CS 764: Topics in Database Management Systems

Lecture 22: Deterministic DBMS

Xiangyao Yu

11/21/2022

Announcement

Two lectures next week are **online mode only**

- 11/28 Monday and 11/30 Wednesday

Today's Paper: Deterministic DBMS

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.

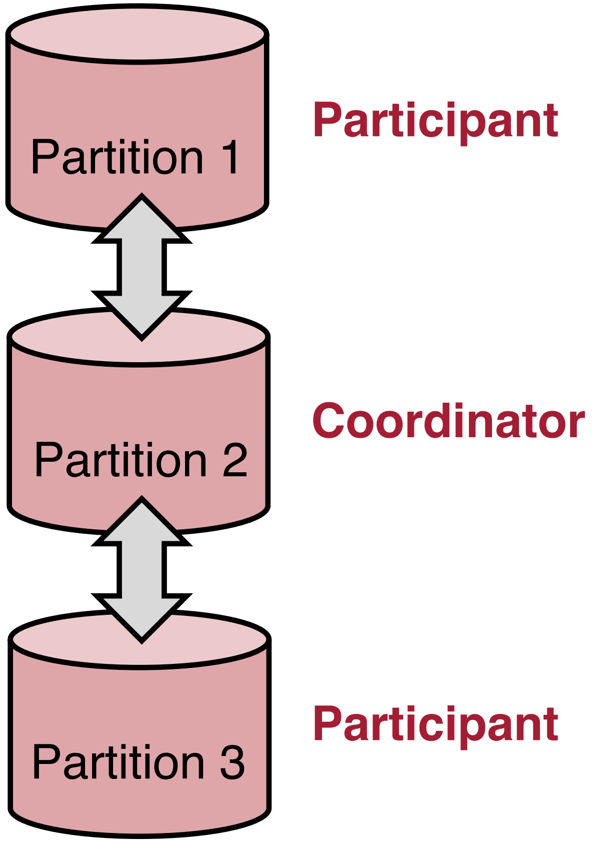
The state-of-the-art deterministic databases, BOHM [19], PWV [18], and Calvin [52], achieve determinism through *dependency graphs* or *ordered locks*. The key idea in BOHM and PWV is that a dependency graph is built from a batch of input transactions based on the read/write sets. In this way, the database 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

