Distributed OLTP Databases
Homework #5: Will be released on Monday Nov 22\textsuperscript{nd}. It is due Dec 2\textsuperscript{nd} @ 11:59pm.

Project #4: Will be released today. It is due Dec 5\textsuperscript{th} @ 11:59pm.
Query Optimization and Acceleration at Dremio

→ Mon Nov 22\textsuperscript{ed} @ 4:30pm ET
LAST CLASS

System Architectures
→ Shared-Memory, Shared-Disk, Shared-Nothing

Partitioning/Sharding
→ Hash, Range, Round Robin

Transaction Coordination
→ Centralized vs. Decentralized
OLTP VS. OLAP

On-line Transaction Processing (OLTP):
→ Short-lived read/write txns.
→ Small footprint.
→ Repetitive operations.

On-line Analytical Processing (OLAP):
→ Long-running, read-only queries.
→ Complex joins.
→ Exploratory queries.
DECENTRALIZED COORDINATOR
DECENTRALIZED COORDINATOR

Application Server

Begin Request

Primary Node

Partitions

P1

P2

P3

P4
DECENTRALIZED COORDINATOR

Application Server

Primary Node

Partitions

P1

P2

P3

P4

Query

Query

Query
DECENTRALIZED COORDINATOR

Application Server

Primary Node

Partitions

P1

P2

P3

P4
DECENTRALIZED COORDINATOR

Application Server

Commit Request

Primary Node

Partitions

P1

P2

P3

P4
DECENTRALIZED COORDINATOR

Application Server

Commit Request

Safe to commit?

Primary Node

Partitions

P1

P2

P3

P4
OBSERVATION

We have not discussed how to ensure that all nodes agree to commit a txn and then to make sure it does commit if we decide that it should.

→ What happens if a node fails?
→ What happens if our messages show up late?
→ What happens if we don't wait for every node to agree?
IMPORTANT ASSUMPTION

We can assume that all nodes in a distributed DBMS are well-behaved and under the same administrative domain.

→ If we tell a node to commit a txn, then it will commit the txn (if there is not a failure).

If you do not trust the other nodes in a distributed DBMS, then you need to use a Byzantine Fault Tolerant protocol for txns (blockchain).
TODAY'S AGENDA

Atomic Commit Protocols
Replication
Consistency Issues (CAP)
Federated Databases
When a multi-node txn finishes, the DBMS needs to ask all the nodes involved whether it is safe to commit.

Examples:
→ Two-Phase Commit
→ Three-Phase Commit (not used)
→ Paxos
→ Raft
→ ZAB (Apache Zookeeper)
→ Viewstamped Replication
When a multi-node txn finishes, the DBMS needs to ask all the nodes involved whether it is safe to commit.

Examples:
- Two-Phase Commit
- Three-Phase Commit (not used)
- Paxos
- Raft
- ZAB (Apache Zookeeper)
- Viewstamped Replication
TWO-PHASE COMMIT (SUCCESS)
TWO-PHASE COMMIT (SUCCESS)
TWO-PHASE COMMIT (SUCCESS)

Application Server

Commit Request

Coordinator

Node 1

Node 2

Node 3
TWO-PHASE COMMIT (SUCCESS)

Application Server → Commit Request → Coordinator → Node 1

Participant

Node 2

Participant

Node 3
TWO-PHASE COMMIT (SUCCESS)

Commit Request

Phase 1: Prepare

Application Server

Coordinator

Node 1

Participant

Node 2

Participant

Node 3
TWO-PHASE COMMIT (SUCCESS)

Application Server

Commit Request

Phase 1: Prepare

Commit Request

OK

OK

Node 1

Coordinator

Node 2

Participant

Node 3

Participant

Application Server
**TWO-PHASE COMMIT (SUCCESS)**

Application Server

Commit Request

Phase 1: Prepare

Phase 2: Commit

Coordinator

Node 1

Node 2

Node 3

Commit Request

OK

Participant

OK

Participant
TWO-PHASE COMMIT (SUCCESS)

Application Server

Commit Request

Phase 1: Prepare

Phase 2: Commit

Commit Request

Node 1

Node 2

Node 3
TWO-PHASE COMMIT (SUCCESS)

Application Server

Coordinator

Node 1

Success!

