Today

Odds and ends:

- More about deadlocks.
- Distributed concurrency issues.

Soon:

- Final exam.
- Life.

Breaking Deadlocks With Rollbacks

- Avoid having to build and use entire waits-for graph.
- A lock request for a locked item can result in either the requester or the holder having to abort.
- Methods are based on a transaction timestamp.
  - Needed whether or not timestamp-based concurrency control is in use.
  - If a transaction aborts, it restarts with the *same* timestamp.
  - In both schemes, the transaction with the earliest timestamp cannot abort, and so will eventually complete (no "starvation").

Wait-Die Scheme

Suppose T requests an element locked by U.

1. If T is earlier than U, T WAITS for the lock.
2. If T is later than U, T DIES (aborts).
   - We later restart T with its original timestamp.

Wound-Wait Scheme

Under the same circumstance:

1. If T is earlier than U, T WOUNDS U (U must release its locks, then abort and restart), and the lock is given to T.
2. If T is later than U, T WAITS for the lock.

Why They Work

Suppose there is a cycle of waiting, say $T_1 \rightarrow T_2 \rightarrow T_3 \rightarrow T_1$.

- There has to be one with the latest timestamp, say T_2.
- In the Wait-Die scheme, you can only wait for later transactions.
  - Why is T_2 waiting for T_3?
- In the Wound-Wait scheme, you can only wait for earlier transactions.
  - Why is T_1 waiting for T_2?

Comparison

- Wound-Wait appears to encourage waiting rather than aborting.
  - ❖ Intuition: it is more likely that a recent transaction will request a lock held by an older transaction than vice-versa.
- On the other hand, Wait-Die aborts only those transactions in the process of gathering locks.
  - ❖ Presumably they have the least work to waste in a rollback.

Distributed Databases

Three common architectures for distributed/parallel systems:

1. *Shared memory*: NUMA, processors on a shared bus.
2. *Shared disk*.
3. *Shared nothing*: networks of workstations, hypercubes, etc.

- Shared-nothing is most popular DB machine.
- (2) and (3) each require expensive "message passing" for communication.

Summary of Issues

1. *Data storage*:
   - Replicated? Good for heavily read data.
   - Relations partitioned among processors? Local autonomy, e.g., chain stores.
2. *Distributed Query Processing*:
   - Parallel pipelining, balancing load among processors.
   - Operations on relations that are not colocated. Move one to the other?
3. *Distributed commit.*
4. *Distributed locking.*

Two-Phase Commit

Assume a transaction T has parts executing at several sites. How can they agree to commit T?

Phase 1

1. Coordinator (site originating the transaction?) does:

   a) Log `<Prepare` T`>`.

   b) Send `prepare` T messages to all sites involved in T.

2. Each site receiving this message decides to commit T or abort T, and

   a) Logs either `<Ready` T`>` or `<Don't commit` T`>`, and

   b) Sends to the coordinator the corresponding message: either `ready` T or `don't commit` T.

Phase 2

1. Coordinator decides to commit or abort. Commit if all `ready` messages received; abort if at least one `abort` or timeout.

   a) Log `<Commit` T`>` or `<Abort` T`>`.

   b) Send `commit` T or `abort` T messages to all sites involved in T.

2. Sites receiving these messages log them and follow instruction.

Recovery

- If a site fails with `<Commit` T`>` or `<Abort` T`>` on its log, it can redo or undo, respectively.
- If a site fails with `<Don't commit` T`>` as its last T entry, it knows the coordinator could not have signaled commit, so it may abort T and undo.
- If a site fails with `<Ready` T`>` as its last T entry, it doesn't know what has happened to T; it must ask the coordinator.

Recovery When Coordinator Fails

Other sites elect a leader and try to figure out what happened.

- Any site with a `<Commit` T`>` or `<Abort` T`>` log record is decisive.
- Otherwise, if any site does *not* have a `<Ready` T`>` log record, then surely no commit instruction was sent anywhere by the coordinator, so abort T.
- Otherwise, no help, must wait for coordinator to recover.

Distributed Locking

Goal: reduce messages.

- Nonreplicated case: let local host lock its part of the DB.
  - ❖ 3 messages: request lock , grant lock, unlock + extra if cannot grant.
  - ❖ Each transaction uses 2PL.

Locking Replicated Elements

Goal: allow reads locally, using shared locks, while writes must be done to every copy, using an exclusive lock.

1. Centralized: coordinator, or primary-copy systems.
2. Distributed locks.

Coordinator

Allow one site to maintain all locks.

- Only 3 messages per granted lock/unlock.
- Bottleneck at coordinator?
- What if coordinator site crashes?

Primary Copy

One site holds lock for each replicated element.

- Similar to coordinator method, but bottleneck reduced because locking responsibility is shared.

Distributed Locks

Suppose A has n copies. Pick s = number of copies that must be locked for SL and x = number of copies to lock for XL.

- As long as $2x > n$ and $s + x > n$, we have the desired properties: only one exclusive lock on A, and no shared and exclusive lock on A.
  - Note that locally, there can be only one XL, but many SL's if no XL.
- Why it works?
  - Since $2x > n$, if two transactions had XL's on A, there would be at least one copy that had granted XL to both.
  - Since $s + x > n$, if one transaction had a SL and one an XL on A, then some copy granted both SL and XL at the same time.

- Interesting cases:
  - ❖ $s = 1; x = n$. *Read-locks-one; write-locks-all.* Favors systems where reading dominates.
  - ❖ $s = x = (n+1)/2$. *Majority locking.*
- Number of messages: $2s + 1$ for read, $2x + 1$ for write.

Comparisons

- Centralized lock schemes (coordinator, primary copy) minimize messages.
  - But read-locks-one comes close if read dominates.
- Distributed locking more resilient to crashes.
  - As long as s and x are each less than n, we can continue even if a site crashes.
  - Centralized systems prevent locks from being taken, even when a copy is available.