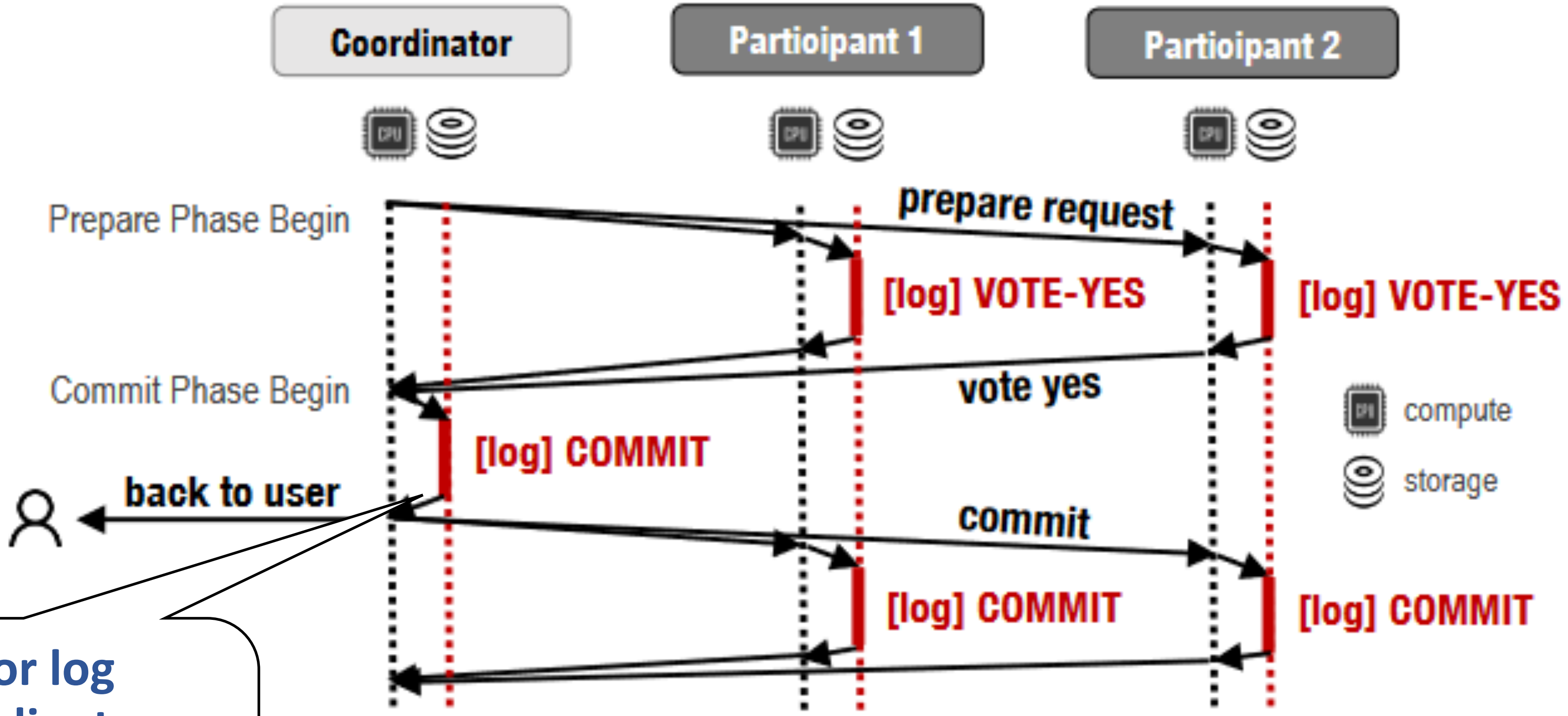
CORNUS: ATOMIC COMMIT FOR A CLOUD DBMS WITH STORAGE DISAGGREGATION

- Sumedha Joshirao

WHY CLOUD DATABASES?

