Epoch-based Commit and Replication in Distributed OLTP Databases

Yi Lu
Massachusetts Institute of Technology
yilu@csail.mit.edu

Xiangyao Yu
University of Wisconsin-Madison
yxy@cs.wisc.edu

Lei Cao
Massachusetts Institute of Technology
lcao@csail.mit.edu

Samuel Madden
Massachusetts Institute of Technology
madden@csail.mit.edu

ABSTRACT

Many modern data-oriented applications are built on top of distributed OLTP databases for both scalability and high availability. Such distributed databases enforce atomicity, durability, and consistency through *two-phase commit* (2PC) and *synchronous replication* at the granularity of every single transaction. In this paper, we present COCO, a new distributed OLTP database that supports epoch-based commit and replication. The key idea behind COCO is that it separates transactions into epochs and treats a whole epoch of transactions as the commit unit. In this way, the overhead of 2PC and synchronous replication is significantly reduced. We support two variants of optimistic concurrency control (OCC) using physical time and logical time with various optimizations, which are enabled by the epoch-based execution. Our evaluation on two popular benchmarks (YCSB and TPC-C) show that COCO outperforms systems with fine-grained 2PC and synchronous replication by up to a factor of four.

PVLDB Reference Format:
Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. Epoch-based Commit and Replication in Distributed OLTP Databases. PVLDB, 14(5): 743 - 756, 2021.
doi:10.14778/3446095.3446098

1 INTRODUCTION

Many modern distributed OLTP databases use a shared-nothing architecture for scale out [9, 55, 64], since the capacity of a single-node database fails to meet their demands. In distributed databases, scalability is achieved through data partitioning [12, 57], where each node contains one or more partitions of the whole database. Partitioning-based systems can easily support *single-partition transactions* that run on a single node and require no coordination between nodes. If a workload consists of only such transactions, it trivially parallelizes across multiple nodes. However, distributed databases become more complex when dealing with *distributed transactions* that touch several data partitions across multiple nodes. In particular, many implementations require the use of two-phase commit (2PC) [40] to enforce atomicity and durability, making the effects of committed transactions recorded to persistent storage and survive server failures.

It is well known that 2PC causes significant performance degradation in distributed databases [3, 12, 46, 61], because a transaction is not allowed to release locks until the second phase of the protocol, blocking other transactions and reducing the level of concurrency [21]. In addition, 2PC requires *two network round-trip delays* and *two sequential durable writes* for every distributed transaction, making it a major bottleneck in many distributed transaction processing systems [21]. Although there have been some efforts to eliminate distributed transactions or 2PC, unfortunately, existing solutions either introduce impractical assumptions (e.g., the read/write set of each transaction has to be known a priori in deterministic databases [19, 61, 62]) or significant runtime overhead (e.g., dynamic data partitioning [12, 31]).

In addition, a desirable property of any distributed database is high availability, i.e., when a server fails, the system can mask the failure from end users by replacing the failed server with a standby machine. High availability is typically implemented using data replication, where all writes are handled at the *primary replica* and are shipped to the *backup replicas*. Conventional high availability protocols must make a tradeoff between performance and consistency. On one hand, *asynchronous replication* allows a transaction to commit once its writes arrive at the primary replicas; propagation to backup replicas happens in the background asynchronously [14]. Transactions can achieve high performance but a failure may cause data loss, i.e., consistency is sacrificed. On the other hand, *synchronous replication* allows a transaction to commit only after its writes arrive at all replicas [9]. No data loss occurs but each transaction holds locks for a longer duration of time and has longer latency — even single-partition transactions need to wait for at least one round-trip of network communication.

In this paper, we make the key observation that the inefficiency of both 2PC and synchronous replication mainly comes from the fact that existing protocols enforce consistency at the granularity of *individual transactions*. By grouping transactions that arrive within a short time window into short periods of time — which we call epochs — it's possible to manage both atomic commit and consistent data replication at the granularity of epochs. In our approach, an epoch is the basic unit at which transactions commit and recover — either all or none of the transactions in an epoch commit — which adheres to the general principle of group commit [15] in single-node databases. However, epoch-based commit and replication focus on reducing the overhead due to 2PC and synchronous

Proceedings of the VLDB Endowment, Vol. 14, No. 5 ISSN 2150-8097.
doi:10.14778/3446095.3446098

replication rather than disk latency as in group commit. As a result, a transaction releases its locks immediately after execution finishes, and logging to persistent storage occurs in the background and is only enforced at the boundary of epochs. Similarly, a transaction no longer needs to hold locks after updating the primary replica, since write propagation happens in the background as in asynchronous replication. Note that the writes of a transaction are visible to other transactions as soon as it commits but they may disappear if a failure occurs and a rollback happens. Therefore, a transaction does not release the result to the end user until the current epoch commits, when the writes of all transactions belong to the epoch are durable. In COCO, consistency is enforced at the boundary of epochs as in synchronous replication. The epoch size, which determines the average latency of transactions, can be selected to be sufficiently low for most OLTP workloads (e.g., 10 ms). Prior work [11, 43, 63] has argued that such latencies are acceptable and are not a problem for most transactional workloads. In addition, when the durable write latency exceeds several hundred microseconds, epoch-based commit and replication can help reduce tail latency compared to a traditional architecture with 2PC [16].

In this paper, we describe COCO, a distributed main-memory OLTP database we built that embodies the idea of epoch-based commit and replication. COCO supports two variants of optimistic concurrency control (OCC) [25] that serialize transactions using physical time and logical time, respectively. In addition, it supports both serializability and snapshot isolation with various optimizations, which are enabled by the epoch-based commit and replication. Our evaluation on an eight-server cluster shows that our system outperforms systems that use conventional concurrency control algorithms and replication strategies by up to a factor of four on YCSB and TPC-C. The performance improvement is even more significant in a wide-area network, i.e., 6x on YCSB and an order of magnitude on TPC-C. In addition, we show that epoch-based commit and replication are less sensitive to durable write latency with up to 6x higher throughput than 2PC.

In summary, this paper makes the following major contributions:

- We present COCO, a distributed and replicated main-memory OLTP framework that implements epoch-based commit and replication.
- We introduce the design of two variants of optimistic concurrency control algorithms in COCO, implement critical performance optimizations, and extend them to support snapshot transactions.
- We report a detailed evaluation on two popular benchmarks (YCSB and TPC-C). Overall, our proposed solution outperforms conventional commit and replication algorithms by up to a factor of four.

2 BACKGROUND

This section discusses the background of distributed concurrency control, 2PC and data replication in distributed databases.

2.1 Distributed Concurrency Control

Concurrency control enforces two critical properties of a database: atomicity and isolation. Atomicity requires a transaction to expose either all or none of its changes to the database. The isolation level specifies when a transaction is allowed to see another transaction's writes.

Two classes of concurrency control protocols are commonly used in distributed systems: two-phase locking (2PL) [5, 17] and optimistic concurrency control (OCC) [25]. 2PL protocols are pessimistic and use locks to avoid conflicts. In OCC, a transaction does not acquire locks during execution; after execution, the database validates a transaction to determine whether it commits or aborts. At low contention, OCC has better performance than 2PL due to its non-blocking execution. Traditionally, both 2PL and OCC support serializability. Multi-version concurrency control (MVCC) [51] was proposed to support snapshot isolation in addition to serializability. An MVCC protocol maintains multiple versions of each tuple in the database. This offers higher concurrency since a transaction can potentially pick from amongst several consistent versions to read, at the cost of higher storage overhead and complexity.

2.2 The Necessity and Cost of 2PC

In distributed databases, a transaction that accesses data on multiple *participant nodes* is usually initiated and coordinated by a *coordinator*. The most common way to commit transactions in a distributed database is via two-phase commit (2PC) [40]. Once a transaction finishes execution, the coordinator begins the two-phase commit protocol, which consists of a *prepare phase* and a *commit phase*. In the prepare phase, the coordinator communicates with each participant node, asking if it is ready to commit the transaction. To tolerate failures, each participant node must make the prepare/abort decision durable before replying to the coordinator. After the coordinator has collected all votes from participant nodes, it enters the commit phase. In the commit phase, the coordinator is able to commit the transaction if all participant nodes agree to commit. It first makes the commit/abort decision durable and then asks each participant node to commit the transaction. In the end, each participant node acknowledges the commit to the coordinator.

Although the above mechanism ensures both atomicity and durability of distributed transactions, it also introduces some problems that significantly limit the performance of distributed database systems. We now summarize the problems it introduces and discuss the implications for distributed databases: (1) Two network round trips: On top of network round trips during transaction execution, two-phase commit requires two additional network round trips, making the cost of running a distributed transaction more expensive than the cost of running a single-partition transaction on a single node [46]; (2) Multiple durable writes: A write is considered durable when it has been flushed to disk (e.g., using `fsync` [58]). Depending on different hardware, the latency of a flush is from tens or hundreds of microseconds on SSDs to tens of milliseconds on spinning disks; (3) Increased contention: Multiple network round trips and durable writes also increase the duration that locks are held. As a result, contention increases, which further impairs the throughput and latency.