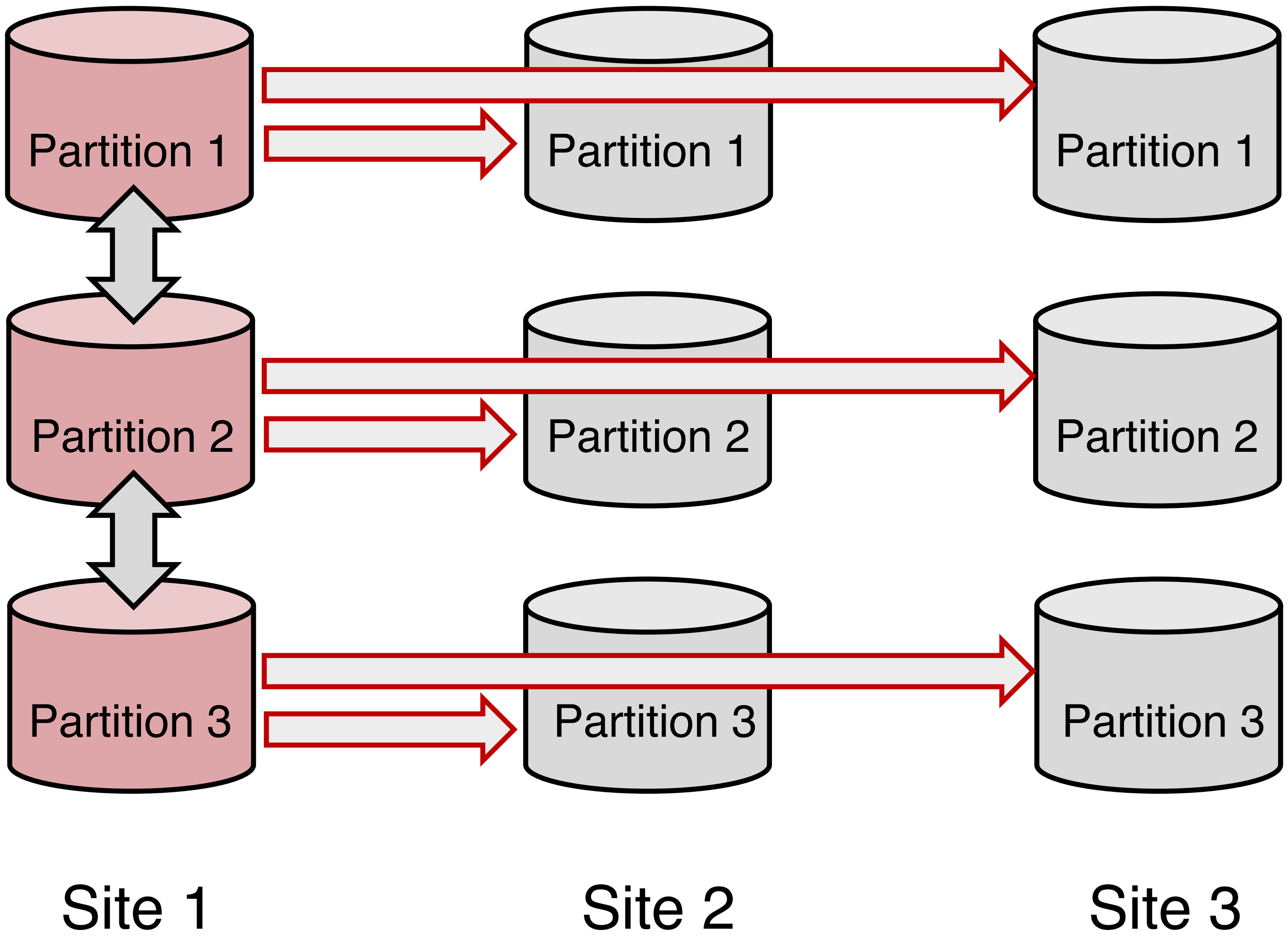
VLDB 2020

Distributed DBMS Overhead: Replication



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

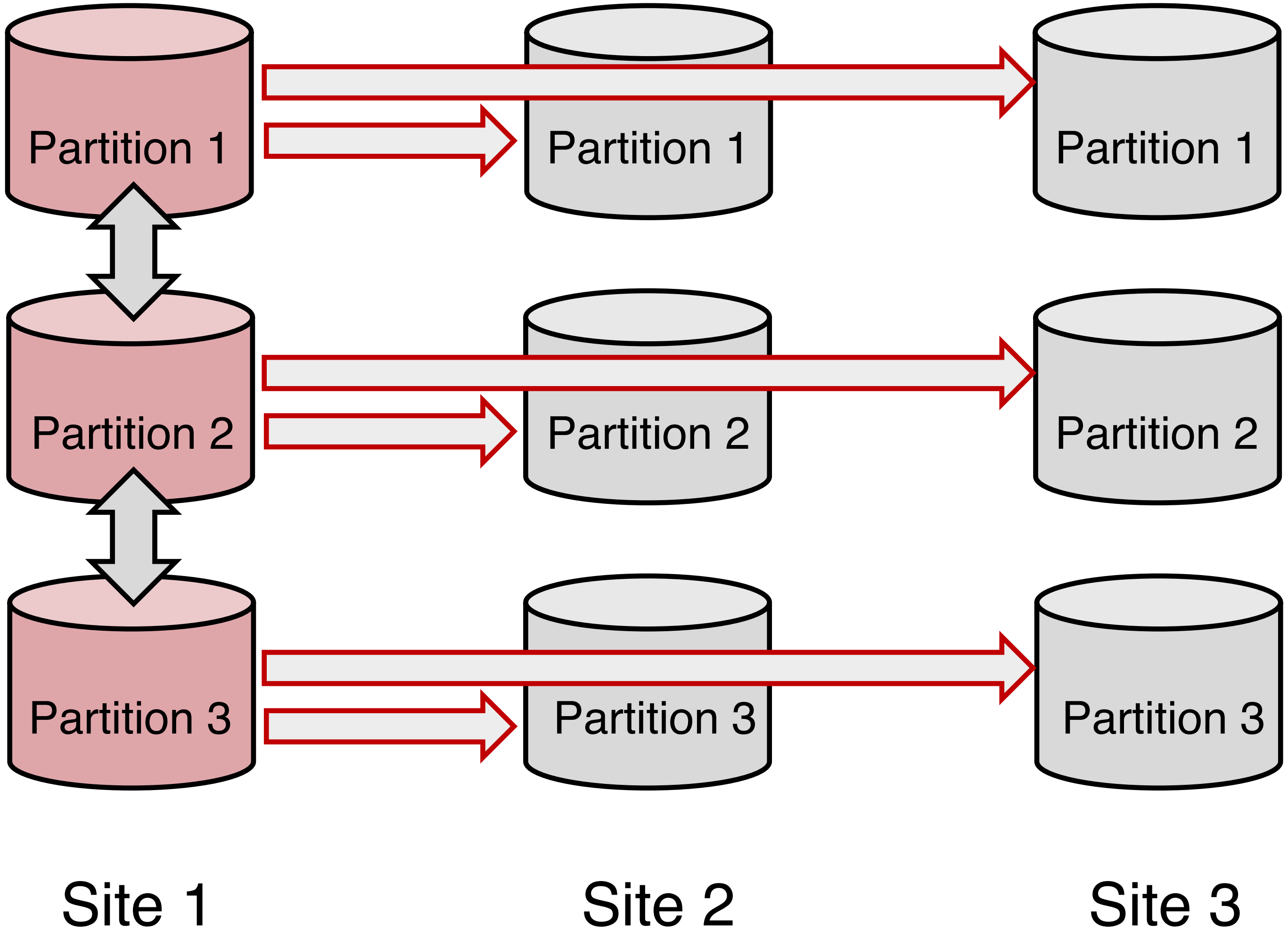
Distributed DBMS Overhead: Replication



