ByShard: Sharding in a Byzantine Environment

Ramesh Adhikari, Graduate Research Assistant
School of Computer and Cyber Sciences, Augusta University

September, 2022

Outline

- Motivation for this paper
- Used Protocol
- Main Idea
- Proposed Model
- Evaluation

Motivation for this paper

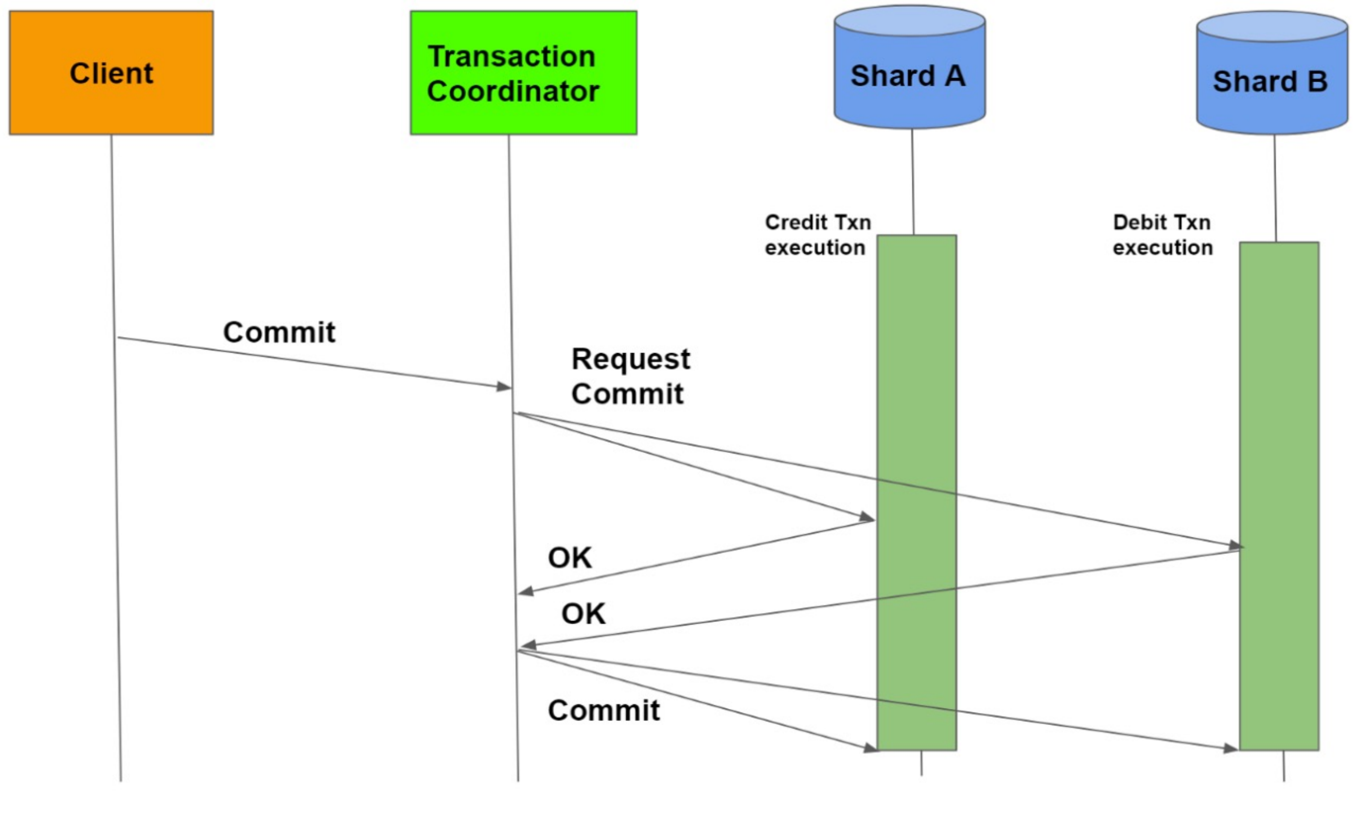
- Development of resilient system than can handle Byzantine failure due to Crashes, Bugs, Malicious behaviors
- Current Sharded resilient system do not provide the flexibility of traditional data management system
- To proposed High-Performance Resilient system

