Lecture #23: Distributed OLTP Databases

15-445/645 Database Systems (Fall 2024)
https://15445.courses.cs.cmu.edu/fall2024/
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 Decentralized Coordinator Setup

The basic senario for distributed database system is that we have an application server and multiple data partitions. One of these partitions is elected to be the primary node. The begin request for transactions goes from application server to primary node and queries are sent directly sent to various nodes if there are no centralized coordinator. The commit request goes from the application server to the primary node and the primary node is reponsible for figuring out amongst all participating nodes whether it is allowed to commit. If all participating nodes agree it is safe to commit, then we can commit the transaction. Two phase locking, MVCC, OCC and other strategies are used to determine whether the transaction can be safely committed on each individual node.

The following sections will talk about how to ensure all nodes agree to commit a transaction and what happens if a node fails/messages show up late/the system does not wait for every node to agree. We'll assume that all nodes in a distributed DBMS are well-behaved and under the smae administrative domain. (If you do not trust the other nodes in a distributed DBMS, then you need to use a *byzantine fault tolerant* protocol (e.g., blockchain) for the transaction.)

3 Replication

The DBMS can replicate data across redundant nodes to increase availability.

Design Decisions:

- Replica Configuration
- Propagation Scheme
- Propagation Timing
- Update Method

Replica Congigurations

In **Primary-Replica**, all updates go to a designated primary for each object. The primary propagates updates to its replicas without an atomic commit protocol, coordinating all updates that come to it. Read-only transactions may be allowed to access replicas if the most up-to-date information is not needed. If the primary goes down, then an election is held to select a new primary.

In **Multi-Primary**, transactions can update data objects at any replica. Replicas must synchronize with each other using an atomic commit protocol.

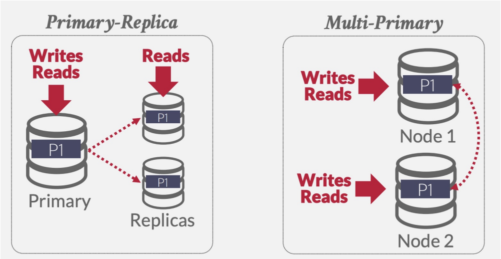

Figure 1: Replica Configurations

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. A higher value of K reduces the risk of losing data. It is a threshold to determine how available a system can be.

Propagation Scheme

When a transaction commits on a replicated database, the DBMS decides whether it must wait for that transaction's changes to propagate to other nodes before it can send the acknowledgement to the application client. There are two propagation levels: synchronous (strong consistency) and asynchronous (eventual consistency).

In a *synchronous* scheme, the primary sends updates to replicas and then waits for them to acknowledge that they fully applied (i.e., logged) the changes. Then, the primary can notify the client that the update has succeeded. It ensures that the DBMS will not lose any data due to strong consistency. This is more common in a traditional DBMS.

In an *asynchronous* scheme, the primary immediately returns the acknowledgement to the client without waiting for replicas to apply the changes. Stale reads can occur in this approach, since updates may not be fully applied to replicas when read is occurring. If some data loss can be tolerated, this option can be a viable optimization. This is used commonly in NoSQL systems.

Propagation Timing

For *continuous* propagation timing, the DBMS sends log messages immediately as it generates them. Note that commit and abort messages also need to be sent. Most systems use this approach.

For *on commit* propagation timing, the DBMS only sends the log messages for a transaction to the replicas once the transaction is committed. This does not waste time for sending log records for aborted transactions. It does make the assumption that a transaction's log records fit entirely in memory.

Active vs Passive

There are multiple approaches to applying changes to replicas. For *active-active*, a transaction executes at each replica independently. At the end, the DBMS needs to check whether the transaction ends up with the same result at each replica to see if the replicas committed correctly. This is difficult since now the ordering of the transactions must sync between all the nodes, making it less common.

For *active-passive*, each transaction executes at a single location and propagates the overall changes to the replica. The DBMS can either send out the physical bytes that were changed, which is more common, or the logical SQL queries. Most systems are active-passive.

4 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 ususally have a notion of **Resource Managers (RMs)** that manage resources 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 need to gurantee the following property:

- **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 failure.
- **Liveness:** The protocol always have some way of progressing forward (e.g enough nodes are alive and connected for the duration of the protocol).

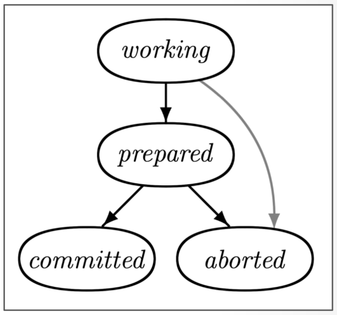

Figure 2: Atomic commit protocols can usually be modeled as state machines.

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.

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

Use exponential back off times for 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.

5 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 is 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).