Database Systems

Save this note as PDF

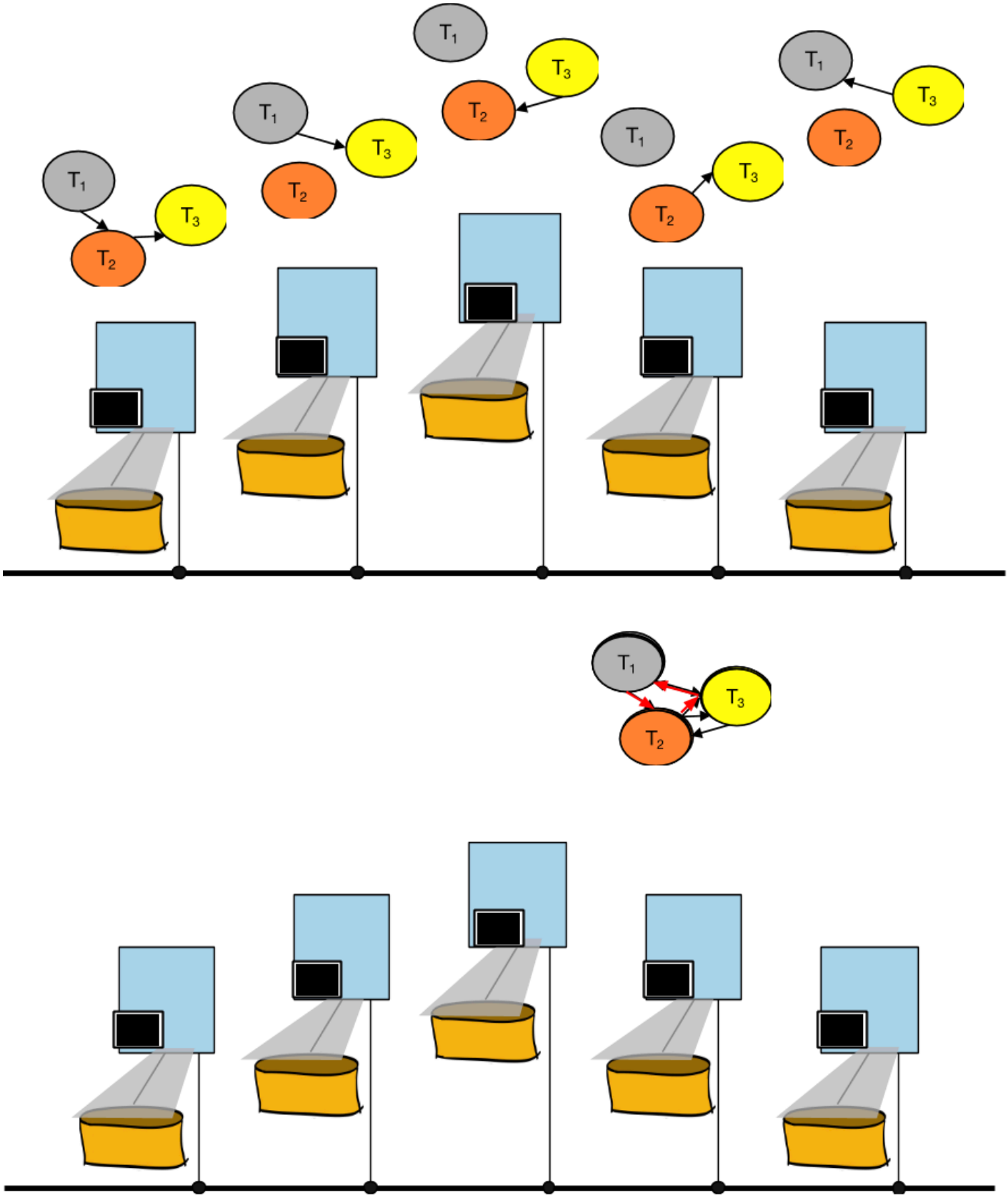
Introduction

For much of this semester, we 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.

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**. This works if the data fits entirely within that node, such as pages or tuples of data. For coarser grained locks on data that spans multiple nodes, such as a table or database, the coarser grained locks can either be given to all nodes containing a partition of that data or be centralized at a predetermined master 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. Do this by drawing all waits-for graphs on top of each other. If a cycle is detected in this unioned graph, then there is a deadlock within the distributed system.



Two Phase Commit (2 PC)

With a distributed database where transactions can be running on multiple nodes, how do the various nodes know whether to commit or abort a transaction? In either case, commit or abort, we want to ensure that all nodes reach **consensus, which is to say 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.