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



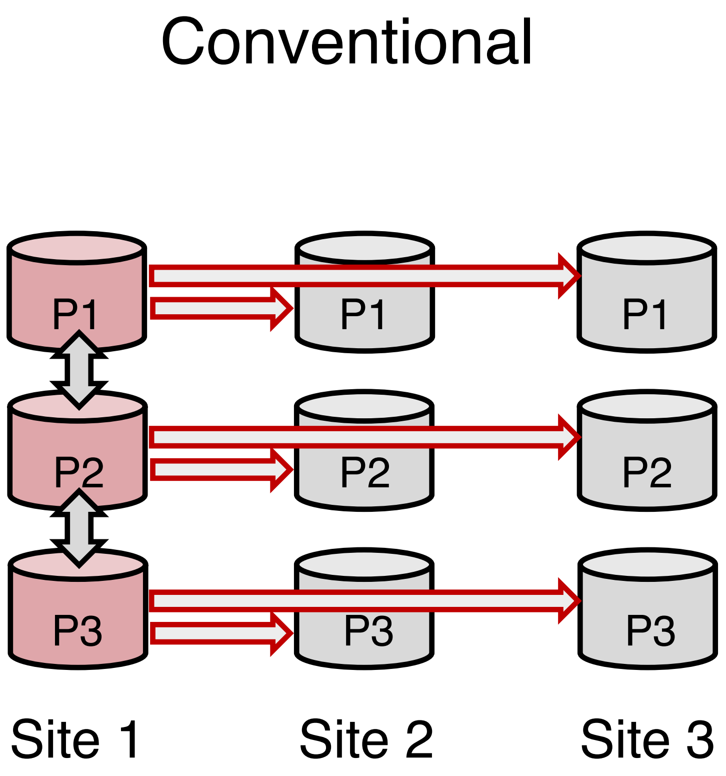
Two-phase commit (2PC) incurs extra network traffic and disk logging

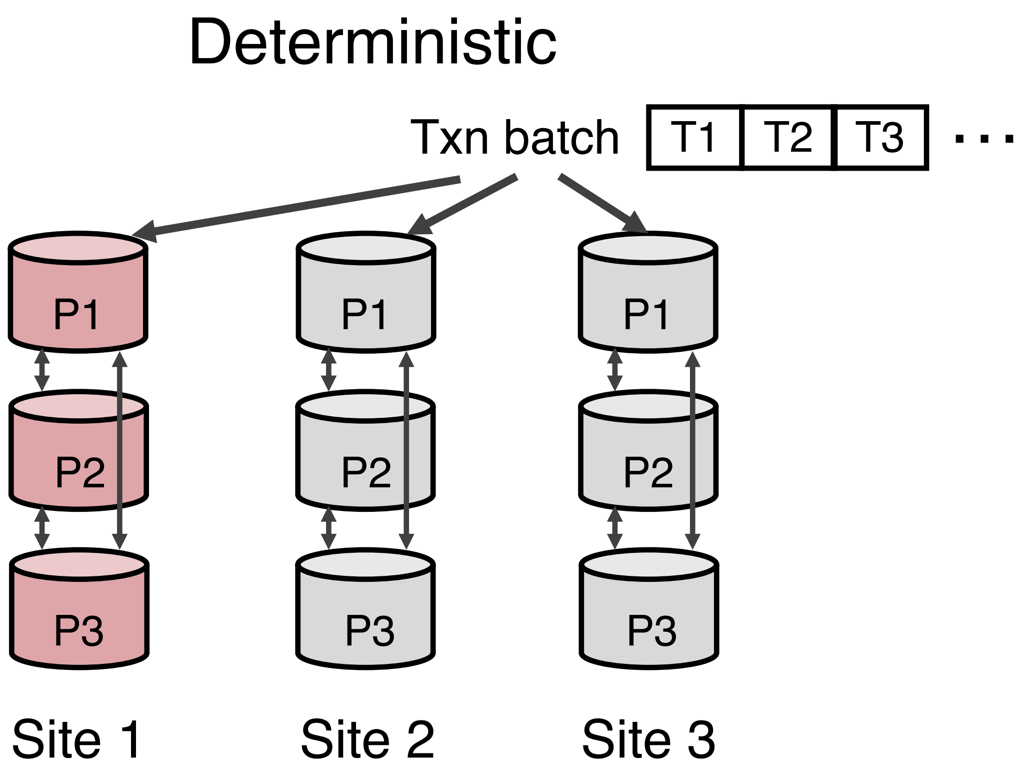
Network can be a bottleneck for log shipping during replication