Used Protocol

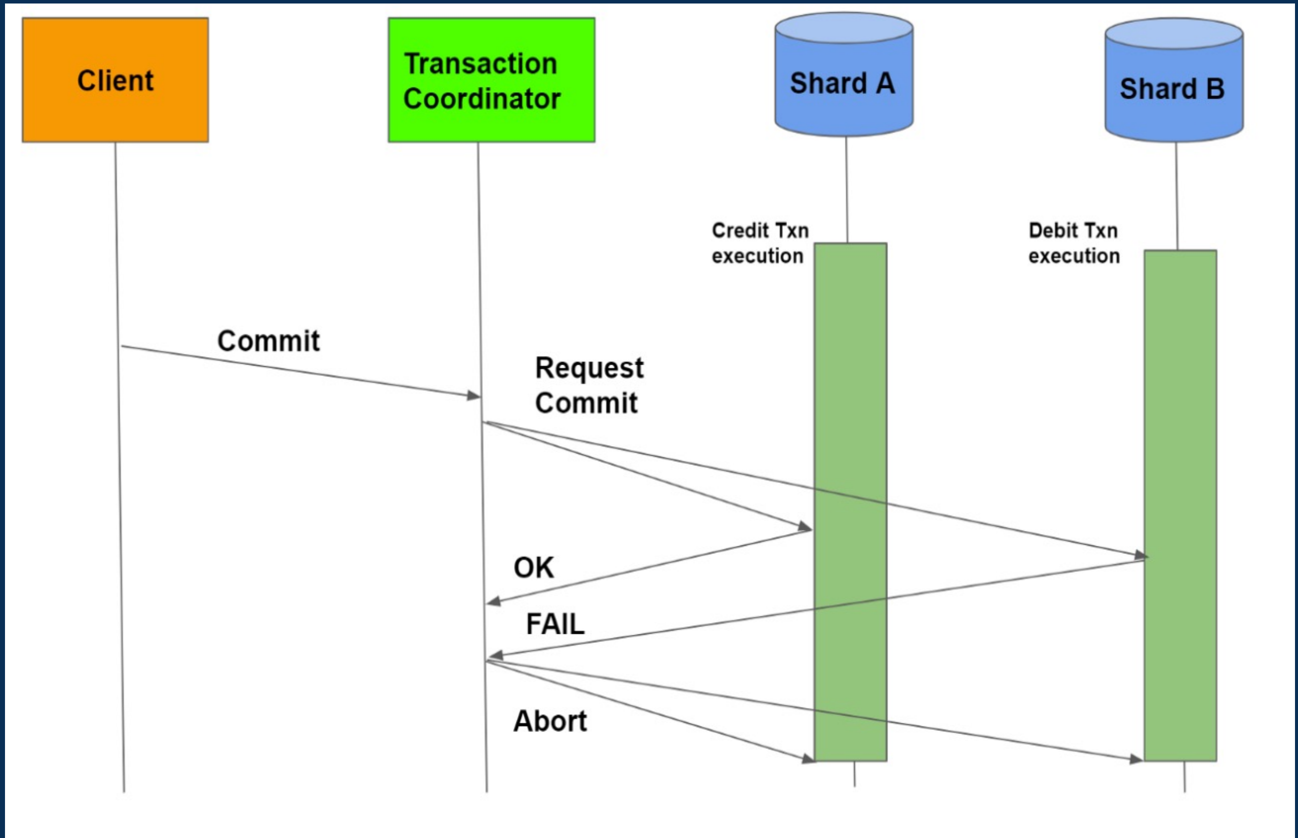
Used two traditional sharded database concept efficiently in Byzantine environment

- Two-phase commit: Atomicity; atomic decision on whether the transaction can be committed or not;
- Two-phase locking: Isolation; provide concurrency control

