Lecture #23

Distributed OLTP Databases
LAST CLASS

System Architectures
→ Shared-Everything, 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

Partitions

P1

P2

P3

P4
DECENTRALIZED COORDINATOR

Application Server

Partitions

P1

P2

P3

P4
DECENTRALIZED COORDINATOR

Application Server

Begin Request

Primary Node

Partitions

P1

P2

P3

P4
DECENTRALIZED COORDINATOR

Application Server

Query

Primary Node

Partitions

P1

Query

P2

Query

P3

Query

P4
DECENTRALIZED COORDINATOR

Application Server

Primary Node

Partitions

P1

P2

P3

P4
DECENTRALIZED COORDINATOR

Application Server

Primary Node

Partitions

Commit Request
DECENTRALIZED COORDINATOR

Application Server

Commit Request

Safe to commit?

Primary Node

Partitions

P1

P2

P3

P4
Recall that our goal is to have multiple physical nodes appear as a single logical DBMS.

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

→ What happens if a node fails?
→ What happens if messages show up late?
→ What happens if the system does not wait for every node to agree to commit?
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).

→ Blockchains are NOT good for high-throughput OLTP workloads (also they are not good for OLAP).
TODAY'S AGENDA

Replication
Atomic Commit Protocols
Consistency Issues (CAP / PACELC)
Google Spanner
REPLICATION

The DBMS can replicate a database across redundant nodes to increase availability.
→ Partitioned vs. Non-Partitioned
→ Shared-Nothing vs. Shared-Disk

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 by shipping logs.
→ 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

Primary

Replicas
REPLICA CONFIGURATIONS

Primary-Replica

Writes

Reads

Primary

Replicas

P1

P1
REPLICA CONFIGURATIONS

Primary-Replica

Writes
Reads

Primary

Replicas
REPLICA CONFIGURATIONS

Primary-Replica

**Writes**

Primary

Replicas

**Reads**

P1

P1
REPLICA CONFIGURATIONS

**Primary-Replica**

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

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

**Multi-Primary**

- **Node 1**
  - P1

- **Node 2**
  - P1
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
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 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*)
**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 #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.
**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.
**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 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.
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
ATOMIC COMMIT PROTOCOL

Coordinating the commit order of txns across nodes in a distributed DBMS.
→ Commit Order = State Machine
→ It does not matter whether the database's contents are replicated or partitioned.

Examples:
→ Two-Phase Commit (1970s)
→ Three-Phase Commit (1983)
→ Viewstamped Replication (1988)
→ Paxos (1989)
→ ZAB (2008?)
→ Raft (2013)
Resource Managers (RMs)
→ Execute on different nodes
→ Need to coordinate to decide on the fate of a txn: Commit or Abort

Properties of the Commit Protocol
→ **Stability**: Once the fate is decided, it can’t be changed.
→ **Consistency**: All RMs end up in the same state.

Assumes “Liveness”:
→ Informally, there is some way of progressing forward; e.g., enough nodes are alive and connected for the duration of the protocol.
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)

Application Server

Coordinator

Node 1

Phase 1: Prepare

Commit Request

OK

Participant

Node 2

OK

Participant

Node 3

Participant
TWO-PHASE COMMIT (SUCCESS)

Commit Request

Phase1: Prepare

Phase2: Commit

Application Server

Node 1

Node 2

Node 3

Coordinator

Participant

Participant
TWO-PHASE COMMIT (SUCCESS)

Commit Request

Phase1: Prepare

Phase2: Commit

OK

OK

OK

OK

Application Server

Coordinator

Node 1

Node 2

Node 3

Participant

Participant
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

Participant

Node 2

Participant

Node 3
TWO-PHASE COMMIT (ABORT)

Commit Request

Phase1: Prepare

Application Server

Coordinator

Node 1

Participant

Node 2

Participant

Node 3
TWO-PHASE COMMIT (ABORT)

Commit Request

Phase1: Prepare

ABORT!

Application Server

Coordinator

Node 1

Participant

Node 2

Participant

Node 3
TWO-PHASE COMMIT (ABORT)

Application Server

Coordinator

Node 1

Node 2

Node 3

Participant

Participant

Aborted

ABORT!
TWO-PHASE COMMIT (ABORT)

Application Server

Coordinator

Node 1

Phase2: Abort

Aborted

Node 2

Participant

Node 3

Participant

ABORT!
TWO-PHASE COMMIT (ABORT)

Application Server

Coordinator

Node 1

Phase2: Abort

Participant

Node 2

OK

OK

Node 3

Participant

Abort!

