Transactions
- A Quick Overview

Ashvin Goel

Electrical and Computer Engineering
University of Toronto

ECE1724

These slides are adapted from Michael Freedman & Wyatt Lloyd's course on Distributed Systems

Transactions

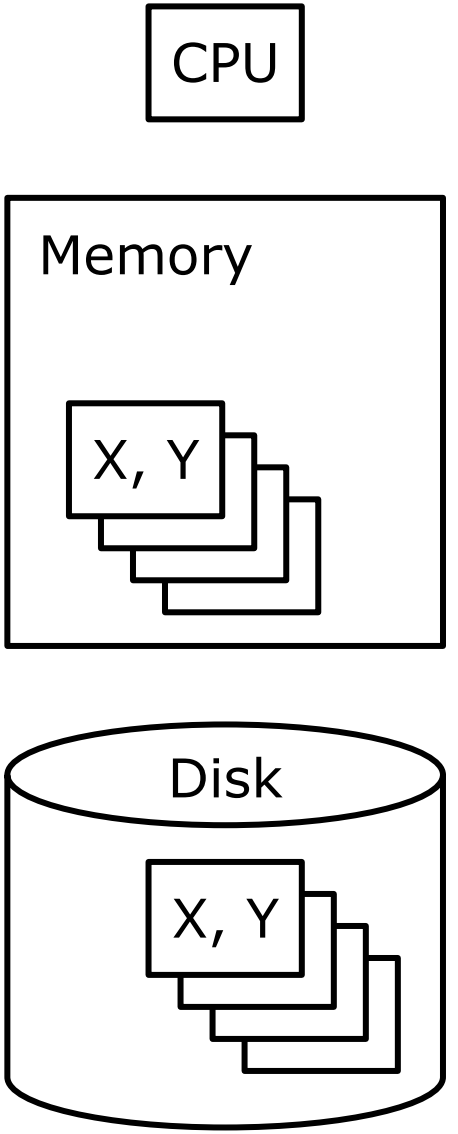
- A unit of work that may perform multiple operations (e.g., reads and writes) on multiple items (e.g., A, B)

```
transfer(A, B):
begin_tx
a = read(A)
if a < 10 then
   abort_tx
else
   write(A, a-10)
   b = read(B)
   write(B, b+10)
   commit_tx
```

```
sum(A, B):
begin_tx
a = read(A)
b = read(B)
print a + b
commit_tx
```

Transaction Execution Model

- input(X)
  - copy the disk block containing object X to memory
- v = read(X)
  - read the value of X into a local variable v
  - execute input(X) first if necessary
- write(X, v)
  - write value v to X in memory
  - execute input(X) first if necessary
- output(X)
  - write memory block containing X to disk

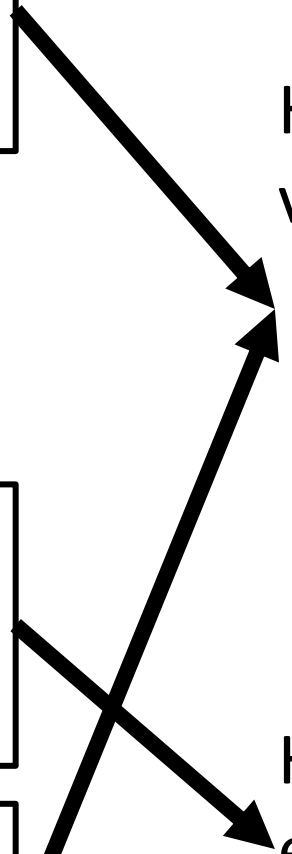


Transaction Properties: ACID

- Atomicity: transaction executes completely or not at all
  - E.g., transfer(A,B) either commits or makes no changes
- Consistency: transaction moves database from one consistent state to another
  - E.g., writes don't violate integrity constraints, avoids database corruption
- Isolation: operations in the transaction appear to happen together at a point in time
  - E.g., sum(A,B) does not read intermediate updates by transfer(A, B)
- Durability: transactions that commit are not lost, even on failure

ACID Challenges

- Atomicity: transaction executes completely or not at all (failure atomicity)
- Consistency: transaction moves database from one consistent state to another
- Isolation: operations in the transaction appear to happen together
- Durability: transactions that commit are not lost, even on failure



How to recover from various failures?

- app-level (txn abort)
- system-level (e.g., oom)
- crash failures
- media failures

How to control execution of concurrent transactions?

Failure Recovery

Failures

- Transaction T aborts or system crashes while T is executing, and partial effects of T were written to disk
  - How do we undo T (atomicity)?
- System crashes after a transaction T commits, and not all effects of T were written to disk
  - How do we complete T (durability)?
- Media fails or data on disk is corrupted
  - How do we reconstruct the database (durability)?
- Key idea for failure recovery: always make a copy before overwriting a block so the copy can be used for recovery

Write-Ahead Logging (WAL)

- Logging: write a sequence of log records to disk, recording a history of changes made to the database
  - Each write becomes two writes, isn't it bad for performance?
- Write-ahead logging: before any object X is overwritten on disk (flushed), log record for X must be flushed
  - Enables failure recovery

Undo Based Write-Ahead-Logging

- Before Transaction T modifies X on disk, use WAL to flush its old value to the log
  - Log format: <Tid, X, old_value_of_X>
    - Tid is transaction id
    - X: physical address of X (block id, offset)
    - old_value_of_X: physical bits (physical logging)
- Force: before commit record of a transaction is flushed to the log, all writes of transaction must be flushed
  - If system crashes before transaction commits, undo updates to X on disk by restoring old value of X from log
  - If system crashes after transaction commits, all updates have already been applied

Undo Logging Example

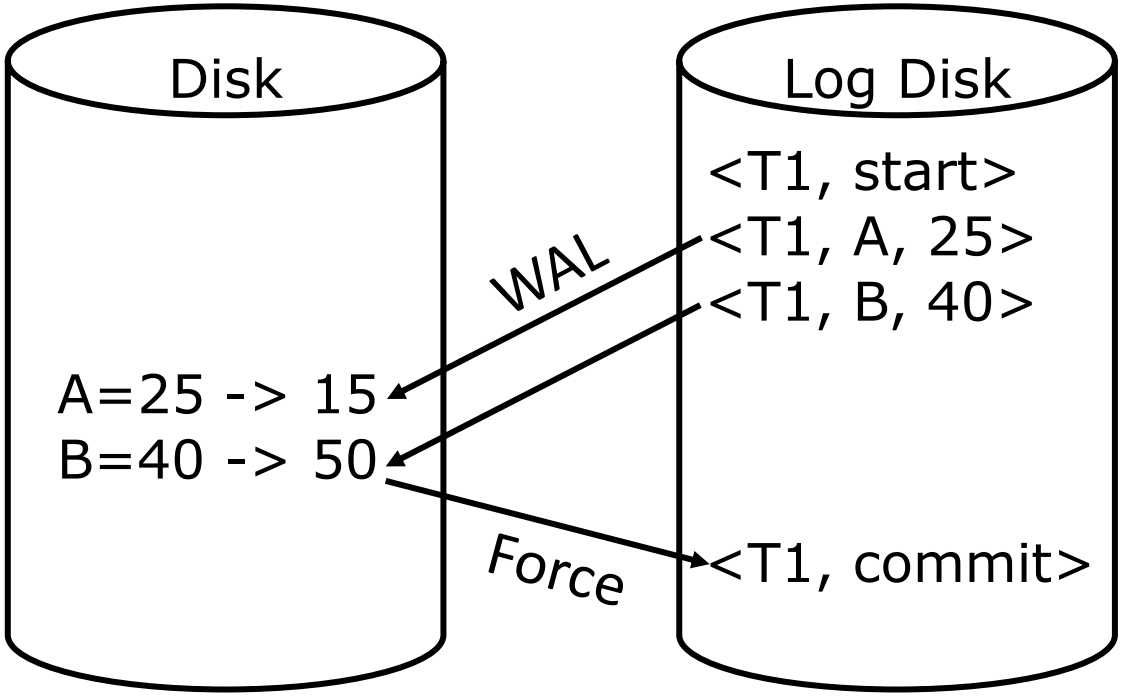
```
T1(A, B):
begin_tx
a = read(A)
write(A, a-10)
b = read(B)
write(B, b+10)
output(A)
output(B)
commit_tx
```

