Homework #5 is due Sunday Dec 4th @ 11:59pm

Project #4 is due Sunday Dec 11th @ 11:59pm

Upcoming Special Lectures:
→ Snowflake (Tuesday Dec 6th)
→ Live Call-in Q&A Lecture (Thursday Dec 8th)

Final Exam is Friday Dec 16th @ 1:00pm.
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

Primary Node

Partitions

P1

P2

P3

P4
DECENTRALIZED COORDINATOR

Application Server

Query

Primary Node

Partitions

P1

P2

P3

P4
DECENTRALIZED COORDINATOR

Application Server

Commit Request

Primary Node

Safe to commit?

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 will 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 / PACELC)
Google Spanner
ATOMIC COMMIT PROTOCOL

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)

1. **Commit Request** from Application Server to Coordinator
2. **Phase 1: Prepare** from Coordinator to Node 1
3. Node 1 to Node 2 and Node 3
4. Coordinator to Node 2 and Node 3

**Phase 1: Prepare** ensures that all participants are ready to commit.
TWO-PHASE COMMIT (SUCCESS)

Commit Request

Phase1: Prepare

OK

Participant

Application Server

Coordinator

Node 1

Node 2

Node 3
TWO-PHASE COMMIT (SUCCESS)

Application Server

Coordinator

Node 1

Phase 1: Prepare

Phase 2: Commit

Commit Request

OK

OK

Node 2

Node 3

Participant

Participant
TWO-PHASE COMMIT (SUCCESS)

Phase 1: Prepare

Coordinator

Node 1

Application Server

Commit Request

Phase 2: Commit

Participant

Node 2

OK

OK

OK

Node 3

OK

OK

OK
TWO-PHASE COMMIT (SUCCESS)

Application Server

Coordinator

Node 1

Success!

Participant

Node 2

Participant

Node 3
TWO-PHASE COMMIT (ABORT)

Application Server

Coordinator

Node 1

Commit Request

Phase 1: Prepare

Node 2

Participant

Participant

Node 3
TWO-PHASE COMMIT (ABORT)

Commit Request

Phase1: Prepare

Application Server

Coordinator

Node 1

Participant

Node 2

Participant

Node 3

ABORT!
TWO-PHASE COMMIT (ABORT)

Application Server

Coordinator

Node 1

Participant

Node 2

Participant

Node 3

Aborted

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

Aborted

Coordinator

Node 1

Phase2: Abort

Node 2

ABORT!

Node 3

Participant

Participant

Application Server
TWO-PHASE COMMIT (ABORT)

Application Server

Coordinator

Node 1

Phase2: Abort

Participant Node 2

ABORT!

Participant Node 3

Aborted

OK

OK

Participants

Coordination

Application Server

Node 1

Phase2: Abort

Participant Node 2

ABORT!

Participant Node 3

Aborted

OK

OK

Participants

Coordination
TWO-PHASE COMMIT

Each node records the inbound/outbound messages and outcome of each phase in a non-volatile storage log.

On recovery, examine the log for 2PC messages:
→ If local txn in prepared state, contact coordinator.
→ If local txn not in prepared, abort it.
→ If local txn was committing and node is the coordinator, send COMMIT message to nodes.
TWO-PHASE COMMIT FAILURES

What happens if coordinator crashes?
→ Participants must decide what to do after a timeout.
→ System is **not** available during this time.

What happens if participant crashes?
→ Coordinator assumes that it responded with an abort if it hasn't sent an acknowledgement yet.
→ Again, nodes use a timeout to determine that participant is dead.
2PC OPTIMIZATIONS

Early Prepare Voting (*Rare*)
→ 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 Ack After Prepare (*Common*)
→ 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**

**Phase 1: Prepare**

- **Commit Request**

**Nodes:**
- Node 1 (Coordinator)
- Node 2 (Participant)
- Node 3 (Participant)

**Connections:**
- Application Server to Node 1
- Node 1 to Node 2 and Node 3
EARLY ACKNOWLEDGEMENT

Application Server

Coordinator

Node 1

Success!

Phase1: Prepare

OK

Node 2

OK

Node 3

Participant

Participant

CMU-DB

15-445/645 (Fall 2022)
**EARLY ACKNOWLEDGEMENT**

- **Application Server**
- **Coordinator**
- **Phase 1: Prepare**
- **Phase 2: Commit**
- **Node 1**
- **Node 2**
- **Node 3**

**Success!**

- **OK**
- **OK**

**Participant**

Diagram shows the interactions between the application server, coordinator, and participants (nodes 1, 2, 3) during the phases of preparing and committing.
EARLY ACKNOWLEDGEMENT

Application Server

Coordinator

Node 1

Phase 1: Prepare

Phase 2: Commit

Success!

Node 2

OK

OK

OK

Node 3

OK

OK

Participant

Participant

Phase 1: Prepare

Phase 2: Commit
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.
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

Node 1

Proposer

Node 2

Acceptor

Node 3

Acceptor

Node 4

Acceptor
Application Server → Proposer (Node 1) → Acceptor (Node 2) → Commit Request → Acceptor (Node 3) → Acceptor (Node 4)
PAXOS

Application Server

Node 1

Proposer

Commit

Node 2

Acceptor

Agree

Node 3

Acceptor

Propose

Node 4

Acceptor

Commit Request

Agree

CMU-DB
PAXOS

Application Server

Proposer

Node 1

Success!

