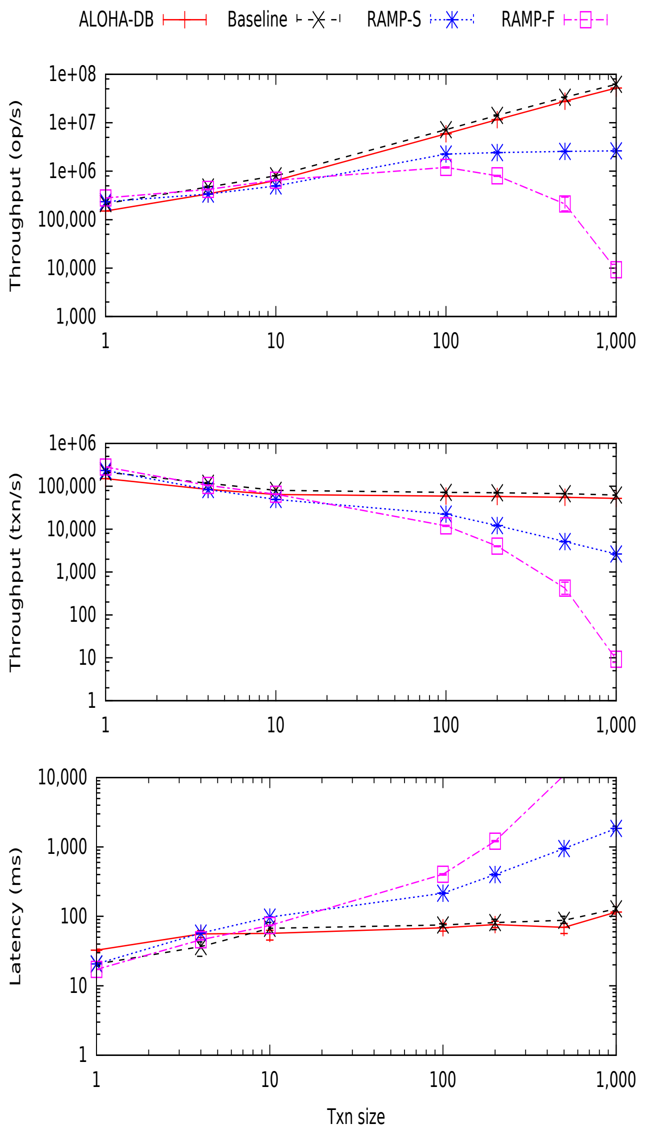

Figure 3: ALOHA-KV vs. RAMP under 20ms read/write epoch duration. Throughput and latency presented using logarithmic scales.

also are interested in larger transactions to determine the benefits of batch processing. For a fair comparison, our experiment uses 5000 synchronous RPC clients in total (same as [4]). Of the three protocol variations in [4], we tested RAMP-Fast and RAMP-Small only since the performance of RAMP-Hybrid is known to be a compromise between the other two.

Figure 3 shows that ALOHA-KV outperforms RAMP in terms of both throughput and latency for large transactions (around >6 for RAMP-S, >10 for RAMP-F). The performance gaps grow as transaction size increases. For transaction size 1000, our system achieves throughput roughly 20× greater than RAMP-S, and over three orders of magnitude greater than RAMP-F. ALOHA-KV scales almost linearly with transaction size in terms of throughput, which demonstrates the benefits of batching operations. We also see that ALOHA-KV performance follows closely the baseline implementation despite providing atomic transaction commitment and serializable isolation.

As reported in the RAMP paper, RAMP-F performs worse than RAMP-S for larger transactions, with the performance gap widening as transaction size increases. This is because RAMP-F metadata grows linearly with transaction size for each key. In the worst case, a read-only transaction of size n may retrieve metadata referring to n^2 keys if each key in the transaction returns metadata containing n other keys. In comparison, the latency of RAMP-S under various transaction sizes is much flatter because it uses constant size metadata. However, RAMP-S requires two round trips for all read transactions, which explains the lower throughput relative to ALOHA-KV. Furthermore, despite the more compact metadata, RAMP-S uses protocol messages whose size is linear in the transaction size.