Aborted
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 has not sent an acknowledgement yet.
→ Again, nodes use a timeout to determine whether a 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 to execute in this txn, 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 theirtxn was successful before the commit phase finishes.
EARLY ACKNOWLEDGEMENT

Application Server

Coordinator

Node 1

Commit Request

Node 2

Participant

Node 3

Participant
EARLY ACKNOWLEDGEMENT

Application Server

Coordinator

Node 1

Commit Request

Phase1: Prepare

Node 2

Participant

Node 3

Participant
EARLY ACKNOWLEDGEMENT

Phase1: Prepare

Commit Request

Coordinator

Node 1

Node 2

Node 3

Participant

Participant

Participants
**EARLY ACKNOWLEDGEMENT**

**Phase1: Prepare**

Application Server

Coordinator

Node 1

Participant

Node 2

Success!

OK

OK

Participant

Node 3
**EARLY ACKNOWLEDGEMENT**

Success!

**Phase 1: Prepare**

**Phase 2: Commit**

Participant

Application Server

Node 1

Coordinator

OK

Node 2

Participant

OK

Node 3

Participant
EARLY ACKNOWLEDGEMENT

Phase1: Prepare
Phase2: Commit
Success!
PAXOS

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

Proposer

Node 1

Commit Request

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

Acceptors

Node 2

Node 3

Node 4

Propose
PAXOS

Application Server

Proposer

Node 1

Commit Request

Agree

Agree

Acceptor

Acceptor

Node 2

Node 3

Node 4

Propose
PAXOS

Application Server

Node 1

Commit Request

Propose

commit

Node 2

Agree

Node 3

Agree

Node 4

Proposer

Acceptors
PAXOS

Application Server

Proposer

Commit Request

Agree

Accept

Node 1

Propose

Commit

Agree

Accept

Node 2

Node 3

Node 4
PAXOS

Application Server

Proposer

Node 1

Success!

Node 2

Node 3

Node 4
PAXOS

Application Server

Proposer

Node 1

Success!

Acceptor

Node 2

Acceptor

Node 3

Acceptor

Node 4
PAXOS

TIME

Proposer

Acceptors

Proposer
PAXOS

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

- Propose(n)
- Agree(n)
- Commit(n)
PAXOS

Proposer
Propose(n)
Commit(n)

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

Proposer
Propose(n+1)
Agree(n+1)

TIME
**PAXOS**

1. **Proposer**
   - Propose(n)
   - Commit(n)

2. **Acceptors**
   - Agree(n)
   - Reject(n,n+1)
   - Agree(n+1)

3. **Proposer**
   - Propose(n+1)
   - Commit(n+1)
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)

TIME
MULTI-PAXOS

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 systemperiodicallyrenewstheleader(knownas 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 VS. RAFT

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.

Raft:
→ Similar to Paxos but with fewer node types.
→ Only nodes with most up-to-date log can become leaders.
CAP THEOREM

Proposed in the late 1990s that is impossible for a distributed database to always be:
→ **C**onsistent
→ **A**lways **V**ailable
→ **N**etwork **P**artition **T**olerant

Extended in 2010 (**PACELC**) to include consistency vs. latency trade-offs:
→ **P**artition **T**olerant
→ **A**lways **V**ailable
→ **C**onsistent
→ **E**Else, choose during normal operations
→ **L**atency
→ **C**onsistency
CONSISTENCY

Application Server

A=1  B=8

Primary

NETWORK

A=1  B=8

Replica

Application Server
CONSISTENCY

Set A=2

Application Server

Primary

A=1
B=8

Replica

A=1
B=8

Application Server

Network
CONSISTENCY

Application Server -> Primary

Set A=2

A=2
B=8

NETWORK

A=1
B=8

Application Server -> Replica
CONSISTENCY

Application Server

Set A=2

Primary

A=2
B=8

Replica

A=2
B=8

Application Server

NETWORK
**CONSISTENCY**

Application Server → Set A=2 → ACK → Application Server

Primary

A=2
B=8

Replica

A=2
B=8

NETWORK
CONSISTENCY

Application Server

Set A=2
ACK

A=2
B=8

Primary

NETWORK

A=2
B=8

Replica

Read A

Application Server

A=1
B=8
A=2

Application
Server

15-445/645 (Spring 2024)
If Primary says the txn committed, then it should be immediately visible on replicas.
AVAILABILITY