- Elasticity
- High availability
- Cost Competitiveness

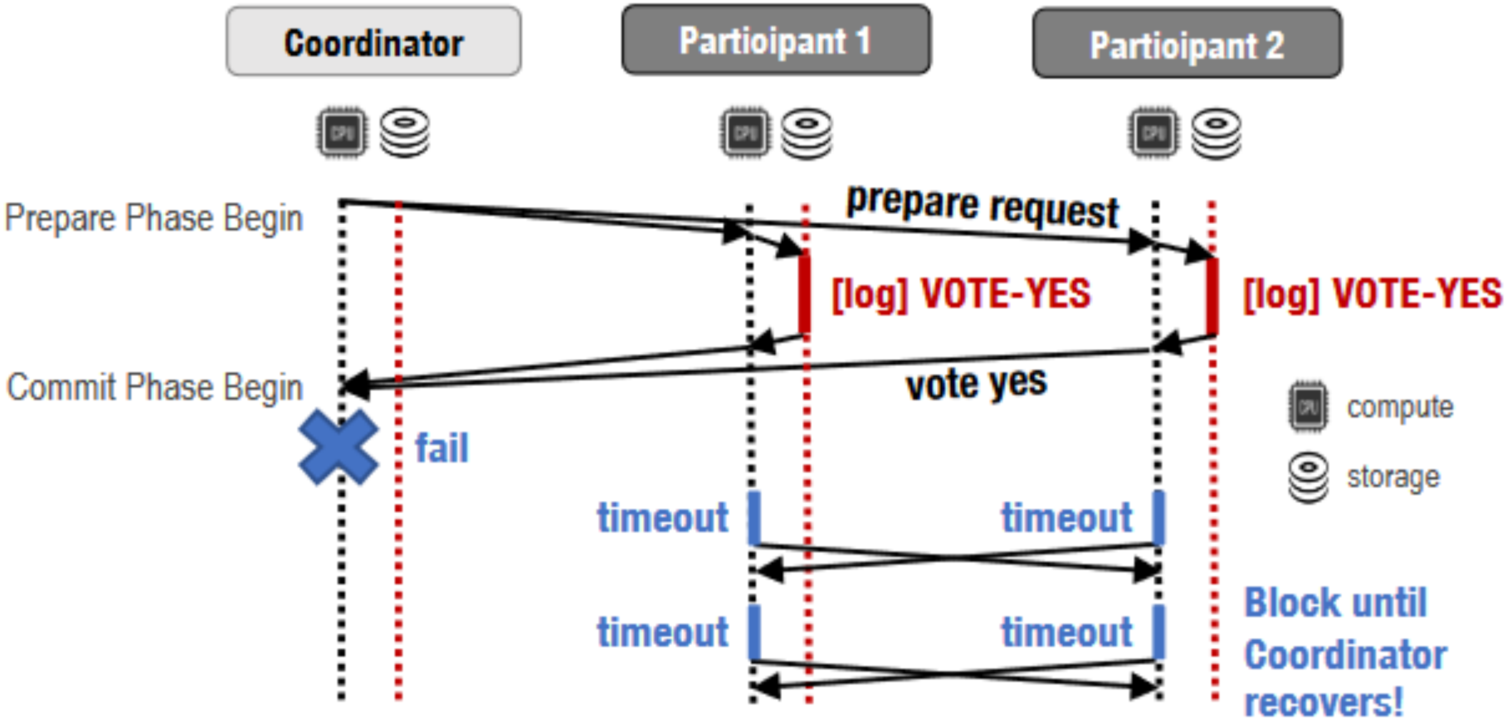
TWO PHASE COMMIT PROTOCOL



- **Coordinator log delay: Coordinator must durably log the decision before reply**

(a) 2PC with no failure.

TWO PHASE COMMIT PROTOCOL



(b) 2PC with coordinator failure (cooperative termination protocol).

ISSUES WITH 2PC PROTOCOL

Latency Issue:

- Average latency of one network round-trip and two logging operations

Blocking

- Decision postponed till coordinator is restored

ISSUES WITH EXISTING SOLUTIONS THAT TRY TO SOLVE THESE PROBLEMS

- Targeted shared-nothing architecture
- Need to Customize storage
- Make strong assumptions about the workload
  - May not be practical for disintegrated storage
  - Mitigate blocking by adding an extra phase -> increase latency

