Aria: A Fast and Practical Deterministic OLTP Database

Yi Lu [1], Xiangyao Yu [2], Lei Cao [1], Samuel Madden[1]
[1]*Massachusetts Institute of Technology, Cambridge, MA, USA*
[2]*University of Wisconsin-Madison, Madison, WI, USA*
{yilu,lcao,madden}@csail.mit.edu, yxy@cs.wisc.edu

ABSTRACT

Deterministic databases are able to efficiently run transactions across different replicas without coordination. However, existing state-of-the-art deterministic databases require that transaction read/write sets are known before execution, making such systems impractical in many OLTP applications. In this paper, we present Aria, a new distributed and deterministic OLTP database that does not have this limitation. The key idea behind Aria is that it first executes a batch of transactions against the same database snapshot in an *execution phase*, and then deterministically (without communication between replicas) chooses those that should commit to ensure serializability in a *commit phase*. We also propose a novel deterministic reordering mechanism that allows Aria to order transactions in a way that reduces the number of conflicts. Our experiments on a cluster of eight nodes show that Aria outperforms systems with conventional nondeterministic concurrency control algorithms and the state-of-the-art deterministic databases by up to a factor of two on two popular benchmarks (YCSB and TPC-C).

PVLDB Reference Format:
Yi Lu, Xiangyao Yu, Lei Cao and Samuel Madden. Aria: A Fast and Practical Deterministic OLTP Database. *PVLDB*, 13(11): 2047-2060, 2020.
DOI: https://doi.org/10.14778/3407790.3407808

1. INTRODUCTION

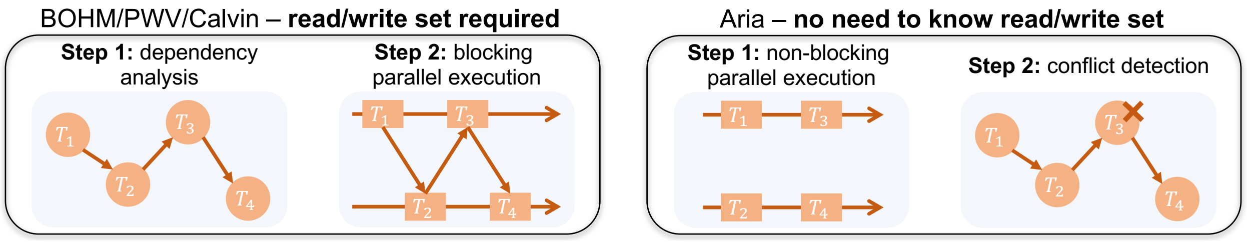
Modern database systems employ replication for high availability and data partitioning for scale-out. Replication allows systems to provide high availability, i.e., tolerance to machine failures, but also incurs additional network round trips to ensure writes are synchronized to replicas. Partitioning across several nodes allows systems to scale to larger databases. However, most implementations require the use of two-phase commit (2PC) [37] to address the issues caused by nondeterministic events such as system failures and race conditions in concurrency control. This introduces additional latency to distributed transactions and impairs scalability and availability (e.g., due to coordinator failures).

Deterministic concurrency control algorithms [18, 19, 51, 52] provide a new way of building distributed and highly available database systems. They avoid the use of expensive commit and replication protocols by ensuring different replicas always *independently* produce the same results as long as the same input transactions are given. Therefore, rather than replicating and synchronizing the updates of distributed transactions, deterministic databases only have to replicate the input transactions across different replicas, which can be done asynchronously and often with much less communication. In addition, deterministic databases avoid the use of two-phase commit, since they naturally eliminate nondeterministic race conditions in concurrency control and are able to recover from system failures by re-executing the same original input transactions.

The state-of-the-art deterministic databases, BOHM [19], PWV [18], and Calvin [52], achieve determinism through *dependency graphs* or *ordered locks*. The key idea in BOHM and PWV is that a dependency graph is built from a batch of input transactions based on the read/write sets. In this way, the database can produce deterministic results as long as the transactions are run following the dependency graph. The key idea in Calvin is that read/write locks are acquired prior to executing the transaction, and according to the ordering of input transactions. A transaction is assigned to a worker thread for execution once all needed locks are granted. As shown in the left side of Figure 1, existing deterministic databases perform dependency analysis before transaction execution, which requires that the read/write set of a transaction be known a priori. For very simple transactions, e.g., that only access to records via equality lookups on a primary key, this can be done easily. However, in reality, many transactions access records through complex predicates over non-key attributes; for such queries, these systems must execute the query at least twice: once to determine the read/write set, once to execute the query, and possibly more times if the pre-determined read/write set changes between these two executions. In addition, Calvin requires the use of a single-threaded lock manager per database partition, which significantly limits the concurrency it can achieve.

In this paper, we propose a new system, Aria, to address the limitations in previous deterministic OLTP databases with a fundamentally different mechanism, which does not require any analysis or pre-execution of input transactions. Aria runs transactions in batches. The key idea is that each replica runs an identical batch of transactions on an iden-

Proceedings of the VLDB Endowment, Vol. 13, No. 11
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3407790.3407808



BOHM/PWV/Calvin – read/write set required
Step 1: dependency analysis
Step 2: blocking parallel execution
T_1
T_2
T_3
T_4
Aria – no need to know read/write set
Step 1: non-blocking parallel execution
Step 2: conflict detection

Figure 1: Comparison of Aria, BOHM, PWV, and Calvin

tical database snapshot, and resolves conflicts in the same way, ensuring deterministic execution. As shown in the right side of Figure 1, each replica reads from the current snapshot of the database and executes all transactions to completion in an *execution phase*, and then chooses deterministically which transactions should commit and which should abort to ensure serializability in a *commit phase*. For good scalability and high transaction throughput, the serializability check in the commit phase is performed in parallel on each transaction independently and no central coordination is required. Aborted transactions will be scheduled for re-execution at the beginning of the next batch. In this way, Aria can enforce determinism without needing to know the read or write sets of input transactions in advance. Note that although optimistic concurrency control (OCC) algorithms [29] also resolve conflicts after execution, transactions in OCC can commit in a non-deterministic order depending on scheduling variations, which is fundamentally different from Aria.

We also introduce a novel deterministic reordering mechanism that brings a new opportunity for reducing conflicts and improving the throughput of Aria. Unlike existing deterministic databases [19, 51] that execute transactions strictly following the ordering of the input, our reordering mechanism allows Aria to commit transactions in an order that reduces aborts. Furthermore, the reordering is done in parallel as a part of the process of identifying aborted transactions and imposes minimal execution overhead.

Our evaluation on a single multicore node shows that Aria outperforms conventional nondeterministic concurrency control algorithms and state-of-the-art deterministic databases by a large margin on YCSB [11]. Further, Aria is able to achieve higher transaction throughput by up to a factor of three with deterministic reordering on a YCSB workload with skewed access. On a subset of TPC-C [1], in which more conflicts exist, Aria achieves competitive performance to PWV and higher performance than all other deterministic databases. Our evaluation on a cluster of eight nodes running on Amazon EC2 shows that Aria outperforms Calvin by up to a factor of two on YCSB and TPC-C. Finally, we show Aria achieves near linear scalability to 16 nodes.

In summary, we make the following major contributions:

- We present Aria, a fast and practical deterministic OLTP database. It supports deterministic transaction execution without any prior knowledge of the input transactions.
- We introduce a deterministic reordering scheme to reduce conflicts during the commit phase in Aria.
- We describe an implementation of Aria and report a detailed evaluation on two popular benchmarks. Overall, Aria outperforms the state-of-the-art deterministic databases by a large margin on a single node and up to a factor of two on a cluster of eight nodes.

2. BACKGROUND

In this section, we summarize the challenges with using replication as a means to achieve high availability, with respect to both existing nondeterministic and deterministic concurrency control protocols.

2.1 Replication in Nondeterministic Databases

Highly available systems normally replicate data across multiple machines, such that the system still provides some degree of availability when one or more servers fail. We broadly classify existing replication strategies into two categories: (1) asynchronous replication [10, 14], and (2) synchronous replication [27]. In asynchronous replication, data is asynchronously propagated between replicas, meaning a transaction can commit before writes arrive at all replicas. Asynchronous replication reduces the latency to commit transactions, but may suffer from data loss when failures occur. Instead, in synchronous replication, writes are propagated across all replicas before a transaction commits. Synchronous replication increases latency but ensures strong consistency between replicas and has been a recent focus of the research community [12, 35, 62].

Most transactional systems provide serializability, which requires transactions to produce the results following *some* serial order. Most concurrency control algorithms (e.g., optimistic concurrency control [29] and strict two-phase locking [17]) are nondeterministic, meaning different replicas may diverge when given the same input, because of timing and performance differences on replicas executing transactions in parallel. As a result, some care is needed to guarantee consistency between replicas; most replication protocols adopt either a primary-backup scheme [7] or a state machine replication scheme [30]. In a primary-backup system, the primary runs transactions and ships the results to one or more backups. The backups apply the changes in order to the database and periodically acknowledge the writes back to the primary. A transaction can commit after the writes have been acknowledged by backups. In systems using state machine replication, transactions can run on any replica, but read/write operations need to be sequenced using consensus protocols (e.g., Paxos [30] or Raft [42]). Thus, nondeterministic systems are able to stay consistent but suffer from several limitations, including needing multiple rounds of synchronous communication to commit transactions, and a need to synchronize typically large output values rather than smaller input operations.

