Intro to Database Systems (15-445/645)

22 Distributed OLTP Databases

Carnegie Mellon University

SPRING 2023

Charlie Garrod
ADDITIONAL INFORMATION

Homework 5 available
→ Due Friday, April 21st at 11:59 p.m.

Project 4 available
→ Due Friday, April 28th at 11:59 p.m.

Interested in TAing this course?
→ https://forms.gle/AvjfUtSaWtrNiJMXA

Final exam Monday, May 1st, 8:30 – 11:30 a.m.
LAST TIME

System architectures
→ Shared memory, shared disk, shared nothing

Partitioning/sharding
→ Hash vs. range

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

Commit Request

Query

Primary Node

Partitions

P1

P2

P3

P4

Query

Say No commit?
**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 atxn, 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 (e.g., 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)

1. Phase 1: Prepare
   - Coordinator
   - Participant

2. Phase 2: Commit
   - Coordinator
   - Participant

Application Server

Node 1

Node 2

Node 3

Success!

OK

OK

OK

OK

OK

OK

Success!
TWO-PHASE COMMIT (ABORT)

Coordinator

Node 1

Phase1: Prepare

Phase2: Abort

Aborted Request

OK

Application Server

Participant

Node 2

OK

Node 3

ABORT!

Commit Request
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

Application Server

Coordinator

Node 1

Phase1: Prepare

Phase2: Commit

Success!

Node 2

Node 3

OK

OK

OK

Participant

Participant

Participant
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

<table>
<thead>
<tr>
<th>Proposer</th>
<th>Acceptors</th>
<th>Proposer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Propose(n)</td>
<td>Agree(n)</td>
<td>Propose(n+1)</td>
</tr>
<tr>
<td>Commit(n,v1)</td>
<td>Reject(n,n+1)</td>
<td>Commit(n+1,v2)</td>
</tr>
<tr>
<td></td>
<td>Agree(n+1)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Accept(n+1,v2)</td>
<td></td>
</tr>
</tbody>
</table>

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 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.
Two-Phase Commit
→ Atomic commit for distributed transactions
→ Blocks if coordinator fails after the prepare message is sent, until coordinator recovers.

Paxos
→ Distributed consensus, can implement atomic commit on top of 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.
REPLICA CONFIGURATIONS

Primary-Replica

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

Multi-Primary

- **Node 1:**
  - Writes
  - Reads
  - P1
- **Node 2:**
  - Writes
  - Reads
  - 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.
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*)
**Propagagation 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’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**
  - Leader
  - Data Center 1
  - Writes + Reads
  - Paxos
  - Snapshot Reads

- **Tablet A**
  - Data Center 2
  - 2PC

- **Tablet A**
  - Data Center 3
  - Snapshot Reads

- **Tablet B-Z Paxos Groups**
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.
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.
All up nodes can satisfy all requests.
If Primary says the txn committed, then it should be immediately visible on replicas.
CAP – PARTITION TOLERANCE

Primary

Set A=2
ACK

A=2
B=8

Network

Set A=3
ACK

A=3
B=8

Replica

Application Server

Set A=2

Application Server

Set A=3

A=1

B=8
CAP FOR OLTP DBMSs

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