Node 2

Participant

Node 3

Participant
TWO-PHASE COMMIT (ABORT)
TWO-PHASE COMMIT (ABORT)

Application Server

Coordinator

Node 1

Commit Request

Participant

Node 2

Participant

Node 3
TWO-PHASE COMMIT (ABORT)

Commit Request

Phase 1: Prepare

Application Server

Coordinator

Node 1

Participant

Participant

Node 2

Node 3
TWO-PHASE COMMIT (ABORT)

Commit Request

Phase 1: Prepare

ABORT!
TWO-PHASE COMMIT (ABORT)

Application Server

Node 1

Coordinator

Node 2

Participant

Aborted

Node 3

Participant

ABORT!
TWO-PHASE COMMIT (ABORT)

Application Server

Coordinator

Node 1

Phase2: Abort

Participant

Node 2

Participant

Node 3

Aborted

ABORT!
TWO-PHASE COMMIT (ABORT)

Application Server

Node 1

Coordinator

Phase2: Abort

Aborted

Node 2

Node 3

Participant

Participant

OK

ABORT!

OK

ABORT!

OK

ABORT!

OK

ABORT!
2PC OPTIMIZATIONS

Early Prepare Voting
→ If you send a query to a remote node that you know will be the last one you execute there, then that node will also return their vote for the prepare phase with the query result.

Early Acknowledgement After Prepare
→ If all nodes vote to commit a txn, the coordinator can send the client an acknowledgement that their txn was successful before the commit phase finishes.
EARLY ACKNOWLEDGEMENT

Application Server

Coordinator

Node 1

Participant

Node 2

Participant

Node 3
EARLY ACKNOWLEDGEMENT

Application Server

Commit Request

Coordinator

Node 1

Node 2

Node 3
Commit Request

Phase1: Prepare
EARLY ACKNOWLEDGEMENT

Application Server

Commit Request

Phase1: Prepare

Coordinator

Node 1

OK

Node 2

OK

Node 3

Commit Request
**EARLY ACKNOWLEDGEMENT**

Phase 1: Prepare

Success!

Application Server

Coordinator

Node 1

Node 2

Node 3

Participant

Participant

Participant
EARLY ACKNOWLEDGEMENT

Phase 1: Prepare

Phase 2: Commit

Application Server

Coordinator

Node 1

OK

Node 2

OK

Node 3

Success!
EARLY ACKNOWLEDGEMENT

Application Server

Coordinator

Phase 1: Prepare

Phase 2: Commit

Success!

Node 1

Node 2

Node 3

Participant

Participant

Participant

Application Server

Coordinator

Phase 1: Prepare

Phase 2: Commit

Success!

Node 1

Node 2

Node 3

Participant

Participant

Participant
TWO-PHASE COMMIT

Each node records the outcome of each phase in a non-volatile storage log.

What happens if coordinator crashes?

What happens if participant crashes?
Each node records the outcome of each phase in a non-volatile storage log.

What happens if coordinator crashes?
→ Participants must decide what to do.

What happens if participant crashes?
Each node records the outcome of each phase in a non-volatile storage log.

What happens if coordinator crashes?
→ Participants must decide what to do.

What happens if participant crashes?
→ Coordinator assumes that it responded with an abort if it hasn't sent an acknowledgement yet.
Consensus protocol where a coordinator proposes an outcome (e.g., commit or abort) and then the participants vote on whether that outcome should succeed. Does not block if a majority of participants are available and has provably minimal message delays in the best case.
PAXOS

Application Server

Node 1

Commit Request

Node 2

Node 3

Node 4
PAXOS

Commit Request

Application Server

Proposer

Node 1

Node 2

Node 3

Node 4
PAXOS

Application Server

Proposer

Node 1

Commit Request

Node 2

Node 3

Node 4

Acceptor

Acceptor

Acceptor
PAXOS

Application Server

Proposer

Node 1

Commit Request

Propose

Node 2

Node 3

Node 4

Acceptor

Acceptor

Acceptor

Acceptor
PAXOS

Application Server

Commit Request

Propose

Agree

Node 1

Node 2

Node 3

Node 4
PAXOS

Application Server

Node 1

Proposer

Commit

Node 2

Acceptor

Node 3

Propose

Node 4

Commit Request

Agree

Accept

Agree

Accept
PAXOS

Application Server

Proposer

Node 1

Success!