SOLUTION: CORNUS

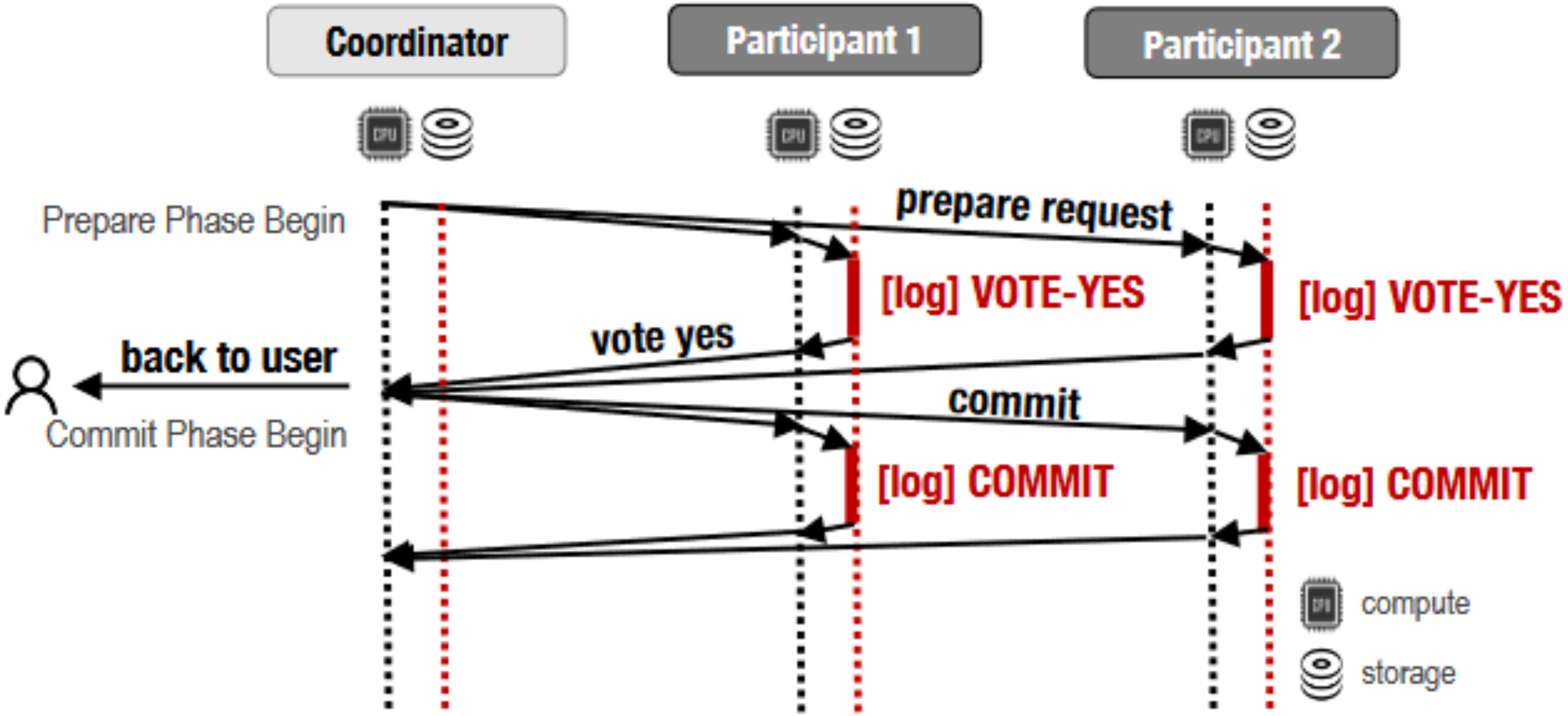
Non-blocking

Low-latency 2PC variant

Only new storage layer function needed is LogOnce()

- Implemented using compare and swap

ILLUSTRATION OF CORNUS



CORNUS APIs

- Remote Procedural Calls (RPC) – communication between participants
    A. Log(txn,type): appends log record of certain type to the end of txn’s log
    B. LogOnce(txn,type): guarantees that transaction’s state is written at the most once