For smaller transaction sizes, RAMP outperforms ALOHA-KV slightly, because the metadata overhead in RAMP is small and and because ALOHA-KV incurs an overhead for epoch switching. Furthermore, the RPC framework differences also account for performance differences in small transaction size. For example at transaction size 1, the throughput of RAMP-F and RAMP-S are 31% and 9.2% higher, respectively, than our baseline (no concurrency control, no atomic commitment). However, the throughput of ALOHA-KV and RAMP-S for large transactions can indicate the performance of batching of small-size transactions, because metadata size is irrelevant to transaction size in these algorithms.

In summary, even though the RAMP protocol provides both synchronization and partition independence [4], the potential benefits of larger transactions are counteracted by the overheads of large metadata in RAMP-F and large messages in RAMP-S. In contrast, ALOHA-KV records minimal metadata and distributes transaction execution across BEs using messages that grow linearly with transaction size.

6.3 Microbenchmark

To better understand the performance of ALOHA-KV under various scenarios, we present the results of microbenchmark experiments. Unless otherwise specified, we use the same deployment as in Sec. 6.1. Clients use the fbthrift async-client API, which allows issuing multiple requests without blocking on the response, and enables potentially higher throughput than synchronous RPCs. The experiments use transaction size 1000, 50% read-only transactions and 100ms read and write epoch duration as default settings.

6.3.1 Epoch duration. Figure 4 shows results under various epoch durations. We use 80 async clients, which ensures that the ALOHA servers are not overloaded. The results confirm the intuition that longer epochs yield both higher throughput by reducing the proportion of time spent on epoch switching, and higher latency by making transactions wait longer for authorization. Latency grows nearly linearly with epoch duration, whereas throughput is fairly flat beyond about 50ms.

The experiment also shows that epoch switching has relatively little impact on performance. In particular, there is only around

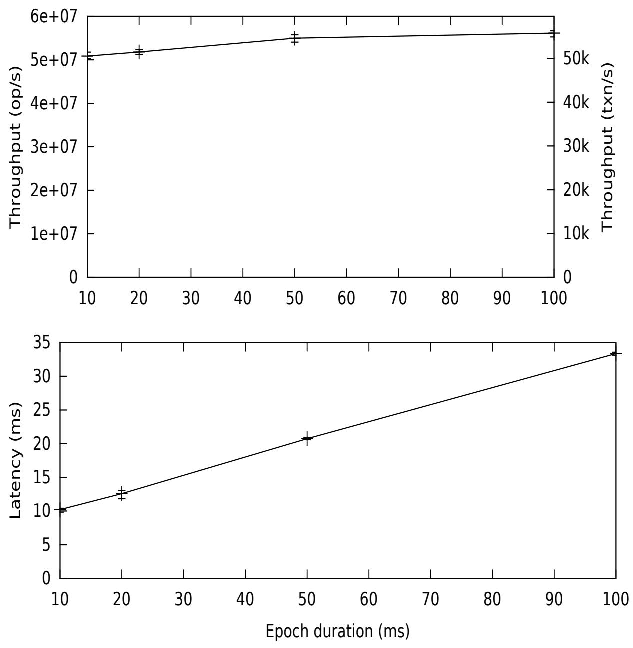

Figure 4: Throughput and avg. latency under various epoch durations.

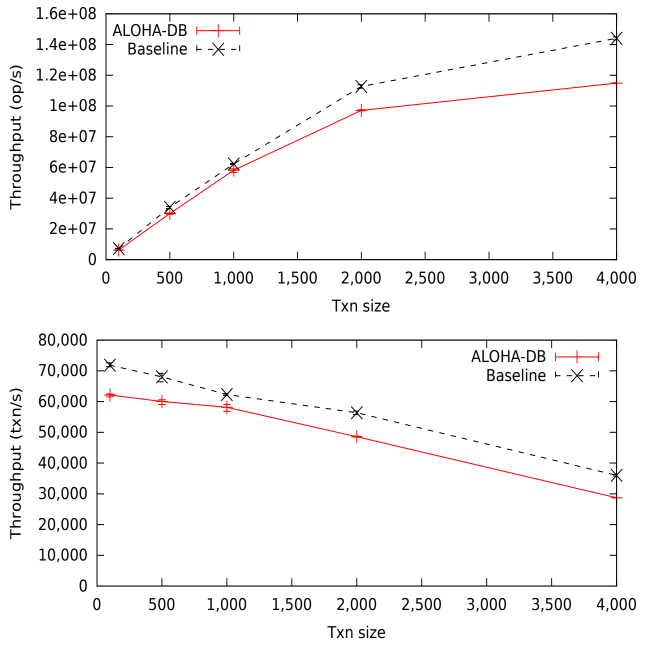

Figure 5: Throughput and latency under various transaction sizes.