Two Phase Commit

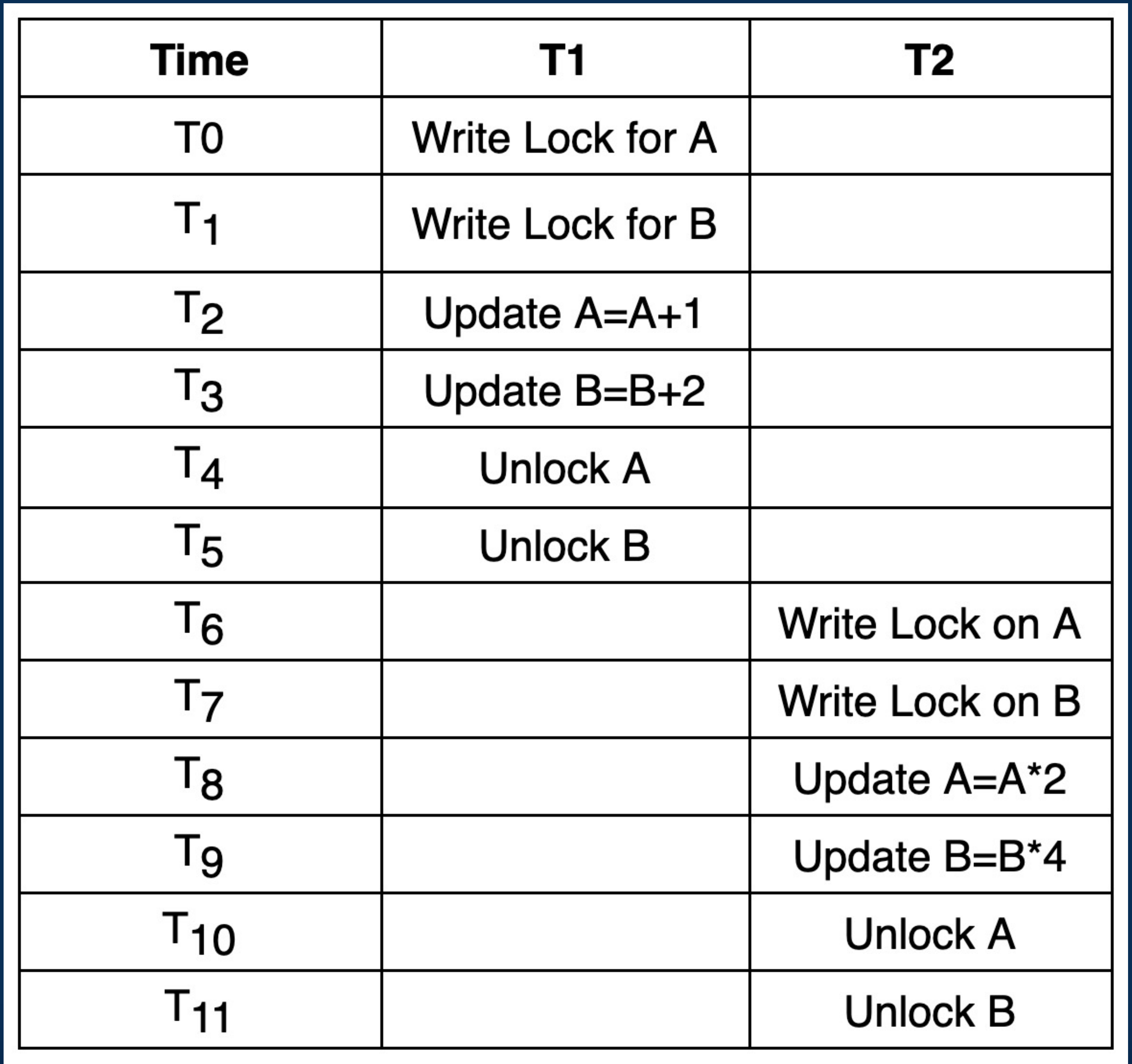


Commit Txn after receiving OK from both the servers



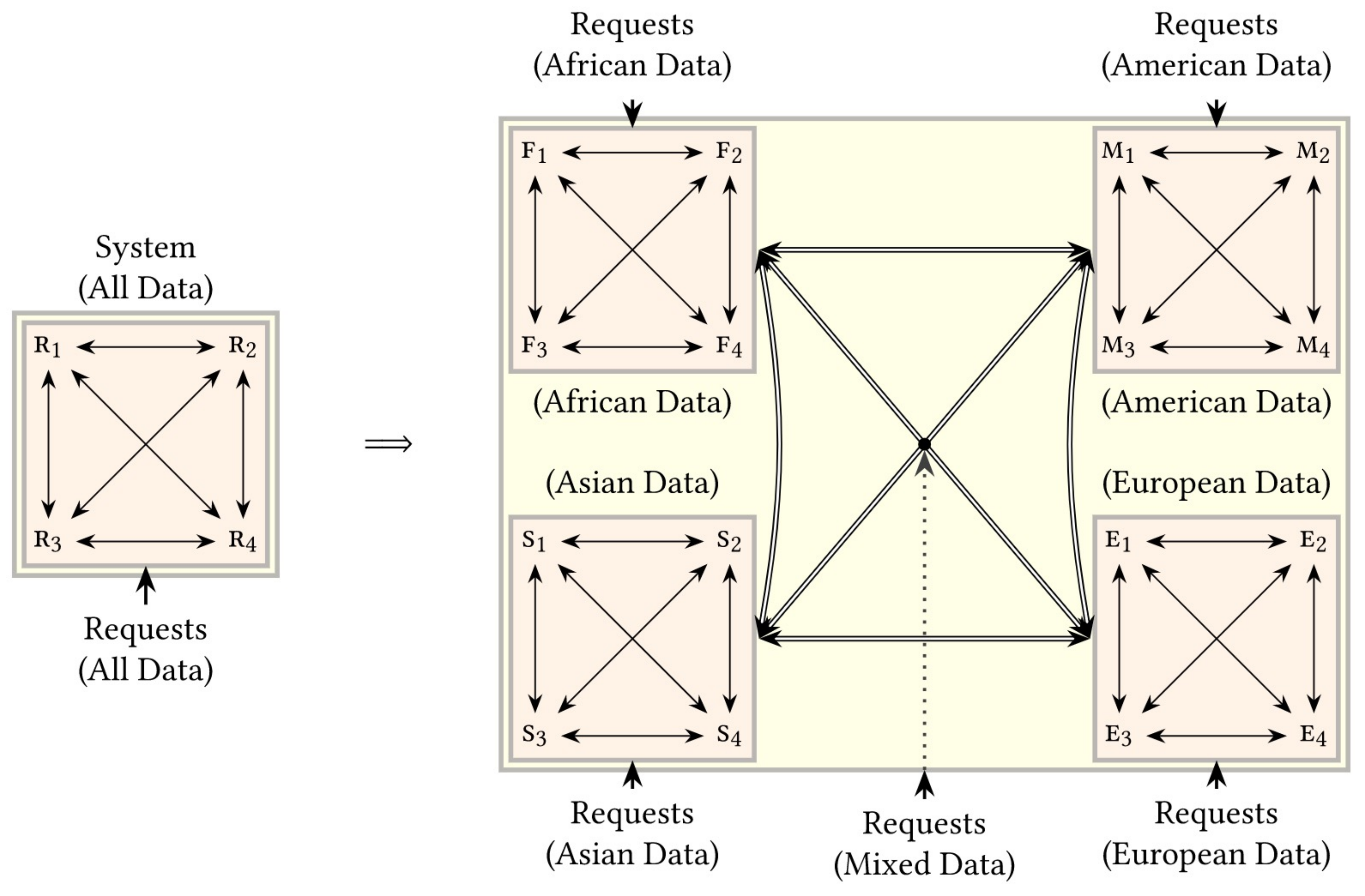
Rollback the transaction in case of failure

Two Phase Locking (Serializability)



Time	T1	T2
T0	Write Lock for A	
T_1	Write Lock for B	
T_2	Update A=A+1	
T_3	Update B=B+2	
T_4	Unlock A	
T_5	Unlock B	
T_6		Write Lock on A
T_7		Write Lock on B
T_8		Update A=A*2
T_9		Update B=B*4
T_{10}		Unlock A
T_{11}		Unlock B

Sharded Design



AUGUSTA UNIVERSITY

Main Idea

- Orchestrate-execution model (OEM) in Byzantine environment
  - Orchestration: Replication of transactions among all involved shards and reaching on atomic decision; used two-phase commit
  - Execution model: Execution of transactions by maintaining data consistency among shards; used two-phase locking
- Uses cluster-sending communication
  - Particular algorithm unspecified
- Uses consensus abstraction as a Blackbox

Communication in Shard

- **Cluster-sending** protocol is used for reliable communication between clusters S_1 and S_2; To send S_1 value v to $S_{2;}$ Provide the following guarantees
  - S_1 can send v to S_2 only if there is agreement on sending v among the non-faulty replicas in S_1;
  - all non-faulty replicas in S_2 will receive the value v; and
  - all non-faulty replicas in S_1 obtain confirmation of receipt.