Memory

A=25 -> 15
B=40 -> 50

Log (in memory)

<T1, start>
<T1, A, 25>
<T1, B, 40>
<T1, commit>



Redo Based Write-Ahead-Logging

- Before Transaction T modifies X on disk, use WAL to flush its new value to the log
  - Log format: <Tid, X, new_value_of_X>
    - Tid is transaction id
    - X: physical address of X (block id, offset)
    - new_value_of_X: physical bits (physical logging)
- No steal: all log records (including commit record) must be flushed to the log, before any writes of transaction are flushed
  - If system crashes before transaction commits, no updates have been applied
  - If system crashes after transaction commits, redo updates to X on disk by using the new value of X from log

Redo Logging Example

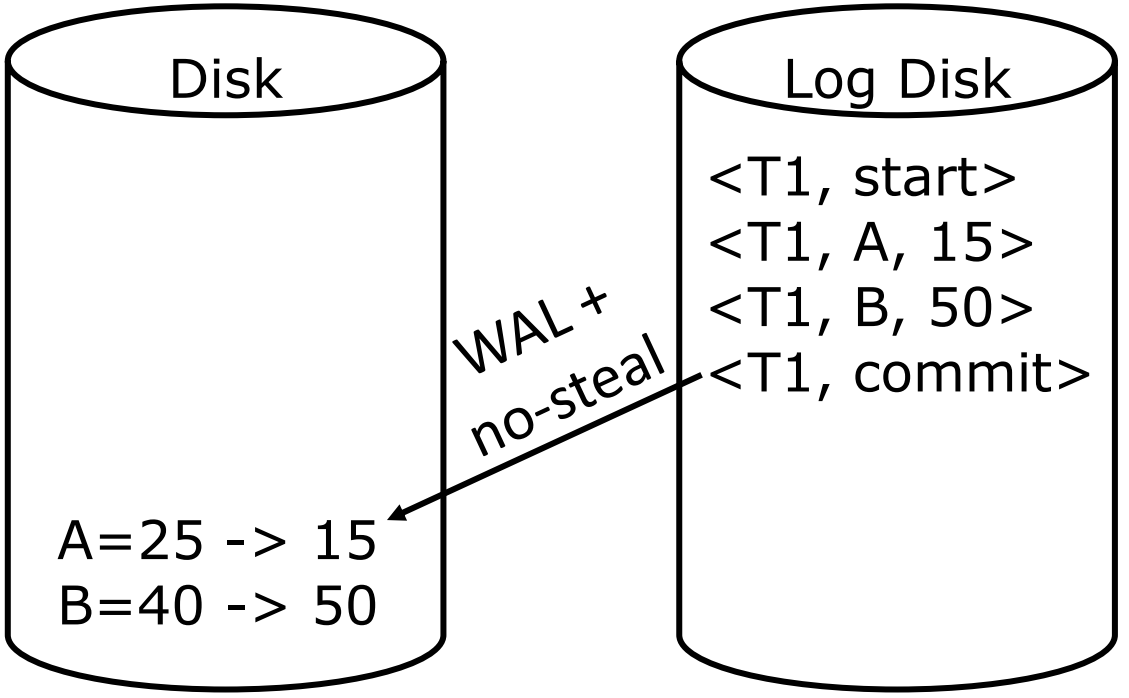
```
T1(A, B):
begin_tx
a = read(A)
write(A, a-10)
b = read(B)
write(B, b+10)
output(A)
output(B)
commit_tx
```

Memory

A=25 -> 15
B=40 -> 50

Log (in memory)

<T1, start>
<T1, A, 15>
<T1, B, 50>
<T1, commit>



Isolation

Isolation

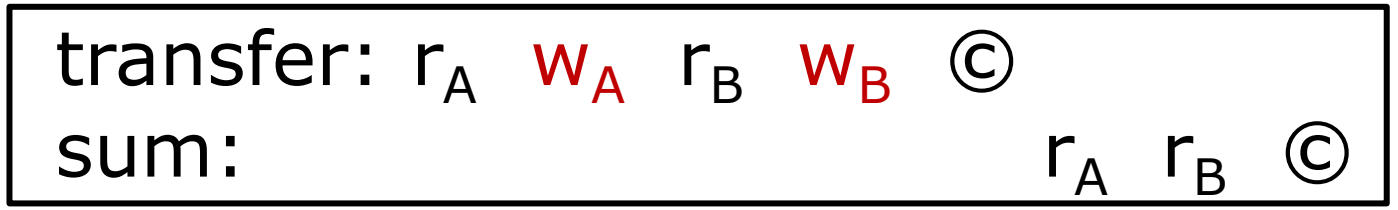
- Goal: operations in the transaction appear to happen together at a point in time
- Serial execution
  - All operations in a transaction are executed before another transaction is run, ensures isolation
  - Problem: poor performance, no concurrency possible
- Concurrent execution
  - Transactions are executed concurrently by interleaving their operations, provides good performance
  - Problem: certain interleavings of operations may violate isolation, need to avoid them

Serializability

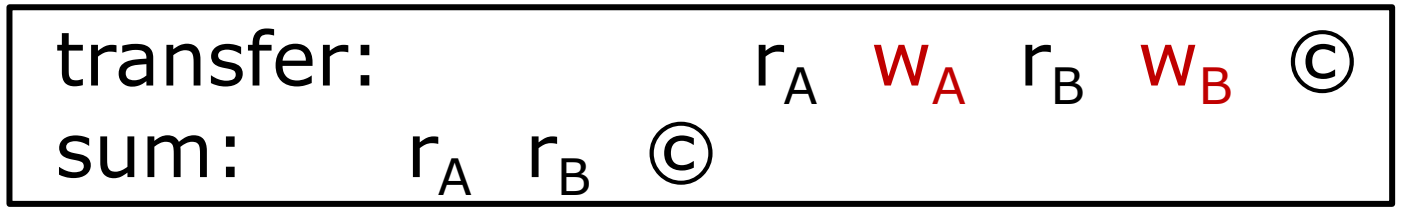
- A schedule for a set of transactions is an ordering of the operations (reads, writes) performed by those transactions
- A schedule is serializable if it is equivalent to some serial schedule
  - A serializable schedule provides isolation
  - i.e., ensures that the operations in a transaction appear to happen together in some serial order (even if they don't)