Application Server

Primary

A=1
B=8

NETWORK

Replica

A=1
B=8

Application Server
AVAILABILITY

Application Server

A=1
B=8

Primary

NETWORK

Replica

Application Server

A=1
B=8
AVAILABILITY

Application Server

Primary

A=1
B=8

Read B

Network

Replica

Application Server

A=1
B=8
AVAILABILITY

Application Server

Read B
B=8

Primary

A=1
B=8

NETWORK

Replica

Application Server
AVAILABILITY

Application Server

A=1
B=8

Primary

Application Server

A=1
B=8

Replica

NETWORK
AVAILABILITY

Application Server

Primary

A=1
B=8

Replica

Read A

Application Server

NETWORK

CMU-DB
15-445/645 (Spring 2024)
PARTITION TOLERANCE

Application Server

Primary

A=1
B=8

Network

Replica

A=1
B=8

Application Server
PARTITION TOLERANCE

Application Server

A=1
B=8

Application Server

A=1
B=8

Primary

Replica

CMU-DB
15-445/645 (Spring 2024)
PARTITION TOLERANCE

Application Server

Primary

A=1
B=8

Application Server

Replica

A=1
B=8
PARTITION TOLERANCE

Application Server

Primary

A=1
B=8

Application Server

Primary

A=1
B=8
PARTITION TOLERANCE

Application Server

Primary

A=1
B=8

Application Server

Primary

A=1
B=8

CMU-DB
15-445/645 (Spring 2024)
PARTITION TOLERANCE

Application Server

Set A=2

Primary

A=1
B=8

Set A=3

Primary

A=1
B=8

Application Server
PARTITION TOLERANCE

Application Server → Set A=2

Primary

Application Server → Set A=3

Primary

A=2
B=8

A=3
B=8
PARTITION TOLERANCE

Application Server

Set A=2
ACK

A=2
B=8

Primary

Set A=3
ACK

A=3
B=8

Primary

Application Server
PARTITION TOLERANCE

Application Server

Primary

Set A=2
ACK

A=2
B=8

Set A=3

A=3
B=8

Application Server

Primary

NETWORK

ACK

ACK
PARTITION TOLERANCE

Application Server

Set A=2
ACK

Primary

A=2
B=8

Network

A=3
B=8

Primary

Application Server

Set A=3
ACK
LATENCY VS. CONSISTENCY

Application Server

Primary (us-east)

Replica (us-west)

Replica (eu-east)
LATENCY VS. CONSISTENCY

Application Server

Set A=2

Primary (us-east)

Replica (us-west)

A=1

Replica (eu-east)

A=1
LATENCY VS. CONSISTENCY

Application Server

Set A=2

Primary (us-east)

A=2

Replica (us-west)

A=1

Replica (eu-east)

Set A=1
LATENCY VS. CONSISTENCY

Application Server

Primary (us-east)

Set A=2

Replica (us-west)

Replica (eu-east)

A=2

A=2

A=2

A=2
LATENCY VS. CONSISTENCY

Application Server

Primary (us-east)

Replica (us-west)

A=2

Replica (eu-east)

A=2

ACK

ACK
LATENCY VS. CONSISTENCY

Application Server

Primary (us-east)

Replica (us-west)

Replica (eu-east)

A=2

A=2

ACK

ACK

???
LATENCY VS. CONSISTENCY

Application Server

???

Primary (us-east)

Replica (us-west)

Replica (eu-east)

ACK

A=2

A=2

A=2

ACK

A=2
LATENCY VS. CONSISTENCY

Application Server

Primary (us-east)

Replica (us-west)

Replica (eu-east)

A=2

ACK

A=2

ACK

A=2

A=2

???
LATENCY VS. CONSISTENCY

Application Server

Primary (us-east)

Replica (us-west)

Replica (eu-east)

ACK

ACK

ACK

A=2

A=2

A=2

???
CAP/PACE/LC FOR OLTP DBMSs

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

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

NoSQL DBMSs
→ No multi-node consistency. Last update wins (common).
→ Provide client-side API to resolve conflicts after nodes are reconnected (rare).
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: CONCURRENCEY 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

Paxos Group

Tablet A

Data Center 1

Data Center 2

Leader

Data Center 3

Writes + Reads
SPANNER TABLETS

Paxos Group

Tablet A

Data Center 1

Tablet A

Data Center 2

Tablet A

Data Center 3

Leader

Writes + Reads

Paxos

Paxos
SPANNER TABLETS

Tablet A
Data Center 1

Paxos

Tablet A
Data Center 2
Leader

Tablet A
Data Center 3

Snapshot Reads
Writes + Reads
Snapshot Reads

Paxos
Paxos
SPANNER TABLETS

Tablet B-Z Paxos Groups

Snapshot Reads

Paxos Group

Tablet A

Data Center 1

Paxos

Tablet A

Data Center 2

Leader

Tablet A

Data Center 3

2PC

 Writes + Reads

Snapshot Reads

Paxos

Paxos Groups

2PC
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.
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. Not suitable for database workloads.

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

Distributed OLAP Systems