Orchestrate-execution model (OEM)

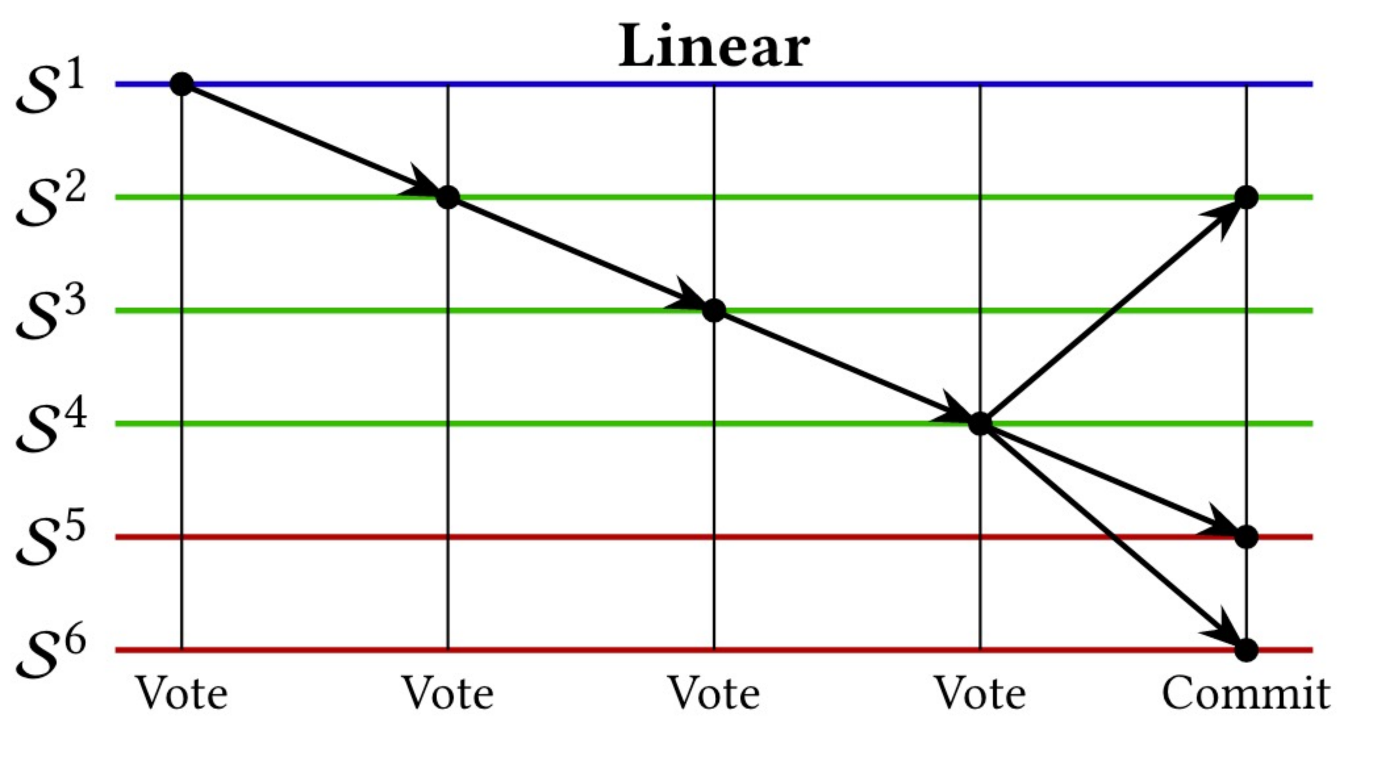
- Processing is broken down into three types of shard-steps
  - Vote-step: Shard (S) Verifies the constraints to determine whether S votes for either commit or abort. And can make local changes, e.g., check conditions, modify local data or acquire locks
  - Commit-step: Shard performs necessary operations to finalize transactions when transactions is committed. E.g., Modify data and release locks
  - Abort-step: Shard performs necessary operations to roll-back transactions when transactions is aborted. E.g., roll -back local changes, release locks

