Announcement

Two lectures next week are **online mode only**
  – 11/28 Monday and 11/30 Wednesday
Aria: A Fast and Practical Deterministic OLTP Database

Yi Lu, Xiangyao Yu, Lei Cao and Samuel Madden
Massachusetts Institute of Technology, Cambridge, MA, USA
University of Wisconsin-Madison, Madison, WI, USA

ABSTRACT
Deterministic databases are able to efficiently run transactions across different replicas without coordination. However, existing deterministic databases require that transactions read/write sets are known before execution, making such systems impractical in many OLTP scenarios. In this paper, we present Aria, a new deterministic and decentralized OLTP database that does not have this limitation. The core idea behind Aria is that it fast executes each transaction against the same database snapshot, and then deterministically (without coordination between replicas) chooses those that should commit to ensure durability in a consistent way. We also propose a novel deterministic checkpointing mechanism that allows Aria to order transactions in a way that reduces the number of conflicts. Our experiments on a cluster of eight nodes shows that Aria outperforms current systems with minimal overhead in deterministic concurrency control algorithms and is comparable to two or two popular benchmarks (YCSB and TPCC).

PYLDB Reference Banner:
DOI: https://doi.org/10.14778/3407878.3407908

1. INTRODUCTION
Modern database systems employ replication for high-availability and data partitioning for workload. Replication offers systems to provide high-availability, i.e., resilience to machine failures, but also incurs additional network round trips to ensure write operations are replicated to replicas. Partiti-
ing across several nodes allows systems to scale to larger databases. However, most implementations require the use of two-phase commit (2PC) to address the issues caused by non-deterministic results such as system failures and race conditions in concurrency control. This introduces addi-
tional latency to distributed transactions and impacts availability (e.g., due to coordinator failure).

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License. To view a copy of the license, visit http://creativecommons.org/licenses/by-nc-sa/4.0/. The license text is also available in the VLDB Journal.

Send inquiries to: yi1@csail.mit.edu, xyu@csail.mit.edu, samm@csail.mit.edu.
Two-phase commit (2PC) incurs extra network traffic and disk logging.
Distributed DBMS Overhead: Replication

Two-phase commit (2PC) incurs extra network traffic and disk logging.

Network can be a bottleneck for log shipping during replication.
Distributed DBMS Overhead: Replication

Two-phase commit (2PC) incurs extra network traffic and disk logging.

Network can be a bottleneck for log shipping during replication.

2PC and replication degrade performance.
Deterministic Concurrency Control

Determine a batch of transactions and their order
- Each replica (i.e., site) executes the batch deterministically
Deterministic Concurrency Control

Step 1: Determine the order for a batch of transactions

Sequencer

User transactions

Partition 1
Partition 2
Partition 3
Site 1

Partition 1
Partition 2
Partition 3
Site 2

Partition 1
Partition 2
Partition 3
Site 3
Deterministic Concurrency Control

**Step 1:** Determine the order for a batch of transactions

**Step 2:** Replicate and persist the inputs of these transactions
- input size < data log size
Deterministic Concurrency Control

Step 1: Determine the order for a batch of transactions

Step 2: Replicate and persist the inputs of these transactions
- input size < data log size

Step 3: Each replica executes transactions deterministically without 2PC or replication
Calvin [1]

**Goal**: Deterministically execute a batch of transactions using parallel hardware

**Calvin** [1]

**Goal**: Deterministically execute a batch of transactions using parallel hardware

**Assumption**: read and write sets are known before execution starts

=> **Limitation 1**: read/write sets not always available

Calvin \[1\]

**Goal**: Deterministically execute a batch of transactions using parallel hardware

**Assumption**: read and write sets are known before execution starts

=> **Limitation 1**: read/write sets not always available

**Execution process**:

- A single thread acquires all locks following the deterministic order
- Worker threads execute transactions when their locks are acquired

=> **Limitation 2**: the single locking thread can be a performance bottleneck

Calvin Example

T1: read(y), write(x)
T2: read(z), write(y)
T3: write(z), write(x)

The locking thread performs the following:
- Lock y (SH) and x (EX) and dispatch T1 for execution
- Lock z (SH) and add T2’s EX lock request into y’s waiting queue
- Add T3’s EX lock requests into z’s and y’s waiting queues
Aria Deterministic Concurrency Control

No requirement of knowing read/write sets
- All transactions in a batch read from the same snapshot and write to local write sets, in parallel
- Deterministically decide what transactions can commit based on the access set.
- If abort, deterministically move to next batch

No global locking thread
Key Technique: Deterministic Reservation [2]

For each write(tuple x) by T

\[
\text{reservation}[\ h(x) \ ] = \min(\text{T.ID}, \ \text{reservation}[\ h(x) \ ])
\]

Key Technique: Deterministic Reservation [2]

For each write(tuple x) by T
\[ \text{reservation}[h(x)] = \min(T.ID, \text{reservation}[h(x)]) \]

After the entire batch is executed, T can commit if
- For every write w, T.ID = reservation[h(w)]
- For every read r, T.ID ≤ writes[h(r)]

