Distributed OLTP Databases
Homework #5: Monday Dec 3rd @ 11:59pm

Project #4: Monday Dec 10th @ 11:59pm

Extra Credit: Wednesday Dec 10th @ 11:59pm

Final Exam: Monday Dec 9th @ 5:30pm
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

Application Server

Begin Request

Partitions

P1

P2

P3

P4
DECENTRALIZED COORDINATOR

Application Server

Partitions

P1

P2

P3

P4

Query

Query

Query
DECENTRALIZED COORDINATOR

Application Server

Commit Request

Safe to commit?

Partitions

P1

P2

P3

P4
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
TWO-PHASE COMMIT (SUCCESS)

Commit Request

Application Server

Coordinator

Node 1

Participant

Node 2

Participant

Node 3
TWO-PHASE COMMIT (SUCCESS)

Commit Request

Phase1: Prepare

Application Server

Coordinator

Node 1

Participant

Node 2

Participant

Node 3
TWO-PHASE COMMIT (SUCCESS)

Commit Request

Phase1: Prepare

OK

Node 1

Coordinator

Node 2

Node 3

Application Server
TWO-PHASE COMMIT (SUCCESS)

Application Server

Commit Request

Phase1: Prepare

Participant Coordinator

Participant

Coordinate

Phase2: Commit

Node 1

OK

Node 2

OK

Node 3
**TWO-PHASE COMMIT (SUCCESS)**

**Phase 1: Prepare**
- **Coordinator**
- **Application Server**
- **Participant 1 (Node 1)**
- **Participant 2 (Node 2)**
- **Participant 3 (Node 3)**

**Commit Request**

**Phase 2: Commit**
- **OK** from all participants
TWO-PHASE COMMIT (SUCCESS)

Application Server

Coordinator

Node 1

Success!

Participant

Node 2

Participant

Node 3
TWO-PHASE COMMIT (ABORT)

Commit Request

Application Server

Coordinator

Node 1

Participant

Participant

Node 2

Node 3
TWO-PHASE COMMIT (ABORT)

Application Server

Commit Request

Phase 1: Prepare

Coordinator

Node 1

Node 2

Node 3

Participant

Participant
TWO-PHASE COMMIT (ABORT)

Commit Request

Phase1: Prepare

ABORT!
**TWO-PHASE COMMIT (ABORT)**

- **Application Server**
- **Node 1**
- **Node 2**
- **Node 3**

Coordinator:

- Sends **ABORT!** to Participant Node 2
- Receives Aborted message from Participant Node 3

Participant Node 2:

- Sends **ABORT!** to Coordinator
- Receives Aborted message from Coordinator

Participant Node 3:

- Sends Aborted message to Coordinator
- Receives ABORT! message from Coordinator
TWO-PHASE COMMIT (ABORT)

Application Server

Coordinator

Phase 2: Abort

Node 1

Participant

Aborted

Node 2

Participant

ABORT!

Node 3
**TWO-PHASE COMMIT (ABORT)**

**Coordinator**
- **Phase2: Abort**

**Application Server**
- **Aborted**

**Node 1**
- **Phase2: Abort**

**Node 2**
- **OK**
- **ABORT!**
- **OK**

**Node 3**
- **OK**
- **ABORT!**
- **OK**
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

Commit Request

Application Server ➔ Coordinator ➔ Node 1 ➔ Node 2 ➔ Node 3
EARLY ACKNOWLEDGEMENT

Phase 1: Prepare

Commit Request

Application Server

Coordinator

Node 1

Node 2

Node 3

Participant

Participant

Participant
Commit Request

Phase 1: Prepare

Node 1

Application Server

Coordinator

Participant

Node 2

Node 3
EARLY ACKNOWLEDGEMENT

Application Server

Phase1: Prepare

Coordinator

Node 1

Node 2

Node 3

Success!

OK

OK

Participant

Participant

Participant
EARLY ACKNOWLEDGEMENT

Phase 1: Prepare

Phase 2: Commit

Application Server

Coordinator

Node 1

Node 2

Node 3

Success!

OK

OK

Participant

Participant
EARLY ACKNOWLEDGEMENT

Application Server

Phase 1: Prepare

Phase 2: Commit

Success!

Node 1

Node 2

Node 3

Coordinator

Participant

Participant

OK

OK

OK

OK
TWO-PHASE COMMIT

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

Commit Request

Proposer

Node 1

Node 2

Node 3

Node 4
PAXOS

Application Server

Proposer

Node 1

Commit Request

Propose

Node 2

Node 3

Node 4
PAXOS

Application Server

Proposer

Node 1

Commit Request

Propose

Node 2

Node 3

Node 4
PAXOS

Commit Request

Propose

Agree

Agree

Application Server

Node 1

Node 2

Node 3

Node 4
PAXOS

Application Server

Commit Request

Propose

Commit

Node 1

Agree

Acceptor

Node 2

Node 3

Node 4

Agree

Acceptor

Acceptor
PAXOS

Commit Request

Propose

Commit

Node 1

Agree

Accept

Node 2

Agree

Accept

Node 4

Node 3

Proposer

Application Server

Accept
PAXOS

Application Server

Node 1
Proposer

Success!