Some solutions have been proposed to address the inefficiency caused by distributed transactions and 2PC [12, 31, 62], but they all suffer from significant limitations. Schism [12] reduces the number of distributed transactions through a workload-driven partitioning and replication scheme. However, distributed transactions are frequent in real world scenarios and fundamentally unable

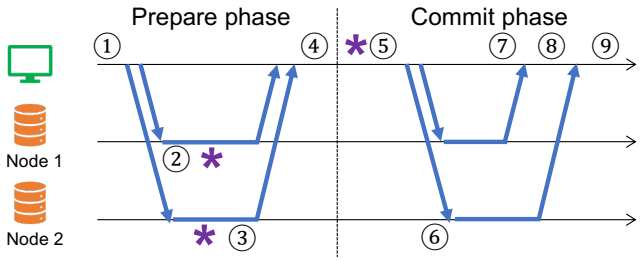


Figure 1: Failure scenarios in epoch-based commit

to be fully partitioned. G-Store [13] and LEAP [31] eliminate distributed transactions by dynamically re-partitioning the database. However, multiple network round trips are still required to move data across a cluster of nodes. Calvin [62] and other deterministic databases [18, 19, 61] avoid 2PC by running transactions deterministically across multiple nodes, but these systems need to know a transaction's read/write set, which is not always feasible. Aria [33] supports deterministic transaction execution without any prior knowledge of the input transactions but it may suffer from undesirable performance in high contention workloads, as we show in our experiments. The major difference of deterministic databases is that they replicate the input of transactions instead of the output. However, to achieve determinism, different systems usually come with different limitations as we introduced above. Note that deterministic databases need to deploy a sequencing layer as well, which requires additional hardware cost.

2.3 Replication in Distributed Databases

Modern database systems support high availability such that when a subset of servers fail, the rest of the servers can carry out the database functionality, thereby end users do not notice the server failures. In most applications, high availability is implemented using data replication, which can be broadly classified into (1) synchronous replication [22], and (2) asynchronous replication [14].

Synchronous replication requires that a transaction does not commit until its writes have been replicated on all replicas. Primary-backup replication and state machine replication are two common ways to implement synchronous replication. In primary-backup replication, the primary node usually holds the write locks until the writes of committed transactions are replicated on backup nodes. In state machine replication, every node can initiate a write to the database, but all of them must agree on the order of data access through a consensus protocol, such as Paxos [26] or Raft [45]. Asynchronous replication allows a transaction to commit as soon as its writes are durable on the primary replica. The writes of committed transactions are usually batched and applied to backup replicas later on. Such design improves the efficiency of replication at the cost of potential data inconsistency and reduced availability when a failure occurs.

Some recent work such as G-PAC [37] and Helios [44] unifies the commit and consensus protocols (e.g., Paxos or Raft) to reduce the latency of transactions in geo-replicated databases. Epoch-based commit and replication does not impose any restrictions on the underlying replication protocol. The users of our system have the flexibility to choose primary-backup replication or state machine replication.

3 EPOCH-BASED COMMIT AND REPLICATION

In this section, we first show how our new commit protocol based on epochs offers superior performance and provides the same guarantees as two-phase commit (2PC). We then discuss how the epoch-based design of COCO reveals opportunities to design a new replication scheme that unifies the best of both synchronous and asynchronous replication schemes. In Section 4 and Section 5, we will discuss how to design distributed concurrency control with epoch-based commit and replication in COCO.

3.1 The Commit Protocol

In COCO, a batch of transactions run and commit in an epoch. However, the result of each transaction is not released until the end of the epoch, when all participant nodes agree to commit all transactions from the current epoch. The system increments the global epoch every few milliseconds (e.g., 10 ms by default) with two phases: a *prepare* phase and a *commit* phase, as in two-phase commit (2PC).

In the prepare phase, the coordinator sends a prepare message to each participant node. Note that the coordinator node is the node coordinating epoch advancement among a cluster of nodes and it's different from the coordinator of distributed transactions as we will see in later sections. The coordinator can be any node in the system or a standalone node outside the system. To prevent the coordinator from being a single point of failure, it can be implemented as a replicated state machine with Paxos [26] or Raft [45]. When a participant node receives a prepare message, it prepares to commit all transactions in the current epoch by force logging a durable *prepared write* record (indicated by a purple star in Figure 1) with all the transaction IDs (TIDs) of ready-to-commit transactions as well as the current epoch number. Note that some transactions may have aborted earlier due to conflicts. The underlying concurrency control algorithms are also required to log all the writes of ready-to-commit transactions durably (See Section 6.2) before the prepared write record. When a participant node durably logs all necessary writes, it then replies an acknowledgement to the coordinator.

In the commit phase, the coordinator first decides if the current epoch can commit. If any participant node fails to reply an acknowledgement due to failures, all transactions from the current epoch will abort. Otherwise, the coordinator writes a durable *commit* record (indicated by a purple star in Figure 1) with the current epoch number and then increments the global epoch. It then sends a commit message to each participant node. Note that if a transaction aborts due to concurrent accesses or integrity violation, it does not stop the current epoch to commit. When a participant node receives a commit message, all the writes of ready-to-commit transactions from the last epoch are considered committed, and the results of these transactions are released to users. In the end, it replies an acknowledgement to the coordinator, and prepares to execute transactions from the next epoch.

Note that COCO is a good fit for workloads in which most transactions are short-lived. In reality, most long-running transactions are read-only analytical transactions, which can be configured to run over a slightly stale database snapshot. COCO is able to support mixed short-lived and long-running update transactions without

sacrificing ACID properties. However, transactions will suffer from higher commit latency. This is because a whole batch of transactions cannot commit until all transactions belonging to the epoch finish execution.

3.2 Fault Tolerance

In this paper, we assume fail-stop failures [14], in which we can assume that any healthy node in the system can detect which node has failed. A *failed node* is one on which the process of a COCO instance has crashed. Since COCO detects failures at the granularity of epochs, all transactions in an epoch will be aborted and re-executed by the system automatically when a failure occurs. Here, we argue that the benefit brought by COCO significantly exceeds the cost to abort and re-run a whole epoch of transactions, since failures are rare on modern hardware. 2PC ensures atomicity and durability at the granularity of every single transaction, but introduces expensive coordination, which is wasted most of the time.

As shown in Figure 1, a failure can occur when an epoch of transactions commit with the epoch-based commit protocol. We classify all failure scenarios into nine categories: (1) before the coordinator sends prepare requests, (2) after some participant nodes receive prepared requests, (3) after all participant nodes receive prepared requests, (4) before the coordinator receives all votes from participant nodes, (5) after the coordinator writes the commit record, (6) before some participant nodes receive commit requests, (7) before the coordinator receives any acknowledgement, (8) after the coordinator receives some acknowledgements, and (9) after the coordinator receives all acknowledgements.

The durable commit record written on the coordinator node indicates the time when the system commits all transactions from the current epoch. Therefore, in cases (1) - (4), the system simply aborts all transactions from the current epoch. Specifically, after recovery, each participant node rollbacks its prepared writes and discards all intermediate results. In cases (5) - (8), transactions from the current epoch are considered committed even though a failure has occurred. After recovery, each participant node can learn the outcome of the current epoch when communicating with the coordinator, and then releases the results to users. Case (9) is the same as case (1), since the system has entered the next epoch.

Once a fault occurs, COCO rollbacks the database to the last successful epoch, i.e., all tuples that are updated in the current epoch are reverted to the states in the last epoch. To achieve this, the database maintains two versions of each tuple. One always has the latest value. The other one has the most recent value up to the last successful epoch.

3.3 Efficient and Consistent Replication

A desirable property of any approach to high availability is strong consistency between replicas, i.e., that there is no way for clients to tell when a failover happened, because the state reflected by the replicas is identical. Enforcing strong consistency in a replicated and distributed database is a challenging task. The most common approach to achieve strong consistency is based on primary-backup replication, where the primary releases locks and commits only after writes have propagated to all replicas, blocking other transactions from accessing modified records and limiting performance.

If a transaction could process reads at replicas and ship writes to replicas asynchronously, it could achieve considerably lower latency and higher throughput, because transactions can read from the nearest replica and release locks before replicas respond to writes. Indeed, these features are central to many recent systems that offer eventual consistency(e.g., Dynamo [14]). Observe that both local reads and asynchronous writes introduce the same problem: the possibility of stale reads at replicas. Thus, they both introduce the same consistency challenge: the database cannot determine whether the records a transaction reads are consistent or not. Therefore, transactions running with two-phase locking (2PL) [5, 17] must always read from the primary replica, since there is no way for them to tell if records are locked by only communicating with backup replicas.

In COCO, atomicity and durability are enforced at the granularity of epochs. Therefore, COCO can replicate writes of committed transactions asynchronously without having to worry about the replication lag within an epoch. The system only needs to ensure that the primary replica is consistent with all backup replicas at epoch boundaries. In addition, COCO uses optimistic concurrency control and each record in the database has an associated TID. The TID of a record usually indicates the last transaction that modified the record and can be used to detect if a read from backup replicas is stale [34, 63, 69, 72]. As a result, a transaction in COCO can read from nearest backup replicas and only validates with the primary replica in the commit phase, which significantly reduces network traffic and latency.

3.4 Limitations

Our experiments show that COCO offers superior performance to distributed transactional databases by running transactions in epochs. We now discuss the limitations of epoch-based commit and replication and how they impact existing OLTP applications.

First, epoch-based commit and replication add a few milliseconds more latency to every single transaction, making it a poor fit for workloads that require extremely low latency. Second, the system could have undesirable performance due to imbalanced running time among transactions. For example, a single long-running transaction can stop a whole batch of transactions from committing until it finishes. Third, since failures are detected at the granularity of epochs, all transactions in an epoch will be aborted and re-executed by the system automatically when a failure occurs. Note that transaction aborts due to conflicts *do not* force all transactions in the epoch to abort and conflicting transactions will be re-run by the system automatically.

