Bernstein & Goodman

This is the go-to reference for concurrency control in general, and distributed concurrency control in particular.

Textbook material -- this is the textbook!

- Partial Failure: DDBMS must account for "one site failing while the rest of the system continues to operate"
- "Computationally equivalent" executions
  - produces the same output (is the order of output relevant?)
  - has the same effect on the database
- Conflict Serializability
- *Serialization Order*: if it's serializable, there must be some order!
- Choosing unique timestamps in a distributed system w/o coordination

Basic Assumptions:

- Ordered, reliable delivery (per channel)
- Data may be partitioned and copied
- No semantics of computation -- just BEGIN, READ, WRITE, END
- "Private workspace", i.e. shadow copies.
- No WAL.
  - dm_write(x) on END. A single dm_write is like a Commit log -- no going back.
  - Assume there's an atomic write scheme (log-based).

Distributed Database Model

- TMs vs DMs. Bipartite -- TMs don't communicate with each other, DMs don't communicate with each other.
- Every transaction gets a "supervisory" or "master" TM.
- Single-site "Two-Phase Commit" (really a primitive WAL scheme)
  - Phase 1:
    - TM sends *prewrite* commands to DMs
    - DMs put private copies into secure storage (like WAL)
  - Phase 2:

        - TM sends *dm-write* commands to DMs.
        - DMs can use private copies to ensure durability
- Distributed "Two-Phase Commit"
    - Need to account for *partial failure*
    - Modification to Phase 1:
        - TM sends *prewrite* commands to DMS, with the list of the other DMs involved in the commit
    - Modification to Phase 2:
        - If the TM fails, the DMs that never got *dm-write*s can gang up with the other DMs to commit. "The details of this procedure are complex and appear in HAMM80" :-)
- Standard DDBMS processing:
    - BEGIN: TM creates a private workspace in some unspecified way
    - READ(X): TM checks the workspace for a copy of X. Return if found, else select some stored copy, and issue *dm-read(xi)* to the relevant DM.
    - WRITE(X, val): Update private workspace copy of *X* (create if necessary)
    - END: "two-phase commit"
        - *prewrite(xi)* the appropriate DM for all the updated *xi*'s
        - DMs store onto secure storage
        - Then issue *dm-writes*
        - DMs install new versions into DB

Separating rw and ww Concurrency Control

Definitions:
- rw conflict
- wr conflict
- ww conflict
- rwr and unspecified conflicts

Theorem 2 [Bern80a]
Execution E is serializable if (a) its rwr conflicts are acyclic, (b) its ww conflicts are acyclic, and (c) there is a total ordering of the transactions consistent with all rwr and ww conflicts.

Say what??

Consider this history:

W1(x) R2(x) W2(y) W1(y)

"The cornerstone of our paradigm for concurrency control."

OK, let's roll with it. We're going to enforce rwr and ww *synchronization* separately. "However, in addition to both rwr and ww being acyclic, there must also be *one* serial order consistent with *all* -> relations."

What will this be for T/O? For 2PL?

Distributed 2PL

Basic 2PL

- Obvious thing to do is co-locate schedulers (lock managers) with data
    - Readlock granted on *dm-read*, release when *dm-write*s go out (commit)
    - Writelock granted on *prewrite,* released on *dm-write*
    - Works for partitioned AND replicated data too!
- Read-lock one, write-lock many
    - You have to write all anyhow (?)
    - This seems arbitrary and we may revisit in later in discussions of *quorums*

Primary Copy 2PL

- [Ston79]: simplest possible scheme!
- Extra communication for readlocks, to talk to master even if you read elsewhere (e.g. local)
- BUT actually kind of nice for writelocks: only the *prewrite(x1)* sets a write lock (others do not).
    - Hint: so maybe Primary Copy is good for ww?
    - Do we care about saving the lock requests?

Voting 2PL

Now, let's talk *consensus*, specifically majority.
A lock is granted if a majority of TMs say so!
Consider w lock:

- Upon issuing *prewrite* requests, you wait until you get the majority, then you go.
    - Only 1 write can have the majority
    - If that transaction is not aborted (e.g. deadlock) it will get to its locked point and issue all its *dm-write*s at commit time
- Seems to solve ww

Why not use it for rwr?

- “Correctness only requires that a single copy of X be locked—namely the one being read—yet this technique requests locks on all copies. For this reason we deem Voting 2PL to be inappropriate for rw synchronization.”
- Huh?

Centralized 2PL

LaaS -- locking as a service!

Deadlock Prevention

Textbook stuff: Wound-Wait and Wait-Die. Generic priorities can be used. Timestamps are useful to ensure that priority goes up over retries.