2PC and replication degrade performance

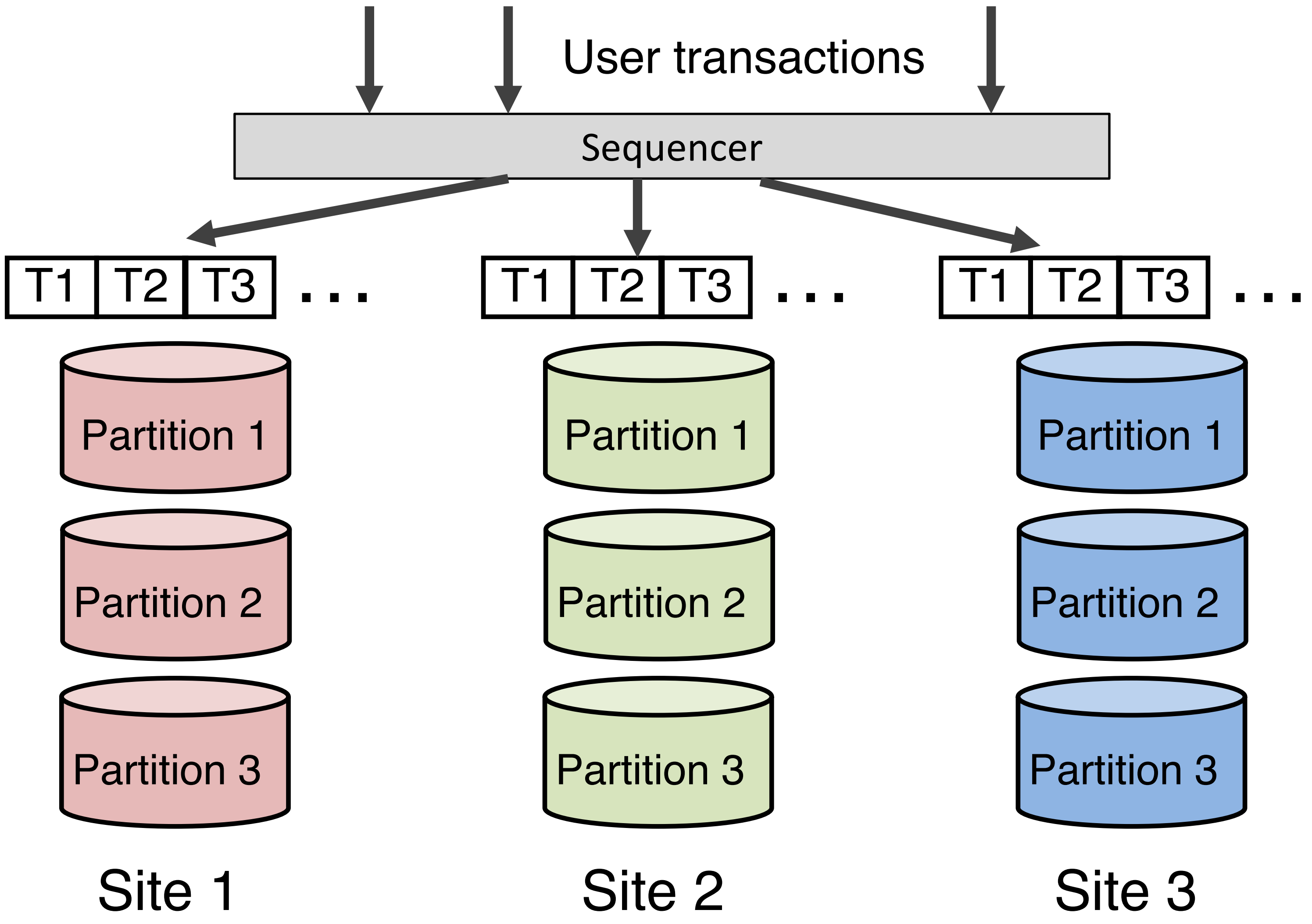
Deterministic Concurrency Control

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



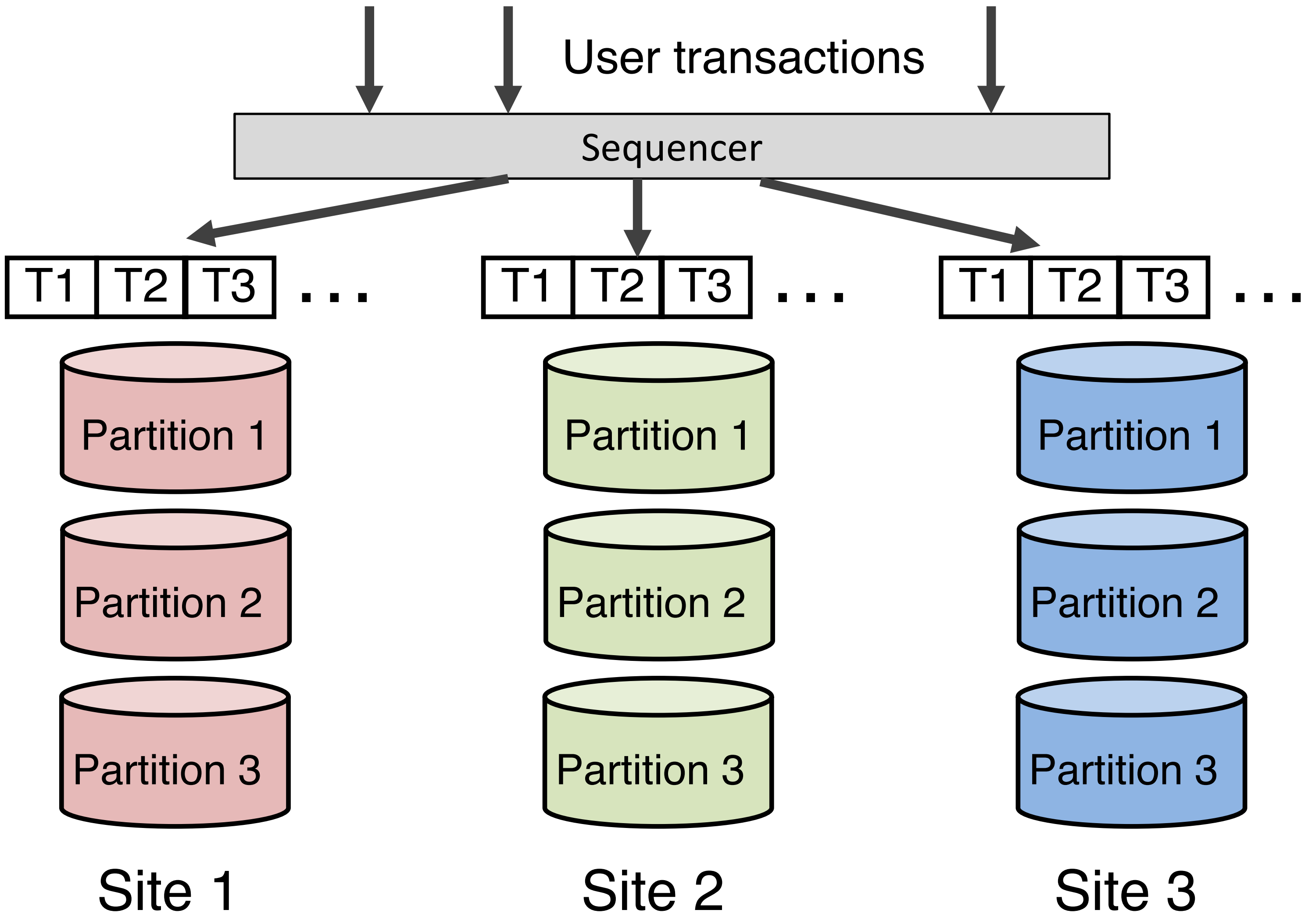


Deterministic Concurrency Control



Step 1: Determine the order for a batch of transactions

Deterministic Concurrency Control

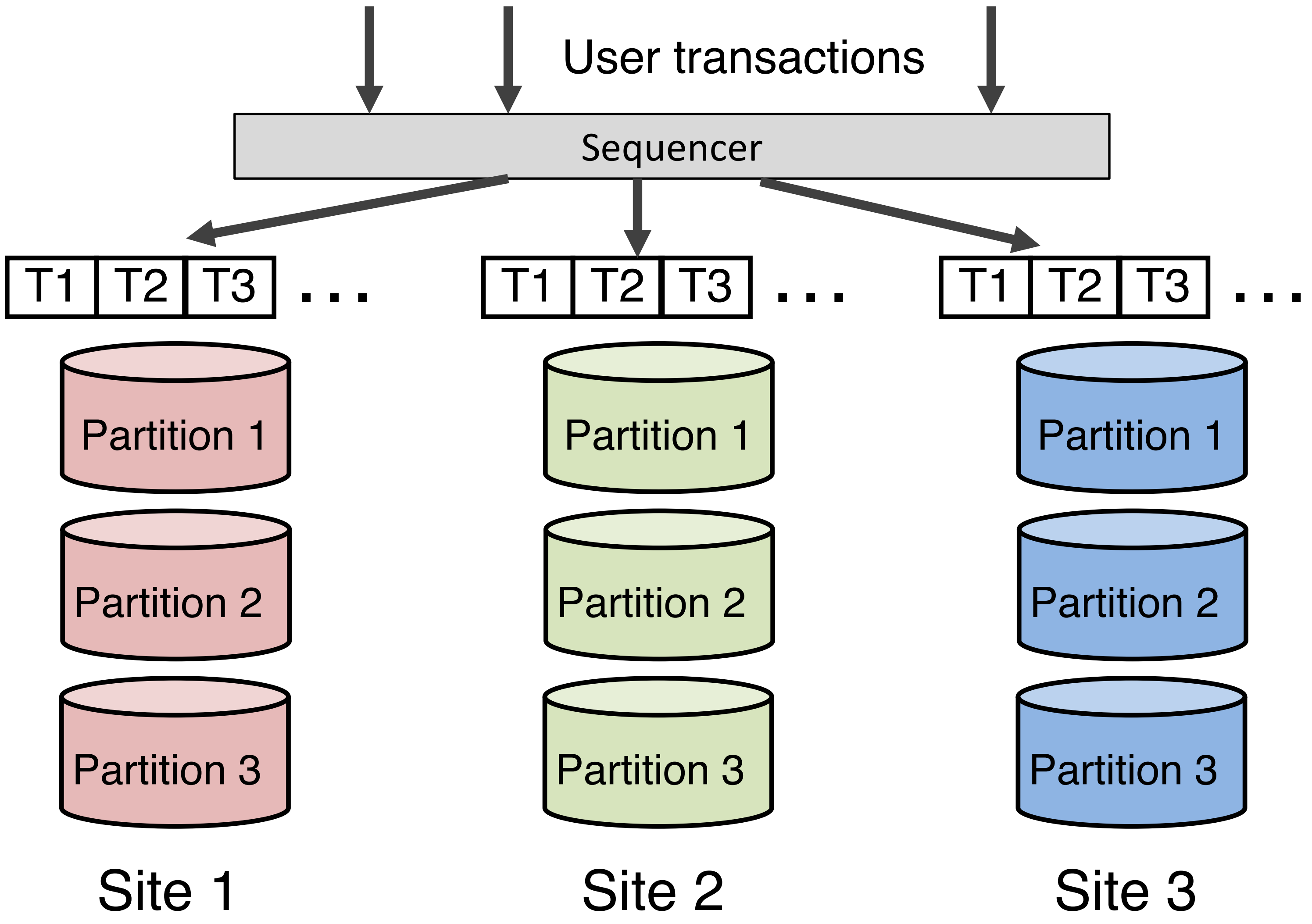


Step 1: Determine the order for a batch of transactions

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

[1] Thomson, Alexander, et al. "Calvin: fast distributed transactions for partitioned database systems." SIGMOD 2012

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

[1] Thomson, Alexander, et al. "Calvin: fast distributed transactions for partitioned database systems." SIGMOD 2012

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

[1] Thomson, Alexander, et al. "Calvin: fast distributed transactions for partitioned database systems." SIGMOD 2012

Calvin Example

T1: read(y), write(x)

T2: read(z), write(y)

T3: write(z), write(x)

The locking thread performs the following:

