Lecture #24: Distributed Database Systems II

15-445/645 Database Systems (Fall 2025)
`https://15445.courses.cs.cmu.edu/fall2025/`
Carnegie Mellon University
Andy Pavlo

1 OLTP vs. OLAP

On-line Transaction Processing (OLTP)

- Short-lived read/write transactions.
- Small footprint.
- Repetitive operations.

On-line Analytical Processing (OLAP)

- Long-running, read-only queries.
- Complex joins.
- Exploratory queries.

2 Atomic Commit Protocols

When a multi-node transaction finishes, the DBMS needs to ask all of the nodes involved whether it is safe to commit. Depending on the protocol, a majority of the nodes or all of the nodes may be needed to commit. Examples include:

- Two-Phase Commit (1970s)
- Three-Phase Commit (1983)
- Viewstamped Replication (1988)
- Paxos (1989)
- ZAB (2008? Zookeeper Atomic Broadcast protocol, **Apache Zookeeper**)
- Raft (2013)

All of these atomic commit protocols have a common structure. They usually have a notion of **Resource Managers (RMs)** that manage resources or databases (or part of a database) on different nodes. The RMs need to coordinate together to decide the fate of each transaction: *Commit* or *Abort*. Using the RMs, an atomic commit protocol needs to guarantee the following properties:

- **Stability:** Once the fate of a transaction is decided, it cannot be changed.
- **Consistency:** All the RMs end up in the same state, even after failures.
- **Liveness:** The protocol must always have some way of progressing forward (e.g., enough nodes are alive and connected for the duration of the protocol).

In the following sections, we will focus on Two-Phase Commit and Paxos. If the coordinator fails after the prepare message is sent, Two-Phase Commit (2PC) blocks until the coordinator recovers. On the other hand, Paxos is non-blocking if a majority of participants are alive, provided that there is a sufficiently long period without further failures. If the nodes are in the same data center, do not fail often, and are not malicious, then 2PC is often preferred over Paxos as 2PC usually results in fewer round trips.

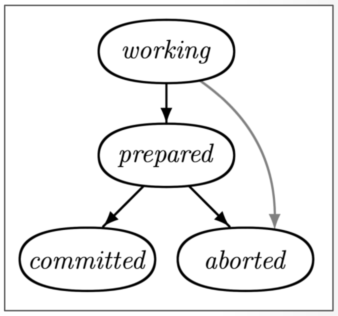

Figure 1: Atomic commit protocols can usually be modeled as state machines.

Two-Phase Commit

The client sends a *Commit Request* to the coordinator. In the first phase of this protocol, the coordinator sends a *Prepare* message, essentially asking the participant nodes if the current transaction is allowed to commit. If a given participant verifies that the given transaction is valid, they send an *OK* to the coordinator. If the coordinator receives an *OK* from all the participants, the system can now go into the second phase in the protocol. If anyone sends an *Abort* to the coordinator, the coordinator sends an *Abort* to the client.

The coordinator sends a *Commit* to all the participants, telling those nodes to commit the transaction if all the participants have sent an *OK*. Once the participants respond with an *OK*, the coordinator can tell the client that the transaction is committed. If the transaction was aborted in the first phase, the participants receive an *Abort* from the coordinator, to which they should respond with an *OK*. Either everyone commits or no one does. The coordinator can also be a participant in the system.

Additionally, in the case of a crash, all nodes keep track of a non-volatile log of the outcomes of each phase. Two-Phase Commit is a **blocking protocol**, which means nodes block until they can figure out the next course of action. If the coordinator crashes, the participants must decide what to do. A safe option is just to abort. Alternatively, the nodes can communicate with each other to see if they can commit without the explicit permission of the coordinator. If a participant crashes, the coordinator assumes that it responded with an abort if it has not sent an acknowledgement yet.

Optimizations:

- *Early Prepare Voting* – If the DBMS sends a query to a remote node that it knows will be the last one executed 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 transaction, the coordinator can send the client an acknowledgement that their transaction was successful before the commit phase finishes.

Paxos

Paxos (along with Raft) is more prevalent in modern systems than 2PC. 2PC is a degenerate case of Paxos; Paxos uses $2F + 1$ coordinators and makes progress as long as at least $F + 1$ of them are working properly, 2PC sets $F = 0$.