Distributed Deadlock Detection

Looking for cycles in a distributed graph.
Suggestion 1: Centralized
Suggestion 2: Hierarchical

Protocol game: Pick a number between 1 and 10. Write it down. Now pick another, write it down. Draw an arrow from the first to the second.

Name a scheme that DOESN’T work here.
What can we say about deadlocks? About detecting them?

Timestamp Ordering

Timestamps

Each TM assigns a unique TS to every entering transaction.
- How?
- What’s the total order of time?
- Can new nodes join the system?
- What could cause the scheme to break?

Basic Single-Site T/O

For rw synchronization:
- Consider transaction T with TS issues *dm-read(x)*:

    - if TS < W-ts(x), *reject and abort T*
    - else R-ts(x):= max(TS, R-ts(x)) and output the *dm-read*
- Consider transaction with TS issues *dm-write(x):*
    - if TS < R-ts(x), *reject and abort T*
    - else W-ts(x) := max(TS, W-ts(x)) and output the *dm-write*

For ww synchronization

- if TS < W-ts(x) *reject and abort T*
- else W-tx(x) = TS; output the *dm-write*

Basic Distributed T/O

As above but:

1. accept/reject on *prewrite* (not on *dm-write*)
    - Accepting a *prewrite* is a promise to accept the *dm-write*
    - Essentially a write lock until commit!
2. *dm-read, dm-write* and *prewrite* are **buffered** by the scheduler.
    - Can release these from buffer when we *know* their time(stamp) has come
    - I.e. for *dm-read(x)*, when its TS precedes the earliest *prewrite* in the buffer (*min-P-ts(x)*)
    - I.e. for *dm-write*, when its TS precedes the earliest *dm-read* in the buffer (*min-R-ts(x)*)

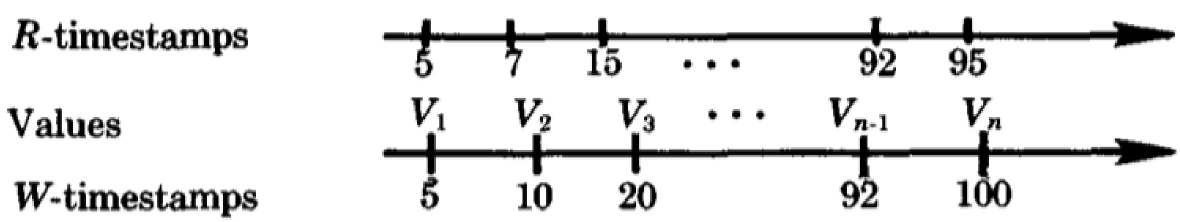
The Thomas Write Rule (TWR)

For ww synchronization, if TS < W-ts(x), *do not abort. Just ignore!*
Idea: this write might as well have arrived earlier, it still would have been overwritten. No harm in ignoring "obsolete" writes!

Note: ww synchronization with TWR requires no 2PC -- all *prewrites* can be accepted, no *dm-writes* ever buffered.

MultiVersion T/O

This is the fun one!
Easiest to explain with a picture:



Process *dm-read(x)* with TS=95.
Process *dm-write(x)* with TS=93.

The (only!) problematic situation:
You may not install a Write between a W-ts and an R-ts in a timeline. If you try, you are aborted.

Proof of correctness: demonstrate that the committed transactions are equivalent to the serial TS-ordered schedule.

Let W be an out-of-order *dm-write(x).* That is, some *dm-read(x)* with higher timestamp arrived before this. Since W was not rejected, that means there was an intervening write after W and before the *dm-read(x)* in the schedule. So W had no effect on that read.

Let R be an out-of-order *dm-read(x)*. That is, some *dm-write(x)* with higher timestamp arrived before this. R will ignore all writes greater than *ts(R),* so will read the same data it would have in the serial execution.

NOTES:

- Reads are *never* rejected in rw!
- Writes are *never* rejected in ww -- hence no need for 2-phase commit!

Pros?
Cons?

Conservative TO

I always find this far-fetched and odd. Lots of constraints. “Optimizations” like transaction classes only make it weirder and more complicated. Some day you can convince me I’m wrong.

Timestamp Management

- Representation?
- Garbage Collection/Compaction?

Combinations?

Yeah maybe.

“Interface” to get a serialization order mixing 2PL and T/O?

- Assign timestamps at locked points!
    - L-TS for each lock request
    - TS is assigned to be bigger than any L-ts for the transaction
- Any problems with this?

Let's Revisit the Postgres Storage Manager

- No distribution.
- 2PL
- MV data
    - Xmin, Xmax
    - Tmin, Tmax: similar to the TS for mixed 2PL and TO?!
- What's the effect on serialization order?