ALGORITHM

```
Function Participant::StartCornus(txn)
    wait for VOTE-REQ from coordinator
        on timeout RPC_sync^{local log}::Log(ABORT) return
    if participant votes yes for txn then
        resp ← RPC_sync^{local log}::LogOnce(VOTE-YES)
        if resp is ABORT then
            # Another participant has logged ABORT for it
            reply ABORT to coordinator
        else
            reply VOTE-YES to coordinator
            wait for decision from coordinator
                on timeout decision ← TerminationProtocol(txn)
            RPC_sync^{local log}::Log(decision)
    else
        RPC_async^{local log}::Log(ABORT)
        reply ABORT to coordinator
```

```
Function Coordinator::StartCornus(txn)
    for p in txn.participants do
        send VOTE-REQ to p asynchronously
    wait for all responses from participants
        on receiving ABORT decision ← ABORT
        on receiving all responses decision ← COMMIT
        on timeout decision ← TerminationProtocol(txn)
    reply decision to the txn caller
    for p in txn.participants do
        send decision to p asynchronously
```

```
Function TerminationProtocol(txn)
    for every paticipant p other than self do
        RPC_async^{p.log}::LogOnce(ABORT)
    wait for responses
        on receiving ABORT decision ← ABORT
        on receiving COMMIT decision ← COMMIT
        on receiving all responses decision ← COMMIT
        on timeout retry from the beginning
    return decision
```

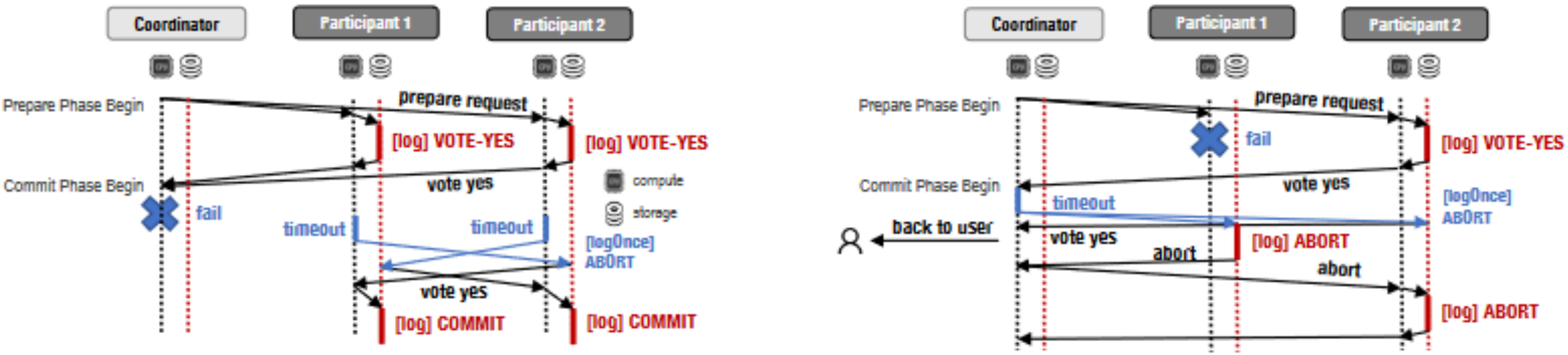
FAILURE AND RECOVERY

- **Coordinator Failure:**
  - **Case 1:** FAILURE BEFORE PROTOCOL STARTS
  - **Case 2:** FAILURE AFTER SENDING SOME BUT NOT ALL VOTE REQUESTS
  - **Case 3:** FAILURE AFTER SENDING ALL VOTE REQUESTS BUT BEFORE SENDING DECISION
  - **Case 4:** FAILURE AFTER SENDING DECISION TO SOME BUT NOT AL
  - **Case 5:** FAILURE AFTER SENDING DECISION TO ALL PARTICIPANTS

FAILURE AND RECOVERY

- **Participant Failure:**
  - **Case 1**: FAILS BEFORE RECEIVING VOTE REQUEST
  - **Case 2:** FAILURE BEFORE LOGGING VOTE BUT AFTER RECEIVING VOTE REQUEST
  - **Case 3:** FAILURE AFTER LOGGING THE VOTE, BEFORE REPLYING TO COORDINATOR
  - **Case 4:** FAILURE AFTER SENDING VOTE