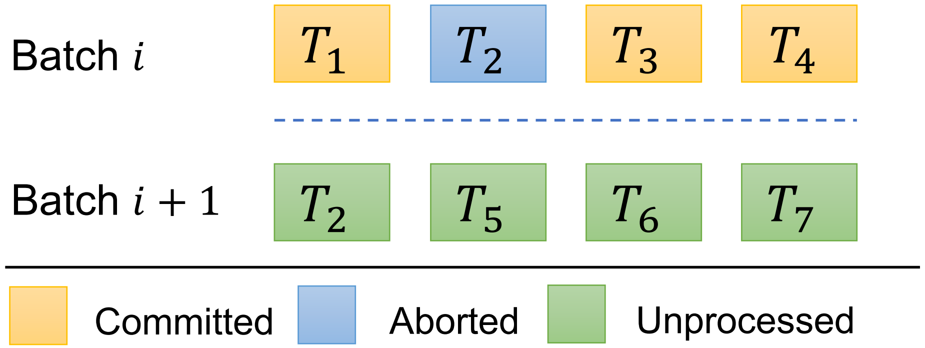
- Lock y (SH) and x (EX) and dispatch T1 for execution
- Lock z (SH) and add T2’s EX lock request into y’s waiting queue
- Add T3’s EX lock requests into z’s and y’s waiting queues

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

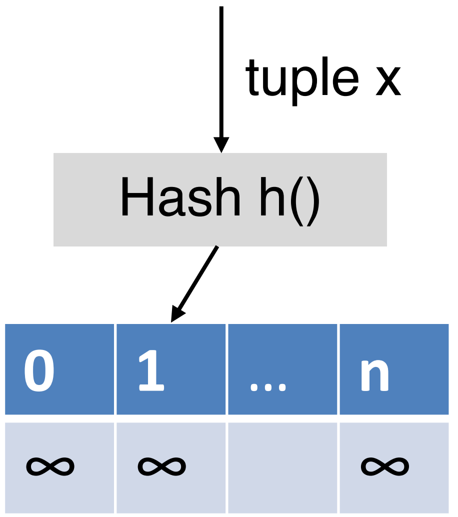
No global locking thread



Key Technique: Deterministic Reservation [2]

For each write(tuple x) by T

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



Write reservation table

[2] Blelloch, Guy E., et al. "Internally deterministic parallel algorithms can be fast." *PPoPP* 2012.

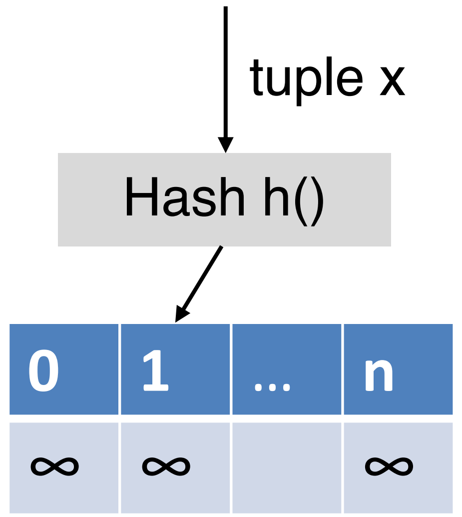
Key Technique: Deterministic Reservation [2]

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)]



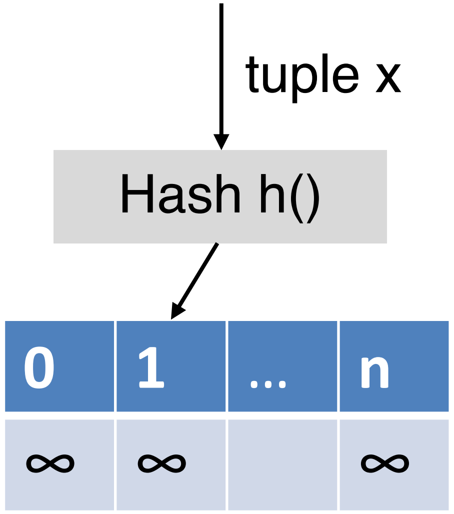
Write reservation table

[2] Blelloch, Guy E., et al. "Internally deterministic parallel algorithms can be fast." *PPoPP* 2012.

Key Technique: Deterministic Reservation [2]

For each write(tuple x) by T

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

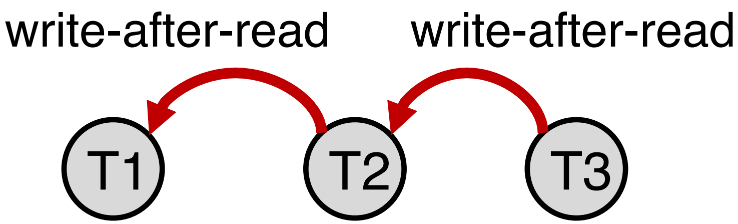


Write reservation table

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



[2] Blelloch, Guy E., et al. "Internally deterministic parallel algorithms can be fast." *PPoPP* 2012.