Schedules

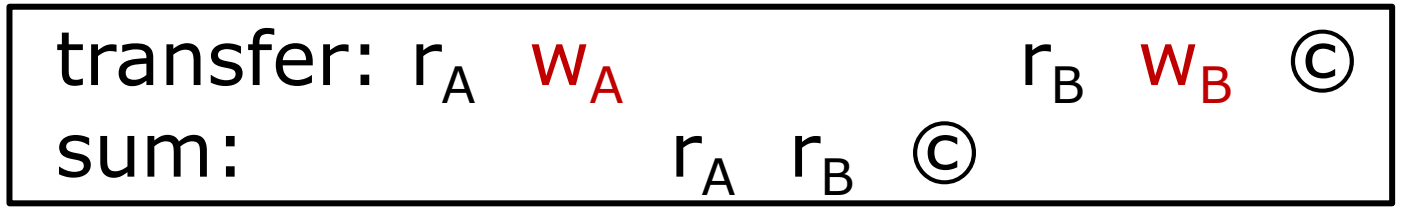
r_A: read row A
w_A: write row A
©: commit txn



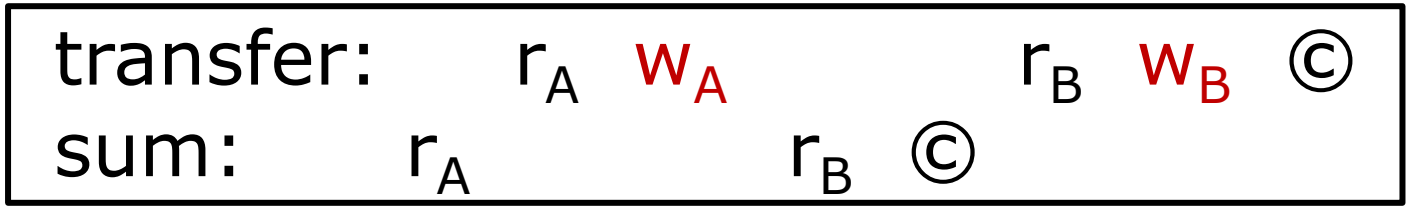
Serializable



Serializable



Non-Serializable

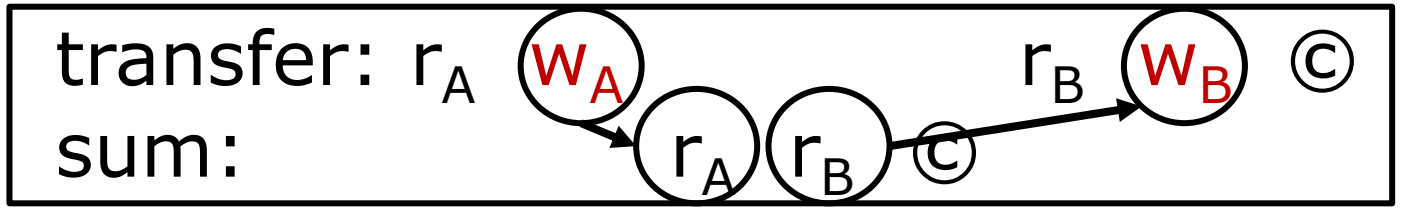


Serializable

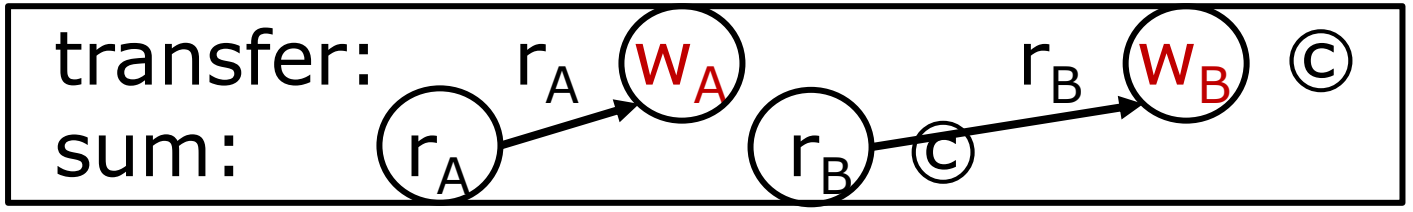
Conflicts

r_A: read row A
w_A: write row A
©: commit txn

- Two operations from different transactions are conflicting if they operate on the same item and at least one of them is write
  - read-write, write-read, write-write operations are conflicting because they are non-commutative
  - For serializability, conflicts must occur in same order



Non-Serializable



Serializable

Linearizability vs. Serializability

- Linearizability: a guarantee about single operations on single objects
  - Reads and writes have a total order
  - Once write completes, all reads that begin later (in real-time order) should reflect that write
- Serializability: a guarantee about multiple operations (transactions) on multiple objects
  - Transactions appear to execute in some serial order
  - Doesn't impose any real-time constraints
- Strict serializability: intuitively serializability + linearizability

Implementing Serializability with Locking

- Concurrent execution can violate serializability
  - We need to **control** concurrent execution to ensure serializability (i.e., so conflicts occur in same order), and so an implementation of isolation is also called concurrency control
- Traditionally, locking is used for concurrency control
- Two types of locks maintained for each data item
  - Shared: Acquire before reading object
  - Exclusive: Acquire before writing object

	Shared (S)	Exclusive (X)
Shared (S)	Yes	No
Exclusive (X)	No	No

Two-Phase Locking (2PL)

- 2PL rule: Once a transaction has released a lock it is not allowed to obtain any other locks
  - Growing phase: transaction acquires locks on its read and write set (i.e., items it reads and writes)
  - Shrinking phase: transaction releases locks

- In practice:
  - Growing phase is the entire transaction
  - Shrinking phase is after commit

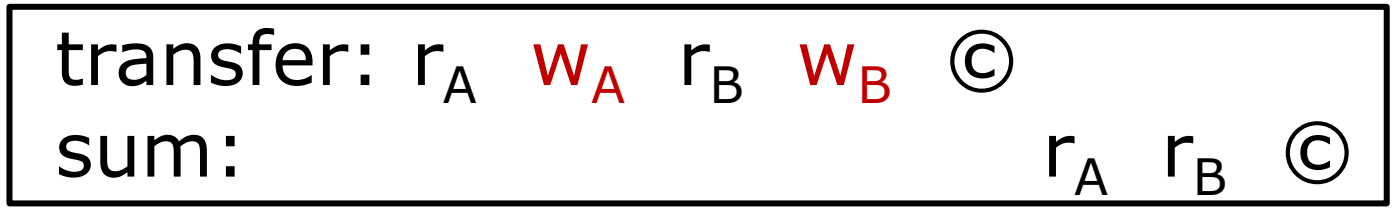
2PL Example

S(O): acquire shared lock on object O
X(O): acquire exclusive lock on object O
U(O): release lock on object O

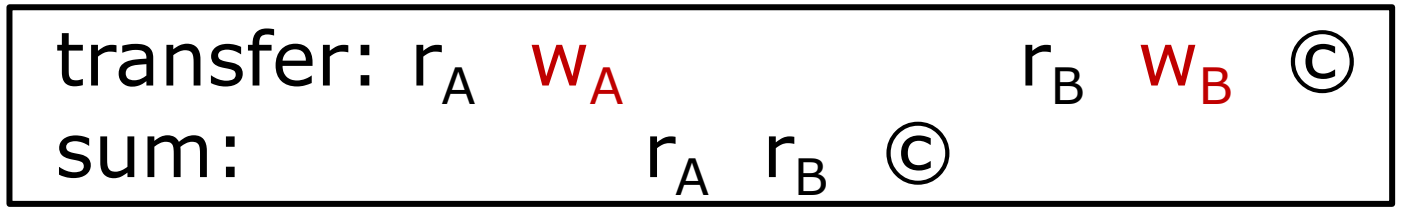
```
transfer(A, B):
begin_tx
a = read(A)          S(A)
if a < 10 then
   abort_tx          U(A)
else
   write(A, a-10)    X(A)
   b = read(B)       S(B)
   write(B, b+10)    X(B)
   commit_tx         U(A,B)
```

```
sum(A, B):
begin_tx
a = read(A)    S(A)
b = read(B)    S(B)
print a + b
commit_tx      U(A,B)
```