Orchestration

- The main goal of it is to replicate the transactions (Tx) to involved shard and obtained the commit/abort decision
- Three type of model
  - Linear (based on Linear 2PC)
  - Centralized (based on 2PC)
  - Decentralized

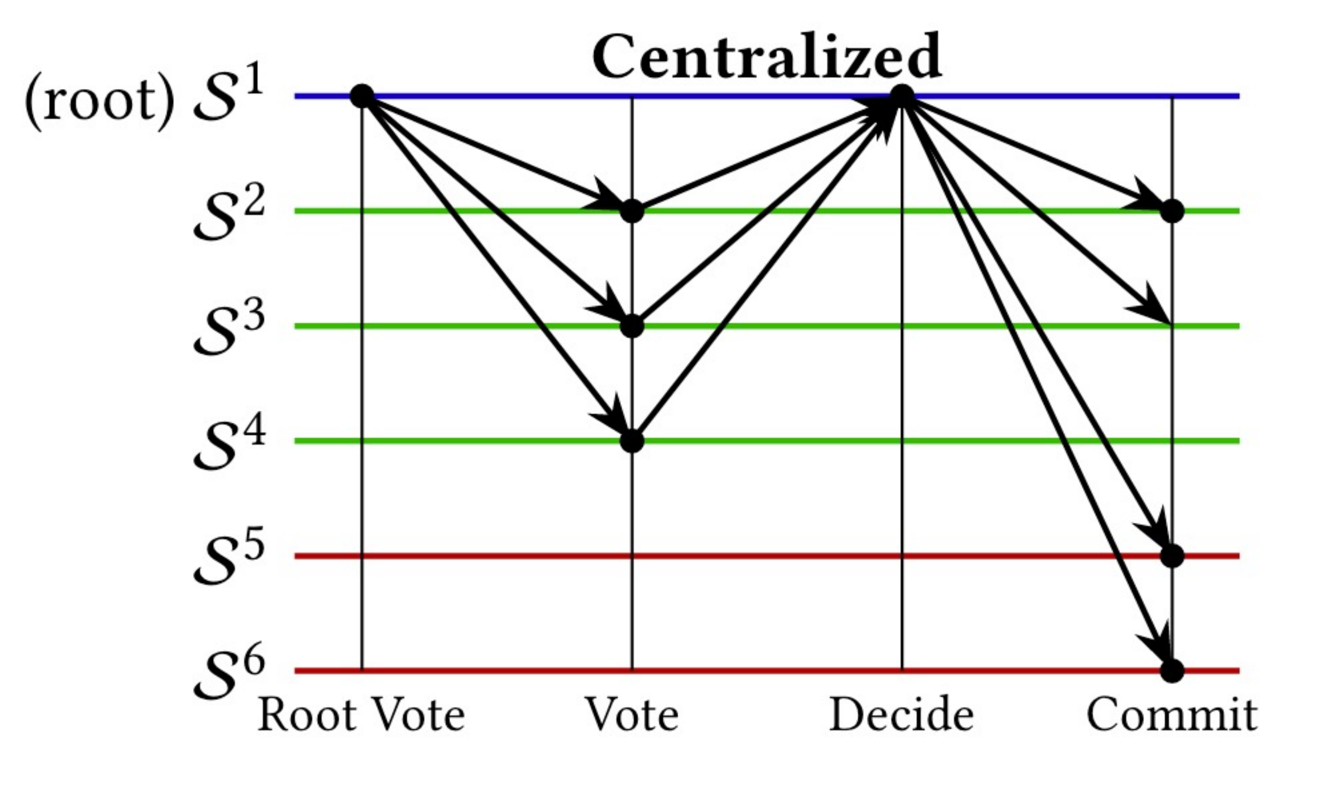
Orchestration - Linear

- Vote Step: Sequence
- Decide: Centralized
- Commit or Abort: Parallel
- Advantage: Early abort
- S^1,S^2,S^3 and S^4 are vote-steps
- S^2,S^5 and S^6 have commit steps
- Every dot represents a single consensus step
- Every arrow a single cluster sending step