Aria Example

Deterministic reservation => deterministic results for parallel execution

T1: read(y), write(x)

T2: read(z), write(y)

T3: write(z), write(x)

x	y	z
∞	∞	∞

Write reservation 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)

x	y	z
T_1	∞	∞

Write reservation 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)

x	y	z
T_1	T_2	∞

Write reservation 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)

x	y	z
T_1	T_2	T_3

Write reservation table

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**

x	y	z
T_1	T_2	T_3

Write reservation table

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

Aria Example

Deterministic reservation => deterministic results for parallel execution

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

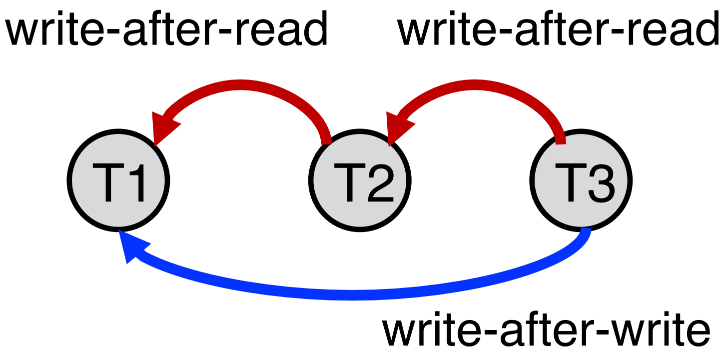
T2: read(z), write(y) **Commit**

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

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

x	y	z
T_1	T_2	T_3

Write reservation table



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)

x	y	z
T_1	T_2	T_3

Write reservation table

Limitation of Basic Aria

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

Example:

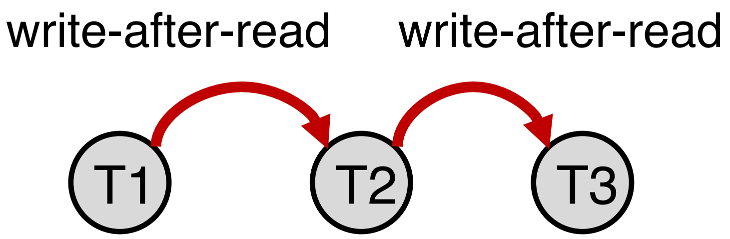
T1: write(x)

T2: read(x), write(y)

T3: read(y), write(z)

x	y	z
T_1	T_2	T_3

Write reservation table



Limitation of Basic Aria

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

Example:

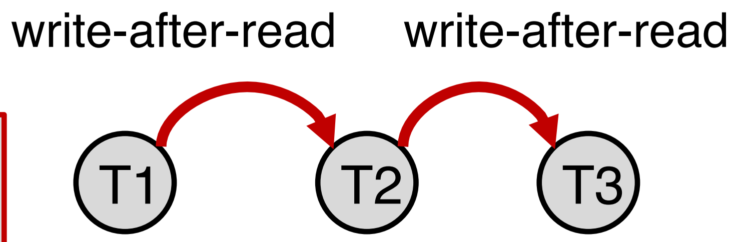
T1: write(x)

T2: read(x), write(y)

T3: read(y), write(z)

x	y	z
T_1	T_2	T_3

Write reservation table



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

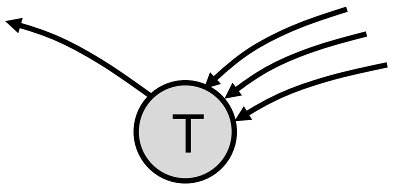
Optimization: Deterministic Reordering

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

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

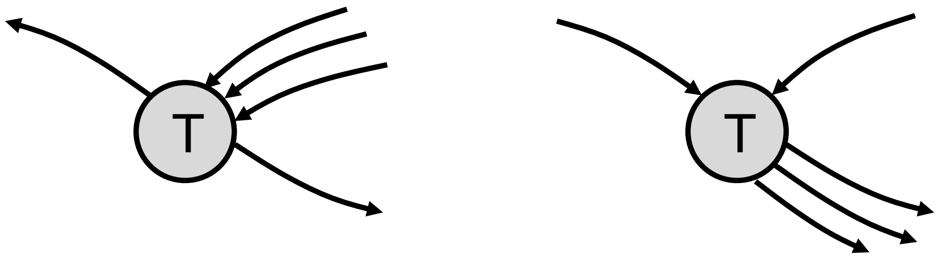
After the entire batch is executed, T can commit if

- 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)]



Node allowed in basic Aria

Only left-pointing arrows permitted



Nodes allowed in optimized Aria

Disallow left-in and left-out turns

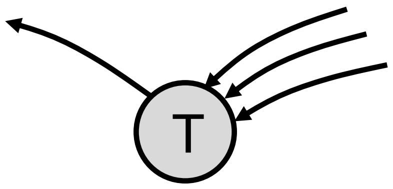
Optimization: Deterministic Reordering

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

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

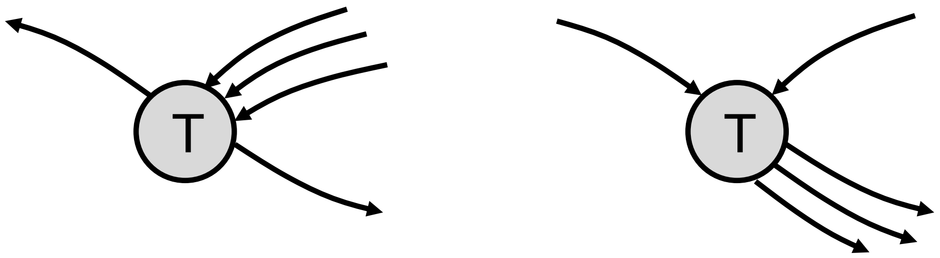
After the entire batch is executed, T can commit if

- 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)]



