Lecture 6 - Concurrency Control Schemes

Today's reading: Bernstein and Goodman

- Survey of CC schemes until 1981, specifically distributed CC schemes
    - Variants of locking and timestamp ordering
    - Two important schemes, validation/OCC and MVCC were not fully fleshed out by then.
- But this sets up most of the "primitives" for future CC approaches
    - And as part of it separates out read/write sync from write/write sync

Basics

A lot of this reading is "textbook"-material - and this is the textbook!

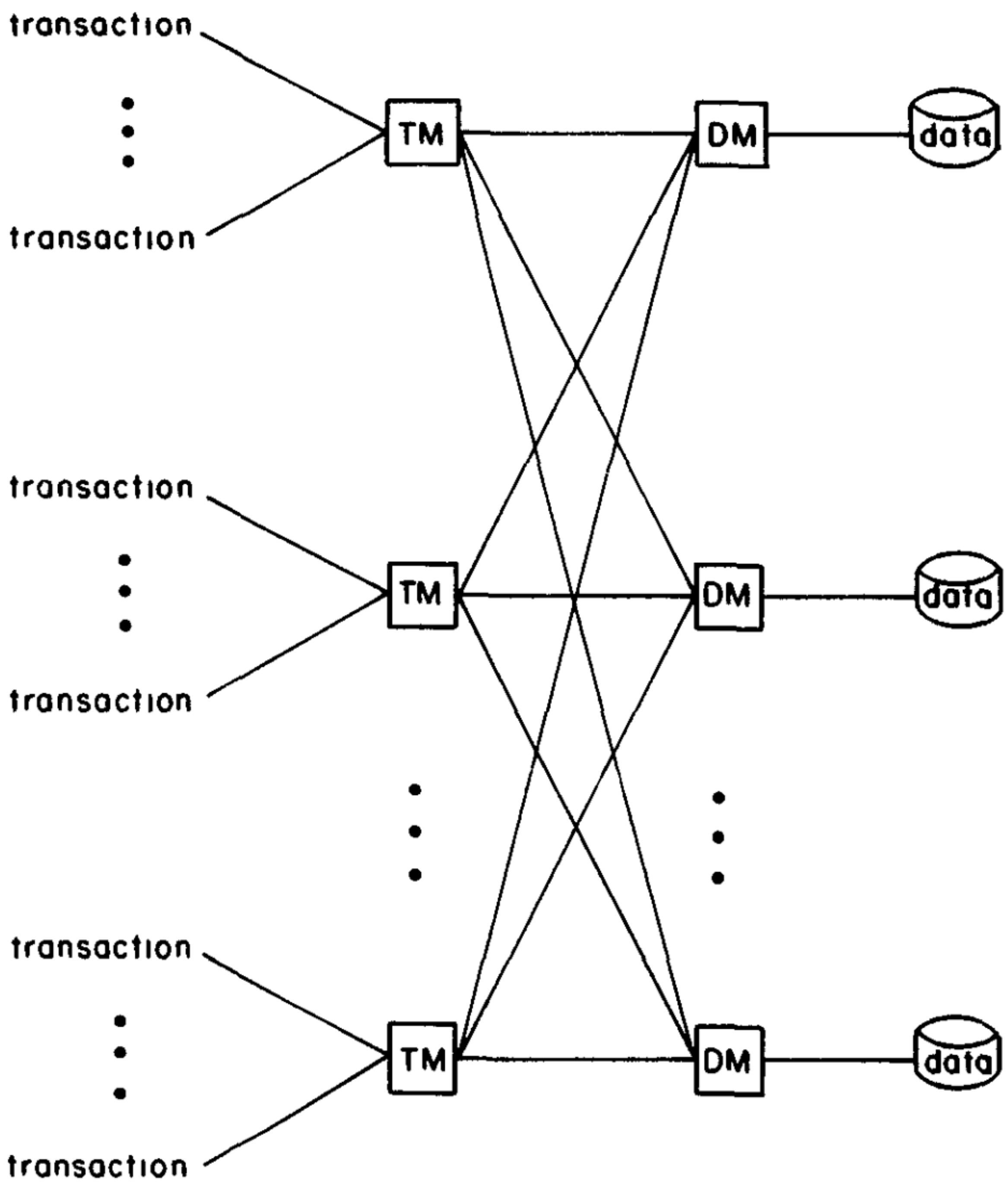
- Starts with the notion of "computationally equivalent" executions
    - These executions must have the same "outputs" - aka reads
    - And must have the same effect on the database - aka writes
    - The paper's main contribution is to separate out these concepts
- For any CC scheme, there has to be a *serialization order*

And distributed systems textbook material:

- Partial Failure: DDBMS must account for “one site failing while the rest of the system continues to operate”
- Timestamps:
    - Choosing unique timestamps in a distributed system w/o coordination
    - Forgetting (garbage collecting) monotonically increasing timestamps that are dominated
- Early discussions of consensus and quorums

Setup

- Abstracts out many real-world details - mathematical model to reason about correctness and algorithms
- Many Transaction Managers (TMs) and many Data Managers (DMs)
    - n:m bipartite graph