2.2 Deterministic Concurrency Control

Deterministic concurrency control algorithms [51] were proposed to address these challenges. In a deterministic database, each replica runs the same set of ordered transactions deterministically, and converts the database from the same initial state to the same final state. Typically, in these systems, transactions first go through a sequencing layer before execution. This sequencing layer acts as the middle layer between clients and the database, and decides a total ordering of transactions submitted by clients. Usually the sequencing layer runs over multiple machines to avoid having a single point of failure and achieves consensus through a replicated log built on top of Paxos [30] or Raft [42] instances. To avoid non-determinism (e.g., due to system calls to get the current time or generate a random number), the sequencing layer pre-processes incoming transactions to eliminate this non-determinism by replacing such function calls with constant values before passing to replicas. As a result, replicas do not need to communicate with one another to remain consistent, making replication simpler and more efficient. In addition, deterministic databases also reduce the overhead of distributed transactions by avoiding two-phase commit (2PC) [37]. In nondeterministic databases, 2PC is used to ensure that the participants in a distributed transaction reach a single commit/abort decision. In contrast, the failure of one participating node does not affect the commit/abort decision in deterministic databases [51].

More recently, deterministic protocols based on dependency graphs [18, 19] or ordered locks [51, 52] have been proposed to provide more parallelism while ensuring determinism. Although the systems above take an important step towards building deterministic databases, they still have one limitation that makes them impractical: the dependency analysis requires that the read/write sets of a transaction (i.e., a collection of primary keys) must be known a priori before execution. For transactions with secondary index lookups or other data dependencies, it may not be possible to determine reads/writes in advance. In such cases, a transaction must run over the database to determine its read/write sets. The transaction then aborts and uses these pre-computed read/write sets for re-execution. However, the transaction may be executed multiple times if any key in the read/write sets changes during this re-execution. We next discuss why the design of existing deterministic databases can lead to undesirable performance.

BOHM, a single-node deterministic database, runs transactions in two steps. In the first step, it inserts placeholders for the primary keys in each input transaction's write set along with the transaction ID (TID) into a multi-version storage layer. In the second step, a transaction must read a particular version for each key in its read set — the one with the largest TID up to the transaction's TID — to enforce determinism. When all the keys in a transaction's read set are ready, the transaction can execute and update the placeholders that were inserted earlier. To avoid contention when inserting placeholders into the storage layer, each worker thread in BOHM scans all input transactions to look for the primary keys for update in its own partition.

PWV, a single-node deterministic database, first decomposes each input transaction into a set of pieces (i.e., sub-transactions) such that each piece only reads from or writes into one partition. It next builds a dependency graph, in which an edge indicates conflicts between pieces within a partition, and uses it to schedule the execution of each piece to meet the requirement of both data dependencies and commit dependencies. Different from BOHM, PWV commits the writes of each transaction at a finer granularity of pieces instead of the whole transaction. In this way, later transactions spend less time waiting to see the writes from earlier transactions at the cost of a more expensive fine-grained dependency graph analysis.

```
Data: write reservations: writes

### the execution phase of batch i ###
for each transaction T in batch i:
  Execute(T, db)
  ReserveWrite(T, writes)

### the commit phase of batch i ###
for each transaction T in batch i:
  Commit(T, db, writes)
```

Figure 2: Execution and commit phases in Aria

Calvin, a distributed deterministic database, uses a single-threaded lock manager to scan the input transactions and grants read/write locks based on pre-declared read/write sets. Once all locks of a transaction are acquired, the lock manager next assigns the transaction to an available worker thread for execution. Once the execution finishes, the worker thread releases the locks. The separate roles of lock manager threads and worker threads increase the system's overall synchronization cost, and locks on one partition must be granted by a single lock manager thread. In machines with many cores, it is difficult to saturate system resource with one single-threaded lock manger, as many worker threads are idle waiting for locks. To solve this issue, the database can be partitioned into multiple partitions per machine, but this introduces additional overhead by introducing more multi-partition transactions with overhead due to redundant execution [52]. For example, a transaction updating both y_1 and y_2 to $f(x)$ must run $f(x)$ twice, if locks on y_1 and y_2 are granted by two different lock mangers.

Note that both BOHM and PWV enforce determinism through dependency graphs, which are built before transaction execution. Calvin runs transactions following dependency graphs as well, but it is achieved implicitly through ordered locks. We will discuss how Aria achieves determinism and addresses the issues above in the following sections.

3. SYSTEM OVERVIEW

Given background, we now give a high level overview how Aria achieves deterministic execution. As in existing deterministic databases, replicas in Aria do not need to communicate with one another, and run transactions independently. This is because the same results will always be produced under deterministic execution. Each transaction passes through a sequencing layer and is assigned a unique transaction ID (TID). The TID indicates a total ordering among transactions; by default it indicates the commit order of transactions, but our deterministic reordering technique, which we will describe in Section 5, may commit transactions in a different order.

Aria runs transactions in batches in two different phases: an *execution phase* and a *commit phase*, separated by a barrier. Within a batch, each Aria replica runs transactions in parallel and in any order with multiple worker threads. The transactions are assigned to worker threads in a round-robin fashion. As shown in Figure 2, in the execution phase, each transaction reads from the current snapshot of the database and keeps the writes in a local write set. Unlike BOHM, Aria adopts a single-version approach, even though it buffers

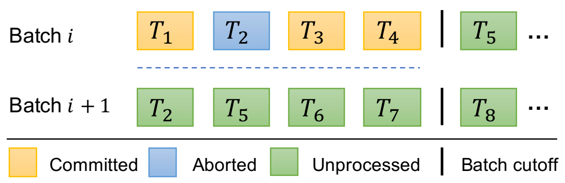


Figure 3: Batch processing in Aria

writes for transactions until the end of a batch. Once all transactions in the batch finish execution on a replica, it enters the commit phase. The commit phase on a replica also runs on multiple worker threads, each executing independently. In the commit phase, a thread aborts transactions that performed conflicting operations with *earlier* transactions, i.e., those with *smaller* TIDs. For example, a transaction would abort if it reads a record modified by some earlier transaction. Aborted transactions are automatically scheduled at the beginning of the next batch for execution, unless the transaction is aborted explicitly (e.g., for violating some integrity constraint). If a transaction does not have conflicts with earlier transactions, the system applies its changes to the database. Consider the example in Figure 3, with four transactions in a batch. Transaction T_2 has conflicts with T_1. To ensure serializability, transaction T_2 aborts and is scheduled at the beginning of the next batch for execution.

In addition to resolving conflicts following the ordering of input transactions, Aria uses a deterministic reordering technique that reduces conflicts in the commit phase. For example, consider a sequence of n transactions, in which T_i reads record i and updates record $i+1$, for $0 < i \leq n$. Thus, each transaction (except T_1) reads a record that has been modified by the previous transaction. Following the default algorithm in Aria, the first transaction is the only transaction that can commit. However, these "conflicting" transactions can still commit under the serial order $T_n \rightarrow T_{n-1} \rightarrow \cdots \rightarrow T_1$. With deterministic reordering, Aria does not physically reorder the input transactions. Instead, it commits more transactions so that the results are still serializable but equivalent to a different ordering.

4. DETERMINISTIC BATCH EXECUTION

In this section, we describe the details of how Aria runs transactions deterministically in two phases, such that the serial order of committed transactions strictly follows the ordering of input transactions. In Section 5, we will present the reordering technique to relax this constraint. We also give a proof to show that Aria produces serializable results. Finally, we discuss the phantom problem and limitations.

4.1 The Execution Phase

During this phase, each transaction reads from the current snapshot of the database, executes its logic, and writes the result to a local write set, as shown from line 1 to line 3 in Figure 4. Since the changes made by transactions are kept locally and are not directly committed to the database, the snapshot read by each transaction in the execution phase is always the same.

Once a transaction finishes execution, it goes through its local write set and makes *write reservations* for each entry in its write set, as shown from line 5 to line 7 in Figure 4.

A write reservation on a previously reserved value can be made only when the reserving transaction has a smaller TID than the previous reserver. If a reservation from a larger TID already exists, the old reservation will be voided and a new reservation with the smaller TID will be made. If a transaction cannot make a reservation, the transaction must abort. In addition, the transaction can also skip the commit phase as a performance optimization, since it knows at least one reservation is not successful. Note that a transaction is not allowed to skip the rest of reservations due to an earlier unsuccessful reservation and must continue making all reservations. This is because reservations are made in parallel by multiple worker threads and omitting the rest of reservations will produce nondeterministic results.

```
 1 Function: Execute(T, db)
 2       Read from the latest snapshot of db
 3       Execute to compute T's read/write set # (RS & WS)
 4
 5 Function: ReserveWrite(T, writes)
 6       for each key in T.WS:
 7           writes[key] = min(writes[key], T.TID)
 8
 9 Function: Commit(T, db, writes)
10       # T aborts if writes are not installed
11       if HasConflicts(T.TID, T.RS ∪ T.WS, writes) == false:
12           Install(T, db)
13
14 Function: HasConflicts(TID, keys, reservations)
15       for each key in keys:
16           if key in reservations and reservations[key] < TID:
17               return true
18       return false
19
20 Function: Install(T, db)
21       for each key in T.WS:
22           write(key, db)
```

Figure 4: Execution and commit protocols in Aria

We now use an example to show how transactions make write reservations.