Node 2

Node 3

Node 4

Acceptors
PAXOS

Proposer

Acceptors

Proposer

TIME
PAXOS

Proposer

**Propose(n)**

Acceptors

Proposer

TIME
Proposer

Propose(n)

Acceptors

Agree(n)

Propose(n+1)

Proposer

TIME
PAXOS

Proposer

Propose($n$)

Agree($n$)

Commit($n$)

Reject($n$, $n+1$)

Proposer

Propose($n+1$)

TIME
PAXOS

Proposer

Propose(n)

Commit(n)

Propose(n+1)

TIME

Agree(n)

Reject(n,n+1)

Propose(n+1)

Acceptors

Agree(n+1)

Proposer

CMU-DB
15-445/645 (Fall 2021)
PAXOS

Proposer
- Propose\( (n) \)
- Commit\( (n) \)

Acceptors
- Agree\( (n) \)
- Reject\( (n, n+1) \)
- Agree\( (n+1) \)

Proposer
- Propose\( (n+1) \)
- Commit\( (n+1) \)

TIME
PAXOS

Proposer

Acceptors

Proposer

Propose(n)

Agree(n)

Propose(n+1)

Commit(n)

Reject(n,n+1)

Agree(n+1)

Accept(n+1)

Commit(n+1)
If the system elects a single leader that oversees proposing changes for some period, then it can skip the Propose phase.

→ Fall back to full Paxos whenever there is a failure.

The system periodically renews who the leader is using another Paxos round.

→ Nodes must exchange log entries during leader election to make sure that everyone is up-to-date.
Two-Phase Commit
→ Blocks if coordinator fails after the prepare message is sent, until coordinator recovers.

Paxos
→ Non-blocking if a majority participants are alive, provided there is a sufficiently long period without further failures.
REPLICATION

The DBMS can replicate data across redundant nodes to increase availability.

Design Decisions:
→ Replica Configuration
→ Propagation Scheme
→ Propagation Timing
→ Update Method
REPLICA CONFIGURATIONS

Approach #1: Primary-Replica
→ All updates go to a designated primary for each object.
→ The primary propagates updates to its replicas without an atomic commit protocol.
→ Read-only txns may be allowed to access replicas.
→ If the primary goes down, then hold an election to select a new primary.

Approach #2: Multi-Primary
→ Txns can update data objects at any replica.
→ Replicas must synchronize with each other using an atomic commit protocol.
Primary-Replica

- Primary
- Replicas
Primary-Replica

**Writes**

**Reads**

- Primary
- Replicas
REPLICA CONFIGURATIONS

Primary-Replica

- **Writes**
  - Primary: P1
  - Replicas: P1

- **Reads**
  - Primary: P1
  - Replicas: P1
REPLICA CONFIGURATIONS

Primary-Replica

Primary

Replicas

Writes

Reads

Reads
REPLICA CONFIGURATIONS

Primary-Replica

- Primary
  - Writes
  - Reads
- Replicas
  - Reads

Multi-Primary

- Node 1
  - P1
- Node 2
  - P1
REPLICA CONFIGURATIONS

Primary-Replica

 Writes

 Reads

Primary

Replicas

Multi-Primary

 Writes

 Reads

Node 1

Node 2

Node 1

Node 2
REPLICA CONFIGURATIONS

Primary-Replica

- **Writes**
  - Primary: P1
  - Replicas: P1

- **Reads**
  - Primary: P1
  - Replicas: P1

Multi-Primary

- **Writes**
  - Node 1: P1
  - Node 2: P1

- **Reads**
  - Node 1: P1
  - Node 2: P1
**K-SAFETY**

*K*-safety is a threshold for determining the fault tolerance of the replicated database.

