CS W186
Fall 2019

Distributed Transactions

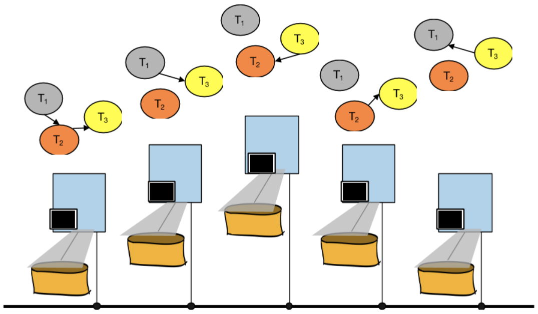
1 Introduction

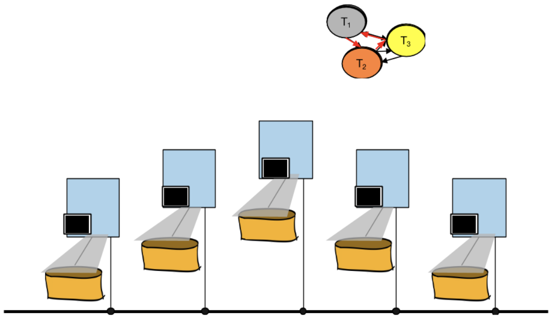
Up until this point, we have always assumed that transactions ran on databases where all of the data existed on one node (machine). This is often the case for databases with lighter workloads, but as demand increases, databases scale out to improve performance by using a **Shared Nothing architecture**. Each node receives a partition of the data set that is distributed based on a range or hash key and is connected to other nodes through a network. Distributed Transactions are needed for executing queries in distributed databases as a transaction may need to perform reads and writes on data that exist on different nodes.

2 Distributed Locking

Since every node contains data that is independent of any other node's data, every node can **maintain its own local lock table**. Coarser grained locks for entire tables or the database can either be given to all nodes containing a partition or be centralized at a predetermined node. This design makes locking simple as **2 phase locking** is performed at every node using local locks in order to guarantee serializability between different transactions.

When dealing with locking, deadlock is always a possibility. To determine whether deadlock has occurred in a distributed database, the waits-for graphs for each node must be unioned to find cycles as transactions can be blocked by other transactions executing on different nodes.





3 Two Phase Commit (2 PC)

In a distributed database, **consensus is the idea that all nodes agree on one course of action**. Consensus is implemented through Two Phase Commit and enforces the property that all nodes maintain the same view of the data. It provides this guarantee by ensuring that a distributed transaction either commits or aborts on all nodes involved. If consensus is not enforced, some nodes may commit the transaction while others abort, causing nodes to have views of data at different points in time.

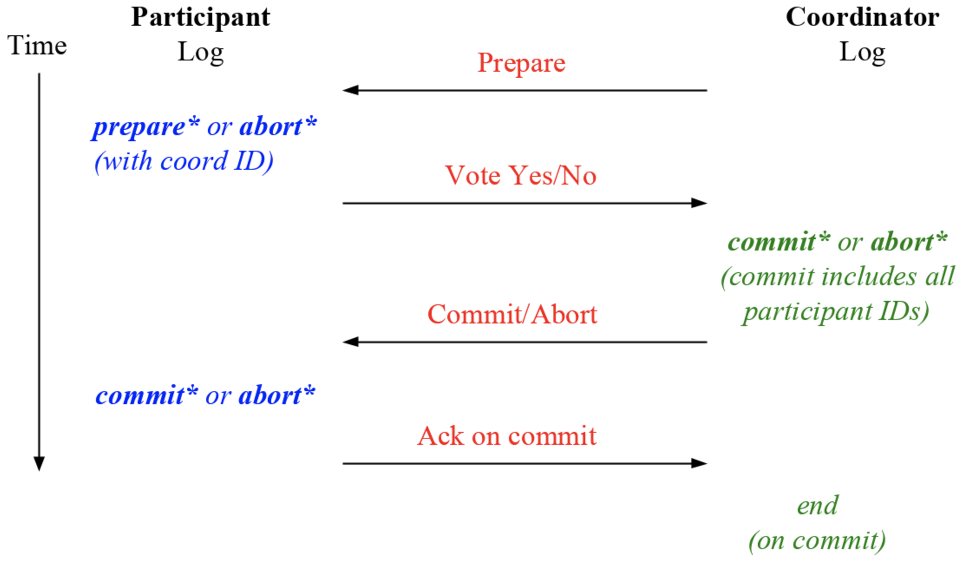
Every distributed transaction is assigned a coordinator node that is responsible for maintaining consensus among all participant nodes involved in the transaction. When the transaction is ready to commit, the coordinator initiates Two Phase Commit.

2 PC's first phase is the **preparation phase**:

1. Coordinator sends prepare message to participants (tells participants to either prepare for commit or abort)
2. Participants generate a prepare or abort record and flush record to disk
3. Participants send yes message to coordinator if prepare record is flushed or a no message if abort record is flushed
4. Coordinator **generates a commit record if unanimous yes votes** or an abort record otherwise and flushes record to disk

2 PC's second phase is the **commit/abort phase**:

1. Coordinator broadcasts (sends message to every node in the db) result of the commit/abort vote based on flushed record
2. Participates generate a commit or abort record based on the received vote message and flush record to disk
3. **If the result is a commit**, participants send an ACK (acknowledgement) message to the coordinator
4. **If the result is a commit**, coordinator generates an end record once all ACKs are received and flushes the record to disk



4 Distributed Recovery

It is also important that the Two-Phase Commit protocol maintains consensus among all nodes ***even in the presence of node failures***. In other words – suppose a node were to fail at an arbitrary point in the protocol. When this node comes back online, it should still end up making the same decision as all the other nodes in the database.