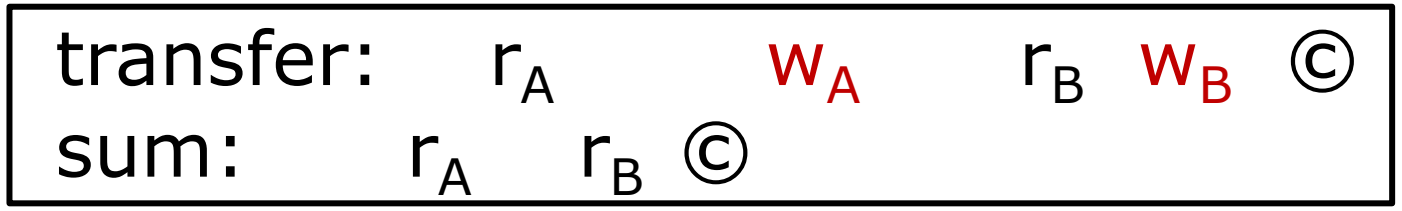
2PL Schedules



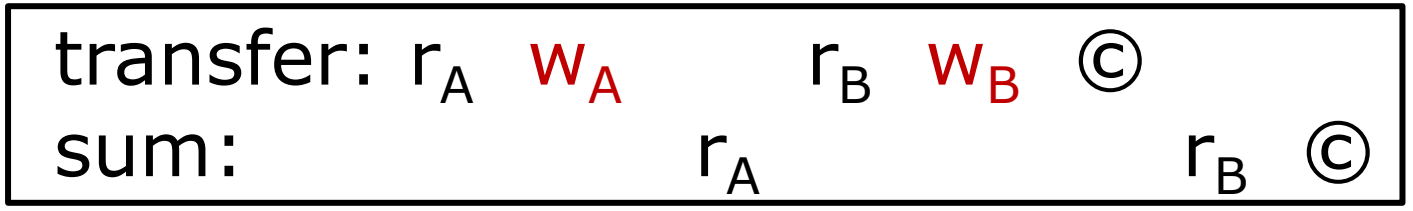
Serializable,
Allowed



Non-Serializable,
Not allowed



Serializable,
Allowed



Serializable,
Not allowed

Issues with 2PL

- What do we do if a lock is unavailable?
  - Wait: wait until lock becomes available?
  - Die: give up immediately, i.e., abort?
  - Wound: force the lock holder to abort to acquire lock?

- Waiting for a lock can result in deadlock
  - Transfer has A locked, waits on B
  - Sum has B locked, waits on A
    - Assuming order A and B are interchanged in the sum() code
  - Many ways to prevent, detect and handle deadlocks
  - Typically wait-die or wound-wait used for prevention

2PL is Pessimistic

- Acquires locks to prevent all potential violations of serializability
- But disallows many concurrent operations that are serializable

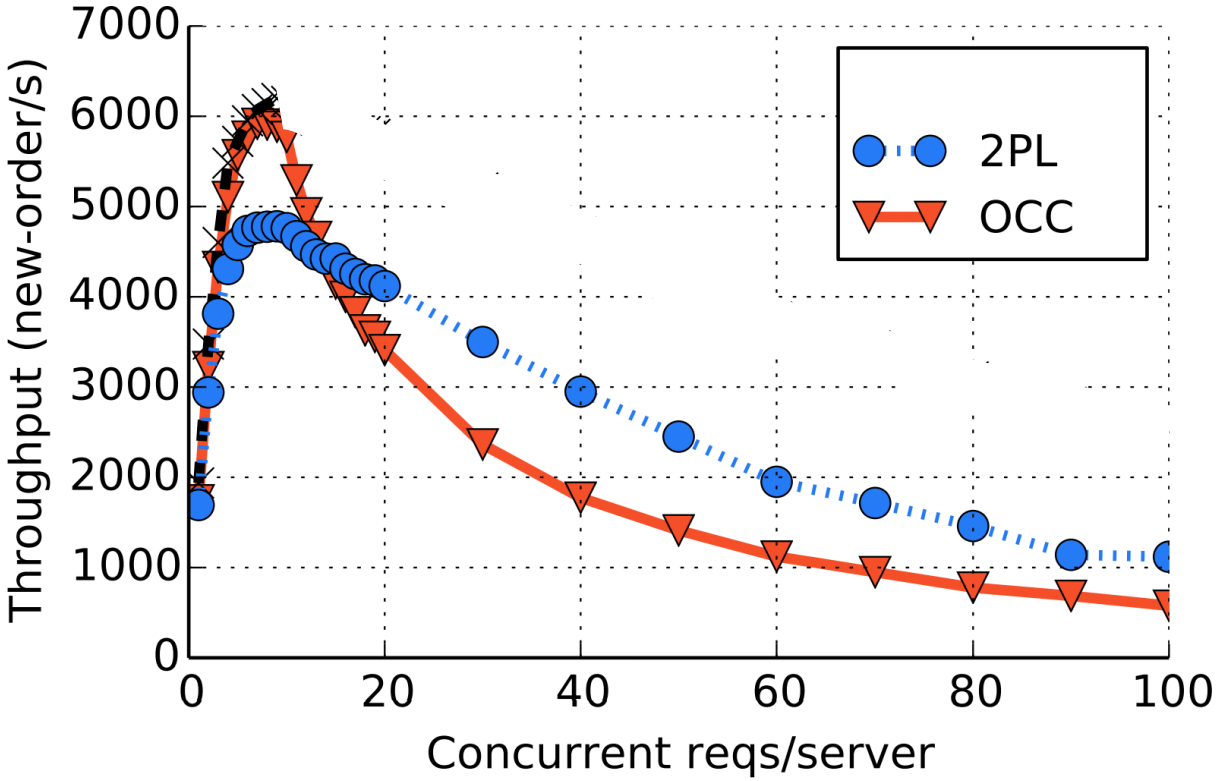
Be Optimistic!