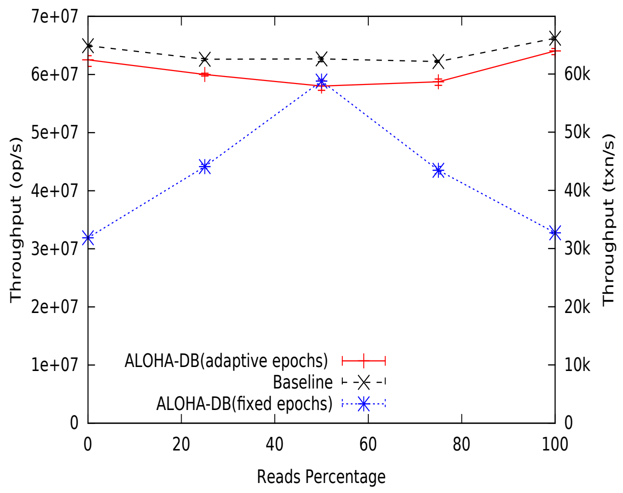

Figure 6: Throughput under various read/write proportions.

10% throughput difference between epoch durations of 10ms and 100ms. Thus, most of the throughput benefits of ECC are realized with fairly short epochs, and little penalty in terms of latency. We notice under large epoch duration (≥ 50), average latencies are less than half of epoch duration. This is because transactions arriving during the correct epoch enjoy a low latency, and transactions that arrive out-of-epoch must wait half of an epoch duration on average before they can begin executing. The average of the two cases is less than half of the epoch duration when epoch duration is large.

6.3.2 Transaction size. Figure 5 illustrates the effect of transaction size on throughput, which increases rapidly up to roughly 1000 operations per transaction. This is because batching boosts network and processing efficiency. Beyond 2000 operations per transaction, the system exhibits diminishing returns, and plateaus around 100 Mops/s. Since the transaction coordinator chops transactions into fragments sent to different partitions, we expect that a larger transaction size is needed to saturate the throughput as the number of servers increases.

6.3.3 Proportion of read-only transactions. In experiments pertaining to the proportion of read-only vs. write-only transactions, we considered two cases: when the proportion is known, and when the proportion is unknown and the default epoch durations are used. These cases are denoted as *adaptive epochs* and *fixed epochs* in Figure 6. We also present the performance of the baseline, where transactions execute without concurrency control and no atomic commitment protocol is used.

The results show that even without any prior knowledge of the read proportion, throughput using fixed epochs is roughly half or more of the level observed using adaptive epochs. The results for adaptive epochs are representative of cases when the workload exhibits a steady read/write mixture that is either known a priori, can be predicted accurately, or can be measured on-the-fly. We also observe low overhead for serializable transactions in ALOHA-KV under various read ratios as compared with the baseline.

6.4 Scalability

We evaluated the scalability of ALOHA-KV with respect to different numbers of servers from two angles: (1) the scale-out throughput

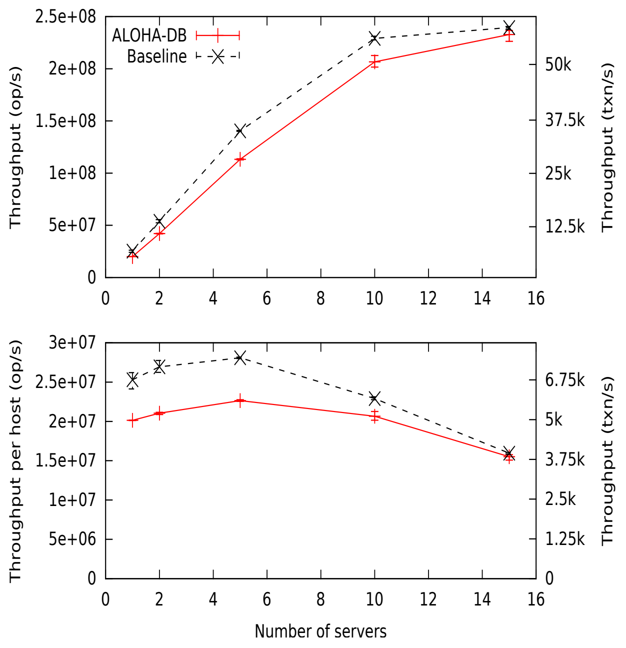

Figure 7: Aggregate and per host throughput using txn size 4000

performance, (2) the overhead of epoch switching. In the first case, we run ALOHA-KV up to a throughput of 233 Mops/s, achieving close to linear throughput scalability. In the second case, we show that a single EM can control hundreds of FEs with only single-digit-millisecond overhead per epoch switch.