EXAMPLE 1. *Consider the following three transactions:* T_1*:* $x = x + 1$*,* T_2*:* $y = x - y$*, and* T_3*:* $x = x + y$

The reservation table is initially empty. Transaction T_1 first makes reservations for its writes. The reservation succeeds and the table now has an entry $\{x : T_1\}$. Transaction T_2 then makes its reservation successfully by creating an entry $\{y : T_2\}$ in the reservation table. Finally, transaction T_3 tries to make a reservation for record x. Since record x has been reserved by transaction T_1, and T_3 has a larger TID, the reservation fails and T_3 must abort. Note that, even though we describe the reservation process in a sequential manner, the whole process can be conducted in parallel and in any order, and the same result will always be produced.

Once all transactions finish execution and reservations are made for writes, the system enters the commit phase.

4.2 The Commit Phase

Aria separates concurrency control from transaction execution. During the commit phase, each transaction is determined to commit or abort deterministically based on the write reservations made in the execution phase. Aria does not require two-phase commit to commit distributed transactions as in non-deterministic databases. The reasons are twofold: (1) the write reservations made in the execution phase do not affect the value of each record in the database, i.e., no rollback is required, and (2) a transaction can always commit (i.e., by applying writes to the database) even a failure occurs, since determinism guarantees that the same result is always produced after re-execution.

We now introduce three types of dependencies that Aria uses to determine whether a transaction can commit or not: (1) transaction T_i has a write-after-write (WAW) dependency on transaction T_j if T_i tries to update some record after T_j has updated it, (2) transaction T_i has a write-after-read (WAR) dependency on transaction T_j if T_i tries to update some record after T_j has read it, and (3) transaction T_i has a read-after-write (RAW) dependency on transaction T_j if T_i tries to read some record after T_j has updated it.

Aria decides if a transaction T_i can commit or not by checking if it has certain types of dependencies with any earlier transaction $T_j, \forall j < i$. There are two types of dependencies that the system must check: (1) WAW-dependency: if T_i WAW-depends on some T_j, it must abort. This is because the updates are installed independently and more than one update goes to the same record, and (2) RAW-dependency: if T_i RAW-depends on some T_j, it must abort as well, because it should have seen T_j's write but did not. In contrast, it's safe for a transaction to update some record that has been read by some earlier transaction. Thus, by default, Aria does not check WAR-dependencies, however, WAR-dependencies can help reduce dependency conflicts with deterministic reordering as will be noted in Section 5.

Note that by aborting transactions with WAW-dependencies or RAW-dependencies, Aria ensures that the serial order of committed transactions is the same as the input order.

RULE 1. *A transaction commits if it has no WAW-dependencies or RAW-dependencies on any earlier transaction.*

Aria does not have to go through all earlier transactions to check WAW-dependencies and RAW-dependencies. Instead, it uses a transaction's read set and write set to probe the write reservation table. Note that some transactions may have aborted in the execution phase and can simply skip the commit phase, e.g., a reservation on some write failed. As shown in Figure 4 (line 9 – 12), function `HasConflicts` returns false when none of the keys in a transaction's read set and write set exists in the write reservation table. When no dependencies exist, the transaction commits and all writes are applied to the database.

As in the execution phase, the process of checking dependencies can be done in parallel and in any order for every transaction. Aborted transactions are automatically scheduled at the beginning of the next batch for execution, unless the transaction is aborted explicitly (e.g., violating some integrity constraint). The relative ordering of aborted transactions is kept the same. As a result, every transaction can commit eventually, since the first transaction in a batch always commits and the position of an aborted transaction always moves forward across batches.

4.3 Determinism and Serializability

We now show how the system achieves determinism and serializability with batch execution.

THEOREM 1. *Aria following Rule 1 enforces determinism and serializability.*

Determinism: In the execution phase, the system builds a write reservation table given a batch of transactions. The reservation table is a collection of key-value pairs. A key x is a record modified by some transaction and the value T is the transaction that modifies record x with the smallest TID. Specifically, no matter what order transactions execute in, the collection of key-value pairs always equals to $\big\{\{x \in \mathcal{WS} : T \in \mathcal{C}(x)\} \mid \mathcal{ID}(T) \leq \mathcal{ID}(T'), \forall T' \in \mathcal{C}(x)\big\}$, where $\mathcal{WS}$ indicates the union of the write sets of all transactions and $\mathcal{C}(x)$ indicates a set of transactions that wrote record x. We use $\mathcal{ID}(T)$ to indicate transaction T's TID.

In the commit phase, the reservation table does not change and is used by the system to detect dependencies between transactions. Following the algorithm in Figure 4, if a transaction will commit or abort depends only on its read/write sets and the reservation table regardless of the order in which transactions run in the commit phase. Since each replica runs an identical batch of transactions on an identical database snapshot, different replicas will produce the identical read/write sets and the reservation table. Therefore, Aria achieves determinism.

Serializability: The following proof is by contradiction.

PROOF. (BY CONTRADICTION.) Assume the ordering of committed transactions is: $\cdots \to T_i \to \cdots \to T_j \to \ldots$, and the result produced by Aria is not serializable. Since transactions are committed by multiple worker threads in parallel, there are two possible outcomes: (1) transaction T_j's updates were overwritten by transaction T_i's updates, and (2) transaction T_j read transaction T_i's writes. Since WAW-dependencies do not exist, T_i's write set does not overlap with T_j's, so concurrent transactions must have updated different records. Similarly, since RAW-dependencies do not exist, the values of records that transaction T_j read must be the same as in the original database snapshot. By Rule 1, both outcomes lead to a contradiction. □

Therefore, Aria without deterministic reordering commits transactions under conflict serializability, and the result of the database is equivalent to running all committed transactions serially following the input order. Note that the properties enforced by Aria are sufficient but not necessary for serializability.

4.4 The Phantom Problem

The phantom problem [17] occurs within a transaction when the same query runs more than once but produces different sets of rows. For example, concurrent updates to secondary indexes can make the same scan query observe different results if it runs twice. Serializability does not allow phantom reads. In Aria, secondary indexes are implemented as regular tables that map secondary keys to a set of primary keys. When an insert or a delete affects a secondary index, a reservation must be made in the execution phase. All transactions that access the secondary index are guaranteed to observe the updates to the index by checking RAW-dependencies in the commit phase. Therefore, these reservations will cause aborts in the usual way, and and phantoms cannot occur as a result of concurrent reads and inserts/deletes. Phantoms are also possible in range queries (e.g., a table scan). However, standard techniques such as multi-granularity locking [22] can be applied.

4.5 Limitations

In Aria, a previous batch of transactions must finish executing before a new batch can begin, since a barrier exists across batches. The system could have undesirable performance due to imbalanced workloads among transactions.

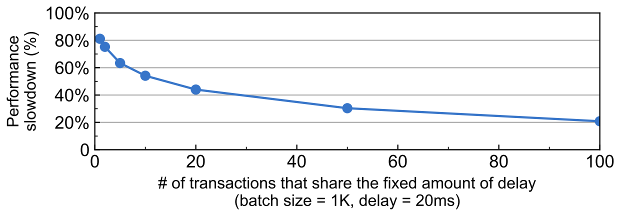


Figure 5: Effect of barriers on performance

Note that because we use round-robin scheduling and execute many transactions in a batch, unless there are very long running transactions, skew in transaction runtimes does not significantly affect the overall runtime of a transaction.

We ran a YCSB workload (see Section 8 for more details) to study the effect of imbalanced workloads on our current implementation. In this experiment, we uniformly distribute a fixed total amount of 20 ms delay across multiple transactions in a batch. For example, each transaction has a delay of 2 ms if the total amount of delay is distributed across ten transactions. In Figure 5, we plot the system's performance slowdown when the delay is distributed across different numbers of transactions. The maximum throughput is measured when all transactions share the fixed amount of delay (i.e., no stragglers or every transaction is a "straggler"). The results with different fixed total amount of delay follow the same trend and are not shown for clarity. In the extreme scenario when only a single transaction has a 20 ms delay (i.e., a single straggler), the slowdown is about 81%. As the delay is distributed across more transactions (i.e., more stragglers but each one has shorter delay), the slowdown goes down. For example, when the delay is distributed across more than 100 transactions, the slowdown is less than 20%.

5. DETERMINISTIC REORDERING

The ordering of input transactions affects which transactions can commit within a batch. In this section, we first introduce a motivating example, and then present the deterministic reordering algorithm, which reduces aborts by transforming RAW-dependencies to WAR-dependencies. Finally, we prove its correctness and give a theoretical analysis of its effectiveness. In Section 6, we will discuss a fallback strategy that Aria adopts when a workload suffers from a high abort rate due to WAW-dependencies.

5.1 A Motivating Example

We use an example to show how the ordering of transactions changes which and how many transactions can commit.

Example 2. *Consider the following three transactions: T_1: $y = x$, T_2: $z = y$, and T_3: Print $y + z$*

As shown on the top of Figure 6, transaction T_2 has a RAW-dependency on transaction T_1, since transaction T_2 reads record y after transaction T_1's write. Following Rule 1, only transaction T_1 can commit, since both transaction T_2 and T_3 have RAW-dependencies. However, if we commit all three transactions, the result is still serializable with an equivalent serial order: $T_3 \rightarrow T_2 \rightarrow T_1$.