- Assume success!
- Optimistic Concurrency Control (OCC)
  - Process transaction as if it will succeed
  - Check for serializability only at commit time
  - If check fails, abort transaction
- Compared to locking, OCC has
  - Higher performance when transactions have few conflicts
  - Lower performance when transactions have many conflicts

Optimistic Concurrency Control

- Optimistic execution
  - Transaction executes initial reads from database (read set)
    - Caches reads locally, re-reads from cache
  - Buffers writes locally (write set)
- Validation and Commit ← Many ways to do validation
  1. Acquire shared locks on read set, exclusive locks on write set
  2. Validate that data in read set hasn't changed
     - i.e., reading data in read set now would give the same result
  3. Apply buffered writes in write set to commit transaction
     - Else abort if locks can't be acquired in 1 or validation fails in 2
  4. Release locks

2PL vs OCC: Increasing Conflict Rate

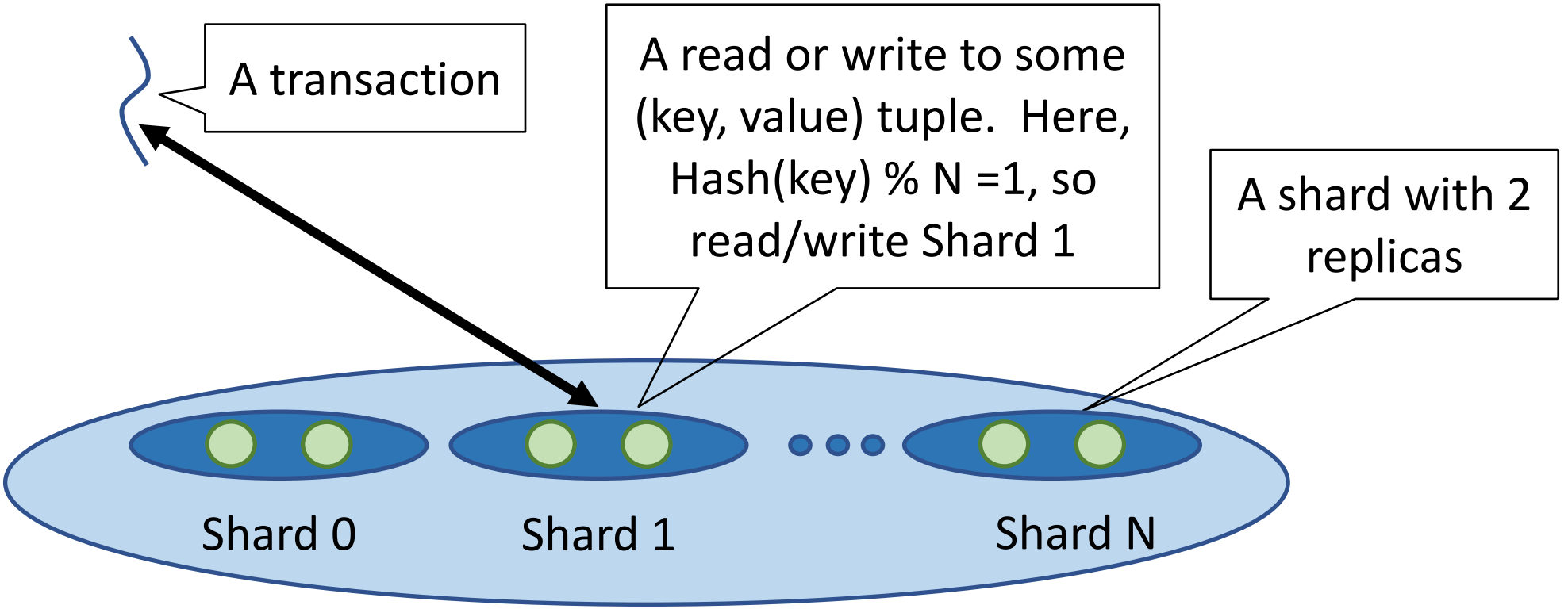


From Rococo, OSDI 2014

Distributed Transactions

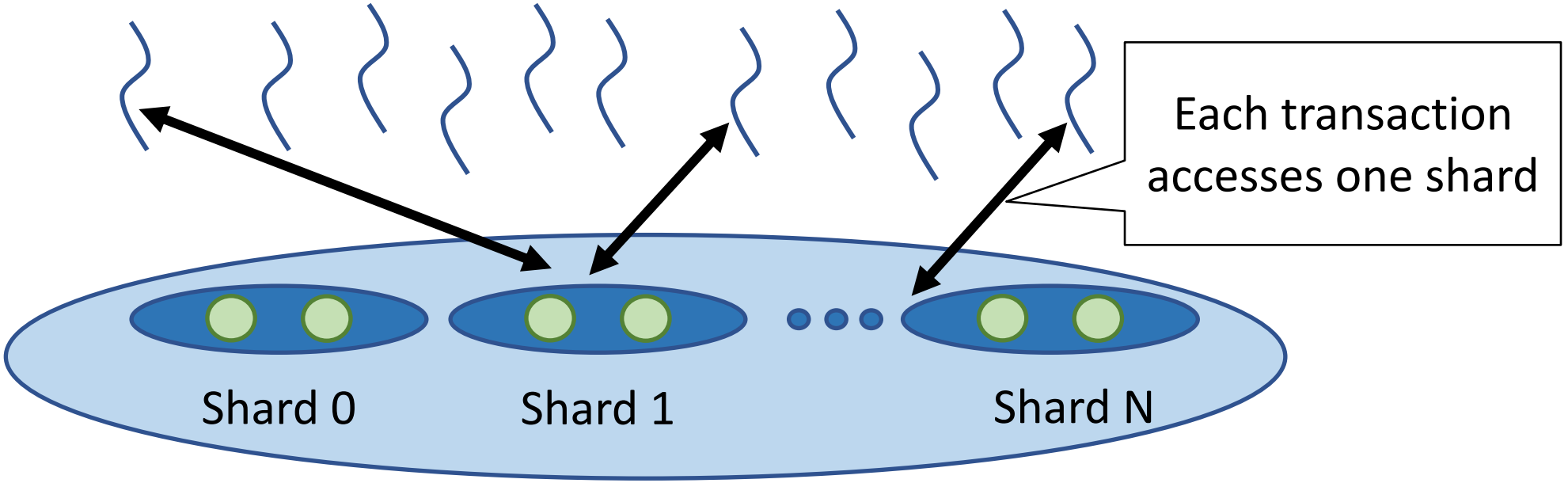
Recap: Sharding Data

- Data is partitioned (sharded) across nodes



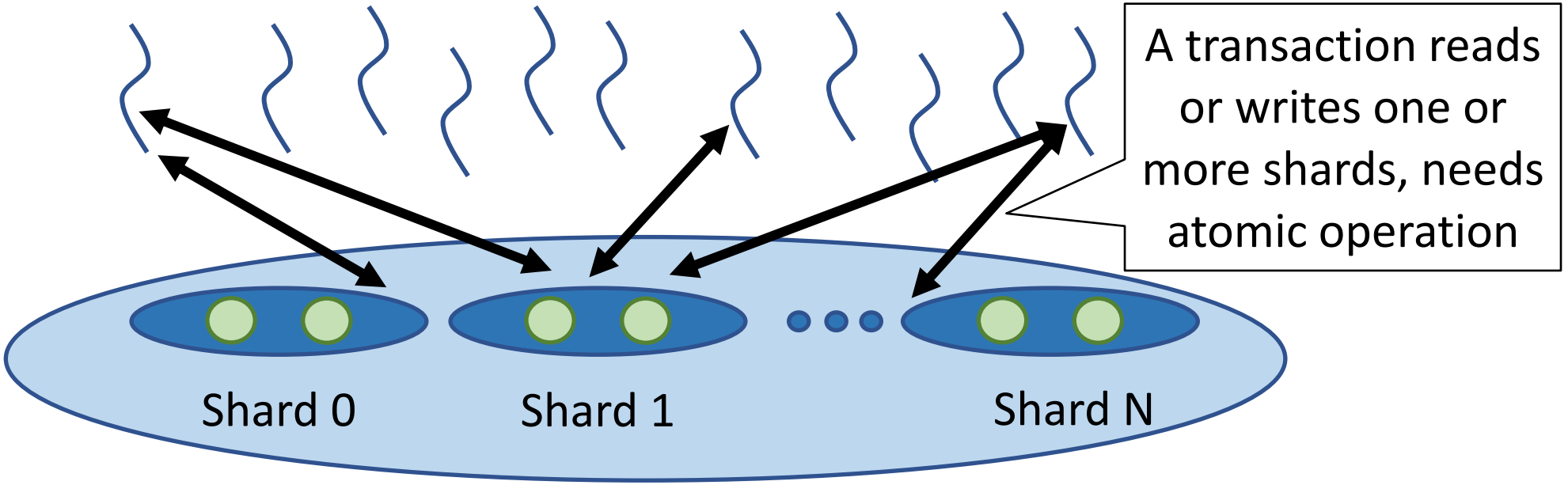
Sharded storage service with N shards,
2 replicated servers per shard

