Distributed OLTP Databases (Part II)
LAST CLASS

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

Partitioning/Sharding
→ Hash, Range, Round Robin

Transaction Coordination
→ Centralized vs. Decentralized
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
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?
TODAY'S AGENDA

Atomic Commit Protocols
Replication
Consistency Issues (CAP)
Federated Databases
ATOMIC COMMIT PROTOCOL

When a multi-node txn finishes, the DBMS needs to ask all of 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)

Application Server

Commit Request

Coordinator

Node 1

Participant

Node 2

Participant

Node 3
TWO-PHASE COMMIT (SUCCESS)

Application Server

Coordinator

Phase 1: Prepare

Commit Request

Node 1

Node 2

Node 3

Participant

Participant
TWO-PHASE COMMIT (SUCCESS)

Commit Request

Phase 1: Prepare

Application Server

Coordinator

Node 1

OK

Node 2

Participant

Node 3

Participant

OK
TWO-PHASE COMMIT (SUCCESS)

Phase 1: Prepare

Phase 2: Commit

Commit Request

OK

Application Server

Coordinator

Node 1

Node 2

Node 3

Participant

Participant
TWO-PHASE COMMIT (SUCCESS)

Commit Request

Phase1: Prepare

Phase2: Commit

Application Server

Coordinator

Node 1

Node 2

Node 3

Participant

Participant

Participant
TWO-PHASE COMMIT (SUCCESS)

Application Server

Coordinator

Node 1

Node 2

Node 3

Success!
TWO-PHASE COMMIT (ABORT)

Application Server

Coordinator

Node 1

Commit Request

Node 2

Node 3

Participant

Participant
TWO-PHASE COMMIT (ABORT)

Commit Request

Phase1: Prepare

Application Server

Coordinator

Node 1

Node 2

Node 3

Participant

Participant
TWO-PHASE COMMIT (ABORT)

Application Server

Aborted

Node 1

Coordinator

Node 2

Participant

Node 3

Participant

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

Application Server

Coordinator

Node 1

Phase2: Abort

Participant

Node 2

OK

Participant

Node 3

Aborted

Node

ABORT!

OK
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

Commit Request

Phase 1: Prepare

Node 2

Participant

Node 3

Participant
EARLY ACKNOWLEDGEMENT

Commit Request

Phase1: Prepare

Application Server

Coordinator

Node 1

Node 2

Node 3
**EARLY ACKNOWLEDGEMENT**

Application Server

Phase 1: Prepare

Coordinator

Node 1

Node 2

Node 3

Success!

OK

OK

Participant

Participant

Participant
EARLY ACKNOWLEDGEMENT

Application Server

Coordinator

Node 1

Phase 2: Commit

Phase 1: Prepare

Node 2

OK

OK

Node 3

Participant

Participant

Success!
EARLY ACKNOWLEDGEMENT

Application Server

Coordinator

Node 1

Phase 1: Prepare

Phase 2: Commit

Success!

Participant

Node 2

OK

OK

Node 3

Participant

OK

OK
TWO-PHASE COMMIT

Each node has to record the outcome of each phase in a stable storage log.

What happens if coordinator crashes?
→ Participants have to 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.
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

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

Commit Request

Node 1

Application Server

Proposer

Node 2

Acceptor

Node 3

Acceptor

Node 4

Acceptor

Propose
PAXOS

Commit Request

Node 1

Proposer

Node 2

Accept

Node 3

Accept

Node 4

Accept

Application Server

Propose
PAXOS

Application Server

Proposer

Node 1

Commit Request

Agree

Node 2

Agree

Node 3

Propose

Node 4
PAXOS

Commit Request

Agree

Propose

Commit

Application
Server

Node 1

Proposer

Node 2

Node 3

Node 4

Acceptor

Acceptor

Acceptor

Acceptor
PAXOS

Commit Request

Propose

Commit

Agree

Accept

Node 1

Node 2

Node 3

Node 4

Accept

Accept

Accept
PAXOS

Application Server

Node 1

Node 2

Node 3

Node 4

Success!
PAXOS

TIME

Proposer

Acceptors

Proposer
**PAXOS**

Proposer

Acceptors

Proposer

\textit{Propose(n)}
PAXOS

Proposer

Acceptors

Proposer

TIME

Propose(n)

Agree(n)
PAXOS

Proposer

\textit{Propose}(n)

TIME

Acceptors

\textit{Agree}(n)

Proposer

\textit{Propose}(n+1)
PAXOS

TIME

Proposer

Propose(n)

Commit(n)

Agree(n)

Acceptors

Commit(n)

Propose(n+1)

Proposer
PAXOS

Proposer

Propose(n)

Agree(n)

Commit(n)

Reject(n,n+1)

Acceptors

Proposer

Propose(n+1)

TIME
PAXOS

TIME

Proposer

Acceptors

Proposer

Propose(n)

Agree(n)

Commit(n)

Reject(n, n+1)

Agree(n+1)

Commit(n+1)

Propose(n+1)

Commit(n+1)
PAXOS

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

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

- **Proposer**
  - Propose(n+1)
  - Commit(n+1)
  - Accept(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 PREPARE phase. → Fall back to full Paxos whenever there is a failure.