The key insight is that by reordering some transactions, RAW-dependencies can be transformed to WAR-dependencies, allowing more transactions to commit. This is because our system does not require aborts as a result of WAR-dependencies.

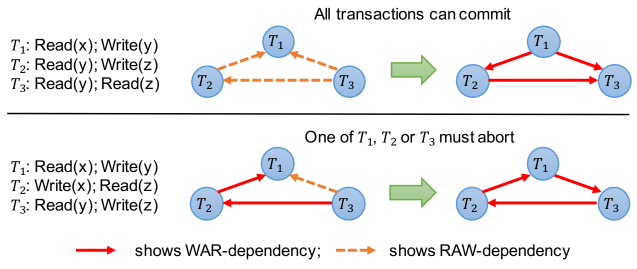


Figure 6: Illustrating deterministic reordering

5.2 The Reordering Algorithm

By slightly modifying Rule 1, the reordering algorithm can commit more transactions within a batch.

Rule 2. *A transaction commits if two conditions are met: (1) it has no WAW-dependencies on any earlier transaction, and (2) it does not have both WAR-dependencies and RAW-dependencies on earlier transactions with smaller TIDs.*

The pseudocode for this algorithm is given in Figure 7. Lines 5–10 encode the checking of Rule 2. This code replaces the Commit function in Figure 4. Like the algorithm presented in the previous section, this code can be run in parallel, allowing each transaction to commit or abort based on its read/write set and the reservation tables alone. Because this algorithm allows transactions with RAW-dependencies to commit as long as they have no WAR-dependencies, it can result in a serial order that is different from the input order. For example in Example 2 (shown on the top of Figure 6), transaction T_2 and T_3 have RAW-dependencies, but there are no WAR-dependencies, and all three transactions can commit with an equivalent order $T_3 \rightarrow T_2 \rightarrow T_1$.

WAW and RAW-dependencies are detected using the write reservation table as described in Section 4. To support WAR-dependency detection, the system also needs to make read reservations in the execution phase as shown from line 1 to line 3 in Figure 7. With the read reservation table, for each record in the database, the system knows which transaction first read it – i.e., the one with the smallest TID. In the commit phase, a WAR-dependency will be detected if any key in a transaction's write set appears in the read reservation table and the read is from a different transaction.

5.3 Proof of Correctness

We now show the correctness of our reordering algorithm.

Theorem 2. *Aria following Rule 2 enforces determinism and serializability.*

We next present two definitions for dependency graphs before presenting the proof of Theorem 2.

Definition 1. *Dependency graph: a directed graph whose vertices are transactions, with edges between transactions with RAW-dependencies and WAR-dependencies, as shown in the left side of Figure 6.*

For ease of presentation, we do not consider transactions that have WAW-dependencies on earlier transactions. They are always aborted in Aria and will be re-executed in the next batch. An edge from transaction T_j to T_i indicates that T_j performed the conflicting operation after T_i. This is

```
Function: ReserveRead(T, reads)
    for each key in T.RS:
        reads[key] = min(reads[key], T.TID)

Function: Commit(T, db, reads, writes)
    # T aborts if writes are not installed
    if WAW(T, writes):
        return
    if WAR(T, reads) == false or RAW(T, writes) == false:
        Install(T, db)

Function: WAW(T, writes) = HasConflicts(T.TID, T.WS, writes)
Function: WAR(T, reads) = HasConflicts(T.TID, T.WS, reads)
Function: RAW(T, writes) = HasConflicts(T.TID, T.RS, writes)
```

Figure 7: Pseudocode for deterministic reordering

because conflicts are initially resolved in the order of input transactions for all edges in the graph.

As shown in the right side of Figure 6, RAW-dependencies can be transformed to WAR-dependencies by reversing the direction of edges in the transformed dependency graph. Reversing a RAW edge from transaction T_j to T_i effectively moves T_j before T_i in the serial order, and thus creates a WAR-dependency from T_i to T_j. Based on this, we define the transformed dependency graph.

Definition 2. *Transformed dependency graph: a dependency graph with all the RAW-dependencies transformed to WAR-dependencies.*

However, not all transactions can commit even with reordering, since a topological ordering may not exist if there are cycles in the transformed dependency graph.

Example 3. *Consider the following three transactions: T_1: $y = x$, T_2: $x = z$, and T_3: $z = y$*

As shown on the bottom of Figure 6, the transformed dependency graph is not cycle free, and one of T_1, T_2 or T_3 must abort to achieve serializability. To commit as many transactions as possible, we have to remove as few vertices (i.e., abort as few transactions) as possible from the transformed dependency graph to make it acyclic. This problem is well-known as the *feedback vertex set* problem[1] and is shown NP-Complete [26].

Next, we show Aria effectively makes the transformed dependency graph acyclic by applying Rule 2 with Lemma 1, which will be used to prove Theorem 2.

Lemma 1. *The transformed dependency graph is acyclic after applying Rule 2.*

Proof. (By contradiction.) Suppose the transformed dependency graph is not acyclic. Hence, there must be a sequence of transactions that forms a cycle, i.e., $T_i \rightarrow T_j \rightarrow \cdots \rightarrow T_k \rightarrow T_i$. Without loss of generality, we assume $i > j$ and $i > k$. In the transformed dependency graph, there is a WAR-dependency from transaction T_k to T_i. Since $i > k$, this WAR-dependency must be a RAW-dependency from transaction T_i to T_k in the original dependency graph. This is because dependencies only exist from transactions with larger TIDs to those with smaller TIDs. Similarly, the WAR-dependency from transaction T_i to T_j must be a WAR-dependency in the original dependency graph. Hence, transaction T_i has both WAR-dependency to T_j and RAW-dependency to T_k simultaneously. According to Rule 2, transaction T_i does not exist, which is a contradiction. ☐

[1]A feedback vertex set of a graph is a set of vertices whose removal leaves a graph without cycles.

We now give the proof to Theorem 2 by showing an acyclic transformed dependency graph indicates that the transactions can commit with a serializable schedule.

Proof. If no cycles exist in the transformed dependency graph, there exists a topological ordering $\mathcal{O}$ such that for every WAR-dependency from transaction T_i to T_j, we have $\mathcal{O}(T_i) < \mathcal{O}(T_j)$. In addition, for any transaction T_i, its read set does not overlap with any earlier transaction's write set. This is because only WAR-dependencies exist in the transformed dependency graph. Hence, the serializable schedule is the same as the topological ordering $\mathcal{O}$ and the effect of executing these transactions in parallel is identical to that running them serially following the topological ordering $\mathcal{O}$. ☐

For example, the serializable schedule of the transformed dependency graph on the top of Figure 6 is $T_1 \rightarrow T_2 \rightarrow T_3$, which is the same as the topological ordering. In contrast, there is no equivalent serializable schedule to the transformed dependency graph on the bottom of Figure 6, since it is not acyclic.

5.4 Effectiveness Analysis

Suppose we have a table with $\mathcal{T}$ records. Let $\mathcal{R}$ and $\mathcal{W}$ be the number of read and write operations in a transaction and assume each data access (i.e., a read or a write operation) is independent and follows a uniform distribution.

We first analyze the probability that an operation does not access to a record that has been updated by earlier transactions. If the operation is from transaction T_i (numbered from 0), we have

$$\Pr(\text{an operation has no conflicts}) = (1 - \frac{\mathcal{W}}{\mathcal{T}})^i$$

and therefore,

$$\Pr(\neg \text{ WAW-dependencies exist}) = (1 - \frac{\mathcal{W}}{\mathcal{T}})^{i\mathcal{W}}$$

$$\Pr(\neg \text{ RAW-dependencies exist}) = (1 - \frac{\mathcal{W}}{\mathcal{T}})^{i\mathcal{R}}$$

likewise, we have

$$\Pr(\neg \text{ WAR-dependencies exist}) = (1 - \frac{\mathcal{R}}{\mathcal{T}})^{i\mathcal{W}}$$

since WAR-dependencies depend on the number of read operations from earlier transactions.

According to Rule 1, transaction T_i can commit if it does not have any WAW-dependencies or RAW-dependencies:

$$\begin{aligned}\Pr(T_i \text{ commits}) &= \Pr(\neg \text{ WAW-dependencies exist}) * \\ &\quad \Pr(\neg \text{ RAW-dependencies exist})\end{aligned}$$

The commit rule can be relaxed by Rule 2. With deterministic reordering, we have

$$\begin{aligned}&\Pr(T_i \text{ commits with deterministic reordering}) \\ = \; &\Pr(\neg \text{ WAW-dependencies exist}) * \big\{1 \\ &-\big[1 - \Pr(\neg \text{ WAR-dependencies exist})\big] \\ &*\big[1 - \Pr(\neg \text{ RAW-dependencies exist})\big]\big\}\end{aligned}$$

Our deterministic reordering algorithm allows each transaction to execute concurrently without coordination, even though a serial but more expensive reordering algorithm could potentially produce a better schedule to reduce more aborts. Note that aborting more transactions than are strictly needed to improve performance is a tried and true technique: i.e., using a coarser granularity of locking creates false sharing, resulting in unnecessary aborts, but can improve performance as fewer locks need to be tracked.