6.4.1 Scale-out. In this experiment, we vary the number of servers hosting BE/FE pairs. The transaction size is set to 4000 to allow batching a significant number of operations in messages sent to each BE. Figure 7 shows that using up to 15 BE/FE servers, ALOHA-KV achieves around 233 Mops/s. Since the probability of conflicts among write-only transactions grows with the observed throughput, the results demonstrate that ECC sustains a high degree of parallelism despite contention. Throughput per server drops slightly as the number of servers increases due to less effective batching, leading to sub-linear scalability. In the extreme case of 1 or 2 servers, the per-server performance is slightly lower than that of 5 servers, due to the single host network performance limit. However, when more servers are added to the system, additional network resources are utilized by the servers.

6.4.2 Epoch switching overhead. As discussed in Section 5.2, epoch management can be implemented either using a single EM, or by organizing the FEs in a scalable hierarchical structure. In this experiment we investigate whether a single EM has enough processing capacity to control hundreds of FEs centrally. To isolate the epoch switch time we set the epoch duration to zero, meaning that each FE responds immediately to an authorization grant by acknowledging authorization revocation. Upon receiving responses from all FEs, the EM issues the next round of epoch switch requests. Each FE instance is run on a distinct physical core, and there is no client workload.

Table 1 shows the average epoch switch time for up to 640 FEs. The epoch switch time grows nearly linearly with the number of FEs, reaching 3.4ms at 640 FEs. As the number of FEs increases, the EM needs more time to process epoch switch messages, whose number grows linearly with the number of FEs. When the FEs are under load from clients, it is expected that epoch switch times will be longer than in this experiment because each FE must finish any in-flight transactions before responding to the EM at the end of an epoch. However, in that case the EM will be more lightly loaded because it has more time to process the same number of messages. Based on the experimental data, we conclude that a centralized EM is sufficient for controlling a cluster of hundreds of ALOHA-KV servers.

6.5 Fault Tolerance

Table 2 shows the performance of various fault tolerance strategies. The system has five partitions, and each partition has a primary server and a backup server in the experiments. In the table, "+log" denotes the strategy of BEs writing operation logs to disk; "+FE" denotes using backup FEs to take over the coordination when the primary crashes; and "+BE" denotes using backup BEs to store a copy of the data held by the primary.

The results show that "+log", "+FE", and "+BE" lead to throughput penalties of 2–5%, 16–21%, and 20–25%, respectively. The "+log+FE" strategy has low overhead and protects against data loss even if an FE or BE crashes (see Section 4.4). The most expensive strategy is "+log+FE+BE", which achieves 35.5 Mops/s—roughly 40% slower than no fault tolerance. In comparison, RAMP performance in some cases is substantially lower without any replication.

7 RELATED WORK

ALOHA-DB's early stage architecture was broadly described in an earlier workshop paper [14]. This paper elaborates on the architecture, improves the protocol, and presents a comprehensive performance evaluation.

Prior distributed storage systems mostly either pay a performance penalty for serializable distributed transaction or sacrifice serializability in favor of weaker isolation models. Sinfonia [2] provides serializable minitransactions, which internally use lock-based concurrency control and a two-phase transaction commitment protocol. Spanner [10], which accommodates a broader class of transactions, uses a similar mechanism combined with optimistic concurrency control, as well as data versioning to avoid locks for read-only transactions. In contrast, VoltDB [28] executes transactions serially in each partition, which avoids the need for concurrency control and enables high throughput for single-partition transactions but forces centralized coordination for multi-partition transactions. MDCC [18] and RAMP [4] use commitment protocols in which the number of network round trips depends on the contention encountered. MDCC provides read committed isolation by default and RAMP provides read atomicity, which is also weaker than serializability. None of the above systems allow distributed write-only transactions to be committed in amortized one round trip in the presence of write-write conflicts.

number of FE instances	80	160	240	320	400	480	560	640
epoch switch time (ms)	1.9	2.1	2.3	2.5	2.7	3.0	3.2	3.4

Table 1: Epoch switch time for various numbers of FE instances.