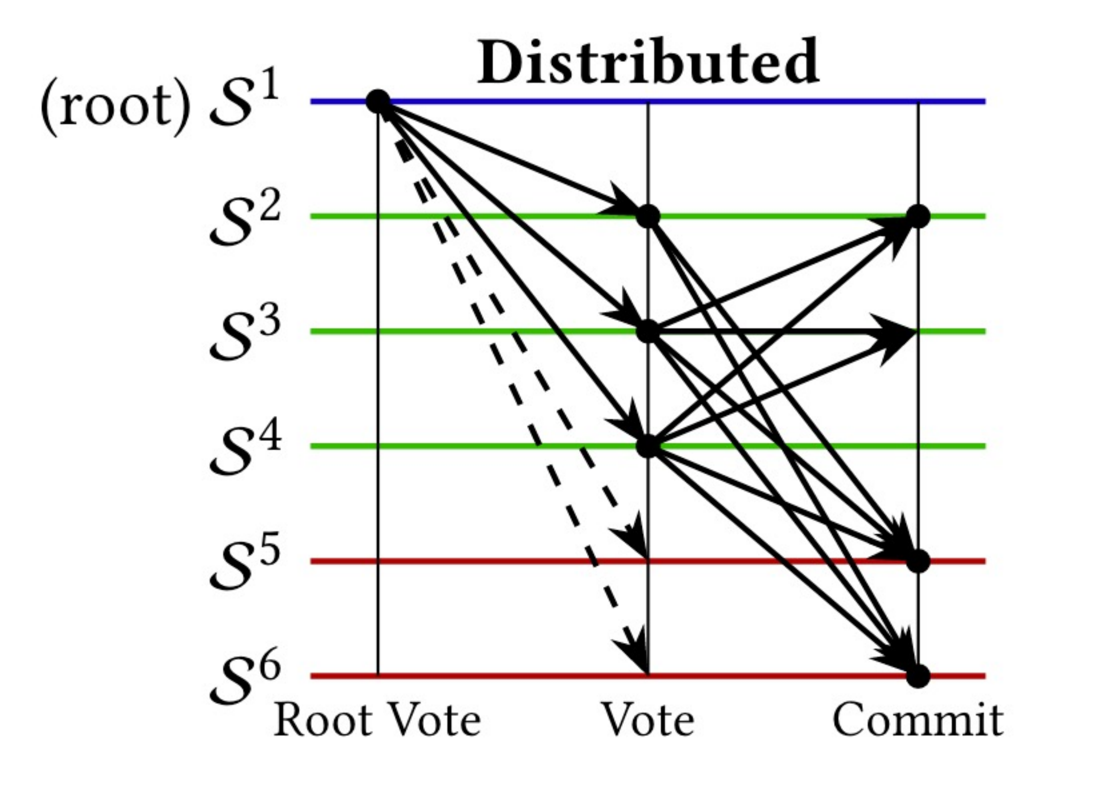
Orchestration - Centralized

- Root/Coordinator is selected for each Tx independently
- Vote Step: Parallel
- Decide: Centralized
- Commit or Abort: Parallel
- Disadvantage: Wait for all message



Orchestration - Decentralized

- Vote Step: Parallel
- Decide: Decentralized
- Commit or Abort: Parallel
- Can be performed in 3 consecutive steps
- Vote aggregation is performed in a single step as well