6. THE FALLBACK STRATEGY

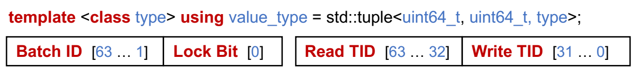
Because Aria does not need to determine read/write sets up front, and can run in parallel on multiple threads within a replica, it provides superior performance to existing deterministic databases when there are few RAW-dependencies and WAW-dependencies between transactions in a batch. The deterministic reordering mechanism can transform RAW-dependencies to WAR-dependencies, allowing many of conflicting transactions to commit under serializability.

In this section, we discuss a fallback strategy that Aria adopts when a workload suffers from a high abort rate due to WAW-dependencies, e.g., two transactions update the same record. The key idea is that we add a new fallback phase at the end of the commit phase and re-run aborted transactions following the dependencies between conflicting transactions. As discussed in Section 3, each transaction reads from the current snapshot of the database in the execution phase and then decides if it can commit in the commit phase. The observation here is that each transaction knows its complete read/write set when the commit phase finishes. The read/write set of each transaction can be used to analyze the dependencies between conflicting transactions. Therefore, many conflicting transactions do not need to be rescheduled in the next batch. Instead, Aria can employ the same mechanism as in existing deterministic databases to re-run those conflicting transactions.

In our current implementation, we use an approach similar to Calvin, which naturally supports distributed transactions, although any deterministic concurrency control is potentially feasible. In the fallback phase, some worker threads work as lock managers to grant read/write locks to each transaction following the TID order. A transaction is assigned to one of the available worker threads, and executes against the current database (i.e., with changes made by transactions with smaller TIDs) once it obtains all locks. After the transaction commits, it releases all locks. As long as the read/write set of the transaction does not change, the transaction can still commit deterministically. Otherwise, it will be scheduled and executed in the next batch in a deterministic fashion.

The original design of Calvin uses only one thread to grant locks in the pre-determined order. However, on a modern multicore node, this single-threaded lock manager cannot saturate all available workers [44]. To better utilize more CPU resources, our implementation uses multiple lock manager threads to grant locks. Each lock manager is responsible for a different portion of the database. Our new design is logically equivalent to but more efficient than the original design [52], which ran multiple Calvin instances on a single node. This is because worker threads now communicate with each other through shared memory rather than sockets.

A moving average of the abort rate in a workload is monitored by the sequencing layer. When the abort rate exceeds the threshold, the sequencing layer deterministically switches the system to using the fallback, and then turns off the fallback when the abort rate drops. Note that the fallback strategy guarantees that Aria is at least as effective as existing deterministic databases in the worst case, since non-conflicting transactions only run once. However, existing deterministic databases run input transactions at least twice: once to determine the read/write set, and once to execute the query.



```
template <class type> using value_type = std::tuple<uint64_t, uint64_t, type>;
```

Figure 8: Per-record metadata format in Aria

7. IMPLEMENTATION

We now describe the data structures used to implement Aria and how the system supports distributed transactions and provides fault tolerance across multiple replicas.

7.1 Data Structures

Aria is a distributed deterministic OLTP database. It provides a relational model, where each table has a pre-defined schema with typed and named attributes. Clients interact with the system by calling pre-compiled *stored procedures* with different parameters, as in many popular systems [52, 53, 59]. A stored procedure is implemented in C++, in which arbitrary logic (e.g., read/write operations) is supported. The system targets one-shot and short-lived transactions and does not support multi-round and interactive transactions. It does not provide a SQL interface.

Each table has a primary key and some number of attributes. Tables are currently implemented as a primary hash table and zero or more secondary hash tables as indexes, meaning that range queries are not supported. This could be easily adapted to tree structures [6, 36, 57]. A record is accessed via probing the primary hash table. For secondary lookups, two probes are needed, one in a secondary hash table to find the primary key of a record, followed by a lookup in the primary hash table.

As discussed in previous sections, a transaction makes reservations for reads and writes in the execution phase, which are used for conflict detection in the commit phase. Essentially, these reservations are collections of key-value pairs. The key is the primary key of the table that a transaction accesses and the value is the transaction's TID. Since these key-value pairs are table specific, we record the TID value along with the batch ID with two 64-bit integers and attach them to each record in the table's primary hash table, as shown on the top of Figure 8.

7.2 Distributed Transactions

Aria supports distributed transactions through partitioning. Each table is horizontally partitioned across all nodes and each partition is randomly hashed to one node. Note that the sequencing layer is allowed to send transactions to any node for execution. However, if a transaction was sent to the node having most data it accesses, network communication could be reduced. In real scenarios, the partitions that a transaction accesses can be inferred from the parameters of a stored procedure in some cases (e.g., the `New-Order` transaction on TPC-C). Otherwise, the transaction can be send to any node in Aria for execution.

As discussed earlier, the execution phase and the commit phase are separated with barriers. In the distributed setting, these barriers are implemented through network round trips. For example, a designated coordinator (selected from among all nodes) decides that the system can enter the commit phase once all nodes finish the execution phase.

In the execution phase, a remote read request is sent, when a transaction reads a record that does not exist on the local node. Similarly, read/write reservation requests are also sent to the corresponding nodes. In the commit phase, network communication is also needed to detect conflicts among transactions. Note that worker threads within the same node do not communicate by exchanging messages. Instead, all operations are conducted directly in main memory. Aria batches these requests to reduce latency and improve the network's performance. Once a transaction commits, write requests are also batched and sent to remote nodes to commit changes to the database.

7.3 Fault Tolerance

Fault tolerance is significantly simplified in a deterministic database. This is because there is no need to maintain physical undo/redo logging, instead, the system only needs to make input transactions durable in the sequencing layer. In addition, replicas do not communicate with one another, meaning global barriers do not exist across replicas. We now discuss when the result of a transaction can be released to the client and how checkpointing works.

Every transaction first goes to the sequencing layer, in which it is assigned a batch ID and a transaction ID. The sequencing layer next forwards the same input transactions to all replicas. We next discuss how the sequencing layer communicates with one replica, since the mechanism is the same for all replicas. Before the execution phase begins, each node within a replica will receive a different portion of a batch of transactions. Once all transactions finish execution, the replica communicates with the sequencing layer again, notifying it the result of each transaction. If a transaction aborts due to conflicts, the sequencing layer will put the transaction into the next batch.

The result of a committed transaction can be released to the client as soon as it commits in the commit phase. This is because its input was durable before execution. Once a failure occurs, every transaction will run again and commit in the same order due to the nature of deterministic execution.

To bound the recovery time, the system can be configured to periodically checkpoint the database to disk. During a checkpoint, a replica stops processing the transactions and saves a consistent snapshot of the database to disk. As long as there is more than one replica, clients are not aware of the pause when checkpoints occur, since each replica can checkpoint independently and transactions are always run by the system through other available replicas. On recovery, a replica loads the latest checkpoint from disk to the database and replays all transactions since the checkpoint.

8. EVALUATION

In this section, we evaluate the performance of Aria focusing on the following key questions:

- How does Aria perform compared to conventional deterministic databases and primary-backup databases?
- What is the scheduling overhead of Aria?
- How effective is the deterministic reordering?
- How does Aria scale?

8.1 Experimental Setup

We ran our experiments on a cluster of `m5.4xlarge` nodes running on Amazon EC2 [2]. Each node has 16 2.50 GHz virtual CPUs and 64 GB RAM running 64-bit Ubuntu 18.04 with Linux kernel 4.15.0 and GCC 7.3.0. The nodes are connected with a 10 GigE network.

To avoid an apples-to-oranges comparison, we implemented the following concurrency control protocols in C++ in the same framework: (1) **Aria:** our algorithm with deterministic reordering. The fallback strategy is disabled, (2) **AriaFB:** different from Aria, AriaFB always uses the fallback strategy to re-run aborted transactions, (3) **BOHM:** a single-node MVCC deterministic database using dependency graphs to enforce determinism [19], (4) **PWV**: a single-node deterministic database that enables early write visibility [18], (5) **Calvin:** a distributed deterministic database using ordered locks to enforce determinism [52], and (6) **PB:** a conventional non-deterministic primary-backup replicated database.

Our implementation of Calvin has the same optimization as in the fallback phase of Aria and we use Calvin-x to denote the number of lock manager threads. The concurrency control protocol used in PB is strict two-phase locking (S2PL). We use a NO_WAIT deadlock prevention strategy, which previous work has shown to be the most scalable protocol [24]. Transactions involving multiple nodes are committed with two-phase commit [37]. PB uses synchronous replication to prevent loss of durability in high-throughput environments. In synchronous replication, locks on the primary are not released until the writes are replicated on the backup. A transaction commits only after the writes are acknowledged by backups.

We use two popular benchmarks, YCSB [11] and a subset of TPC-C [1], to study the performance of Aria. These benchmarks were selected because they are all key-value benchmarks where it is possible for existing deterministic databases to pre-compute read/write sets. Many other workloads cannot easily run on them due to the need for this pre-computation. The Yahoo! Cloud Serving Benchmark (YCSB) is designed to evaluate the performance of key-value systems. There is a single table with ten columns. The primary key of the table is a 64-bit integer and each column has ten random bytes. To make it a transactional benchmark, we wrap operations within transactions [56, 61]. By default, each transaction has 10 operations. The TPC-C benchmark models an order processing application, in which customers place orders in one of ten districts within a local warehouse. We support the `New-Order` and the `Payment` transaction in this benchmark. 88% of the standard mix consists of these two transactions. The other three transactions require range scans, which are currently not supported. By default, there are 10% `New-Order` and 15% `Payment` transactions that access multiple warehouses. On average, one `NewOrder` transaction is followed by one `Payment` transaction.