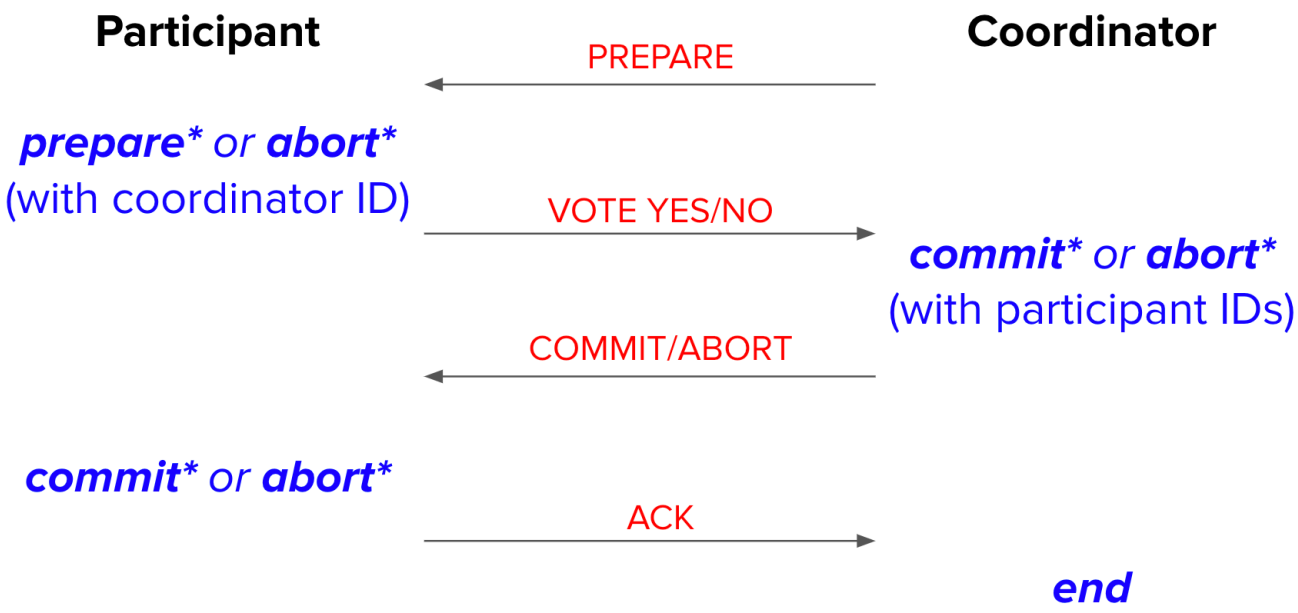
Every distributed transaction is assigned a coordinator node that is responsible for maintaining consensus among all participant nodes involved in the transaction. When the transaction is ready to commit, the coordinator initiates Two Phase Commit.

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

1. Coordinator sends prepare message to participants to tell 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 vote to coordinator if prepare record is flushed or no vote if abort record is flushed
4. Coordinator **generates a commit record if it receives unanimous yes votes** or an abort record otherwise, and flushes the record to disk

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

1. Coordinator broadcasts (sends message to every participant) the result of the commit/abort vote based on flushed record
2. Participants generate a commit or abort record based on the received vote message and flush record to disk
3. Participants send an ACK (acknowledgement) message to the coordinator
4. Coordinator generates an end record once all ACKs are received and flushes the record sometime in the future



The asterisk * in the diagram above means that the node must wait for that log record to flush to disk before sending the next message.

Distributed Recovery (2PC)

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 (for most failure scenarios).

An important assumption we will make for now is that failures are temporary and that all nodes eventually recover. Many of 2PC's invariants break down without this assumption, but that is beyond the scope of this class.

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 votes are sent after flushing the log record to disk).
  - Since it has not sent out any vote messages, it **aborts** the transaction locally. **No messages need to be sent out** (the participant has no knowledge of the coordinator ID).
- **Participant** is recovering, and sees a **prepare record**.
  - This situation is trickier. Looking at the diagram 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 participant node's recovery process must ask the coordinator whether a commit happened ("Did the coordinator log a commit?"). The coordinator can be determined from the coordinator ID stored in the prepare log record.
  - The coordinator will respond with the commit/abort decision, and the **participant resumes 2PC from phase 2**.
- **Coordinator** is recovering, and sees **no commit record**.
  - The coordinator crashed at some point before receiving the votes of all participants and logging a commit decision.
  - The coordinator will **abort** the transaction locally. **No messages need to be sent out** (the coordinator has no knowledge of the participant IDs involved in the transaction since it does not log its own prepare record).
  - If the coordinator receives an inquiry from a participant about the status of the transaction, respond that the transaction aborted.
- **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, **rerun phase 2** (send out commit messages to participants). The participants can be determined from the participant IDs stored in the commit log record.
- **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**. (The coordinator can be determined from the coordinator ID stored in the commit log record.)
- **Coordinator** is recovering, and sees an **end record**.
    - This means that everybody already finished the transaction and there is no recovery to do.