EXAMPLE



(a) Coordinator fails before sending decision

(b) Participant fails before logging vote

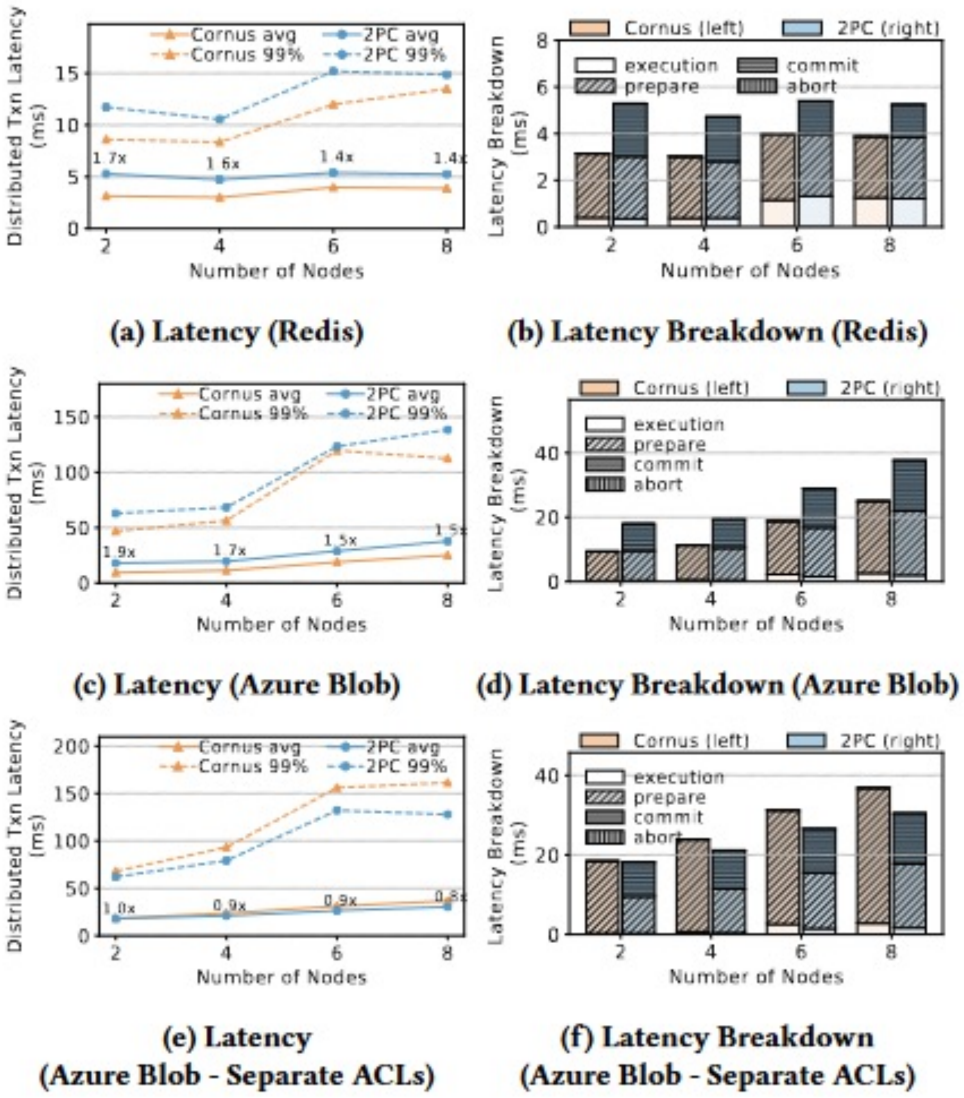
Figure 4: Cornus under Failures — The behavior of Cornus under two failures scenarios.

EXPERIMENTAL ANALYSIS

- **SETUP**
  - Cloud Storage Services: Microsoft Azure Blob Storage, Microsoft Azure Cache for Redis
  - Workloads:
    - Yahoo! Cloud Serving Benchmark
    - 10 GB data partitions -> 1 KB Tuples
    - Each transaction -> 16 tuples with 50 % reads and 50% writes
  - Parameter Setup
    - Maximum 8 compute nodes
    - Eight worker threads per node execute transaction logic
    - Eight worker threads per node serve remote requests