We run 12 worker threads and 2 threads for network communication on each node. Note that, in Calvin-x, there are x worker threads granting locks and 12-x worker threads running transactions. By default, we set the number of partitions equal to the total number of worker threads in the cluster, meaning there are 12 partitions on each node. This is because existing deterministic databases [18, 19, 52] require the use of partitionings to use multiple worker threads for better performance. In YCSB, the number of keys per

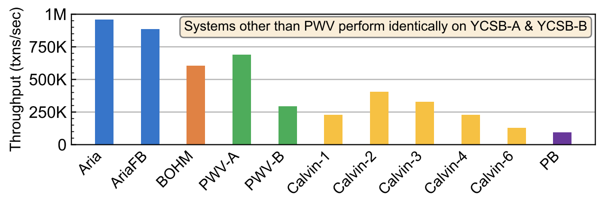


Figure 9: Performance on YCSB-A and YCSB-B

partition is 40K. In TPC-C, the database is partitioned by warehouse and the number of partitions is equal to the number of warehouses.

There is no sequencing layer in our experiments. This is because the sequencing layer is the same in all deterministic databases, and adding the sequencing layer or not does not affect our experimental conclusions. To measure the maximum performance that each system is able to achieve, we use a built-in deterministic workload generator to get an ordered batch of transactions on each node. Transactions are run under serializability. All reported results are the average of ten runs.

We only measure the performance of Aria, AriaFB, BOHM, PWV, and Calvin at one replica, since all replicas achieve comparable performance. We consider two variations in our experiments: (1) a single-node setting, used in from Section 8.2 to Section 8.5, and (2) a multi-node setting (i.e., a cluster of eight nodes), used in Section 8.6 and 8.7. In all deterministic databases, we set the batch size to 1K on YCSB and 500 on TPC-C in the single-node setting, and 80K on YCSB and and 4K on TPC-C in the multi-node settings. These parameters ensure the database to achieve the best performance. For PB, a dedicated backup node stands by for replication in the single-node setting. In the multi-node setting, each partition has a primary partition and a backup partition, which are always assigned to two different nodes.

8.2 YCSB Results

We first study the performance of each system on the YCSB benchmark. We run a workload mix of 80/20, meaning each access in a transaction has an 80% probability of being a read operation and a 20% probability of being an update operation. All accesses follow a uniform distribution, which makes most transactions multi-partition transactions.

We use two variants of YCSB: (1) YCSB-A: each read or write is independent, and (2) YCSB-B: the value of a write depends on all reads. Note that only PWV is sensitive to the two variations above, which affect when the writes of a transaction are visible to other transactions. In contrast, conflict detection or dependency analysis stays the same for Aria, AriaFB, BOHM and Calvin. Likewise, PB always acquires all read/write locks before execution and only releases them when a transaction commits. In this experiment, we use PWV-A and PWV-B to denote the performance of PWV on the two variations above respectively. For other systems, the results on YCSB-A and YCSB-B are the same and we only report the result on YCSB-A.

We report the results in Figure 9. AriaFB has a slightly lower throughput than Aria, since the abort rate in this workload is about 0.4% and the overhead of the fallback strategy exceeds its benefits. PWV on YCSB-A performs the best among all baselines, since each read/write is independent and it allows early write visibility. In contrast, PWV has much lower throughput on YCSB-B, since a write is only visible to other transactions when all reads are finished. BOHM has the second highest throughput, but the overhead of managing multiple versions of each record makes it achieve only two thirds of Aria's throughput. The performance of Calvin depends on the number of threads used for granting locks. One lock manager fails to saturate the system. Calvin achieves the best performance when two lock managers are used in this workload. PB's throughput is about 11% of Aria's throughput. This is because one network round trip is needed in PB for each transaction with write operations due to synchronous replication. In our test environment, the round trip time is about 100 μs.

Overall, Aria achieves 1.1x, 1.6x, 1.4x, 3.3x, 2.4x and 10.3x higher throughput than AriaFB, BOHM, PWV-A, PWV-B, Calvin and PB on YCSB, respectively.

8.3 Scheduling Overhead

We now study the scheduling overhead of each concurrency control algorithm. In this experiment, we use the same YCSB workload as in Section 8.2, but all transactions access only one partition. This is because H-Store [49] has minimal overhead in concurrency control when transactions do not span partitions.

We first compare Aria with our implementation of H-Store in the same framework, which can be used to estimate the upper bound of the transaction throughput that a workload can achieve. We report the result in Figure 10(a) and observe that Aria achieves about 75% of H-Store's throughput.

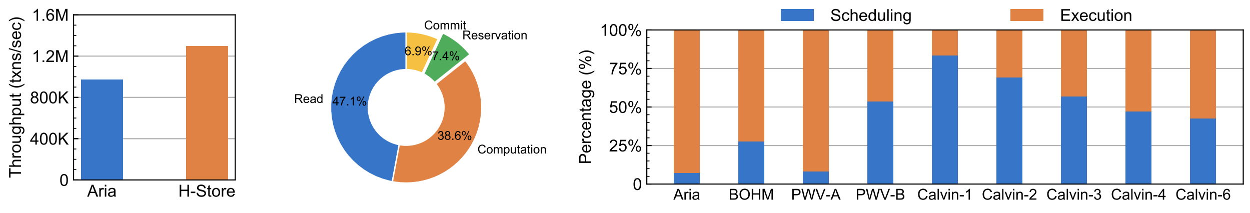
Second, we show a performance breakdown on Aria in Figure 10(b). There are three steps in the execution phase, namely, (1) reading the database snapshot, (2) computing the transaction logic, and (3) making reservations. Note that Aria uses read/write reservations to achieve determinism and serializability. Therefore, we consider the third step to be the scheduling overhead, which is about 7.4%.

Third, we show the scheduling overhead of each deterministic concurrency control in Figure 10(c). Existing deterministic databases run transactions following a dependency graph, meaning there exists a scheduling overhead in each algorithm (e.g., the waiting time wasted in each worker thread). In addition, we consider the time spent on each lock manager in Calvin and the time spent on dependency graph construction in BOHM and PWV to be scheduling overhead. The time spent on placeholder insertion and garbage collection in BOHM is also considered to be scheduling overhead. Among all algorithms, Aria and PWV-A have the lowest scheduling overhead, which partially explains why they have the best performance in Figure 9.

8.4 Effectiveness of Deterministic Reordering

In this section, we study the effectiveness of the deterministic reordering technique introduced in Section 5. There are many factors that influence the effectiveness of this technique, such as the percentage of read-only transactions, access distribution, the database size, and the read/write ratio. In this experiment, we ran the same workload as in Section 8.2, but varied the skew factor [23] from 0 (i.e. uniform distribution) to 0.999 in the workload. Aria w/o D.R. shows the performance of Aria without deterministic reordering. For clarity, in this experiment and all following experiments, we only report the performance of Calvin with the optimal number of lock managers.

We first study the effect of the skew factor on Aria. As shown in Figure 11, the throughput goes down when the we



(a) Aria vs. H-Store (b) Performance breakdown (c) Scheduling vs. Execution

Figure 10: A study of scheduling overhead on each system on YCSB

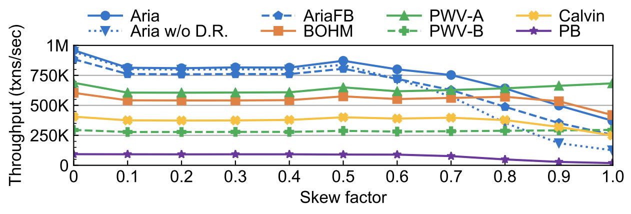


Figure 11: Effectiveness of deterministic reordering

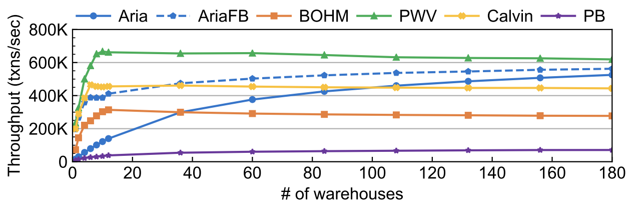


Figure 12: Performance on TPC-C

increase the skew factor. For example, when the skew factor is 0.999, Aria achieves only 39% of its original throughput when there is no skew. This is because a larger skew factor leads to a higher abort rate due to conflicts and many transactions have to be re-run multiple times. Similarly, a transaction has more difficulty getting read/write locks in PB as the skew factor increases. For this reason, the throughput of PB is only about 20% of the original throughput when there is no skew. BOHM and Calvin also have lower throughput when the skew factor is larger due to less parallelism. However, PWV is less sensitive to the skew factor due to its early write visibility mechanism. The throughput of Aria is lower than PWV-A and BOHM when the skew factor is larger than 0.8 and 0.9 respectively.