4 THE LIFECYCLE OF A TRANSACTION

In this section, we discuss the lifecycle of a distributed transaction in COCO, which contains an execution phase and a commit phase. Note that worker threads run transactions in the order they are submitted. After a worker thread finishes the execution phase of a transaction, it immediately starts the commit phase of the transaction and runs it to completion.

```
1  Function: transaction_read(T, key)
2    ns = get_replica_nodes(key)
3    if node_id() in ns: # a local copy is available
4      record = db_read(key)
5    else: # n is the nearest node chosen from ns
6      record = call_n(db_read, key)
7    T.RS.push_back(record)
8
9  Function: db_read(key)
10   # atomically load value and TID
11   return db[tuple.key].{value, tid}
```

Figure 2: Pseudocode to read from the database

4.1 The Execution Phase

A transaction in COCO runs in two phases: an execution phase and a commit phase. We say the node initiating a transaction is the *coordinator node*, and other nodes are *participant nodes*.

In the execution phase, a transaction reads records from the database and maintains local copies of them in its *read set* (RS). Each entry in the read set contains the value as well as the record's associated transaction ID (TID). For a read request, the coordinator node first checks if the request's primary key is already in the read set. This happens when a transaction reads a data record multiple times. In this case, the coordinator node simply uses the value of the first read. Otherwise, the coordinator node reads the record from the database.

A record can be read from any replica in COCO. To avoid network communication, the coordinator node always reads from its local database if a local copy is available. As shown in Figure 2, the coordinator first locates the nodes *ns* on which there exists a copy of the record. If the coordinator node happens to be from *ns*, a transaction can simply read the record from its local database. If no local copy is available, a read request is sent to the nearest node *n* chosen from *ns*. In COCO, TIDs are associated with records at both primary and backup replicas. For a read request, the system returns both the value and the TID of a record; and both are stored in the transaction's local read set.

All computation is performed in the execution phase. Since COCO's algorithm is optimistic, writes are not applied to the database but are stored in a per-transaction *write set* (WS), in which, as with the read set, each entry has a value and the record's associated TID. For a write operation, if the primary key is not in the write set, a new entry is created with the value and then inserted into the write set. Otherwise, the system simply updates the write set with the new value. Note that for updates to records that are already in the read set, the transaction also copies the TIDs to the entry in the write set, which are used for validation later on.

4.2 The Commit Phase

After a transaction finishes its execution phase, it must be successfully validated before it commits. We now describe the three steps to commit a transaction: (1) lock all records in the transaction's write set; (2) validate all records in the transaction's read set and generate a TID; (3) commit changes to the database. Since a later step cannot start before an earlier step finishes, each step must run in a different network round trip. However, the third step can run asynchronously and a worker thread does not need to wait for the completion of writes and replication. In Section 5, we will discuss the details of two concurrency control algorithms.

```
1  Function: transaction_write(T)
2    for record in T.WS:
3      n = get_primary_node(record.key)
4      call_n(db_write, record.key, record.value, T.tid)
5      for i in get_replica_nodes(record.key) \ {n}:
6        call_i(db_replicate, record.key, record.value, T.tid)
7
8  Function: db_write(key, value, tid)
9    db[key] = {value, tid}
10   unlock(db[key])
11
12 Function: db_replicate(key, value, tid)
13   # begin atomic section
14   if db[key].tid < tid: # Thomas write rule
15     db[key] = {value, tid}
16   # end atomic section
```

Figure 3: Pseudocode to write to the database

4.2.1 Locking the write set. A transaction first tries to acquire locks on each record in the write set to prevent concurrent updates from other transactions. In COCO, a locking request is only sent to the primary replica of each record. To avoid deadlocks, we adopt a `NO_WAIT` deadlock prevention policy, which was shown as the most scalable protocol [21]. In `NO_WAIT`, if the lock is already held on the record, the transaction does not wait but simply aborts. For each acquired lock, if any record's latest TID does not equal to the stored TID, the transaction aborts as well. This is because the record has been changed at the primary replica since the transaction last read it.

4.2.2 Validating the read set & TID assignment. When a transaction has locked each record in its write set, it begins to validate each record in its read set. Unlike the execution phase, in which a read request is sent to the nearest node with a copy of the record, a read validation request is always sent to the primary replica of each record. A transaction may fail to validate a record due to concurrent transactions that access the same record. For example, a record cannot be validated if it is being locked by another transaction or the value has changed since its last read.

COCO assigns a TID to each transaction as well in this step. The assignment can happen either prior to or after read validation [34, 63, 69–72], depending on whether the TID is used during read validation. There are some conditions to assign a TID. For example, it must be able to tell the order of conflicting transactions. We now assume a TID is correctly assigned and leave the details of TID assignment to Section 5.

4.2.3 Writing back to the database. If a transaction fails the validation, it simply aborts, unlocks the acquired locks, and discards its local write set. Otherwise, it will commit changes in its write set to the database. COCO applies the writes and replication asynchronously to reduce round-trip communication. Other transactions can observe the writes of committed transactions at each replica as soon as a write is written to the database. Note that, to ensure the writes of committed transactions durable across failures, the result of a committed transaction is not released to clients until the end of the current epoch.

As illustrated in Figure 3, the value of each record in a transaction's write set and the generated TID are sent to the primary and backup replicas from the coordinator node. There are two scenarios that a write is applied to the database: (1) the write is at the primary replica: Since the primary replica is holding the lock, upon receiving the write request, the primary replica simply updates the value and the TID, and then unlocks the record; (2) the write is at a backup replica: Since asynchronous replication is employed in COCO, upon receiving the write request, the lock on the record is not necessarily held on the primary replica, meaning replication requests to the same record from multiple transactions could arrive out of order. COCO determines whether a write at a backup replica should be applied using the Thomas write rule [60]: the database only applies a write at a backup replica if the record's current TID is less than the TID associated with the write (line 14 – 15 of Figure 3). Because the TID of a record monotonically increases at the primary replica, this guarantees that backup replicas apply the writes in the same order as the order to commit transactions at the primary replica.

5 THE TWO VARIANTS OF OCC

In this section, we discuss how to adapt two popular singe-node concurrency control algorithms (i.e., Silo [63] and Tictoc [70]) into the framework of COCO.

Before introducing the details of our modifications, we first summarize our contributions when generalizing these two single-node OCC protocols to the distributed environment: (1) we make a key observation that the transaction ID (TID) in Silo and TicToc satisfies the requirement of epoch-based commit and replication, i.e., the TID assigned to each record increases monotonically and agrees with the serial order, and (2) we make extensions to support snapshot transactions in Silo and TicToc and propose an optimization to allow snapshot transactions to have one fewer round trip in the commit phase.

5.1 PT-OCC – Physical Time OCC

Many recent single-node concurrency control protocols [34, 43, 72, 74] adopted the design philosophy of Silo [63], in which "anti-dependencies" (i.e., write-after-read conflicts) are not tracked to avoid scaling bottlenecks. Instead, anti-dependencies are only enforced across epochs boundaries, which naturally fits into the design of COCO based on epoch-based commit and replication.

We now discuss how to adapt Silo to a distributed environment in COCO and present the pseudocode on the top of Figure 4. Note that we use n to denote the primary node of a record in the database and different records may have different values for n. A transaction first locks each record in its write set. If any record is locked by another transaction, the transaction simply aborts. The transaction next validates the records that only appear in its read set. The validation would fail in two scenarios: (1) the record's TID has changed, meaning the record was modified by other concurrent transactions; (2) the record is locked by another transaction. In either case, the transaction must abort, unlock the acquired locks, and discard its local write set. If the transaction successfully validates its read set, the TID is next generated. There are three criteria [63] to generate the TID for each transaction in PT-OCC: (1) it must be in the current global epoch; (2) it must be larger than the TID of any record in the read/write set; (3) it must be larger than the worker thread's last chosen TID. At last, the transaction commits changes in its write set to the database. The value of each record in the transaction's write set and the TID are sent to the primary replica. As discussed in Section 4, the writes are also asynchronously replicated to backup replicas from the coordinator node.

The protocol above guarantees serializability because all written records have been locked before validating the TIDs of read records. A more formal proof of correctness through reduction to strict two-phase locking can be found in Silo [63].

5.2 LT-OCC – Logical Time OCC

Many concurrency control algorithms [28, 52, 68] allow read-only snapshot transactions to run over a consistent snapshot of the database and commit back in time. TicToc [70], a single-node concurrency control algorithm, takes a further step by allowing read-write serializable transactions to commit back in time in the space of logical time. In TicToc, each record in the database is associated with two logical timestamps, which are represented by two 64-bit integers: $\left[\texttt{wts}, \texttt{rts}\right]$. The `wts` is the logical write timestamp, indicating when the record was written, and the `rts` is the logical *read validity* timestamp, indicating that the record can be read at any logical time ts such that $\texttt{wts} \leq ts \leq \texttt{rts}$. The key idea is to dynamically assign a logical timestamp (i.e., TID) to each transaction on commit so that each record in the read/write set is available for read and update at the same logical time.