Node 2
Node 3
Node 4

Acceptors
PAXOS

TIME

Proposer

Acceptors

Proposer
Proposer

Propose(n)

TIME

Acceptors

Proposer
PAXOS

Proposer

Propose(n)

Acceptors

Agree(n)

TIME

Proposer
PAXOS

Proposer

Propose(n)

Acceptors

Agree(n)

Proposer

Propose(n+1)
PAXOS

- Proposer: Propose(n), Propose(n+1)
- Acceptors: Agree(n)
- Commit(n)

TIME
PAXOS

Proposer

Propose($n$)

Agree($n$)

Commit($n$)

Reject($n,n+1$)

Acceptors

Proposer

Propose($n+1$)

TIME
PAXOS

**Proposer**

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

**Acceptors**

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

**TIME**
PAXOS

Proposer

Propose($n$)

Commit($n$)

Agree($n$)

Reject($n,n+1$)

Propose($n+1$)

Agree($n+1$)

Commit($n+1$)

Acceptors

TIME

Proposer
PAXOS

Proposer

Propose(n)

Commit(n)

Acceptors

Agree(n)

Reject(n, n+1)

Proposer

Propose(n+1)

Agree(n+1)

Accept(n+1)

Commit(n+1)

TIME
MULTI-PAXOS

If the system elects a single leader that is in charge of proposing changes for some period of time, 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.
2PC VS. PAXOS

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

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

Master-Replica

- Master
- Replicas

Multi-Master

- Node 1
- Node 2
**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 master sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.
Approach #1: Synchronous
→ The master sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.
Approach #1: Synchronous
→ The master sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes.

Approach #2: Asynchronous
→ The master 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 master-replica vs. multi-master
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

**C**onsistency

**A**vailability

**P**artition Tolerant

- Linearizability
  - All up nodes can satisfy all requests.

- Impossible

- Still operate correctly despite message loss.
CAP – CONSISTENCY

Application Server -> Master
A=1, B=8

Set A=2

Master <-> Replica
A=1, B=8

NETWORK
CAP – CONSISTENCY

Application Server

Set A=2

Master

Network

Replica

Application Server

A=2
B=8

A=1
B=8
CAP – CONSISTENCY

Application Server

Application Server

Set A=2

Master
A=2
B=8

Network

Replica
A=2
B=8
CAP – CONSISTENCY

Application Server

Set A=2
ACK

Master

A=2
B=8

Replica

A=2
B=8

Application Server

NETWORK
CAP – CONSISTENCY

Application Server

Set A=2
ACK

A=2
B=8

Master

NETWORK

A=2
B=8

Replica

Read A

Application Server

A=2
B=8
If master says the txn committed, then it should be immediately visible on replicas.
CAP – AVAILABILITY

Application Server

Master

Replica

A=1
B=8

NETWORK

Application Server

A=1
B=8

CMU 15-445/645 (Fall 2019)
CAP – AVAILABILITY

Application Server

Read B

Master

A=1
B=8

NETWORK

Replica

Application Server

A=1
B=8
CAP – AVAILABILITY

Application Server

A=1
B=8

Application Server

Read B

B=8

Master

A=1
B=8

NETWORK

Replica

X
CAP – Availability

Application Server

Network

Master

Application Server

Replica

A = 1

B = 8

Read A
CAP – AVAILABILITY

Application Server

Master

A=1
B=8

Network

Replica

Application Server

Read A

A=1
CAP – PARTITION TOLERANCE

Application Server

A=1
B=8

Master

NETWORK

A=1
B=8

Replica

Application Server
CAP – PARTITION TOLERANCE

Application Server

A=1
B=8

Master

Application Server

A=1
B=8

Master
CAP – PARTITION TOLERANCE

Application Server → Set A=2

Master

A=1
B=8

Application Server → Set A=3

Master

A=1
B=8
CAP – PARTITION TOLERANCE

Application Server

Set A=2
ACK

A=2
B=8

Master

Set A=3
ACK

A=3
B=8

Master

Application Server
CAP – PARTITION TOLERANCE

Application Server

Set A=2
ACK

A=2
B=8

A=3
B=8

ACK

Master

Network

Application Server

Set A=3

Master
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

Query Requests

Middleware

Connectors

Back-end DBMSs

MySQL

mongoDB

redis
FEDERATED DATABASE EXAMPLE

Application Server

PostgreSQL

Foreign Data Wrappers

Query Requests

Connectors

Back-end DBMSs

MySQL

mongoDB

SUBWAY

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.
NEXT CLASS

Distributed OLAP Systems