The system has to periodically renew who the leader is.
**2PC VS. PAXOS**

**Two-Phase Commit**
→ Blocks if coordinator fails after the prepare message is sent, until coordinator recovers.

**Paxos**
→ Non-blocking as long as 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
REPLICA CONFIGURATIONS

Approach #1: Master-Replica
→ All updates go to a designated master for each object.
→ The master then propagates those updates to its replicas.
→ 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 synchronize with each other.
REPLICA CONFIGURATIONS

Master-Replica

- **Writes**
  - Master
  - Replicas

- **Reads**
  - Master
  - Replicas

Multi-Master

- **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 exist at all times.

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 has to decide whether it has to wait for that txn's changes to propagate to other nodes before it can send the acknowledgement to application.

Propagation levels:
→ Synchronous
→ Asynchronous
→ Semi-Synchronous
**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 #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 SCHEME**

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 SCHEME

Approach #3: Semi-Synchronous
→ Replicas immediately send acknowledgements without logging them.
Approach #3: Semi-Synchronous
→ Replicas immediately send acknowledgements without logging them.
Approach #3: Semi-Synchronous
→ Replicas immediately send acknowledgements without logging them.
Approach #3: Semi-Synchronous

→ Replicas immediately send acknowledgements without logging them.

Applications can make trade-offs on protecting the integrity of the database versus performance.
PROPA.GATION 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 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.
→ Not the same as master-replica vs. multi-master
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

Consistency (C)

Availability (A)

Partition Tolerant (P)

Linearizability

All up nodes can satisfy all requests.

Impossible

Still operate correctly despite message loss.
CAP – CONSISTENCY

Application Server

Set A=2

Master

A=1
B=8

Replica

A=1
B=8

Application Server

NETWORK
CAP – CONSISTENCY

Set A=2

Application Server

Master

A=2
B=8

Replica

A=1
B=8

Application Server

Network
**CAP – CONSISTENCY**

- **Application Server**
  - Set A=2

- **Master**
  - A=2
  - B=8

- **Replica**
  - A=2
  - B=8

- **Network**

---

CMU 15-445/645 (Fall 2018)
CAP – CONSISTENCY

Application Server

Set A=2
ACK

A=2
B=8

A=2
B=8

Master

NETWORK

Replica
**CAP – CONSISTENCY**

- **Set A=2**
  - ACK
- **Read A**

**Application Server**
- **Master**
  - A=2
  - B=8
- **Replica**
  - A=2
  - B=8

**NETWORK**
CAP – CONSISTENCY

Application Server

Set A=2
ACK

A=2
B=8

NETWORK

Master

Read A
A=2

A=2
B=8

Replica

Application Server
If master says the txn committed, then it should be immediately visible on replicas.
CAP – A VAILABLEITY

Application Server

A=1
B=8

Master

NETWORK

A=1
B=8

Replica

Application Server
CAP – Availability

Application Server

Master

A=1
B=8

Replica

X

Network

Application Server
CAP – AVAILABILITY

Application Server

Read B

Master

A=1
B=8

NETWORK

Replica

Application Server

A=1
B=8
Application Server

A=1
B=8

Read B

B=8

Master

Application Server

Replica

NETWORK

A=1

CAP – AVAILABILITY
CAP – AVAILABILITY

Application Server

A=1
B=8

Master

NETWORK

Replica

Read A

Application Server

A=1
B=8

Application
Server

CMU 15-445/645 (Fall 2018)
CAP – Availability

Application Server

Master

Network

Replica

Application Server

Read A

A = 1

A = 1

B = 8

A = 1

B = 8
CAP – PARTITION TOLERANCE

Application Server

Master

A=1
B=8

Network

Replica

Application Server

A=1
B=8

CMU 15-445/645 (Fall 2018)
CAP – PARTITION TOLERANCE

Application Server

Set A=2

Master

A=1
B=8

Set A=3

Master

A=1
B=8

Application Server
CAP – PARTITION TOLERANCE

Application Server

Set A=2

A=2
B=8

Master

Set A=3

A=3
B=8

Application Server
CAP – PARTITION TOLERANCE

Application Server

Set A=2
ACK

A=2
B=8

Application Server

Set A=3
ACK

A=3
B=8

Master

Master
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

Set A=2
ACK

A=1
B=8

Application Server

Set A=3
ACK

A=1
B=8
CAP FOR OLTP DBMSs

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

Back-end DBMSs:
- MySQL
- MongoDB
- redis
FEDERATED DATABASE EXAMPLE

Query Requests
Application Server
Middleware
Connectors
Back-end DBMSs
MySQL
mongoDB
redis
FEDERATED DATABASE EXAMPLE

Query Requests

Application Server

PostgreSQL

Foreign Data Wrappers

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 have to use different protocols to commit transactions.
NEXT CLASS

Distributed OLAP Systems