	No fault tolerance	+log	+FE	+log+FE	+BE	+log+BE	+FE+BE	+log+FE+BE
Throughput (M ops/s)	58.3	56.4	49.0	46.7	46.6	44.4	36.8	35.5
Throughput (K txn/s)	58.3	56.4	49.0	46.7	46.6	44.4	36.8	35.5

Table 2: Evaluation of various fault tolerance strategies.

Other systems address distributed transactions using deterministic scheduling. Deterministic databases such as Calvin [23, 30, 31] process transactions in deterministic order, which makes it possible to process write-only transactions in one round trip. However, transactions cannot abort using a second round even if some partition is overloaded or failed. Our evaluation does not include the open-source Calvin implementation because it relies on two optimizations that make an apples-to-apples comparison difficult: (1) it merges the benchmark client and server into one binary, and (2) it batches multiple transactions in one request message. However, we predict that Calvin's performance may suffer from the single-threaded lock manager for write-only transactions, whereas ALOHA-KV and RAMP both use multiversioning and nearly avoid locking for writing a new version.

Previous works use epochs to structure transaction processing in various ways [17, 20, 32]. Silo [32] and its more scalable relative FOEDUS [17] use epochs to ensure serializability on recovery from failures, to facilitate garbage collection, and to provide read-only snapshots. Phase reconciliation [20] repeatedly cycles *split phases*, epochs that only process commutative operations, and *joined phases*, epochs that process other operations. Phase reconciliation resembles ECC on first impression, but isolation of reads from writes in our system is orthogonal to isolation of commutative and non-commutative operations in their system. For example, two writes on the same key do not commute and thus cannot be processed concurrently in phase reconciliation, but can be processed in parallel in ALOHA-KV. Additionally, these other systems target single machine transactions, whereas ALOHA-KV targets distributed transactions.

Efficient read-only/write-only transaction protocols are proposed in prior works, such as Eiger [19], Walter[27], and G-store [11], and can be used in batch processing systems [9, 16]. For these transaction types, ALOHA-KV also benefits from batching, and achieves throughput-optimized performance even under high contention. However, none of these previous works provide serializable isolation as ALOHA-KV.

8 FUTURE WORK

ECC support lightweight read-only and write-only transactions for distributed key-value stores. Recent research [6, 7, 15] proposed building distributed read-write transactions using blind multi-key writing by atomically writing a log entry in Corfu [5]. We believe ECC can be extended to support ACID read-write transactions for scale-out database systems, and plan to demonstrate this point.

ECC introduces an interesting tuning knob for adaptivity with respect to workload variations, such as changes between read-heavy and write-heavy workloads, or large versus small transactions. Our various read/write ratio experiments show that fixed epochs achieve at least half of the throughput possible using adaptive epochs. However, in future work we plan to develop an intelligent self-tuning mechanism that will adjust the epoch durations dynamically in response to the workload.

In addition, we will consider techniques for reducing epoch switch time, which hurts throughput. Aside from software techniques such as admission control, we will leverage more advanced network hardware with support for remote direct memory access (RDMA), enabling fast epoch switching.

Inspired by recent developments in transaction processing on a single many-core machine [32, 34], we would also like to explore whether the benefits of ECC demonstrated in a distributed message passing system can be realized in shared memory as well.

9 CONCLUSION

This paper addresses the problem of supporting high-throughput multi-partition read-only and write-only transactions, a.k.a *multi-put* and *multi-get*. We propose an ECC mechanism for these transactions, which guarantees serializability. ECC avoids read-write conflicts among transactions by partitioning transaction execution into disjoint read and write epochs, and mitigates write-write conflicts by storing multiple versions of key-value pairs. Thus, concurrent writes can be processed in parallel with low-overhead, even when their write sets overlap. Using ECC as the central building block, we describe a distributed protocol for serializable read-only and write-only transactions, which requires amortized one round trip to commit a transaction in the absence of failures irrespective of contention. We implement the protocol in a key-value storage system called ALOHA-KV, and show experimentally that it can process around 233 million operations per second on 15 servers when transactions contain thousands of operations. Compared to RAMP, our system achieves much higher throughput for large transactions, despite guaranteeing stronger transaction isolation.