We also report the result of AriaFB in this experiment. When the skew factor is small, AriaFB achieves slightly lower throughput than Aria, since it's more efficient to simply retry the aborts in the next batch than re-running the aborts with the fallback strategy. As we increase the skew factor, the throughput of AriaFB exceeds Aria w/o D.R., but it is still lower than Aria, showing the effectiveness of our reordering technique. Under high skew, AriaFB achieves similar throughput to Calvin, since most transactions are executed using the fallback strategy.

We now study how much performance gain that Aria can achieve with deterministic reordering. When each access follows a uniform distribution, deterministic reordering achieves similar throughput to the one without deterministic reordering. This is because there are few conflicts and the number of aborts is low. Note that the deterministic reordering mechanism does not slow down the system even though when the contention is low. As we increase the skew factor in the workload, more conflicts exist and Aria with deterministic reordering achieves significantly higher throughput. For example, when the skew factor is 0.999, Aria achieves 3.0x higher throughput than Aria w/o D.R. .

Overall, the deterministic reordering technique is effective to reduce aborts due to RAW-dependencies, making Aria achieve higher throughput than AriaFB on YCSB.

8.5 TPC-C Results

We next study the performance of each system on TPC-C. In this experiment, we varied the number of partitions (i.e., the number of warehouses) from 1 to 180. As shown in Figure 12, all other deterministic databases have stable performance when the number of partitions exceeds 12, since the maximum parallelism is achieved when each thread has one partition on TPC-C. The throughput of Aria goes up as more partitions exist due to fewer conflicts. When the number of partitions is 180, Aria achieves 85% of the throughput of PWV.

Note that there are contended writes in TPC-C, i.e., writes made on the `d_ytd` field in the district table, making Aria perform worse when there are a small number of warehouses. However, AriaFB can commit non-conflicting transactions in the normal commit phase and re-run conflicting transactions with the fallback strategy, making it achieve the best of both worlds. Therefore, the throughput of AriaFB always exceeds both Aria and Calvin on TPC-C.

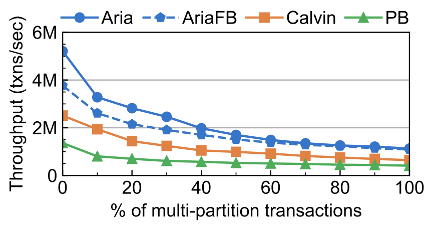
In summary, with 180 partitions, Aria achieves 1.9x, 1.2x and 7.4x higher throughput than BOHM, Calvin and PB respectively and has comparable throughput to PWV, which enables early write visibility but requires complex transaction decomposition.

8.6 Results on Distributed Transactions

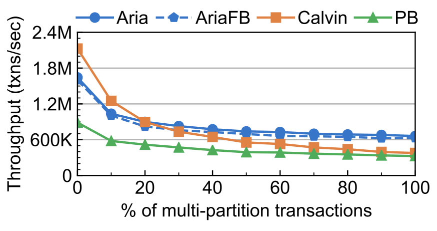
In this section, we study the performance of Aria, AriaFB, Calvin, and PB in a multi-node setting using 8 Amazon EC2 [2] instances. We omit the results of BOHM and PWV, which are designed to be single-node deterministic databases.

We first ran a YCSB workload with a varying percentage of multi-partition transactions and report the results in Figure 13(a). In this experiment, a multi-partition transaction only accesses two partitions. Different from the single-node setting, the throughput of all approaches goes down with more multi-partition transactions. This is because more network communication occurs. In addition, AriaFB has lower throughput than Aria, since there are two more barriers per batch when the fallback strategy is employed. In summary, Aria achieves up to 1.4x higher throughput than AriaFB, 1.6x - 2.1x higher throughput than Calvin and 2.7x - 4.1x higher throughput than PB on YCSB.

We also ran a workload of TPC-C and report the result in Figure 13(b). We set the number of partitions (i.e., the number of warehouses) to 108 per node, meaning there are 864



(a) YCSB - batch size: 10K * 8 = 80K



(b) TPC-C - batch size: 500 * 8 = 4K

Figure 13: Performance on YCSB and TPC-C in the multi-node setting

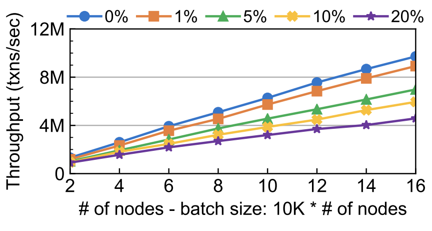


Figure 14: Scalability of Aria

partitions in total. Aria achieves higher throughput than Calvin when more multi-partition transactions exist. This is because more work needs to be done in multi-partition transactions in Calvin as discussed in Section 2. In addition, Aria and AriaFB achieve similar throughput in this workload, since the performance gain from the fallback strategy is about the same as the overhead of two additional barriers. Overall, Aria achieves up to 1.7x higher throughput than Calvin and up to 2x higher throughput than PB on TPC-C.

8.7 Scalability Experiment

Finally, we study the scalability of Aria. We ran the same YCSB workload as in Section 8.2, but we scale the number of partitions from 24 to 192. The batch size is set to 10K times by the number of nodes. We report the results on five different configurations in Figure 14, in which x% denotes the percentage of multi-partition transactions. Again, a multi-partition transaction accesses only two partitions. We can observe that Aria achieves almost linear scalability in all configurations. For example, it achieves around 6.3x higher throughput with 16 nodes compared to the one with 2 nodes when 10% multi-partition transactions exist.

9. RELATED WORK

Aria builds on many pieces of related work for its design, including transaction processing, highly available systems, and deterministic databases.

Transaction Processing. There has been a resurgent interest in transaction processing, as both scale-up multicore databases [15, 28, 32, 41, 53, 60, 61] and scale-out distributed databases [13, 24, 31, 34, 38, 39, 49] are becoming more popular. Silo [53] avoids shared-memory write by dividing time into short epochs and each thread generates its own timestamp by embedding the global epoch. Aria uses a similar batch-based execution model to run transactions. ROCOCO [38] tracks conflicting constituent pieces of transactions and reorders them in a serializable order before execution. Aria deterministically detects conflicts in a separate commit phase after all transactions finish execution. Doppel [41] allows different orderings of updates as long as updates are commutative. Aria can be extended to support commutative updates as well to reduce conflicts between transactions. More recent database systems [9, 16, 58, 59] build on RDMA and HTM. Aria can use RDMA to further decrease the overhead of network communication and barriers across batches.

Highly Available Systems. Modern database systems support high availability, such that an end user does not experience any noticeable downtime when a subset of servers fail. High availability is usually achieved through replication [8, 21, 48, 54], which itself can be either asynchronous [10, 14] or synchronous [27]. Asynchronous replication reduces latency during replication by sacrificing consistency when failures occur. For this reason, synchronous replication which enables strong consistency is becoming more popular [12, 35].

Many recent protocols [43, 50, 55] are developed with high availability in mind. TAPIR [63] builds consistent transactions through inconsistent replication. SiloR [64] uses parallel recovery via value replication, in which logs can be replayed in any order. Most existing work above achieves high availability through shipping the output. Instead, Aria achieves high availability through replicating the input and running transactions deterministically across replicas.

Deterministic Databases. The initial design of deterministic database systems dates back to the late 1990s [25, 40]. Each replica achieves consistency through a deterministic scheduler that produces exactly the same thread interleaving [25]. As more systems favor full ACID properties, deterministic systems have attracted a great deal of attention recently [3, 20, 45, 46, 47, 51]. Existing deterministic databases [18, 19, 52] require that the read set and write set of a transaction must be known a priori. Hyder [5], FuzzyLog [33] and Tango [4], unlike conventional deterministic databases [18, 19, 52], allow each node to read the same database via a shared log, which establishes a global order across all updates. However, the result could be different if multiple shared-log systems are deployed independently. Instead, conventional deterministic databases always produce the same result given the same input transactions.

10. CONCLUSION

In this paper, we presented Aria, a new distributed and deterministic OLTP database. By employing deterministic execution, Aria is able to efficiently run transactions across different replicas without coordination, and without prior knowledge of read and write sets as in existing work. The system executes a batch of transactions against the same database snapshot in an execution phase, and then chooses deterministically the ones that should commit to ensure serializability in a commit phase. We also propose a novel deterministic reordering mechanism so that Aria commits transactions in an order that reduces the number of conflicts. Our results on two popular benchmarks show that Aria outperforms systems with conventional nondeterministic concurrency control algorithms and state-of-the-art deterministic databases by a large margin on a single node and up to a factor of two on a cluster of eight nodes.

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

11. REFERENCES

[1] TPC Benchmark C. http://www.tpc.org/tpcc/, 2010.

[2] Amazon EC2. https://aws.amazon.com/ec2/, 2020.

[3] D. J. Abadi and J. M. Faleiro. An overview of deterministic database systems. *Commun. ACM*, 61(9):78–88, 2018.

[4] M. Balakrishnan, D. Malkhi, T. Wobber, M. Wu, V. Prabhakaran, M. Wei, J. D. Davis, S. Rao, T. Zou, and A. Zuck. Tango: distributed data structures over a shared log. In *SOSP*, pages 325–340, 2013.

[5] P. A. Bernstein, C. W. Reid, and S. Das. Hyder - A transactional record manager for shared flash. In *CIDR*, pages 9–20, 2011.

[6] R. Binna, E. Zangerle, M. Pichl, G. Specht, and V. Leis. HOT: A height optimized trie index for main-memory database systems. In *SIGMOD Conference*, pages 521–534, 2018.