Single Node (Local) Transactions



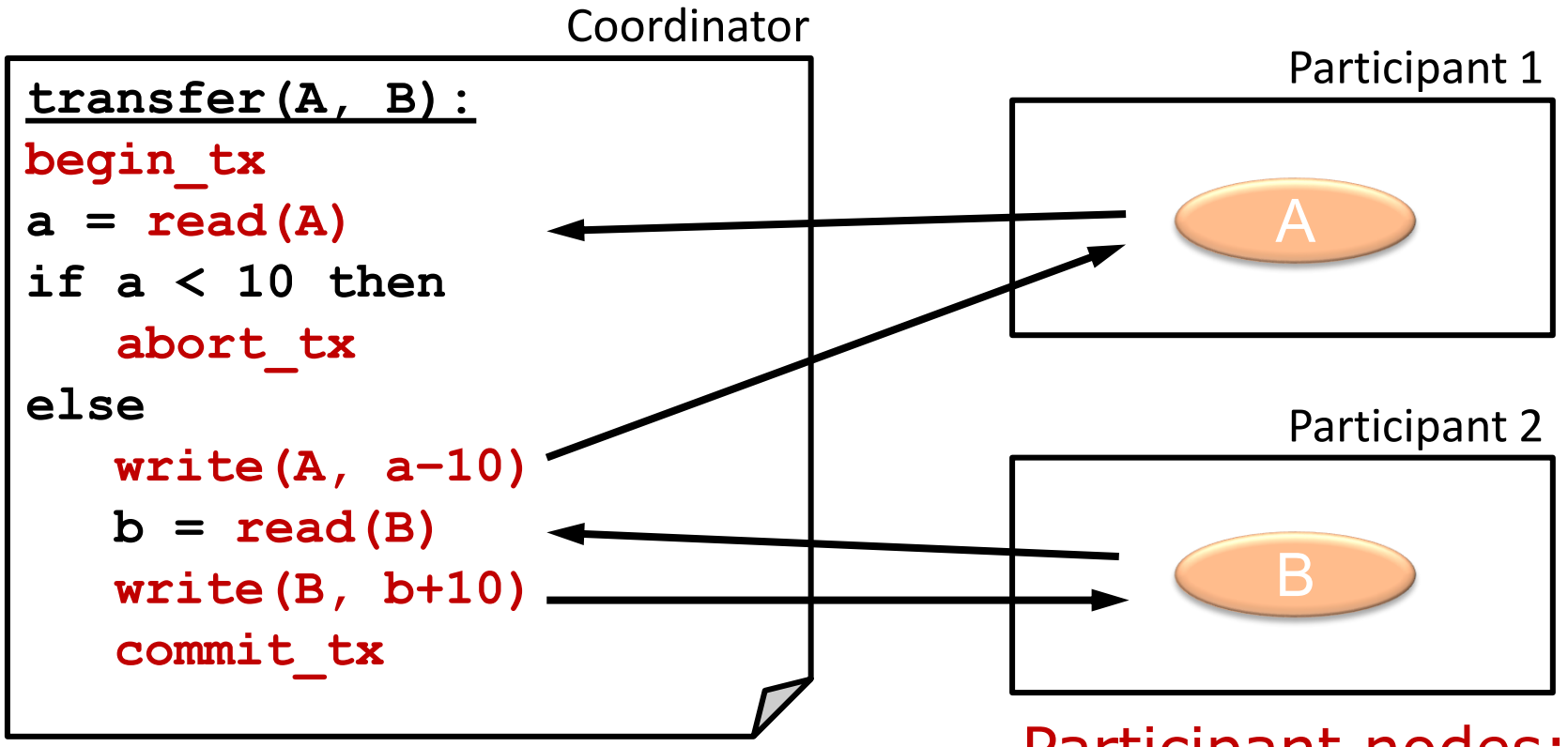
If each transaction does all its work at just one shard, never needing to access two or more shards, then sharding scales well

Distributed Transactions



Transactions that touch multiple shards hold locks for long time, need 2-phase commit (agreement protocol) for atomicity, hard to scale … let's see why in detail

Distributed Txn Execution Model



Coordinator node:
runs transaction code,
coordinates participants,
uses WAL for recovery

Participant nodes:
store transaction data,
acquire/release locks,
use WAL for recovery