Paxos is a consensus protocol where a coordinator proposes an outcome (e.g., commit or abort) and then the participants vote on whether that outcome should succeed. This protocol does not block if a majority of participants are available and has provably minimal message delays in the best case. For Paxos, the coordinator is called the **proposer** and participants are called **acceptors**.

The client will send a *Commit Request* to the proposer. The proposer will send a *Propose* to the other nodes in the system, or the acceptors. A given acceptor will send an *Agree* if they have not already sent an *Agree* on a higher logical timestamp. Otherwise, they send a *Reject*.

Once the majority of the acceptors have sent an *Agree*, the proposer will send a *Commit*. The proposer must wait to receive an *Accept* from a majority of acceptors before sending the final message to the client saying that the transaction is committed, unlike 2PC.

Use exponential backoff times when trying to propose again after a failed proposal, to avoid dueling proposers.

Multi-Paxos: If the system elects a single leader that oversees proposing changes for some period, then it can skip the propose phase. The system periodically renews who the leader is using another Paxos round. When there is a failure, the DBMS can fall back to full Paxos.

3 CAP Theorem

The *CAP Theorem*, proposed by Eric Brewer and later proved in 2002 at MIT, explained that it is impossible for a distributed system to always be Consistent, Available, and Partition Tolerant. Only two of these three properties can be chosen.

Consistency is synonymous with linearizability for operations on all nodes. Once a write completes, all future reads should return the value of that write applied or a later write applied. Additionally, once a read has been returned, future reads should return that value or the value of a later applied write. NoSQL systems compromise this property in favor of the latter two. Other systems will favor this property and one of the latter two.

Availability is the concept that all nodes that are up can satisfy all requests.

Partition tolerance means that the system can still operate correctly despite some message loss between nodes that are trying to reach consensus on values. If consistency and partition tolerance are chosen for a system, updates will not be allowed until a majority of nodes are reconnected, typically done in traditional or NewSQL DBMSs.

There is a modern version that considers consistency vs. latency trade-offs: *PACELC Theorem*. In case of network partitioning (P) in a distributed system, one has to choose between availability (A) and consistency (C), else (E), even when the system runs normally without network partitions, one has to choose between latency (L) and consistency (C).

4 Distributed Join Algorithms

For analytical workloads, the majority of time is spent computing joins and reading from disk, so optimizing joins is important. The efficiency of a distributed join depends on the target tables' partitioning schemes.

One approach is to put entire tables on a single node and then perform the join there. However, the DBMS loses the parallelism of a distributed DBMS, defeating the purpose of making it distributed. This option also entails costly data transfer over the network.

To join tables R and S, the DBMS needs to get the proper tuples on the same node. Once there, it executes the same join algorithms discussed earlier in the semester. One should always send the minimal amount needed to compute the join.

There are four scenarios for distributed join algorithms.

Scenario 1

One of the tables is replicated at every node and the other table is partitioned across nodes. Each node joins its local data in parallel and then sends its results to a coordinating node.

Scenario 2

Both tables are partitioned on the join attribute, with IDs matching on each node. Each node performs the join on local data and then sends them to a node for coalescing.

Scenario 3

Both tables are partitioned on different keys. If one of the tables is small, then the DBMS broadcasts that table to all nodes. This takes us back to Scenario 1. Local joins are computed and then those joins are sent to a common node to perform the final join. This is known as a *broadcast join*.

Scenario 4

This is the worst-case scenario. Both tables are not partitioned on the join key. The DBMS partitions the tables across nodes via the shuffle operator such that nodes have data matching on the join ID. This recovers Scenario 2. Local joins are computed and then the results are sent to a common node for the final join. If there isn't enough disk space, a failure is unavoidable. This is called a *shuffle join*.

Semi-Join Optimization

Before pulling data from another node, a *semi-join filter* can be used to reduce data movement. This filter blocks rows from the data that will be discarded in the result anyway due to a predicate. An approximate filter like a Bloom filter can be used here.

Shuffle Operation

In addition to being useful in joins, a shuffle operation is generally useful. It can be used to rebalance data based on observed characteristics, and adjust the capacity allocated to a subcomputation. The shuffle operator is basically the repartition type of exchange operator discussed previously in lecture.