We now discuss how to adapt TicToc [70] to a distributed environment in COCO and present the pseudocode on the bottom of Figure 4. A transaction first locks each record in its write set. Since concurrent transactions may have extended the `rts` of some records, LT-OCC updates the `rts` of each record in the transaction's write set to the latest one at the primary replica, which is available when a record is locked. The transaction next validates the records that only appear in its read set. The validation requires a TID, which is the smallest timestamp that meets the following three conditions [72]: (1) it must be in the current global epoch; (2) it must be not less than the `wts` of any record in the read set; (3) it must be larger than the `rts` of any record in the write set. The TID is first compared with the `rts` of the record in its read set. A read validation request is sent only when a record's `rts` is less than the TID. In this case, the transaction tries to extend the record's `rts` at the primary replica. The extension would fail in two scenarios: (1) the record's `wts` has changed, meaning the record was modified by other concurrent transactions; (2) the record is locked by another transaction and the `rts` is less than the TID. In either case, the `rts` cannot be extended and the transaction must abort. Otherwise, the transaction extends the record's `rts` to the TID. If the transaction fails the validation, it simply aborts, unlocks the acquired locks, and discards its local write set. Otherwise, it will commit changes in its write set to the database. As in PT-OCC, the writes are also asynchronously replicated to backup replicas from the coordinator node.

LT-OCC is able to avoid the need to validate the read set against the primary replica as long as the logical commit time falls in all time intervals of data read from replicas, even if the replicas are not fully

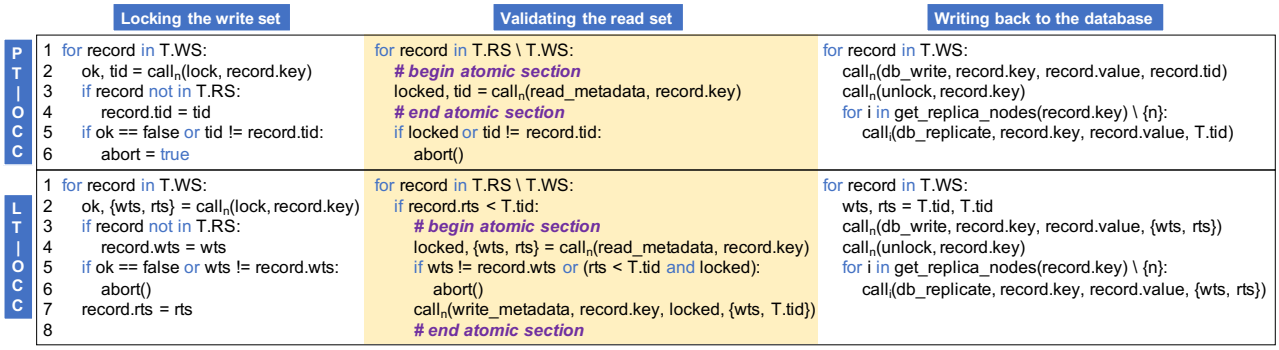


Figure 4: Pseudocode of the commit phase in PT-OCC and LT-OCC

up to date. Informally, the protocol above guarantees serializability because all the reads and writes of a transaction happen at the same logical time. From logical time perspective, all accesses happen simultaneously. A more formal proof of how logical timestamps enforce serializability can be found in TicToc [70].

5.3 Snapshot Transactions

Serializability allows transactions to run concurrently while ensuring the state of the database is equivalent to some serial ordering of the transactions. In contrast, snapshot transactions only run over a consistent snapshot of the database, meaning read/write conflicts are not detected. As a result, a database system running snapshot isolation has a lower abort rate and higher throughput.

Many systems adopt a multi-version concurrency control (MVCC) algorithm to support snapshot isolation (SI). In an MVCC-based system, a timestamp is assigned to a transaction when it starts to execute. By reading all records that have overlapping time intervals with the timestamp, the transaction is guaranteed to observe the state of the database (i.e., a consistent snapshot) at the time when the transaction began. Instead of maintaining multiple versions for each record, we made minor changes to the algorithm discussed in Section 5 to support snapshot isolation. The key idea behind is to ensure that all reads are from a consistent snapshot of the database and there are no conflicts with any concurrent updates made since that snapshot. We now describe the extensions making both PT-OCC and LT-OCC support snapshot transactions.

5.3.1 Snapshot transactions in PT-OCC. In PT-OCC, a transaction first locks all records in the write set and next validates each record in the read set to see if there exists any concurrent modification. If the validation succeeds, the transaction can safely commit under serializability. There is a short period of time after all records in the write set are locked and before any record in the read set is validated. The serialization point can be any physical time during the period.

To support snapshot transactions, a transaction can validate its read set without locks being held for each record in its write set. As long as no changes are detected, it is guaranteed that the transaction reads a consistent snapshot from the database. Likewise, there is a short period of time after the transaction finishes all reads and before the validation, and the snapshot is taken at some physical time during the period above. Snapshot transactions are more likely to commit than serializable transactions, since read validation can happen right after execution without locks being held.

5.3.2 Snapshot transactions in LT-OCC. In LT-OCC, the TID is assigned to each serializable transaction after all records in the write set have been locked. The TID must be larger than the `rts` of each record in the write set and not less than the `wts` of each record in the read set. To support snapshot transactions, LT-OCC assigns an additional TID_{SI} to each transaction, which is the smallest timestamp not less than the `wts` of each record in the read set. Since the new TID_{SI} is not required to be larger than the `rts` of each record in the write set, it's usually less than the TID for serializable transactions allowing more transactions to commit. During read validation, a transaction can safely commit under snapshot isolation as long as each record in the read set does not change until logical time TID_{SI}.

5.3.3 Parallel locking and validation optimization. As we discussed in sections above, a snapshot transaction can validate its read set regardless of its write set in both PT-OCC and LT-OCC. In particular, read validation can happen before the records in the write set have been locked in PT-OCC. Likewise, the calculation of TID_{SI} does not depend on the `rts` of each record in the write set in LT-OCC, which indirectly implies that read validation can happen independently.

We now introduce an optimization we call *parallel locking and validation optimization*, which combines the first two steps in the commit phase. In order words, locking the write set and validating the read set are now allowed to happen in parallel at the same time, making both PT-OCC and LT-OCC have one fewer round trip of network communication. This is the key reason that COCO running transactions in snapshot isolation has higher throughput even when the contention in a workload is low.

6 IMPLEMENTATION

This section describes COCO's underlying data structures, disk logging and checkpointing for durability and recovery, and implementation of insert and delete database operations.

6.1 Data Structures

COCO is a distributed in-memory OLTP database, in which each table in COCO has a pre-defined schema with typed and named attributes. Transactions are submitted to the system through pre-compiled *stored procedures* with different parameters, as in many popular systems [34, 62, 63, 67, 70]. Arbitrary logic (e.g., read/write and insert/delete operations) can be implemented in a stored procedure in C++.

Tables are currently implemented as a collection of hash tables — a primary hash table and zero or more secondary hash tables. A record is accessed through the probing primary hash table. Two probes are needed for secondary index lookups, i.e., one in a secondary hash table to find the primary key, followed by a lookup on the primary hash table. The system currently does not support range queries, but can be easily adapted to tree structures [6, 38, 66]. Note that the two concurrency control algorithms [63, 70] have native support for range queries with phantom prevention.

In COCO, we use TIDs to detect conflicts. We record the TID value as one or more 64-bit integers depending on the underlying concurrency control algorithm, and the TID is attached to each record in a table's primary hash table. The high 35 bits of each TID contain an epoch number, which indicates the epoch that a transaction comes from. The middle 27 bits are used to distinguish transactions within the same epoch. The remaining two bits are the status bits showing if the record has been deleted or locked. Likewise, we use two 64-bit integers as a TID to represent `wts` and `rts` in LT-OCC. Note that status bits only exist in the integer indicating the `wts`. By default, the epoch size is 10 ms. The number of bits reserved for epochs is sufficient for ten years, and the number of bits reserved for transactions are sufficient for over 100 million transactions per epoch.

6.2 Disk Logging and Checkpointing

As shown in Section 3.2, a commit record is written to disk when an epoch commits. On recovery, the system uses the commit record to decide the outcome of the whole epoch. However, the commit record does not have the writes of each transaction from an epoch. As a result, COCO requires that the underlying concurrency control algorithm must properly log to disk as well.

We now take PT-OCC as an example to show how transactions are logged to disk. In COCO, each transaction is run by a single worker thread, which has a local recovery log. The writes of committed transactions are first buffered in memory. They are flushed to disk when the local buffer fills or when the system enters to the next epoch. A log entry contains the information of a single write to a record in the database with the following information: (1) table and partition IDs, (2) TID, (3) primary key, and (4) value. A log entry in LT-OCC is the same as in PT-OCC, except for the TID, which has both `wts` and `rts`.

To bound the recovery time, a separate checkpointing thread can be used to periodically checkpoint the database to disk as in SiloR [74]. The checkpointing thread first logs the current epoch number e_c to disk, and next scans the whole database and logs each record to disk. Note that the epoch number e_c indicates when the checkpoint begins, and all log entries with embedded epoch numbers smaller than e_c can be safely deleted after the checkpoint finishes.

On recovery, COCO recovers the database through checkpoints and log entries. The system first loads the most recent checkpoint if available, and next replays all epochs since the checkpoint. An epoch is only replayed if the commit record indicates committed. Note that the database can be recovered in parallel with multiple worker threads. A write can be applied to the database as long as its TID is larger than the latest one in the database.

6.3 Deletes and Inserts

A record deleted by a transaction is not immediately deleted from the hash table. Instead, the system only marks the record as deleted by setting the delete status bit and registers it for garbage collection. This is because other concurrent transactions may read the record for validation. If a transaction finds a record it read has been deleted, it aborts and retries. All marked records from an epoch can be safely deleted, when the system enters the next epoch.