Node allowed in basic Aria

Only left-pointing arrows permitted

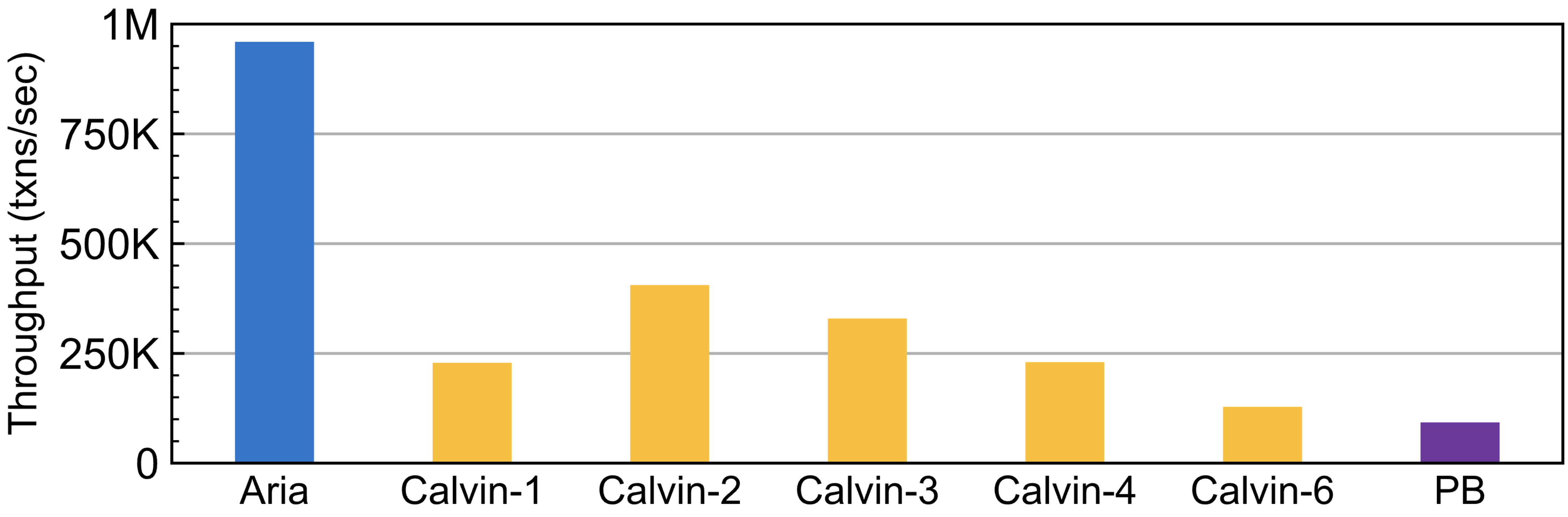


Nodes allowed in optimized Aria

Disallow left-in and left-out turns

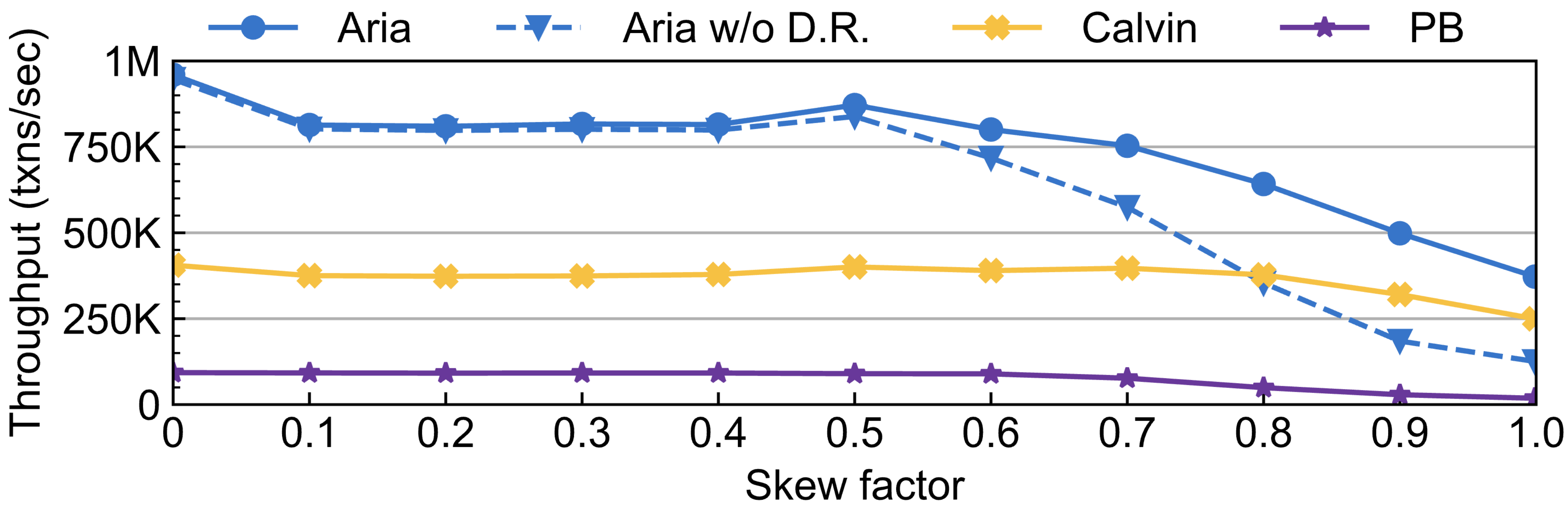
The algorithm is **deterministic** with **no central bottleneck**

Evaluation – Overall (YCSB)



A YCSB workload: 480k keys , 80/20 read/write, 10 keys per transaction, uniform distribution, 12 threads

Evaluation – Deterministic Reordering



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

Q/A – Deterministic DBMS

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? (a txn is constantly pushed to next batch)

Next Monday

Alexandre Verbitski, et al., Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. SIGMOD, 2017