ACKNOWLEDGMENTS

The authors would also like to thank the anonymous referees and shepherd Nathan Bronson for their valuable comments and helpful suggestions. The authors are supported in part by Hewlett-Packard Labs and the Natural Sciences and Engineering Research Council (NSERC) of Canada.

REFERENCES

[1] Marcos K. Aguilera, Joshua B. Leners, and Michael Walfish. 2015. Yesquel: Scalable SQL Storage for Web Applications. In *Proceedings of SOSP*. 245–262. https://doi.org/10.1145/2815400.2815413

[2] Marcos K. Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. 2009. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. *ACM Trans. Comput. Syst.* 27, 3, Article 5 (Nov. 2009), 48 pages. https://doi.org/10.1145/1629087.1629088

[3] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 185–196. https://doi.org/10.14778/2735508.2735509

[4] Peter Bailis, Alan Fekete, Joseph M. Hellerstein, Ali Ghodsi, and Ion Stoica. 2014. Scalable Atomic Visibility with RAMP Transactions. In *Proc. of SIGMOD*. 27–38. https://doi.org/10.1145/2588555.2588562

[5] Mahesh Balakrishnan, Dahlia Malkhi, John D. Davis, Vijayan Prabhakaran, Michael Wei, and Ted Wobber. 2013. CORFU: A Distributed Shared Log. *ACM Trans. Comput. Syst.* 31, 4, Article 10 (Dec. 2013), 24 pages. https://doi.org/10.1145/2535930

[6] Phil Bernstein, Colin Reid, and Sudipto Das. 2011. Hyder - A Transactional Record Manager for Shared Flash. In *Proc. of CIDR*. 9–20.

[7] Philip A. Bernstein, Sudipto Das, Bailu Ding, and Markus Pilman. 2015. Optimizing Optimistic Concurrency Control for Tree-Structured, Log-Structured Databases. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1295–1309. https://doi.org/10.1145/2723372.2737788

[8] Philip A. Bernstein, Vassos Hadzilacos, and Nathan Goodman. 1986. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[9] James Cipar, Greg Ganger, Kimberly Keeton, Charles B. Morrey, III, Craig A.N. Soules, and Alistair Veitch. 2012. LazyBase: Trading Freshness for Performance in a Scalable Database. In *Proc. of EuroSys*. 169–182. https://doi.org/10.1145/2168836.2168854

[10] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google&Rsquo;s Globally Distributed Database. *ACM Trans. Comput. Syst.* 31, 3, Article 8, 22 pages. https://doi.org/10.1145/2491245

[11] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2010. G-Store: A Scalable Data Store for Transactional Multi Key Access in the Cloud. In *Proceedings of the 1st ACM Symposium on Cloud Computing (SoCC '10)*. ACM, New York, NY, USA, 163–174. https://doi.org/10.1145/1807128.1807157

[12] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. 2007. Dynamo: Amazon's Highly Available Key-value Store. *SIGOPS Oper. Syst. Rev.* 41, 6 (Oct. 2007), 205–220. https://doi.org/10.1145/1323293.1294281

[13] Facebook. 2015. fbthrift. https://github.com/facebook/fbthrift. (2015).

[14] Hua Fan. 2015. High Performance multi-partition transaction. In *Proc. of VLDB PhD Workshop*.

[15] A. K. Goel, J. Pound, N. Auch, P. Bumbulis, S. MacLean, F. Färber, F. Gropengiesser, C. Mathis, T. Bodner, and W. Lehner. 2015. Towards Scalable Real-time Analytics: An Architecture for Scale-out of OLxP Workloads. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1716–1727. https://doi.org/10.14778/2824032.2824069

[16] Charles Johnson, Kimberly Keeton, Charles B. Morrey, Craig A. N. Soules, Alistair Veitch, Stephen Bacon, Oskar Batuner, Marcelo Condotta, Hamilton Coutinho, Patrick J. Doyle, Rafael Eichelberger, Hugo Kiehl, Guilherme Magalhaes, James McEvoy, Padmanabhan Nagarajan, Patrick Osborne, Joaquim Souza, Andy Sparkes, Mike Spitzer, Sebastien Tandel, Lincoln Thomas, and Sebastian Zangaro. 2014. From Research to Practice: Experiences Engineering a Production Metadata Database for a Scale Out File System. In *Proc. of FAST*. 191–198. https://www.usenix.org/conference/fast14/technical-sessions/presentation/johnson

[17] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 691–706. https://doi.org/10.1145/2723372.2746480