The value *K* represents the number of replicas per data object that must always be available.

If the number of replicas goes **below** this threshold, then the DBMS halts execution and takes itself offline.
PROPAGATION SCHEME

When a txn commits on a replicated database, the DBMS decides whether it must wait for that txn's changes to propagate to other nodes before it can send the acknowledgement to application.

Propagation levels:
→ Synchronous (Strong Consistency)
→ Asynchronous (Eventual Consistency)
**Approach #1: Synchronous**

- The primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.
**Propogation Scheme**

**Approach #1: Synchronous**

→ The primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.
Approach #1: Synchronous

→ The primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.
Approach #1: Synchronous

→ The primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.
Propagación: Modelado #1: Sincrónico

→ El principal envía actualizaciones a los replicados y luego espera que ellos confirmean (i.e., loggin) que han aplicado (i.e., logged) los cambios.
Propagating Synchronous

The primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.
Approach #1: Synchronous

→ The primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.
**PROPOSITION SCHEME**

Approach #1: Synchronous

→ The primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.

Approach #2: Asynchronous

→ The primary immediately returns the acknowledgement to the client without waiting for replicas to apply the changes.
PROPAGATION SCHEME

Approach #1: Synchronous
→ The primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.

Approach #2: Asynchronous
→ The primary immediately returns the acknowledgement to the client without waiting for replicas to apply the changes.
PROPAGATION SCHEME

Approach #1: Synchronous
→ The primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.

Approach #2: Asynchronous
→ The primary immediately returns the acknowledgement to the client without waiting for replicas to apply the changes.
PROPAGATION TIMING

Approach #1: Continuous
→ The DBMS sends log messages immediately as it generates them.
→ Also need to send a commit/abort message.

Approach #2: On Commit
→ The DBMS only sends the log messages for a txn to the replicas once the txn is commits.
→ Do not waste time sending log records for aborted txns.
→ Assumes that a txn's log records fits entirely in memory.
ACTIVE VS. PASSIVE

Approach #1: Active-Active
→ A txn executes at each replica independently.
→ Need to check at the end whether the txn ends up with the same result at each replica.

Approach #2: Active-Passive
→ Each txn executes at a single location and propagates the changes to the replica.
→ Can either do physical or logical replication.
→ Not the same as Primary-replica vs. multi-Primary
CAP THEOREM

Proposed by Eric Brewer that it is impossible for a distributed system to always be:
→ Consistent
→ Always Available
→ Network Partition Tolerant

Proved in 2002.
CAP THEOREM

Proposed by Eric Brewer that it is impossible for a distributed system to always be:
→ Consistent
→ Always Available
→ Network Partition Tolerant

Proved in 2002.

Pick Two! Sort of...
Consistency
Availability
Partition Tolerant
CAP THEOREM

- Consistency
- Availability
- Partition Tolerant

Linearizability
CAP THEOREM

- Consistency
- Availability
- Partition Tolerant

Linearizability

All up nodes can satisfy all requests.
CAP THEOREM

Consistency
Availability
Partition Tolerant

Linearizability
All up nodes can satisfy all requests.

Still operate correctly despite message loss.
CAP THEOREM

Consistency
Availability
Partition Tolerant

Linearizability

All up nodes can satisfy all requests.