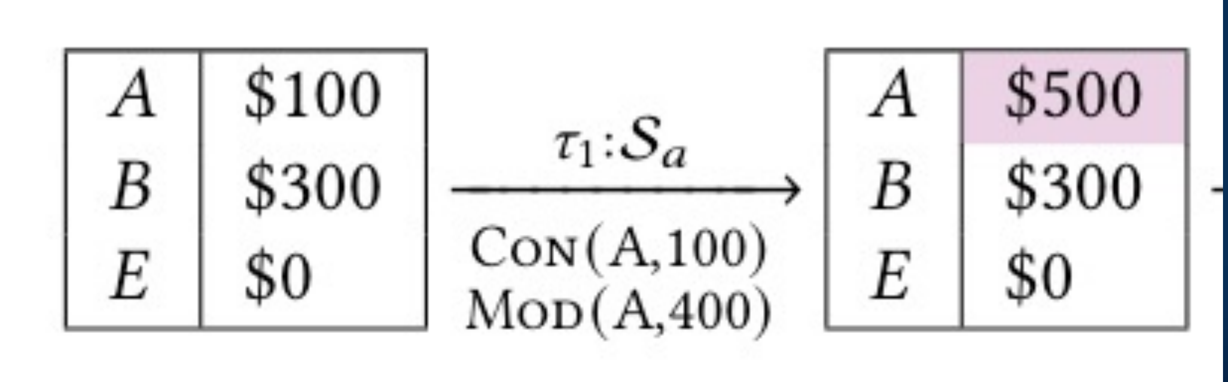


Execution Model

- Execution part consider the isolation
  - The above orchestrations allow to read uncommitted data
  - Two-phase locking is proposed to cope with that
  - A Tx is split to Constraint and Modification steps

Execution-Isolation free execution

- If S has a condition – update is made in the vote step
- Abort steps are generated for all such modification
- If S has no condition – modifications are made in the commit step, no abort step needed
- Disadvantage- Dirty read are possible



A	\$100
B	\$300
E	\$0

$$\xrightarrow[\textsc{Con}(A,100)\ \textsc{Mod}(A,400)]{\tau_1 : S_a}$$

A	\$500
B	\$300
E	\$0

Execution – Lock-based execution

- Read/Write locks are used
- Modes:
  - Read uncommitted : Dirty Read
  - Read Committed : avoid a dirty read, but reads the same row twice and gets a different value each time
  - Serializable: read and write locks are used in a usual way; Two Phase Locking; data consistency; isolation

Example of the OEM

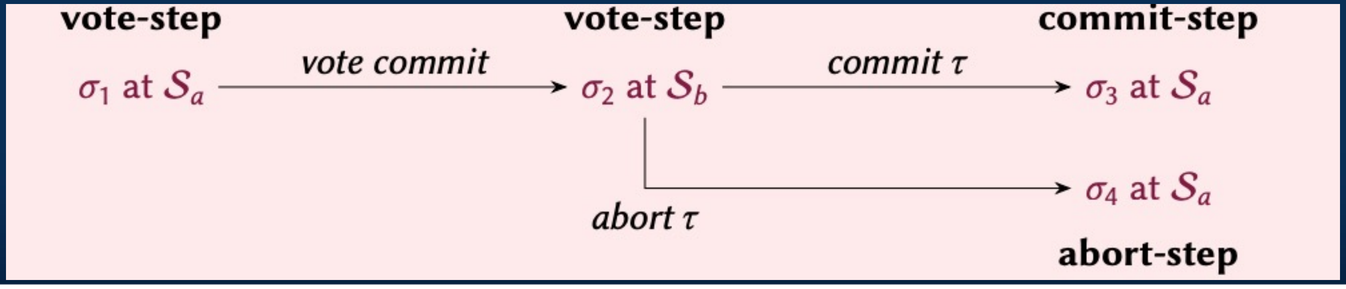
Shard accounts by first letter of name

Representations: τ is transaction; σ is shard-step;

- τ = “if Ana has $500 and Bo has $200, then move $400 from Ana to Bo.
- σ1 = “LOCK(Ana); if Ana has $500, then forward σ2 to Sb (Commit vote)

  else RELEASE(Ana) (Abort vote)”
- σ2 = “LOCK(**Bo**); if Bo has $200, then add $400 to Bo; RELESE(Bo);
  and forward σ3 to Sa (Commit)

  else RELEASE(Bo) and forward σ4 to Sa (Abort)”
- σ3 = “remove $400 from Ana, Commit τ and RELEASE(Ana)”
- σ4 = “Abort τ and RELEASE(Ana)”



vote-step

σ_1 at $\mathcal{S}_a$ $\xrightarrow{\textit{vote commit}}$ σ_2 at $\mathcal{S}_b$

vote-step

σ_2 at $\mathcal{S}_b$ $\xrightarrow{\textit{commit } \tau}$ σ_3 at $\mathcal{S}_a$

commit-step

abort τ → σ_4 at $\mathcal{S}_a$

abort-step

Evaluation

- Consensus steps were abstracted in evaluations
- Experiment done on 5000 Tx
- Tx affects 16 accounts, 8 accounts have constrained
- 64 Shards
- 8k accounts
- Scalability increases with number shards while keeping other parameters constant.

Thank you!