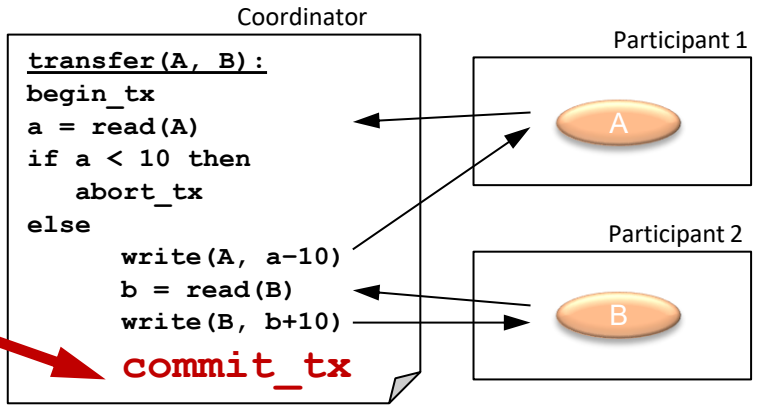
Atomic Commit

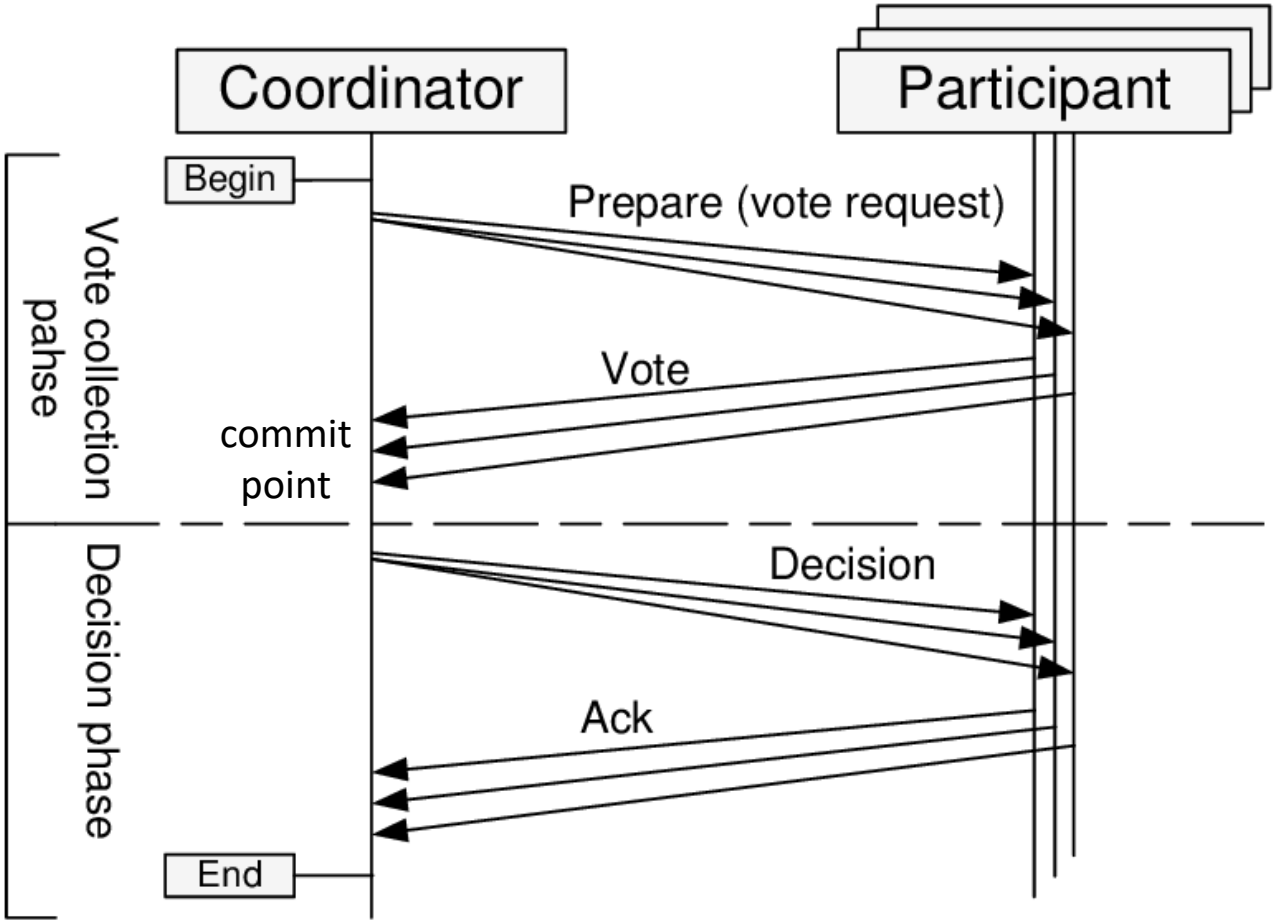
- Problem: Participant node may not be able to complete its operation
  - Cannot acquire required lock (e.g., deadlock)
  - No memory or disk space available to do write
  - Transaction constraint fails (e.g., a < 10)
  - Node crashes
- Atomic: All or nothing
  - Either all participants agree to commit (commit) or no participant does anything (abort)
    - i.e,. abort even if one participant says no

Why?

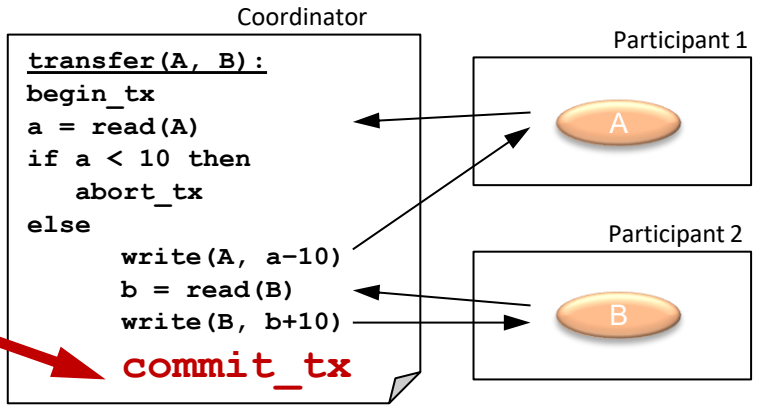
- Common use: commit a distributed transaction that updates data on different shards

Two-Phase Commit



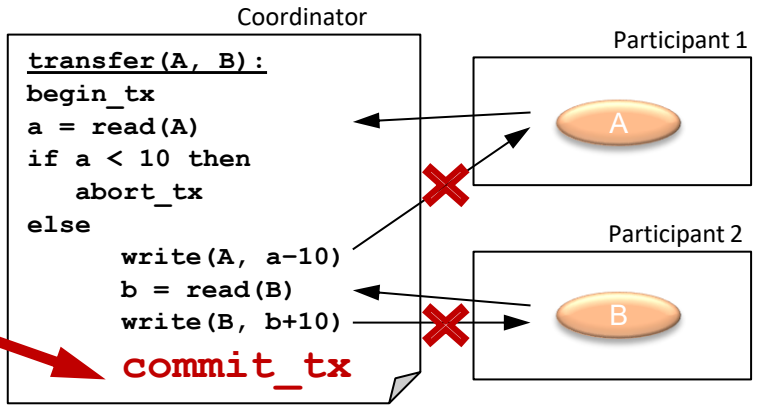


2PL Two-Phase Commit



- Phase 1
  - Coordinator sends Prepare requests to all participants
  - Each participant votes yes or no
    - Records vote in its log
    - Sends yes or no vote back to coordinator
- Coordinator inspects all votes
  - If all yes, then commit, else abort
  - Records commit/abort status in log (commit point)
- Phase 2
  - Coordinator sends Commit or Abort to all participants
  - Each participant commits or aborts changes
  - Each participant releases any locks it holds
  - Each participant sends an Ack back to the coordinator