Acceptor

Node 2

Acceptor

Node 3

Acceptor

Node 4
PAXOS

Proposer

Propose(n)
Commit(n)

Acceptors

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

Proposer

Propose(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 the leader (known as a *lease*) using another Paxos round.

→ Nodes must exchange log entries during leader election to make sure that everyone is up-to-date.
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.
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.
**REPLICA CONFIGURATIONS**

**Primary-Replica**

- **Writes**
  - Primary
  - Replicas

- **Reads**
  - Primary
  - Replicas

**Multi-Primary**

- **Writes**
  - Node 1
  - Node 2

- **Reads**
  - 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 atxn 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)
**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.
**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.
**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
GOOGLE SPANNER

Google’s geo-replicated DBMS (>2011)
Schematized, semi-relational data model.
Decentralized shared-disk architecture.
Log-structured on-disk storage.
Concurrency Control:
→ Strict 2PL + MVCC + Multi-Paxos + 2PC
→ Externally consistent global write-transactions with synchronous replication.
→ Lock-free read-only transactions.
SPANNER: CONCURRENCY CONTROL

MVCC + Strict 2PL with Wound-Wait Deadlock Prevention

DBMS ensures ordering through globally unique timestamps generated from atomic clocks and GPS devices.

Database is broken up into tablets (partitions):
→ Use Paxos to elect leader in tablet group.
→ Use 2PC for txns that span tablets.
SPANNER TABLETS

Tablet A
Data Center 1

Tablet A
Data Center 2
Leader

Tablet A
Data Center 3

Paxos Groups
2PC

Snapshot Reads
Writes + Reads
Snapshot Reads

Paxos
Paxos
SPANNER: TRANSACTION ORDERING

DBMS orders transactions based on physical "wall-clock" time.
→ This is necessary to guarantee strict serializability.
→ If $T_1$ finishes before $T_2$, then $T_2$ should see the result of $T_1$.

Each Paxos group decides in what order transactions should be committed according to the timestamps.
→ If $T_1$ commits at $\text{time}_1$ and $T_2$ starts at $\text{time}_2 > \text{time}_1$, then $T_1$'s timestamp should be less than $T_2$'s.
CAP THEOREM

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

One flaw is that it ignores consistency vs. latency trade-offs.
→ See PACELC Theorem
Proposed by Eric Brewer that it is impossible for a distributed system to always be:

→ Consistent
→ Always Available
→ Network Partition Tolerant

One flaw is that it ignores consistency vs. latency trade-offs.

→ See PACELC Theorem
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

---

Primary

Set A=2

A=1
B=8

---

Replica

A=1
B=8

Application Server

NETWORK
CAP - CONSISTENCY

Application Server

Set A=2
ACK

A=2
B=8

Primary

A=2
B=8

Replica

Application Server

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

Application Server

A=1
B=8

Primary

NETWORK

Application Server

A=1
B=8

Replica
CAP - AVAILABILITY

Application Server

Read B

Primary

A=1
B=8

NETWORK

Replica

Application Server
CAP - AVAILABILITY

Application Server

Read B
B=8

A=1
B=8

Primary

NETWORK

A=1
B=8

Replica

Application Server
CAP - AVAILABILITY

Application Server

Read A

Primary

A=1
B=8

NETWORK

Replica

Application Server

A=1
B=8
CAP - AVAILABILITY

Application Server -> Primary

A=1
B=8

NETWORK

Read A
A=1

Application Server

Replica

A=1

CAP - PARTITION TOLERANCE

Application Server

Primary

A=1
B=8

Application Server

Replica

A=1
B=8
CAP – PARTITION TOLERANCE

Application Server

Primary

A=1
B=8

Application Server

Primary

A=1
B=8
CAP – PARTITION TOLERANCE

Application Server

Set A=2

A=2
B=8

Primary

Set A=3

A=3
B=8

Primary

Application Server
CAP – PARTITION TOLERANCE

Application Server

Set A=2
ACK

Primary

A=2
B=8

Application Server

Set A=3
ACK

Primary

A=3
B=8

Application Server

Set A=2
ACK

Primary

A=2
B=8

Application Server

Set A=3
ACK

Primary

A=3
B=8

Application Server

Set A=2
ACK

Primary

A=2
B=8

Application Server

Set A=3
ACK

Primary

A=3
B=8
CAP – PARTITION TOLERANCE

Application Server

Set A=2
ACK

A=2
B=8
Primary

Set A=3
ACK

A=3
B=8
Primary

ACK

Application Server

NETWORK

Set A=1

A=1
B=8
How a DBMS handles failures determines which elements of the CAP theorem they support.

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

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

Maintaining transactional consistency across multiple nodes is hard. Bad things will happen.

Blockchain databases assume that the nodes are adversarial. You must use different protocols to commit transactions. This is stupid.

More info (and humiliation):
→ Kyle Kingsbury's Jepsen Project
NEXT CLASS

Distributed OLAP Systems