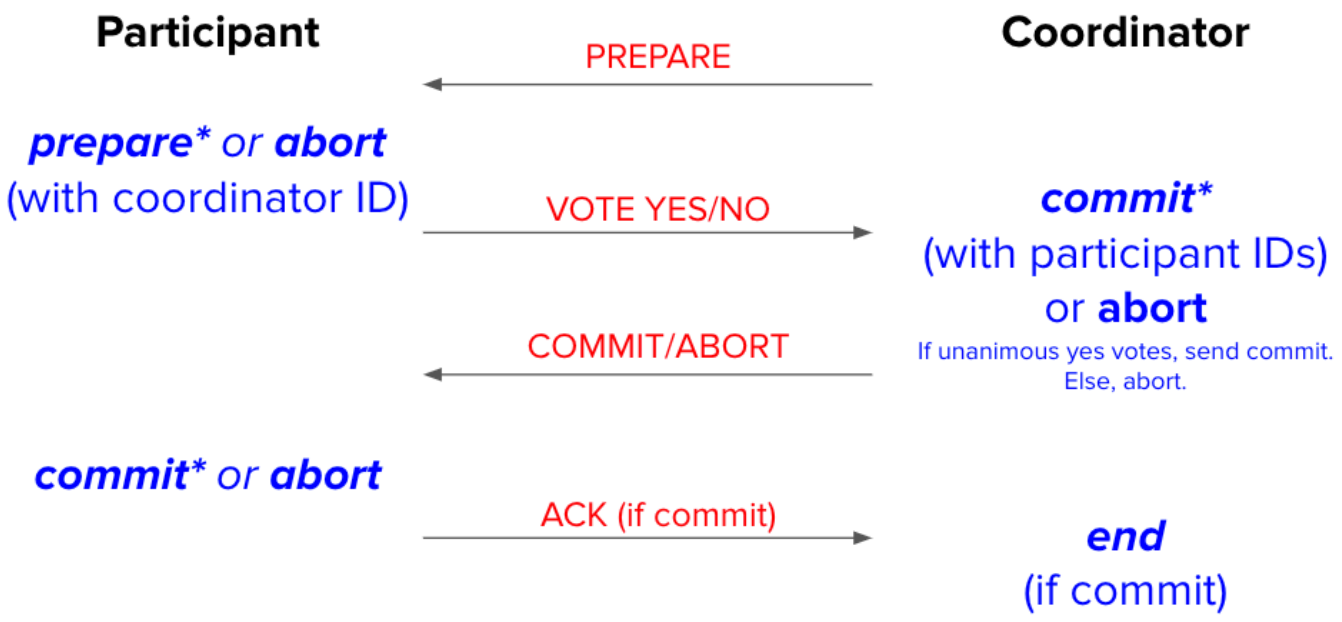
The list above only discusses commit records. **What about abort records?**

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?

Distributed Recovery (2PC with Presumed Abort)

It turns out if all participants and the coordinator understand that **no log records means abort**, we wouldn't have to bother with recovering aborted transactions. Instead, we can assume we should abort if we see no log record. 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.

The protocol with the presumed abort optimization looks like this:



The asterisk * in the diagram above means that the node must wait for that log record to flush to disk before sending the next message. **Notice that in the presumed abort optimization, abort records no longer need to be flushed to disk.**

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

- **Participant** is recovering, and sees **no phase 1 abort record**.
  - Without presumed abort: 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 votes are sent after flushing the log record to disk).
  - With presumed abort: It is possible that the participant decided to abort and sent a "no" vote to the coordinator before the crash.
  - With or without presumed abort, the participant **aborts** the transaction locally. **No messages need to be sent out** (the participant has no knowledge of the coordinator ID).
- **Participant** is recovering, and sees a **phase 1 abort record**.
  - Without presumed abort: Abort the transaction locally and send "no" vote to the coordinator. (The coordinator can be determined from the coordinator ID stored in the abort log record.)

    - With presumed abort: Abort the transaction locally. No messages need to be sent out! (The coordinator will timeout after not hearing from the participant and presume abort.)