SCALABILITY

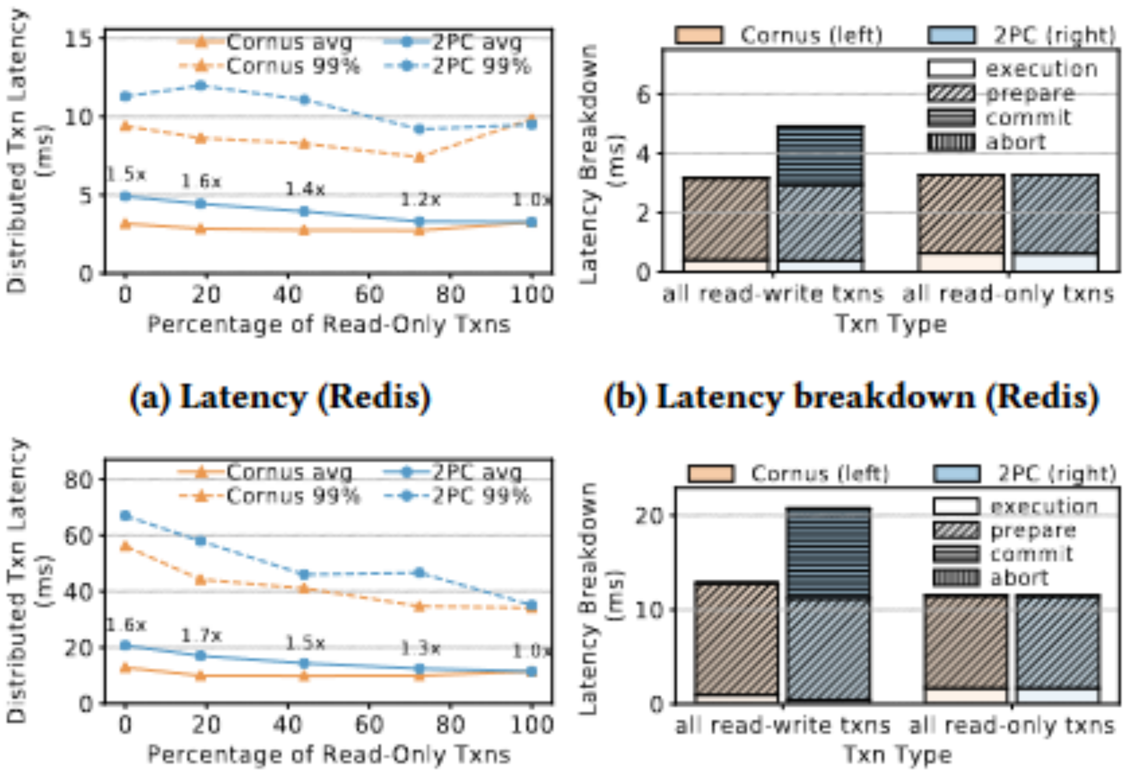
- As nodes increase latency of both 2PC and Cornus increases linearly
- Speedup of Cornus over 2PC on average latency slightly decreases as the number of nodes increase
- Current version of Azure Blob cannot benefit from Cornus for applications that want separate access control between data and transaction states



(a) Latency (Redis)
(b) Latency Breakdown (Redis)
(c) Latency (Azure Blob)
(d) Latency Breakdown (Azure Blob)
(e) Latency (Azure Blob - Separate ACLs)
(f) Latency Breakdown (Azure Blob - Separate ACLs)

PERCENTAGE OF R-ONLY TRANSACTIONS

- Improvements of Cornus increases with decrease in percentage of R-only transactions -> 1.7 times improvement
- Improves latency for RW transactions
- Spends more time in prepare phase



(a) Latency (Redis)