When a transaction makes an insert during execution, a placeholder is created in the hash table with TID 0. This is because a transaction needs to lock each record in its write set. If a transaction aborts due to conflicts, the placeholder is marked as deleted for garbage collection. Otherwise, the transaction writes a new value to the placeholder, updates the TID, and unlocks the record.

7 EVALUATION

In this section, we study the performance of each distributed concurrency control algorithm with 2PC and epoch-based commit focusing on the following key questions:

- How does each distributed concurrency control algorithm perform with 2PC and epoch-based commit?
- How does epoch-based commit perform compared to deterministic databases?
- What's the effect of durable write latency affect on 2PC and epoch-based commit?
- How does network latency affect each distributed concurrency control algorithm?
- How much performance gain of snapshot isolation over serializability?
- What's the effect of different epoch sizes?

7.1 Experimental Setup

We run our experiments on a cluster of eight `m5.4xlarge` nodes on Amazon EC2 [2], each with 16 2.50 GHz virtual CPUs and 64 GB RAM. Each node runs 64-bit Ubuntu 18.04 with Linux kernel 4.15.0. `iperf` shows that the network between each node delivers about 4.8 Gbits/s throughput. We implement COCO in C++ and compile it using GCC 7.4.0 with `-O2` option enabled.

In our experiments, we run 12 worker threads and 2 threads for network communication on each node. In addition, we have a thread coordinating all worker threads and IO threads and we leave one more CPU thread for other processes. Each worker thread has an integrated workload generator. Aborted transaction are re-executed with an exponential back-off strategy. To study the performance gain that epoch-based commits are able to achieve in the worst case, we disable logging and checkpointing for 2PC and epoch-based commit by default, since any additional latency from durable writes gives epoch-based commits more performance advantage. All results are the average of five runs.

7.1.1 Workloads. To evaluate the performance of COCO, we run a number of experiments using the following two popular benchmarks:

YCSB: The Yahoo! Cloud Serving Benchmark (YCSB) is a simple transactional workload. It's designed to be a benchmark for facilitating performance comparisons of database and key-value

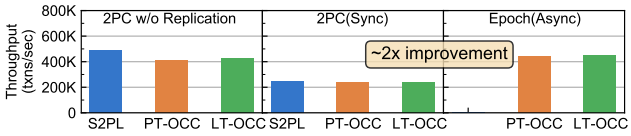


Figure 5: Throughput on YCSB

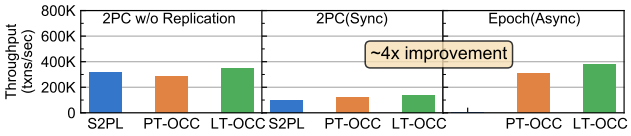


Figure 6: Throughput on TPC-C

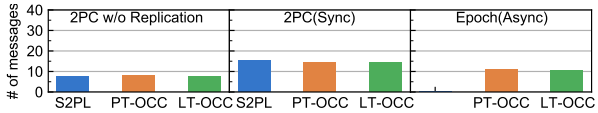


Figure 7: Average # of messages on YCSB

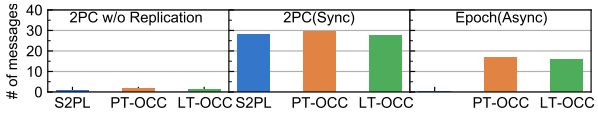


Figure 8: Average # of messages on TPC-C

systems [8]. There is a single table and each row has ten attributes. The primary key of the table is a 64-bit integer and each attribute has 10 random bytes. We run a workload mix of 80/20, i.e., each transaction has 8 read operations and 2 read/write operation. By default, we run this workload with 20% multi-partition transactions that access to multiple partitions.

TPC-C: The TPC-C benchmark is a popular benchmark to evaluate OLTP databases [1]. It models a warehouse-centric order processing application. We support the `NewOrder` and the `Payment` transaction in this benchmark, which involves customers placing orders and making payments in their districts within a local warehouse. 88% of the standard TPC-C mix consists of these two transactions. We currently do not support the other three transactions that require range scans. By default, a NewOrder transaction is followed by a Payment transaction, and 10% of NewOrder and 15% of Payment transactions are multi-partition transactions.

In YCSB, we set the number of records to 400K per partition and the number of partitions to 96, which equals to the total number of worker threads in the cluster. In TPC-C, we partition the database by warehouse and there are 96 warehouses in total. Note that we replicate the read-only `Item` table on each node. We set the number of replicas to 3 in the replicated setting, i.e., each partition has a primary partition and two backup partitions, which are always hashed to three different nodes.

7.1.2 Distributed concurrency control algorithms. We study the following distributed concurrency control algorithms in COCO. To avoid an apples-to-oranges comparison, we implemented all algorithms in C++ in our framework.

S2PL: This is a distributed concurrency control algorithm based on strict two-phase locking. Read locks and write locks are acquired as a worker runs a transaction. To avoid deadlock, the same `NO_WAIT` policy is adopted as discussed in Section 4. A worker thread updates all records and replicates the writes to replicas before releasing all acquired locks.

PT-OCC: Our physical time OCC from Section 5.1.

LT-OCC: Our logical time OCC from Section 5.2.

In addition to each concurrency control algorithm, we also support three different combinations of commit protocols and replication schemes.

2PC w/o Replication: A transaction commits with two-phase commit and no replication exists.

2PC(Sync): A transaction commits with two-phase commit. A transaction's write locks are not released until all writes are replicated on all replicas.

Epoch(Async): A transaction commits with epoch-based commit and replication, i.e., a transaction's write locks are released as soon as the writes are applied to the primary replica.

Note that a transaction cannot run with S2PL and commit with *Epoch(Async)*, since it's not straightforward to asynchronously apply writes on replicas without write locks being held in S2PL. By default, optimistic concurrency control protocols (i.e., PT-OCC and LT-OCC) are allowed to read from the nearest replica in both *2PC(Sync)* and *Epoch(Async)*.

7.2 Performance Comparison

We now study the performance of each concurrency control algorithm with different combinations of commit protocols and replication schemes.

We first ran a YCSB workload and report the result in Figure 5. S2PL with synchronous replication only achieves about 50% of the original throughput when there is no replication. This is because each transaction cannot commit before writes are replicated and even a single-partition transaction now has a round-trip delay. Similarly, both PT-OCC and LT-OCC have about 40% performance slowdown, even though they are allowed to read from the nearest replica. We now study how epoch-based commit and replication affect the performance of PT-OCC and LT-OCC, which is shown in *Epoch(Async)*. As shown in the right side of Figure 5, both PT-OCC and LT-OCC have about 2x performance improvement compared to the ones (shown in the middle) with *2PC(Sync)*. This is because the write locks can be released as soon as the writes have been applied at the primary replica. In this way, transactions no longer pay the cost of a round-trip delay.

We also ran a TPC-C workload and report the result in Figure 6. We observe that there exist a 4x performance improvement from *Epoch(Async)* over *2PC(Sync)*. The throughput even exceeds the ones (shown in the left side) with *2PC w/o Replication*, since PT-OCC and LT-OCC are allowed to read from the nearest replica.

Next, we study the number of messages sent for a transaction during execution and commit, and report the results in Figure 7 and Figure 8. We observe that a transaction in *2PC(Sync)* sends more messages than that in *2PC w/o Replication* due to replication. Compared to *2PC(Sync)*, *Epoch(Async)* effectively reduces the number of messages in both PT-OCC and LT-OCC, requiring 26% fewer messages on YCSB and 42% fewer messages on TPC-C on average. This is because *Epoch(Async)* reduces the number of network round trips during commit.

To study how *Epoch(Async)* reduces contention of a workload, we report the abort rate in TPC-C in Figure 9. We do not report the

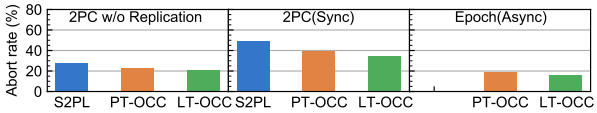


Figure 9: Abort rate on TPC-C

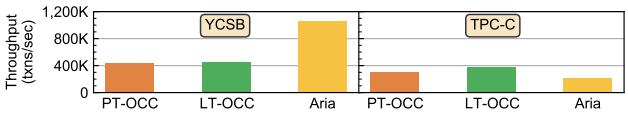


Figure 10: Comparison with deterministic databases

results for YCSB, since each read or write follows a uniform distribution and the abort rate is close to zero in all configurations. Each concurrency control method has a higher abort rate in *2PC(Sync)* than that in *2PC w/o Replication*. This is because locks are held for a longer period of time due to synchronous replication. Compared to *2PC(Sync)*, *Epoch(Async)* effectively reduces the abort rate in both PT-OCC (from 40% to 19%) and LT-OCC (from 34% to 16%).

Due to space limitations, we do not show the latency of each configuration in this experiment. Interested readers can refer to the data points with zero write latency in Figure 11.

In summary, concurrency control protocols are able to achieve 2 ∼ 4x higher throughput through epoch-based commit and replication compared to the ones with *2PC(Sync)*.

7.3 Comparison with Deterministic Databases

We now compare COCO with Aria [33], a deterministic concurrency control and replication algorithm. Aria runs transactions in batches and commits transactions deterministically upon receiving the input batch of transactions. There is no need for replicas to communicate with one another to achieve consistency. Note that Aria is implemented in the same framework as COCO to ensure a fair comparison. In this experiment, we measure the performance of Aria on one replica in the same cluster of eight nodes.