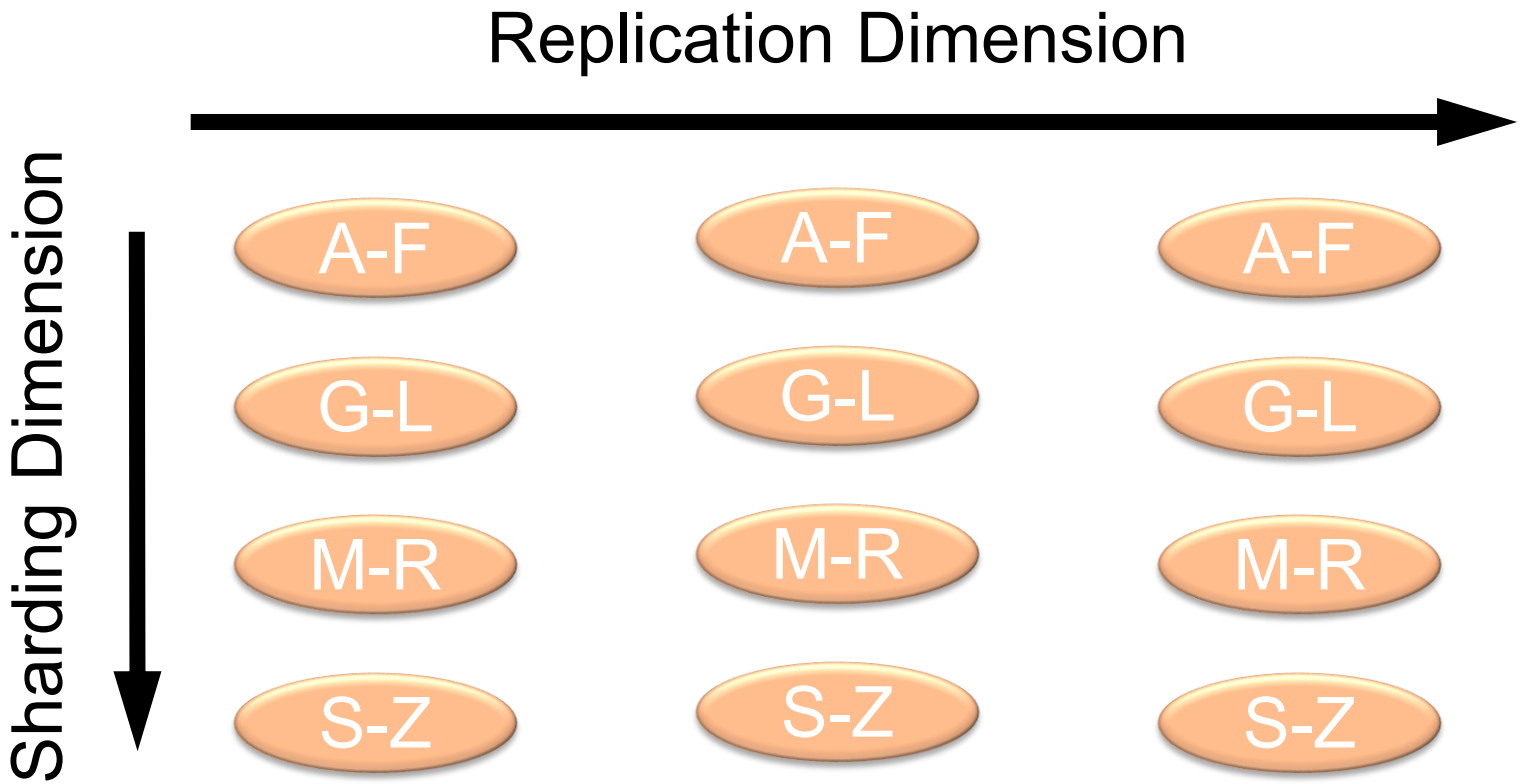
OCC Two-Phase Commit



- Phase 1
  - Coordinator sends Prepare requests to all participants
  - Prepare includes read values and buffered writes for each participant
    - Participant acquires shared locks on read set, exclusive locks on write set
    - Participant validates that data in read set hasn't changed
  - Each participant votes yes or no
    - Records vote in its log
    - Sends yes vote or no vote back to coordinator
- Coordinator inspects all votes
  - If all yes, then commit, else abort
  - Records commit/abort status in log (commit point)
- Phase 2
  - Coordinator sends Commit or Abort to all participants
  - Each participant commits or aborts changes
  - Each participant releases any locks it holds
  - Each participant sends an Ack back to the coordinator

OCC's validation and commit during 2PC

Distributed Transactions and Replication



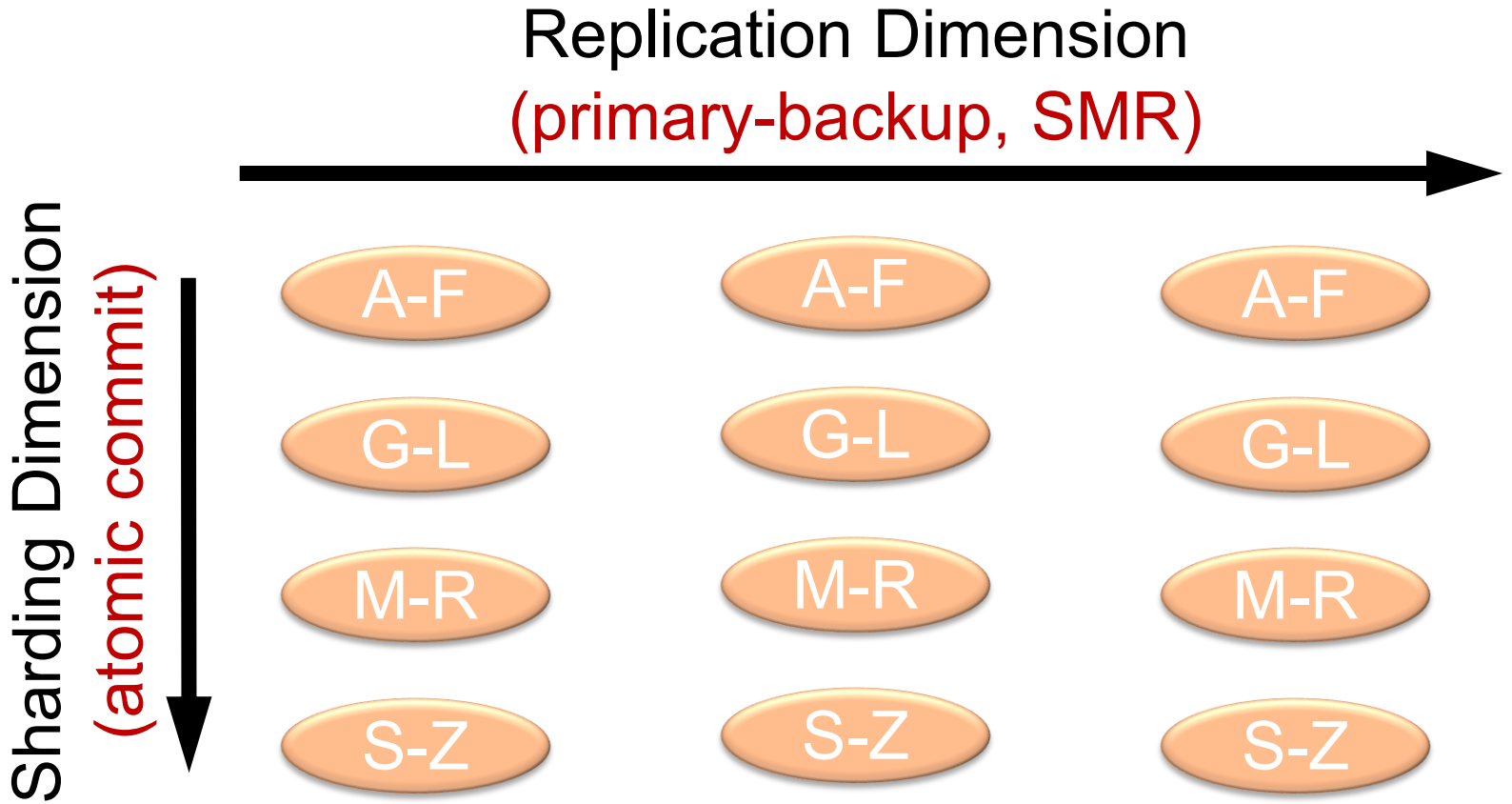
Replication, Sharding, Atomic Commit

- Replication (e.g., primary-backup, state-machine replication) is about doing the same thing in multiple places, primarily to provide fault tolerance

- Sharding is about doing different things in multiple places, primarily for scalability

- Atomic commit is about doing different things in multiple places together (all or nothing)

Distributed Transactions and Replication



Motivation for Today's Paper

- Distributed transactions are expensive
  - Two-phase commit requires two additional round trips, in addition to the read and write requests made to participants
  - Locks are held from the time reads and writes are performed until the end of the two-phase commit
  - Other transactions waiting on locks are also delayed

- Key idea: limit the power of transactions to enable scaling distributed transactions