Key Technique: Deterministic Reservation [2]

For each write(tuple x) by T

reservation[ h(x) ] = min(T.ID, reservation[ h(x) ])

After the entire batch is executed, T can commit if
– For every write w, T.ID = reservation[ h(w) ]
– For every read r, T.ID ≤ writes[ h(r) ]

Intuition: Write-after-read (WAR) dependencies must point from right to left

Deterministic reservation $\Rightarrow$ deterministic results for parallel execution

T1: read(y), write(x)
T2: read(z), write(y)
T3: write(z), write(x)
Deterministic reservation => deterministic results for parallel execution

T1: read(y), write(x)
T2: read(z), write(y)
T3: write(z), write(x)
Aria Example

Deterministic reservation => deterministic results for parallel execution

T1: read(y), write(x)
T2: read(z), write(y)
T3: write(z), write(y)

Write reservation table

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>∞</td>
</tr>
</tbody>
</table>
Aria Example

Deterministic reservation => deterministic results for parallel execution

T1: read(y), write(x)
T2: read(z), write(y)
T3: write(z), write(x)
Aria Example

Deterministic reservation => deterministic results for parallel execution

T1: read(y), write(x)  
   Commit

T2: read(z), write(y)  
   Commit

T3: write(z), write(x)  
   Abort

For every write w, T.ID = reservation[ h(w) ]
For every read r, T.ID ≤ writes[ h(r) ]
Deterministic reservation => deterministic results for parallel execution

T1: read(y), write(x)  Commit
T2: read(z), write(y)  Commit
T3: write(z), write(x)  Abort

For every write w, T.ID = reservation[ h(w) ]
For every read r, T.ID ≤ writes[ h(r) ]
Limitation of Basic Aria

Observation: sometimes cannot commit in T1, T2, T3 order, but can commit in T3, T2, T1 order
Limitation of Basic Aria

**Observation**: sometimes cannot commit in T1, T2, T3 order, but can commit in T3, T2, T1 order

Example:

T1: write(x)
T2: read(x), write(y)
T3: read(y), write(z)

Write reservation table

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_2$</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_3$</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Limitation of Basic Aria

**Observation**: sometimes cannot commit in T1, T2, T3 order, but can commit in T3, T2, T1 order

**Example:**
- T1: write(x)
- T2: read(x), write(y)
- T3: read(y), write(z)
Limitation of Basic Aria

**Observation**: sometimes cannot commit in T1, T2, T3 order, but can commit in T3, T2, T1 order

Example:
- T1: write(x)
- T2: read(x), write(y)
- T3: read(y), write(z)

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td>T1</td>
<td>T2</td>
<td>T3</td>
</tr>
</tbody>
</table>

Write reservation table

Write-after-read

Basic Aria requires all WAR dependencies to point left, which is too restrictive!
Optimization: Deterministic Reordering

**Goal:** Deterministically change the transaction order

**Key Idea:** The execution is serializable as long as the dependency graph has no cycle
Optimization: Deterministic Reordering

For each write(tuple x) \( \text{write-res[ h(x) ]} = \min(\text{T.ID}, \text{write-res[ h(x) ]}) \)

For each read(tuple x) \( \text{read-res[ h(x) ]} = \min(\text{T.ID}, \text{read-res[ h(x) ]}) \)

After the entire batch is executed, T can commit if
- For every write w, T.ID = write-res[ h(w) ]
- For every read r, T.ID ≤ write-res[ h(r) ] or for every write w, T.ID ≤ read-res[ h(w) ]
Optimization: Deterministic Reordering

For each write(tuple $x$) $\text{write-res}[h(x)] = \min(T.ID, \text{write-res}[h(x)])$

For each read(tuple $x$) $\text{read-res}[h(x)] = \min(T.ID, \text{read-res}[h(x)])$

After the entire batch is executed, $T$ can commit if
- For every write $w$, $T.ID = \text{write-res}[h(w)]$
- For every read $r$, $T.ID \leq \text{write-res}[h(r)]$ or for every write $w$, $T.ID \leq \text{read-res}[h(w)]$

The algorithm is deterministic with no central bottleneck
A YCSB workload: 480k keys, 80/20 read/write, 10 keys per transaction, uniform distribution, 12 threads
Evaluation – Deterministic Reordering

![Graph showing throughput vs skew factor for different systems]

A YCSB workload: we vary the skew factor from 0 to 1
Conclusions

Aria supports deterministic transaction execution with no prior knowledge of the read/write sets.

Aria does not use a single thread to lock tuples sequentially.

Deterministic reordering further improves the performance of Aria.
Deterministic DBs used in production systems? If not, what are the limitations?

How are deterministic DBs different from state machine replication?

Clients may observe a higher transaction latency

Optimal batch size?

Better solution to handle high skew factor?

How to ensure TIDs are globally unique?

Straggler transaction delay the completion time of an entire batch?

Can there be livelock? (atxn is constantly pushed to next batch)
Alexandre Verbitski, et al., *Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases*. SIGMOD, 2017