- **Coordinator** is recovering, and sees **no abort record**.
    - Without presumed abort: The coordinator crashed at some point before reaching a commit/abort decision.
    - With presumed abort: It is possible that the coordinator decided to abort and sent out abort messages to the participants before the crash.
    - With or without presumed abort, the coordinator will **abort** the transaction locally. No messages need to be sent out (the coordinator has no knowledge of the participant IDs involved in the transaction).
    - If the coordinator receives an inquiry from a participant about the status of the transaction, respond that the transaction aborted.
- **Coordinator** is recovering, and sees an **abort record**.
    - Without presumed abort: Rerun phase 2 (sending out abort messages to participants). The participants can be determined from the participant IDs in the abort log record.
    - With presumed abort: Abort the transaction locally. No messages need to be sent out! (Participants who don't know the decision will ask the coordinator later.)
- **Participant** is recovering, and sees a **phase 2 abort record.**[1]
    - Without presumed abort: Abort the transaction locally, and send back ACK to coordinator. (The coordinator can be determined from the coordinator ID stored in the abort log record.)
    - With presumed abort: Abort the transaction locally. No messages need to be sent out! (ACKs only need to be sent back on commit.)

Note that the recovery processes for **commit records** is the same in Two-Phase Commit and Two-Phase Commit with Presumed Abort. For instance, the scenarios where the participant recovers from Two-Phase Commit with Presumed Abort and sees a **prepare record** or **commit record** is handled in the same way as in regular Two-Phase Commit. The same goes for when a coordinator recovers and sees a **commit record**.

To wrap everything up, here are some subtleties 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. If the coordinator believes a participant is dead, it can respawn the participant on a new node based on the log of the original participant, and ignore the original participant if it does come back online. However, 2PC struggles to handle scenarios where the coordinator is dead. For example, consider a scenario where all participants vote yes in Phase 1, but the coordinator crashes before sending out a commit decision. The participants will keep pinging the dead coordinator for the status of the transaction, and the system is blocked from making progress.

Practice Questions

1. Suppose that there are some transactions happening concurrently. If no two concurrent transactions ever operate on rows that are being stored in the same node, can a deadlock still happen?
2. Suppose a participant receives a prepare message for a transaction. Assuming the participant node remains online and does not fail, why might it decide to log an abort record and vote no?
3. Suppose a participant receives a prepare message for a transaction and replies `VOTE-YES`. Suppose that they were running a wound-wait deadlock avoidance policy, and a transaction comes in with higher priority. Will the new transaction abort the prepared transaction?
4. True or false - if we, as a participant node, are recovering and see a `PREPARE` record, it means we must have sent out a `YES` vote.
5. True or false - if the coordinator is recovering and sees a `COMMIT` record, we can locally commit and end the process there.
6. How many messages and log flushes does the presumed abort optimization skip if the transaction commits? What about if it aborts due to the participants aborting?

Solutions

1. You may be inclined to say no, since a transaction does not seem like it will ever wait on the locks of another transaction in this scenario. But remember that in addition to each node maintaining its own local lock table, **coarser grained locks, such as locks for tables or the entire database, can still be shared between nodes**, and the transactions may still conflict via these coarser locks.

2. The participant may decide to abort the transaction and vote no to avoid deadlock. For example, if we are using deadlock detection and we discover that the union of the waits-for graphs for each node contains a cycle the transaction is involved in, we may decide to abort the transaction. If we are using a deadlock avoidance policy, we will abort the transaction if acquiring the locks necessary to perform the transaction's operations would require us to wait on a lower priority transaction (for wait-die) or higher priority transaction (for wound-wait).

3. No - transactions that are prepared **must** be ready to commit if the coordinator tells them to, so they cannot be aborted by anyone other than the coordinator.

4. False! We could have crashed between logging `PREPARE` and sending `VOTE-YES`.

5. False! There is no guarantee that all participants received a `COMMIT` record, so we must rerun phase 2 first.

6. If the transaction commits, no messages or log flushes are skipped - the messages sent and records logged are the same as the protocol without presumed abort.

   If the participants abort, they do not flush any log records (skipping up to two flushes) and only have to send one message (they don't have to send the ACK). The coordinator also does not have to flush any log records (skipping one flush), but does not skip any messages (it still sends the messages it would without presumed abort).

Past Exam Problems

- Sp22 Final Q5 pg 20-24
- Fa21 Final Q9 pg 32-35
- Sp21 Final Q10 pg 42-46
- Fa20 Final Q7 pg 33-37

1 In reality, there is **no difference** between a phase 1 and phase 2 abort record. Therefore, for the without presumed abort case, a participant can simply respond with NO to the coordinator upon seeing an abort record. The coordinator will then treat that message as either a phase 1 vote or a phase 2 ACK accordingly. ↩