Lecture #23: Distributed OLTP Databases

15-445/645 Database Systems (Fall 2020)
https://15445.courses.cs.cmu.edu/fall2020/
Carnegie Mellon University
Prof. Andy Pavlo

1 Distributed Transactions

A transaction is "distributed" if it accesses data on multiple nodes. Executing these transactions is more challenging than single-node transactions because now when the transaction commits, the DBMS has to make sure that all the nodes agree to commit the transaction. The DBMS ensure that the database provides the same ACID guarantees as a single-node DBMS even in the case of node failures or message loss.

One can assume that all nodes in a distributed DBMS are well-behaved and under the same administrative domain. In other words, given that there is not a node failure, a node which is told to commit a transaction will commit the transaction. If the other nodes in a distributed DBMS cannot be trusted, then the DBMS needs to use a *byzantine fault tolerant* protocol (e.g., blockchain) for transactions.

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 (Common)
- Three-Phase Commit (Uncommon)
- Paxos (Common)
- Raft (Common)
- ZAB (**Apache Zookeeper**)
- Viewstamped Replication (first provably correct protocol)

Two-Phase Commit (2PC) blocks if coordinator fails after the prepare message is sent, until the coordinator recovers. Paxos, on the other hand, is non-blocking if a majority participants are alive, provided there is a sufficiently long period without further failures. 2PC is used often if the nodes are in the same data center because of the number of round trips could be less than for Paxos, assuming that nodes do not fail often and are not malicious.

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. It is a less strict version of 2PC. This 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 the *Accept* before sending the final message to the client saying that the transaction is committed, unlike 2PC.

To prevent a situation where values keep on getting rejected due to multiple proposers sending higher and higher timestamps, **Multi-Paxos** is useful, where there is a leader node for the Paxos group. If the system elects a single leader that is in charge of proposing changes for some period of time, then it can skip the propose phase. When there is a failure, the DBMS can fall back to full Paxos. The system periodically renews who the leader is using another Paxos round (e.g., the system elects a new leader every 10 seconds). Use exponential back off times for trying to propose again to avoid the original problem of not reaching consensus, in this case determining who the leader is.

3 Replication

The DBMS can replicate data across redundant nodes to increase availability. In other words, if a node goes down, the data is not lost, and the system is still alive and does not need to be rebooted. One can use Paxos to determine which replica to write data to.

Number of Primary Nodes

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 hold an election via Paxos 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 like Paxos or 2PC.

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. A higher value of K reduces 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 in NoSQL systems.

Propagation Timing

For *continuous* propagation timing, the DBMS sends log messages immediately as it generates them. Note that a commit or abort message needs to also 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.

4 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 up nodes 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.

5 Federated Databases

These are distributed architectures that connect together multiple DBMSs into a single logical system. This is more popular in bigger companies. A query can access data at any location. This is hard due to different data models, query languages, and limitations of each individual DBMS. Additionally, there is no easy way to optimize queries. Lastly, there is a lot of data copying that is involved.

For example, say there is an application server which makes some queries. These queries then go through a middleware layer (which will convert the query into a readable format for a given DBMS used in the bigger system) that via *connectors*, will go through the multiple back-end DBMSs that are deployed in the system. The middleware will then handle the results returned from the DBMSs.

PostgreSQL is in the best position to successfully deploy a federated database using its *foreign data wrappers*. It allows a user to use data from another system within a given Postgres session.