We run both YCSB and TPC-C and report the results in Figure 10. In YCSB, 20% of transactions are multi-partition transactions. In TPC-C, 10% of NewOrder and 15% of Payment transactions are multi-partition transactions. The batch size in Aria is set to 40K on YCSB and 2K on TPC-C. For convenience, the same results of PT-OCC and LT-OCC with *Epoch(Async)* in Section 7.2 are shown in Figure 10 as well. We observe that Aria achieves about 2.3x higher throughput than *Epoch(Async)* on YCSB. This is because the abort rate is close to zero on YCSB, making it an ideal workload for deterministic databases such as Aria. Even though Aria achieves higher throughput than COCO, it does not achieve the same availability. This is because there exist three copies of each record in COCO, however, there is only one replica deployed in Aria in the cluster. For TPC-C, PT-OCC and LT-OCC with *Epoch(Async)* achieve 1.5x and 1.8x higher throughput than Aria respectively. This is because there are contended writes in this workload and many transactions must abort and re-execute in Aria. In COCO, the performance advantage is larger when a workload has a higher contention. This is because epoch-based commit and replication enables the locks of a transaction to be held for a shorter period of time.

7.4 Effect of Durable Write Latency

We next study the effect of durable write latency on throughput and latency. For databases with high availability, the prepare and commit records of 2PC and epoch-based commit must be written to durable secondary storage, such as disks and replication to a remote node. To model the latency of various durable secondary storage, we add an artificial delay to 2PC and epoch-based commit through spinning.

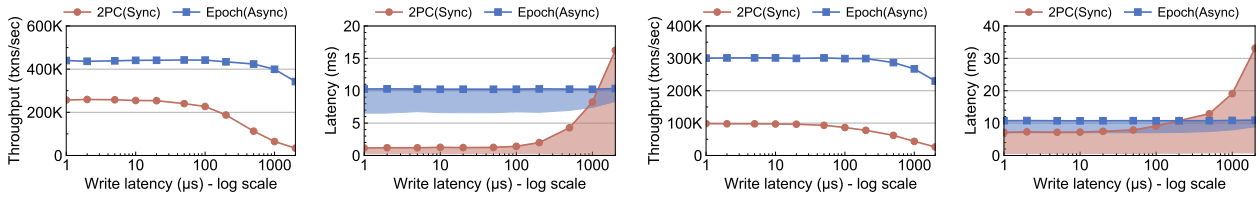
In this experiment, we vary the durable write latency from 1 μs to 2 ms and run both YCSB and TPC-C. Figures 11(a) and 11(b) show the throughput and the latency at the 99th percentile of S2PL with *2PC(Sync)* and PT-OCC with *Epoch(Async)* on YCSB. Transactions using *2PC(Sync)* have a noticeable throughput decline or a latency increase when the durable write latency is larger than 20 μs. In contrast, transactions using *Epoch(Async)* have stable throughput until the durable write latency exceeds 200 μs. In addition, the latency is more stable as well, since the epoch size can be dynamically adjusted based on different durable write latency. Likewise, we report the results of TPC-C in Figure 11(c) and 11(d). A noticeable throughput decline or latency increase is observed in *2PC(Sync)* when the durable write latency is larger than 50 μs. Furthermore, when the durable write latency exceeds 1000 μs on YCSB and 200 μs on TPC-C, the latency of *2PC(Sync)* at the 99th percentile exceeds the latency of *Epoch(Async)*, showing that epoch-based commit also helps reduce tail latency. For interested readers, we also report the latency at the 50th percentile, which is shown at the bottom of the shaded band in Figures 11(b) and 11(d). In COCO, the latency at the 50th percentile is about 6.5 ms on YCSB and 7 ms on TPC-C, since we set the epoch size to 10 ms. In *2PC(Sync)*, the latency at the 50th percentile is not sensitive to durable write latency, and it is about 270 μs on YCSB and 600 μs on TPC-C. This is because durable write latency exists only in distributed transactions and more than 80% transactions are single-partition transactions in our experiment.

Overall, epoch-based commit and replication trade latency for higher throughput. The performance advantage is more significant when durable write latency is larger. For example, with 1 ms durable write latency, *Epoch(Async)* has roughly 6x higher throughput than *2PC(Sync)* on both YCSB and TPC-C.

7.5 Wide-Area Network Experiment

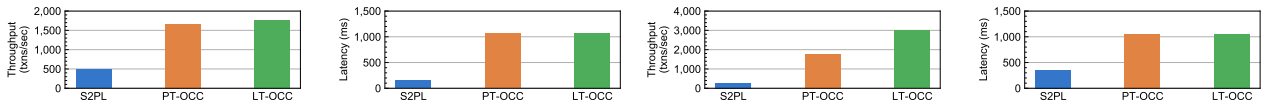
In this section, we study how epoch-based commit and replication perform compared to S2PL with *2PC(Sync)* in the wide-area network (WAN) setting. For users concerned with very high availability, wide-area replication is important because it allows the database to survive the failure of a whole data center (e.g., due to a power outage or a natural disaster).

We use three m5.4xlarge nodes running on Amazon EC2 [2]. The three nodes are in North Virginia, Ohio, and North California respectively. Each partition of the database is fully replicated in all area zones, meaning each one has a primary partition and two backup partitions. The primary partition is randomly chosen from 3 nodes. The round trip times between any two nodes are as follows: (1) North Virginia to Ohio: 11.3 ms, (2) North Virginia to North California: 60.9 ms, and (3) Ohio to North California: 50.0 ms. In this experiment, we set the epoch size to one second, and use *2PC(Sync)* in S2PL and *Epoch(Async)* in PT-OCC and LT-OCC.



(a) YCSB - Throughput (b) YCSB - Latency (c) TPC-C- Throughput (d) TPC-C- Latency

Figure 11: Performance of 2PC(Sync) and Epoch(Async) with varying durable write latency



(a) YCSB - Throughput (b) YCSB - Latency (c) TPC-C- Throughput (d) TPC-C- Latency

Figure 12: Performance of 2PC(Sync) and Epoch(Async) in a wide-area network

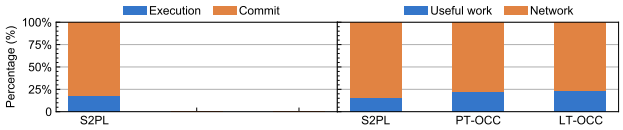


Figure 13: Breakdown on YCSB in a wide-area network

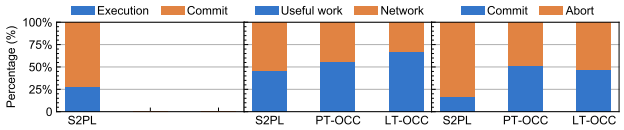


Figure 14: Breakdown on TPC-C in a wide-area network

In this experiment, we ran both YCSB and TPC-C. We report the throughput and the latency at the 99th percentile in Figure 12. Figure 12(a) and Figure 12(c) show that the throughput of *Epoch(Async)* is 4x higher than S2PL with *2PC(Sync)* on YCSB. On TPC-C, the throughput improvement is up to one order of magnitude. In contrast, S2PL has lower latency at the 99th percentile than PT-OCC and LT-OCC. For example, transactions with *2PC(Sync)* only have 150 ms latency on YCSB and 350 ms latency on TPC-C as shown in Figure 12(b) and Figure 12(d). The latency at the 99th percentile in *Epoch(Async)* is about one second, since the epoch size is set to one second in this experiment.

To understand why *Epoch(Async)* achieves higher throughput than S2PL with *2PC(Sync)*, we show some performance breakdown in Figure 13 and Figure 14. First, we observe that around 75% to 80% of time is spent in the commit phase in S2PL with *2PC(Sync)* due to multiple network round trips. The result of *Epoch(Async)* is not reported due to the difficulty in measuring how much time is spent in the execution phase. This is because a transaction in our experiment knows its complete write set from the parameters of the stored procedure and can start the commit phase immediately after the execution phase without waiting for the remote reads to return. In contrast, S2PL requires that all locks must be acquired before the commit phase begins. Second, we show how much time a transaction spends on useful work (e.g., running transaction logic or processing messages) and network I/O. We observe that a transaction in *Epoch(Async)* spends less time in network I/O than that in S2PL with *2PC(Sync)*, since fewer network round trips are needed. At last, we show the abort rate on TPC-C. The result of YCSB is not shown for the same reason as in Section 7.2. We observe that a significantly lower abort rate achieved (i.e., 50% vs. 83%) in *Epoch(Async)* than that in S2PL with *2PC(Sync)* for the same reason as above.

In summary, the performance advantage of epoch-based commit and replication over *2PC(Sync)* is more significant in the WAN setting.

7.6 Snapshot Transactions

We now study how much performance gain that PT-OCC and LT-OCC are able to achieve when they run under snapshot isolation versus serializability. In this experiment, we report the result on YCSB in Figure 15. To increase read/write conflicts, we make each access follow a Zipfian distribution [20].

We consider three different skew factors: (1) No skew (0), (1) Medium skew (0.8), and (3) High skew (0.999). As there is more contention with a larger skew factor, the performance of both algorithms under both snapshot isolation and serializability goes down. This is because the system has a higher abort rate when the workload becomes more contended. Meanwhile, both PT-OCC and LT-OCC have higher throughput under snapshot isolation because of a lower abort rate and the parallel locking and validation optimization as discussed in Section 5. Overall, both PT-OCC and LT-OCC have about 20% higher throughput running under snapshot isolation than serializability.