[7] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault-tolerance. In *SOSP*, pages 1–11, 1995.

[8] E. Cecchet, G. Candea, and A. Ailamaki. Middleware-based database replication: The gaps between theory and practice. In *SIGMOD Conference*, pages 739–752, 2008.

[9] Y. Chen, X. Wei, J. Shi, R. Chen, and H. Chen. Fast and general distributed transactions using RDMA and HTM. In *EuroSys*, pages 26:1–26:17, 2016.

[10] B. F. Cooper, R. Ramakrishnan, U. Srivastava, A. Silberstein, P. Bohannon, H. Jacobsen, N. Puz, D. Weaver, and R. Yerneni. PNUTS: Yahoo!'s hosted data serving platform. *PVLDB*, 1(2):1277–1288, 2008.

[11] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears. Benchmarking cloud serving systems with YCSB. In *SoCC*, pages 143–154, 2010.

[12] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. C. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally-distributed database. In *OSDI*, pages 251–264, 2012.

[13] J. A. Cowling and B. Liskov. Granola: Low-overhead distributed transaction coordination. In *ATC*, pages 223–235, 2012.

[14] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In *SOSP*, pages 205–220, 2007.

[15] B. Ding, L. Kot, and J. Gehrke. Improving optimistic concurrency control through transaction batching and operation reordering. *PVLDB*, 12(2):169–182, 2018.

[16] A. Dragojevic, D. Narayanan, E. B. Nightingale, M. Renzelmann, A. Shamis, A. Badam, and M. Castro. No compromises: Distributed transactions with consistency, availability, and performance. In *SOSP*, pages 54–70, 2015.

[17] K. P. Eswaran, J. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Commun. ACM*, 19(11):624–633, 1976.

[18] J. M. Faleiro, D. Abadi, and J. M. Hellerstein. High performance transactions via early write visibility. *PVLDB*, 10(5):613–624, 2017.

[19] J. M. Faleiro and D. J. Abadi. Rethinking serializable multiversion concurrency control. *PVLDB*, 8(11):1190–1201, 2015.

[20] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *SIGMOD Conference*, pages 15–26, 2014.

[21] J. Gray, P. Helland, P. E. O'Neil, and D. E. Shasha. The dangers of replication and a solution. In *SIGMOD Conference*, pages 173–182, 1996.

[22] J. Gray, R. A. Lorie, G. R. Putzolu, and I. L. Traiger. Granularity of locks in a large shared data base. In D. S. Kerr, editor, *VLDB*, pages 428–451. ACM, 1975.

[23] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. J. Weinberger. Quickly generating billion-record synthetic databases. In *SIGMOD Conference*, pages 243–252, 1994.

[24] R. Harding, D. V. Aken, A. Pavlo, and M. Stonebraker. An evaluation of distributed concurrency control. *PVLDB*, 10(5):553–564, 2017.

[25] R. Jiménez-Peris, M. Patiño-Martínez, and S. Arévalo. Deterministic scheduling for transactional multithreaded replicas. In *SRDS*, pages 164–173, 2000.

[26] R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103, 1972.

[27] B. Kemme and G. Alonso. Don't be lazy, be consistent: Postgres-r, A new way to implement database replication. In *VLDB*, pages 134–143, 2000.

[28] K. Kim, T. Wang, R. Johnson, and I. Pandis. ERMIA: fast memory-optimized database system for heterogeneous workloads. In *SIGMOD Conference*, pages 1675–1687, 2016.

[29] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. *ACM Trans. Database Syst.*, 6(2):213–226, 1981.

[30] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[31] J. Li, E. Michael, and D. R. K. Ports. Eris: Coordination-free consistent transactions using in-network concurrency control. In *SOSP*, pages 104–120, 2017.

[32] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *SIGMOD Conference*, pages 21–35, 2017.

[33] J. Lockerman, J. M. Faleiro, J. Kim, S. Sankaran, D. J. Abadi, J. Aspnes, S. Sen, and M. Balakrishnan. The fuzzylog: A partially ordered shared log. In A. C. Arpaci-Dusseau and G. Voelker, editors, *OSDI*, pages 357–372.

[34] H. A. Mahmoud, V. Arora, F. Nawab, D. Agrawal, and A. El Abbadi. MaaT: Effective and scalable coordination of distributed transactions in the cloud. *PVLDB*, 7(5):329–340, 2014.

[35] H. A. Mahmoud, F. Nawab, A. Pucher, D. Agrawal, and A. El Abbadi. Low-latency multi-datacenter databases using replicated commit. *PVLDB*, 6(9):661–672, 2013.

[36] Y. Mao, E. Kohler, and R. T. Morris. Cache craftiness for fast multicore key-value storage. In *EuroSys*, pages 183–196, 2012.

[37] C. Mohan, B. G. Lindsay, and R. Obermarck. Transaction management in the R* distributed database management system. *ACM Trans. Database Syst.*, 11(4):378–396, 1986.

[38] S. Mu, Y. Cui, Y. Zhang, W. Lloyd, and J. Li. Extracting more concurrency from distributed transactions. In *OSDI*, pages 479–494, 2014.

[39] S. Mu, L. Nelson, W. Lloyd, and J. Li. Consolidating concurrency control and consensus for commits under conflicts. In *OSDI*, pages 517–532, 2016.

[40] P. Narasimhan, L. E. Moser, and P. M. Melliar-Smith. Enforcing determinism for the consistent replication of multithreaded CORBA applications. In *SRDS*, pages 263–273, 1999.

[41] N. Narula, C. Cutler, E. Kohler, and R. T. Morris. Phase reconciliation for contended in-memory transactions. In *OSDI*, pages 511–524, 2014.

[42] D. Ongaro and J. K. Ousterhout. In search of an understandable consensus algorithm. In *ATC*, pages 305–319, 2014.

[43] D. Qin, A. Goel, and A. D. Brown. Scalable replay-based replication for fast databases. *PVLDB*, 10(13):2025–2036, 2017.

[44] K. Ren, J. M. Faleiro, and D. J. Abadi. Design principles for scaling multi-core OLTP under high contention. In *SIGMOD Conference*, pages 1583–1598, 2016.

[45] K. Ren, A. Thomson, and D. J. Abadi. Lightweight locking for main memory database systems. *PVLDB*, 6(2):145–156, 2012.

[46] K. Ren, A. Thomson, and D. J. Abadi. An evaluation of the advantages and disadvantages of deterministic database systems. *PVLDB*, 7(10):821–832, 2014.

[47] K. Ren, A. Thomson, and D. J. Abadi. VLL: a lock manager redesign for main memory database systems. *VLDB J.*, 24(5):681–705, 2015.

[48] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[49] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era (it's time for a complete rewrite). In *VLDB*, pages 1150–1160, 2007.

[50] Y. Taleb, R. Stutsman, G. Antoniu, and T. Cortes. Tailwind: Fast and atomic rdma-based replication. In *ATC*, pages 851–863, 2018.

[51] A. Thomson and D. J. Abadi. The case for determinism in database systems. *PVLDB*, 3(1):70–80, 2010.

[52] A. Thomson, T. Diamond, S. Weng, K. Ren, P. Shao, and D. J. Abadi. Calvin: Fast distributed transactions for partitioned database systems. In *SIGMOD Conference*, pages 1–12, 2012.

[53] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden. Speedy transactions in multicore in-memory databases. In *SOSP*, pages 18–32, 2013.

[54] B. Vandiver, H. Balakrishnan, B. Liskov, and S. Madden. Tolerating byzantine faults in transaction processing systems using commit barrier scheduling. In *SOSP*, pages 59–72, 2007.

[55] T. Wang, R. Johnson, and I. Pandis. Query fresh: Log shipping on steroids. *PVLDB*, 11(4):406–419, 2017.

[56] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *PVLDB*, 10(2):49–60, 2016.

[57] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. G. Andersen. Building a bw-tree takes more than just buzz words. In *SIGMOD Conference*, pages 473–488, 2018.

[58] X. Wei, Z. Dong, R. Chen, and H. Chen. Deconstructing rdma-enabled distributed transactions: Hybrid is better! In *OSDI*, pages 233–251, 2018.

[59] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen. Fast in-memory transaction processing using RDMA and HTM. In *SOSP*, pages 87–104, 2015.

[60] Y. Wu, J. Arulraj, J. Lin, R. Xian, and A. Pavlo. An empirical evaluation of in-memory multi-version concurrency control. *PVLDB*, 10(7):781–792, 2017.

[61] X. Yu, A. Pavlo, D. Sánchez, and S. Devadas. TicToc: Time traveling optimistic concurrency control. In *SIGMOD Conference*, pages 1629–1642, 2016.

[62] E. Zamanian, X. Yu, M. Stonebraker, and T. Kraska. Rethinking database high availability with RDMA networks. *PVLDB*, 12(11):1637–1650, 2019.

[63] I. Zhang, N. K. Sharma, A. Szekeres, A. Krishnamurthy, and D. R. K. Ports. Building consistent transactions with inconsistent replication. In *SOSP*, pages 263–278, 2015.

[64] W. Zheng, S. Tu, E. Kohler, and B. Liskov. Fast databases with fast durability and recovery through multicore parallelism. In *OSDI*, pages 465–477, 2014.