Impossible

Still operate correctly despite message loss.
CAP – CONSISTENCY

Application Server

Set A=2

Primary

A=1
B=8

Replica

A=1
B=8

NETWORK
CAP – CONSISTENCY

Application Server

Set A=2

Primary

A=2
B=8

Replica

A=2
B=8

APPLICATION SERVER

NETWORK
Network

Application Server

Set A=2
ACK

Application Server

Read A

Primary

A=2
B=8

Replica

A=2
B=8

Application Server

A=1
B=8

Network

Set A=2
ACK

Read A

Primary

A=1
B=8

Replica

A=2
B=8

Application Server

CAP – CONSISTENCY
CAP – CONSISTENCY

Application Server

Set A=2
ACK

A=2
B=8

Primary

Application Server

Read A

A=2

Replica

A=2
B=8

NETWORK
If Primary says the txn committed, then it should be immediately visible on replicas.
CAP – AVAILABILITY

Application Server

Primary

A=1
B=8

Application Server

Replica

A=1
B=8

NETWORK
CAP – AVAILABILITY

Application Server

A=1
B=8

Primary

NETWORK

Replica

Application Server
CAP – AVAILABILITY

Application Server

Primary

A=1
B=8

NETWORK

Replica

Read B

Application Server
Application Server - Application Server

Primary

A=1
B=8

Replica

A=1
B=8

Network

Read B
B=8

CAP – AVAILABILITY
CAP – AVAILABILITY

Application Server

Primary

A=1
B=8

NETWORK

Read A
A=1

Replica

Application Server
CAP – PARTITION TOLERANCE

Application Server

A=1
B=8

Primary

NETWORK

A=1
B=8

Replica

Application Server
CAP – PARTITION TOLERANCE

Application Server

Primary

A=1
B=8

Application Server

Primary

A=1
B=8
CAP – PARTITION TOLERANCE

Application Server

A=1
B=8

Primary

Application Server

A=1
B=8

Primary
CAP – PARTITION TOLERANCE

Application Server

Set A=2

Primary

Set A=3

Primary

Application Server

A=1
B=8

A=1
B=8
CAP – PARTITION TOLERANCE

Application Server

Set A=2

Primary

Set A=3

Primary

Application Server

A=2
B=8

A=3
B=8

A=1

B=8

A=2

B=8

A=3

B=8
CAP – PARTITION TOLERANCE

Application Server → Set A=2
ACK

A=2
B=8
Primary

A=3
B=8
Primary

Application Server → Set A=3
ACK
**CAP – PARTITION TOLERANCE**

![Diagram showing partition tolerance with application servers and primary databases.](image-url)
CAP – PARTITION TOLERANCE

Application Server

Set A=2
ACK

Set A=3
ACK

Primary

A=2
B=8

A=3
B=8

Application Server

Primary

NETWORK
How a DBMS handles failures determines which elements of the CAP theorem they support.

**Traditional/NewSQL DBMSs**
→ Stop allowing updates until a majority of nodes are reconnected.

**NoSQL DBMSs**
→ Provide mechanisms to resolve conflicts after nodes are reconnected.
OBSERVATION

We have assumed that the nodes in our distributed systems are running the same DBMS software.

But organizations often run many different DBMSs in their applications.

It would be nice if we could have a single interface for all our data.
FEDERATED DATABASES

Distributed architecture that connects together multiple DBMSs into a single logical system.
A query can access data at any location.

This is hard and nobody does it well
→ Different data models, query languages, limitations.
→ No easy way to optimize queries
→ Lots of data copying (bad).
FEDERATED DATABASE EXAMPLE

Application Server

Middleware

Back-end DBMSs

MySQL

mongoDB

SUBWAY

redis
FEDERATED DATABASE EXAMPLE

Application Server → Middleware → Back-end DBMSs

- Query Requests
- MySQL
- MongoDB
- SUBWAY
- redis
FEDERATED DATABASE EXAMPLE

Middleware

Application Server

Query Requests

Back-end DBMSs

MySQL
mongoDB
redis

CMU-DB
15-445/645 (Fall 2021)
FEDERATED DATABASE EXAMPLE

Application Server

Query Requests

Foreign Data Wrappers

PostgreSQL

Connectors

Back-end DBMSs

MySQL

mongoDB

redis
CONCLUSION

We assumed that the nodes in our distributed DBMS are friendly.

Blockchain databases assume that the nodes are adversarial. This means you must use different protocols to commit transactions.

More info (and humiliation):
→ Kyle Kingsbury's Jepsen Project
Distributed OLAP Systems