7.7 Effect of Epoch Size

We next study the effect of epoch size on throughput, latency, and abort rate. We varied the epoch size from 1 ms to 100 ms, and report the throughput and the latency at the 99th percentile of PT-OCC on TPC-C in Figure 16. The result of LT-OCC is almost the same and is not reported.

When an epoch-based commit happens, the system uses a global barrier to guarantee that all writes of committed transactions are replicated across all replicas. Intuitively, the system spends less time on synchronization with a larger epoch size. As we increase the epoch size, the throughput of PT-OCC continues increasing

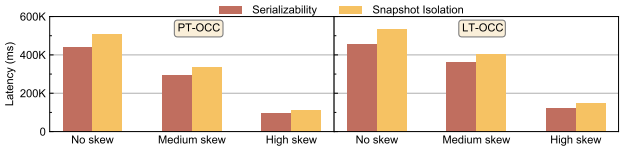


Figure 15: Snapshot transactions

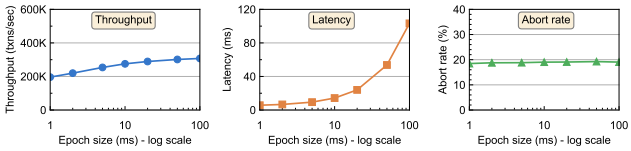


Figure 16: Effect of epoch size

and levels off beyond 50 ms. The latency at the 99th percentile roughly equals to the epoch size. The abort rate stays almost the same regardless of the epoch size, since epoch-based commit only groups the durable writes from all transactions belonging to the same epoch and does not affect the commit phase of each individual transaction.

In summary, with an epoch size of 10 ms, the system can achieve more than 93% of its maximum throughput (the one achieved with a 100 ms epoch size) and have a good balance between throughput and latency.

8 RELATED WORK

The design of COCO is inspired by many pieces of related work, including transaction processing, consistency and replication.

Transaction Processing. The seminal survey by Bernstein et al. [4] summarizes classic distributed concurrency control protocols. As in-memory databases become more popular, there has been a resurgent interest in transaction processing in both multicore processors [23, 30, 63, 68, 70] and distributed systems [10, 21, 35, 42, 53]. Many recent transactional databases [11, 34, 63] employ epochs to trade latency for higher throughput. Silo [63] reduces shared-memory writes through not detecting anti-dependencies within epochs. Obladi [11] delays updates within epochs to increase system throughput and reduce bandwidth cost. STAR [34] separates the execution of single-node and distributed transactions and runs them in different epochs. COCO is the first to leverage epochs to reduce the cost of two-phase commit (2PC) for distributed transactions and achieves strong consistency between replicas with asynchronous writes. Warranties [32] reduces coordination on read validation by maintaining time-based leases to popular records, but writes have to be delayed. In COCO, LT-OCC is able to reducere coordination without penalizing writes. This is because writes instantly make the read validity timestamps on old records expired. Eris [29] uses the datacenter network itself as a concurrency control mechanism for assigning transaction order without explicit coordination. In contrast, the epoch-based commit and replication protocol in COCO does not rely on any additional hardware and can reduce the cost of coordination even across data centers.

Replicated Systems. High availability is typically implemented through replication. Paxos [26] and other leader-less consensus protocols such as EPaxos [41] are popular solutions to coordinate the concurrent reads and writes to different copies of data while providing consistency. Spanner [9] is a Paxos-based transaction processing system based on a two-phase locking protocol. Each master replica initiates a Paxos protocol to synchronize with backup replicas. The protocol incurs multiple round-trip messages for data accesses and replication coordination. TAPIR [73] eliminates the overhead of Paxos by allowing inconsistency in the storage system and building consistent transactions using inconsistent replication. Ganymed [47, 48] runs update transactions on a single node and propagates writes of committed transactions to a potentially unlimited number of read-only replicas. In COCO, the writes of committed transactions are asynchronously applied to each replica and transactions are allowed to read from the nearest replica to reduce round-trip communication. Some recent systems [24, 36, 44, 69] optimize the latency and number of round trips needed to commit transactions across replicas. MDCC [24] is an OCC protocol that exploits generalized Paxos [27] to reduce the coordination overhead, in which a transaction can commit with a single network round trip in the normal operation. Replicated Commit [36] needs less round-trip communication by replicating the commit operation rather than the transactional log. In contrast, COCO avoids the use of 2PC and treats an epoch of transactions as the commit unit that amortizes the cost of 2PC.

Consistency and Snapshot Isolation. Due to the overhead of implementing strong isolation, many systems use weaker isolation levels instead (e.g., PSI [56], causal consistency [39], eventual consistency [59], or no consistency [50]). Lower isolation levels trade programmability for performance and scalability. In this paper, we focus on serializability and snapshot isolation, which are the gold standard for transactional applications. By maintaining multiple data versions, TxCache [49] ensures that a transaction always reads from a consistent snapshot regardless of whether each read operation comes from the database or the cache. Binning et al. [7] show how only distributed snapshot transactions pay the cost of coordination. Serial Safety Net (SSN) [65] is able to make any concurrency control protocol to support serializability by detecting dependency cycles. RushMon [54] reports the number of isolation anomalies based on the number of cycles in the dependency graph, using a sampling approach. Similar, LT-OCC in COCO is also able to detect concurrency anomalies when running snapshot transactions through a simple equality check.

9 CONCLUSION

In this paper, we presented COCO, a new distributed OLTP database, which enforces atomicity, durability and consistency at the boundaries of epochs. By separating transactions into epochs and treating a whole epoch of transactions as the commit unit, COCO is able to address inefficiency found in conventional distributed databases with two-phase commit and synchronous replication. In addition, the system supports two variants of optimistic concurrency control (OCC) using physical time and logical time with various optimizations, which are enabled by epoch-based commit and replication. Our results on two popular benchmarks show that COCO outperforms systems with conventional commit and replication schemes by up to a factor of four.

ACKNOWLEDGMENTS

We thank the reviewers for their valuable comments. Yi Lu is supported by the Facebook PhD Fellowship.