How do we accomplish this? Luckily, the 2PC protocol tells us to log the prepare, commit, and abort records. This means that between looking at our own log, and talking to the coordinator node, we will have all the information needed to accomplish recovery correctly!

Let's look at the specifics. We will analyze what happens for a failure at each possible point in the protocol, for either the participant or the coordinator.
Note that in some cases, we can determine what recovery decisions to make just by looking at our own log. In other cases, however, we might also have to talk to the coordinator; we wrap this logic in a separate process called the *recovery process*.

The possible failures, in chronological order:

- **Participant** is recovering, and sees **no prepare record**.
  - This probably means that the participant has not even started 2PC yet – and if it has,

it hasn't yet sent out any *vote messages* (since those happen after flushing prepare to disk).

  - Since it has not sent out any vote messages, it is safe to **abort** the transaction.

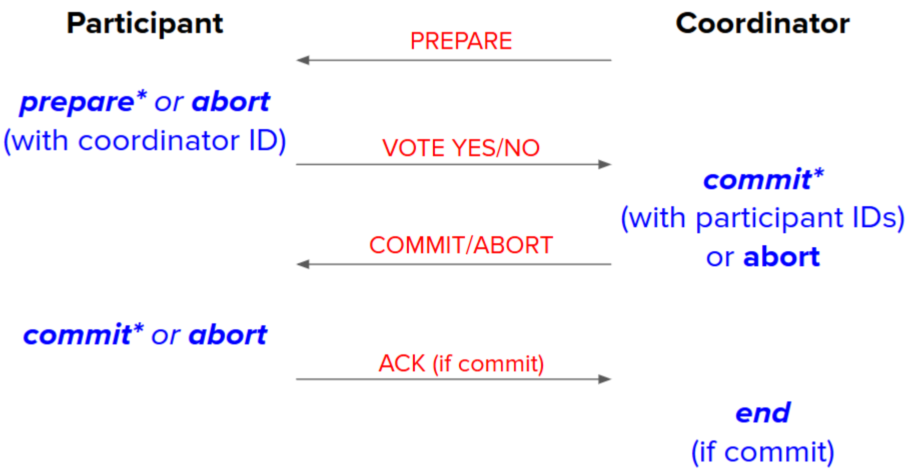
- **Participant** is recovering, and sees a **prepare record**.
  - This situation is trickier. Looking at the graph above, a lot of things could have happened between logging the prepare record and crashing – for instance, we don't even know if we managed to send out our YES vote!
  - Specifically, we don't know whether or not the coordinator made a commit decision. So the **recovery process must ask the coordinator whether a commit happened ("Did the coordinator log a commit?")**.
  - The coordinator will respond with the commit/abort decision, and the **participant resumes 2PC from phase 2**.
- **Coordinator** is recovering, and sees a **commit record**.
  - We'd like to commit, but we don't know if we managed to tell the participants.
  - So, just **rerun phase 2** (send out commit messages).
- **Participant** is recovering, and sees a **commit record**.
  - We did all our work for this commit, but the coordinator might still be waiting for our ACK, so **send ACK to coordinator**.
- **Coordinator** is recovering, and sees an **end record**.
  - This means that everybody already finished the transaction and there is no recovery to do!

The list above only discusses commit records. **What about abort records?**

We could handle them the same way as commit records (e.g. tell participants, send acks). This works fine, but is it really necessary? What if nodes just didn't bother recovering aborted transactions?

It turns out this works if everybody understands that **no log records means abort**. This optimization is called **presumed abort**, and it means that **abort records never have to be flushed** – not in phase 1 or phase 2, not by the participant or the coordinator.

This results in the protocol looking like this:



And this is how we would recover from failures in aborted transactions, with and without presumed abort:

- **Participant** is recovering, and sees a **phase 1 abort record**.
  - Without presumed abort: Send "no" vote to coordinator.
  - With presumed abort: Do nothing! (Coordinator will already presume abort anyway.)
- **Coordinator** is recovering, and sees an **abort record**.
  - Without presumed abort: Rerun phase 2 (tell participants about abort decision).
  - With presumed abort: Do nothing! (Participants who don't know the decision will just ask later.)
- **Participant** is recovering, and sees a **phase 2 abort record**.
  - Both with and without presumed abort: Do nothing (the participant never does anything after writing a phase 2 abort record).

To wrap everything up, here are some subtleties we mentioned that you should be explicitly be aware of:

- The 2PC recovery decision is **commit *if and only if* the coordinator has logged a commit record**.
- Since 2PC requires unanimous agreement, it will only make progress if all nodes are alive. This is true for the recovery protocol as well – for recovery to finish, all failed nodes must eventually come back alive. Protocols that continue operating despite extended failures are out of the scope of this course, but a good example is *"Paxos Commit"*.