[18] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-data Center Consistency. In *Proc. of EuroSys*. 113–126. https://doi.org/10.1145/2465351.2465363

[19] Wyatt Lloyd, Michael J. Freedman, Michael Kaminsky, and David G. Andersen. 2013. Stronger Semantics for Low-latency Geo-replicated Storage. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation (nsdi'13)*. USENIX Association, Berkeley, CA, USA, 313–328. http://dl.acm.org/citation.cfm?id=2482626.2482657

[20] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. 2014. Phase Reconciliation for Contended In-memory Transactions. In *Proc. of OSDI*. USENIX Association, Berkeley, CA, USA, 511–524. http://dl.acm.org/citation.cfm?id=2685048.2685088

[21] Christos H. Papadimitriou. 1979. The Serializability of Concurrent Database Updates. *J. ACM* 26, 4 (Oct. 1979), 631–653. https://doi.org/10.1145/322154.322158

[22] Francisco Perez-Sorrosal, Marta Patiño Martinez, Ricardo Jimenez-Peris, and Bettina Kemme. 2011. Elastic SI-Cache: Consistent and Scalable Caching in Multi-tier Architectures. *The VLDB Journal* 20, 6 (Dec. 2011), 841–865. https://doi.org/10.1007/s00778-011-0228-8

[23] Kun Ren, Alexander Thomson, and Daniel J. Abadi. 2014. An Evaluation of the Advantages and Disadvantages of Deterministic Database Systems. *Proc. VLDB Endow.* 7, 10 (June 2014), 821–832. https://doi.org/10.14778/2732951.2732955

[24] Lawrence G. Roberts. 1975. ALOHA Packet System with and Without Slots and Capture. *SIGCOMM Comput. Commun. Rev.* 5, 2 (April 1975), 28–42. https://doi.org/10.1145/1024916.1024920

[25] Sudip Roy, Lucja Kot, Gabriel Bender, Bailu Ding, Hossein Hojjat, Christoph Koch, Nate Foster, and Johannes Gehrke. 2015. The Homeostasis Protocol: Avoiding Transaction Coordination Through Program Analysis. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 1311–1326. https://doi.org/10.1145/2723372.2723720

[26] Artyom Sharov, Alexander Shraer, Arif Merchant, and Murray Stokely. 2015. Take Me to Your Leader!: Online Optimization of Distributed Storage Configurations. *Proc. VLDB Endow.* 8, 12 (Aug. 2015), 1490–1501. https://doi.org/10.14778/2824032.2824047

[27] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional Storage for Geo-replicated Systems. In *Proc. of SOSP (SOSP '11)*. ACM, New York, NY, USA, 385–400. https://doi.org/10.1145/2043556.2043592

[28] Michael Stonebraker and Ariel Weisberg. 2013. The VoltDB Main Memory DBMS. *IEEE Data Eng. Bull.* 36, 2 (2013), 21–27. http://sites.computer.org/debull/A13june/VoltDB1.pdf

[29] Rebecca Taft, Essam Mansour, Marco Serafini, Jennie Duggan, Aaron J. Elmore, Ashraf Aboulnaga, Andrew Pavlo, and Michael Stonebraker. 2014. E-store: Fine-grained Elastic Partitioning for Distributed Transaction Processing Systems. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 245–256. https://doi.org/10.14778/2735508.2735514

[30] Alexander Thomson and Daniel J. Abadi. 2010. The Case for Determinism in Database Systems. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 70–80. https://doi.org/10.14778/1920841.1920855

[31] Alexander Thomson, Thaddeus Diamond, Shu-Chun Weng, Kun Ren, Philip Shao, and Daniel J. Abadi. 2012. Calvin: Fast Distributed Transactions for Partitioned Database Systems. In *Proc. of SIGMOD*. 1–12. https://doi.org/10.1145/2213836.2213838

[32] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proc. of SOSP*. 18–32. https://doi.org/10.1145/2517349.2522713

[33] Venkateshwaran Venkataramani, Zach Amsden, Nathan Bronson, George Cabrera III, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Jeremy Hoon, Sachin Kulkarni, Nathan Lawrence, Mark Marchukov, Dmitri Petrov, and Lovro Puzar. 2012. TAO: How Facebook Serves the Social Graph. In *Proc. of SIGMOD*. 791–792. https://doi.org/10.1145/2213836.2213957

[34] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220. https://doi.org/10.14778/2735508.2735511