REFERENCES

[1] 2010. TPC Benchmark C. http://www.tpc.org/tpcc/.
[2] 2020. Amazon EC2. https://aws.amazon.com/ec2/.
[3] Peter Bailis, Alan Fekete, Michael J. Franklin, Ali Ghodsi, Joseph M. Hellerstein, and Ion Stoica. 2014. Coordination Avoidance in Database Systems. *PVLDB* 8, 3 (2014), 185–196.
[4] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (1981), 185–221.
[5] Philip A. Bernstein, David W. Shipman, and Wing S. Wong. 1979. Formal Aspects of Serializability in Database Concurrency Control. *IEEE Trans. Software Eng.* 5, 3 (1979), 203–216.
[6] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2018. HOT: A Height Optimized Trie Index for Main-Memory Database Systems. In *SIGMOD Conference*. 521–534.
[7] Carsten Binnig, Stefan Hildenbrand, Franz Färber, Donald Kossmann, Juchang Lee, and Norman May. 2014. Distributed Snapshot Isolation: Global Transactions Pay Globally, Local Transactions Pay Locally. *VLDB J.* 23, 6 (2014), 987–1011.
[8] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking Cloud Serving Systems with YCSB. In *SoCC*. 143–154.
[9] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson C. 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. 2012. Spanner: Google's Globally-Distributed Database. In *OSDI*. 261–264.
[10] James A. Cowling and Barbara Liskov. 2012. Granola: Low-Overhead Distributed Transaction Coordination. In *ATC*. 223–235.
[11] Natacha Crooks, Matthew Burke, Ethan Cecchetti, Sitar Harel, Rachit Agarwal, and Lorenzo Alvisi. 2018. Obladi: Oblivious Serializable Transactions in the Cloud. In *OSDI*. 727–743.
[12] Carlo Curino, Yang Zhang, Evan P. C. Jones, and Samuel Madden. 2010. Schism: a Workload-Driven Approach to Database Replication and Partitioning. *PVLDB* 3, 1 (2010), 48–57.
[13] Sudipto Das, Divyakant Agrawal, and Amr El Abbadi. 2010. G-Store: A Scalable Data Store for Transactional Multi Key Access in the Cloud. In *SoCC*. 163–174.
[14] 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. In *SOSP*. 205–220.
[15] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *SIGMOD Conference*. 1–8.
[16] Bailu Ding, Lucja Kot, and Johannes Gehrke. 2018. Improving Optimistic Concurrency Control Through Transaction Batching and Operation Reordering. *PVLDB* 12, 2 (2018), 169–182.
[17] Kapali P. Eswaran, Jim Gray, Raymond A. Lorie, and Irving L. Traiger. 1976. The Notions of Consistency and Predicate Locks in a Database System. *Commun. ACM* 19, 11 (1976), 624–633.
[18] Jose M. Faleiro, Daniel Abadi, and Joseph M. Hellerstein. 2017. High Performance Transactions via Early Write Visibility. *PVLDB* 10, 5 (2017), 613–624.
[19] Jose M. Faleiro and Daniel J. Abadi. 2015. Rethinking Serializable Multiversion Concurrency Control. *PVLDB* 8, 11 (2015), 1190–1201.
[20] Jim Gray, Prakash Sundaresan, Susanne Englert, Kenneth Baclawski, and Peter J. Weinberger. 1994. Quickly Generating Billion-Record Synthetic Databases. In *SIGMOD Conference*. 243–252.
[21] Rachael Harding, Dana Van Aken, Andrew Pavlo, and Michael Stonebraker. 2017. An Evaluation of Distributed Concurrency Control. *PVLDB* 10, 5 (2017), 553–564.
[22] Bettina Kemme and Gustavo Alonso. 2000. Don't Be Lazy, Be Consistent: Postgres-R, A New Way to Implement Database Replication. In *VLDB*. 134–143.
[23] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD Conference*. 1675–1687.
[24] Tim Kraska, Gene Pang, Michael J. Franklin, Samuel Madden, and Alan Fekete. 2013. MDCC: Multi-Data Center Consistency. In *Eurosys*. 113–126.
[25] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (1981), 213–226.
[26] Leslie Lamport. 2001. Paxos Made Simple. *ACM SIGACT News* 32, 4 (2001), 18–25.
[27] Leslie Lamport. 2005. Generalized Consensus and Paxos. https://www.microsoft.com/en-us/research/publication/generalized-consensus-and-paxos/.
[28] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB* 5, 4 (2011), 298–309.
[29] Jialin Li, Ellis Michael, and Dan R. K. Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *SOSP*. 104–120.
[30] Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. 2017. Cicada: Dependably Fast Multi-Core In-Memory Transactions. In *SIGMOD Conference*. 21–35.
[31] Qian Lin, Pengfei Chang, Gang Chen, Beng Chin Ooi, Kian-Lee Tan, and Zhengkui Wang. 2016. Towards a Non-2PC Transaction Management in Distributed Database Systems. In *SIGMOD Conference*. 1659–1674.
[32] Jed Liu, Tom Magrino, Owen Arden, Michael D. George, and Andrew C. Myers. 2014. Warranties for Faster Strong Consistency. In *NSDI*. 503–517.
[33] Yi Lu, Xiangyao Yu, Lei Cao, and Samuel Madden. 2020. Aria: A Fast and Practical Deterministic OLTP Database. *PVLDB* 13, 11 (2020), 2047–2060.
[34] Yi Lu, Xiangyao Yu, and Samuel Madden. 2019. STAR: Scaling Transactions through Asymmetric Replication. *PVLDB* 12, 11 (2019), 1316–1329.
[35] Hatem A. Mahmoud, Vaibhav Arora, Faisal Nawab, Divyakant Agrawal, and Amr El Abbadi. 2014. MaaT: Effective and Scalable Coordination of Distributed Transactions in the cloud. *PVLDB* 7, 5 (2014), 329–340.
[36] Hatem A. Mahmoud, Faisal Nawab, Alexander Pucher, Divyakant Agrawal, and Amr El Abbadi. 2013. Low-Latency Multi-Datacenter Databases using Replicated Commit. *PVLDB* 6, 9 (2013), 661–672.
[37] Sujaya Maiyya, Faisal Nawab, Divy Agrawal, and Amr El Abbadi. 2019. Unifying Consensus and Atomic Commitment for Effective Cloud Data Management. *PVLDB* 12, 5 (2019), 611–623.
[38] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *EuroSys*. 183–196.
[39] Syed Akbar Mehdi, Cody Littley, Natacha Crooks, Lorenzo Alvisi, Nathan Bronson, and Wyatt Lloyd. 2017. I Can't Believe It's Not Causal! Scalable Causal Consistency with No Slowdown Cascades. In *NSDI*. 453–468.
[40] C. Mohan, Bruce G. Lindsay, and Ron Obermarck. 1986. Transaction Management in the R* Distributed Database Management System. *ACM Trans. Database Syst.* 11, 4 (1986), 378–396.
[41] Iulian Moraru, David G. Andersen, and Michael Kaminsky. 2013. There Is More Consensus in Egalitarian Parliaments. In *SOSP*. 358–372.
[42] Shuai Mu, Yang Cui, Yang Zhang, Wyatt Lloyd, and Jinyang Li. 2014. Extracting More Concurrency from Distributed Transactions. In *OSDI*. 479–494.
[43] Neha Narula, Cody Cutler, Eddie Kohler, and Robert Morris. 2014. Phase Reconciliation for Contended In-Memory Transactions. In *OSDI*. 511–524.
[44] Faisal Nawab, Vaibhav Arora, Divyakant Agrawal, and Amr El Abbadi. 2015. Minimizing Commit Latency of Transactions in Geo-Replicated Data Stores. In *SIGMOD Conference*. 1279–1294.
[45] Diego Ongaro and John K. Ousterhout. 2014. In Search of an Understandable Consensus Algorithm. In *ATC*. 305–319.
[46] Andrew Pavlo, Carlo Curino, and Stanley B. Zdonik. 2012. Skew-aware Automatic Database Partitioning in Shared-nothing, Parallel OLTP Systems. In *SIGMOD Conference*. 61–72.
[47] Christian Plattner and Gustavo Alonso. 2004. Ganymed: Scalable Replication for Transactional Web Applications. In *Middleware Conference*. 155–174.
[48] Christian Plattner, Gustavo Alonso, and M. Tamer Özsu. 2008. Extending DBMSs with Satellite Databases. *VLDB J.* 17, 4 (2008), 657–682.
[49] Dan R. K. Ports, Austin T. Clements, Irene Zhang, Samuel Madden, and Barbara Liskov. 2010. Transactional Consistency and Automatic Management in an Application Data Cache. In *OSDI*. 279–292.
[50] Benjamin Recht, Christopher Ré, Stephen J. Wright, and Feng Niu. 2011. Hogwild: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent. In *NIPS*. 693–701.
[51] David P. Reed. 1978. *Naming and Synchronization in a Decentralized Computer System.* Ph.D. Dissertation. MIT, Cambridge, MA, USA.
[52] David P. Reed. 1983. Implementing Atomic Actions on Decentralized Data. *ACM Trans. Comput. Syst.* 1, 1 (1983), 3–23.
[53] Alex Shamis, Matthew Renzelmann, Stanko Novakovic, Georgios Chatzopoulos, Aleksandar Dragojevic, Dushyanth Narayanan, and Miguel Castro. 2019. Fast General Distributed Transactions with Opacity. In *SIGMOD Conference*. 433–448.
[54] Zechao Shang, Jeffrey Xu Yu, and Aaron J. Elmore. 2018. RushMon: Real-time Isolation Anomalies Monitoring. In *SIGMOD Conference*. 647–662.
[55] Jeff Shute, Radek Vingralek, Bart Samwel, Ben Handy, Chad Whipkey, Eric Rollins, Mircea Oancea, Kyle Littlefield, David Menestrina, Stephan Ellner, John Cieslewicz, Ian Rae, Traian Stancescu, and Himani Apte. 2013. F1: A Distributed SQL Database That Scales. *PVLDB* 6, 11 (2013), 1068–1079.
[56] Yair Sovran, Russell Power, Marcos K. Aguilera, and Jinyang Li. 2011. Transactional Storage for Geo-replicated Systems. In *SOSP*. 385–400.
[57] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*. 1150–1160.
[58] Andrew S. Tanenbaum and Herbert Bos. 2014. *Modern Operating Systems* (4th ed.). Prentice Hall Press.
[59] Douglas B. Terry, Marvin Theimer, Karin Petersen, Alan J. Demers, Mike Spreitzer, and Carl Hauser. 1995. Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System. In *SOSP*. 172–183.
[60] Robert H. Thomas. 1979. A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases. *TODS* 4, 2 (1979), 180–209.
[61] Alexander Thomson and Daniel J. Abadi. 2010. The Case for Determinism in Database Systems. *PVLDB* 3, 1 (2010), 70–80.

[62] 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 *SIGMOD Conference*. 1–12.

[63] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *SOSP*. 18–32.

[64] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *SIGMOD Conference*. 1041–1052.

[65] Tianzheng Wang, Ryan Johnson, Alan Fekete, and Ippokratis Pandis. 2017. Efficiently Making (almost) Any Concurrency Control Mechanism Serializable. *VLDB J.* 26, 4 (2017), 537–562.

[66] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD Conference*. 473–488.

[67] Xingda Wei, Jiaxin Shi, Yanzhe Chen, Rong Chen, and Haibo Chen. 2015. Fast In-memory Transaction Processing Using RDMA and HTM. In *SOSP*. 87–104.

[68] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *PVLDB* 10, 7 (2017), 781–792.

[69] Xinan Yan, Linguan Yang, Hongbo Zhang, Xiayue Charles Lin, Bernard Wong, Kenneth Salem, and Tim Brecht. 2018. Carousel: Low-Latency Transaction Processing for Globally-Distributed Data. In *SIGMOD Conference*. 231–243.

[70] Xiangyao Yu, Andrew Pavlo, Daniel Sánchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *SIGMOD Conference*. 1629–1642.

[71] Xiangyao Yu, Yu Xia, Andrew Pavlo, Daniel Sánchez, Larry Rudolph, and Srinivas Devadas. 2018. Sundial: Harmonizing Concurrency Control and Caching in a Distributed OLTP Database Management System. *PVLDB* 11, 10 (2018), 1289–1302.

[72] Yuan Yuan, Kaibo Wang, Rubao Lee, Xiaoning Ding, Jing Xing, Spyros Blanas, and Xiaodong Zhang. 2016. BCC: Reducing False Aborts in Optimistic Concurrency Control with Low Cost for In-Memory Databases. *PVLDB* 9, 6 (2016), 504–515.

[73] Irene Zhang, Naveen Kr. Sharma, Adriana Szekeres, Arvind Krishnamurthy, and Dan R. K. Ports. 2015. Building Consistent Transactions with Inconsistent Replication. In *SOSP*. 263–278.

[74] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *OSDI*. 465–477.