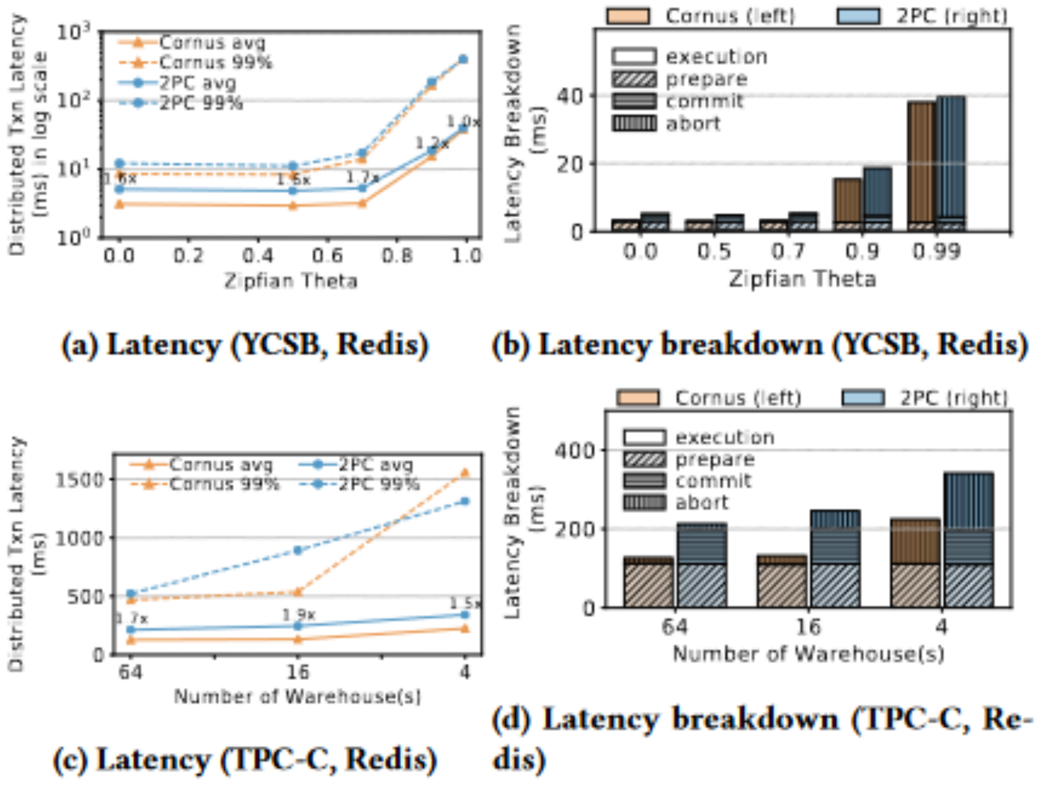
(b) Latency breakdown (Redis)

(c) Latency (Azure Blob)

(d) Latency breakdown (Azure Blob)

CONTENTION

- Provides less improvement under high contention as abort time dominates the total transaction elapsed time

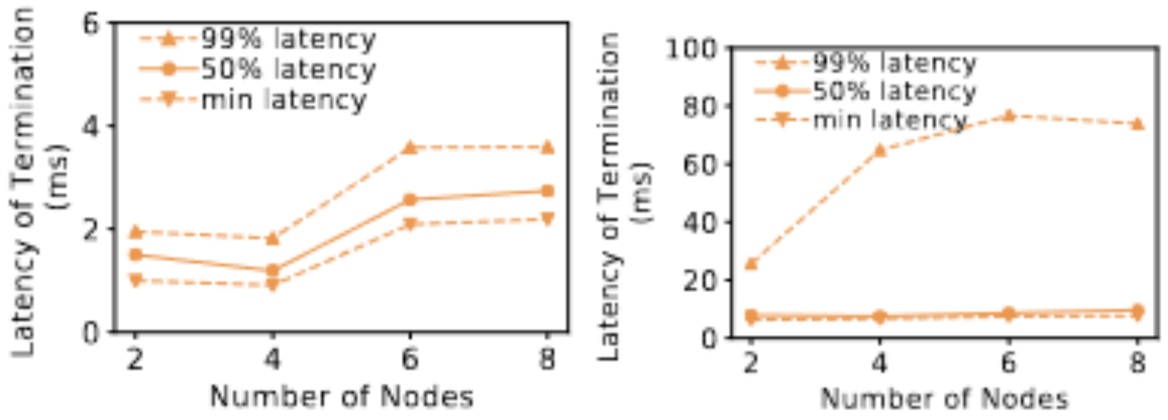


(a) Latency (YCSB, Redis)
(b) Latency breakdown (YCSB, Redis)
(c) Latency (TPC-C, Redis)
(d) Latency breakdown (TPC-C, Redis)

Figure 7: Varying workload contention

TIME TO TERMINATE TRANSACTIONS ON FAILURE

- Always terminates transaction in 4ms upto 8ms on Redis and upto 20 ms on Azure Blob
- Tail latency of Azure Blob increases more than Redis as number of nodes increases



(a) Latency (Redis)

(b) Latency (Azure Blob)

CONCLUSION

- Cornus solves the long latency and blocking problem in 2PC
- Evaluations show a speedup of 1.9x in latency

Questions?

THANK YOU!!!