-
    - each DM is responsible for various items X, Y, Z
    - but each item can be redundantly stored at various sites, e.g., X_1, X_2, ...
- Transactions are issued to a single TM, TMs can communicate with DMs, but TMs don't communicate with each other, and likewise for DMs
- Transactions can R/W items
- Centralized setup (single TM and single DM):
    - TM initializes private workspace (PW) for T; essentially shadow copies
    - `READ(X)`, if X is not available in PW, translates to `dm_read(x)`, and stored in PW
    - `WRITE(X,val)`, similar, updates are made in PW
    - `END` = commit; we do this via 2PC:
        - Phase 1 (prepare): issues `prewrite` commands in DM for each updated item `x`
        - Phase 2 (commit): issue `dm_write(x,val)` for each updated item - each `dm_write` permanently copies prewrites into locations

    - This is all compatible with WAL
- Distributed setup (single TM, many DMs):
  - For reads, can pick some stored copy in some DM, say `x_i`, and issue `dm_read(x_i)`
  - Phase 1:
    - TM sends prewrite commands to all DMs that store `x`
    - 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”
  - To account for failure:
    - Phase 1 prewrites also indicate which other DMs are involved in commit
    - In Phase 2, if the TM fails, the DMs that never got dm-writes can gang up with the other DMs to commit.
      - *“The details of this procedure are complex and appear in HAMM80”* :-)

Definitions of Serializability

- You should know about the definitions of Serializability vs Conflict Serializability!
- Q: What is serializability? What is conflict serializability?
- Two operations conflict if they operate on the same item and one of them is a `dm_write`
- Detour: view serializability
  - Weaker than conflict serializability.
  - Two schedules are view-equivalent if:
    - They read from the same writes
    - They have the same final write for every item
  - Essentially w-w conflicts don't matter if they aren't read from "*if a tree falls in the forest*"
  - Conflict-serializable ⊂ View-serializable.
  - Thomas Write Rule schedules are view-serializable but not conflict-serializable.
- Example schedules:
  - Serializable, but not VS or CS: $W_1(X, 100), R_2(X), R_3(X), W_3(X), W_2(X)$
  - Serializable and VS but not CS: $W_1(X, 100), W_2(X, 100), R_1(X)$
- Distributed stuff makes things hard:
  - DM A: $r_1(x_1)w_1(y_1)r_2(y_1)w_3(x_1)$
  - DM B: $w_2(z_2)w_1(y_2)$
  - DM C: $w_2(z_3)r_3(z_3)$
  - Here, each individual site is serial but there is no total order consistent with this
- Theorem 1: A set of logs (an execution E) is serializable if there is a total ordering of T_i s.t. for each pair of conflicting operations in two txns, they match the total order.

Separating rw and ww Concurrency Control

- rw conflict: $T_i \rightarrow_{rw} T_j$ if for some log, T_i reads something that T_j then writes
- wr conflict: similar
- ww conflict: $T_i \rightarrow_{ww} T_j$ if for some log, T_i writes something that T_j then writes
- “rwr” (rw ∨ wr): $T_i \rightarrow_{rwr} T_j$
- $T_i \rightarrow T_j$ union of all; unspecified conflicts
- Theorem 1 (restated): E is serializable if there is a total ordering that is consistent with $\rightarrow$

Theorem 2 (from Bern80a: Timestamp based algorithms for concurrency control in distributed database systems)

> 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.

Trivially follows from Theorem 1 based on (c)...

- but helpful restatement in that rwr and ww conflicts can synchronized independently
  - i.e., each ensuring acylicity
- as long as there (c) is satisfied: a total ordering consistent with rwr and ww is present
- *"This serial order is the cement that binds together the rw and ww synchronization techniques"*

Distributed 2PL

- First, 2PL has a growing and shrinking phase.
  - The point where growing stops is the *locked point*
  - For any E where 2PL is used, the $\rightarrow_{rwr}$ and $\rightarrow_{ww}$ relation is identical to that of a serial execution E where a transaction executes at its locked point

Basic 2PL

- Obvious thing to do is co-locate schedulers (lock managers) with data
- Read-lock granted on dm-read, release when dm-writes go out (i.e., start of shrinking phase)
- Write-lock granted on prewrite, released on dm-write
- Works for partitioned (shared-nothing) AND replicated data too!
  - For replicas, read-lock one, write-lock all copies
    - You have to write all anyhow

Primary Copy 2PL

- From Distributed Ingres (Stonbraker'79)
- One copy of each logical item ends up being *primary copy* (x1)
- Extra communication for read-locks, to talk to primary even if you read elsewhere (e.g. local)
  - So communication with two DMs
- BUT actually kind of nice for write-locks:
  - only `prewrite(x1)` sets a write lock (others do not).
- Benefits for ww:
  - All ww conflicts are serialized at a single location.
  - Write-lock contention is centralized.
  - Avoids distributed write-write deadlock cycles.

Voting 2PL

- Now, let’s talk consensus, specifically majority.
- A lock is granted if a majority of TMs say so!
- Consider w lock under 2PC:
  - Upon issuing prewrite requests, you wait until you get the majority, then proceed
  - Only 1 txn can have the majority, and therefore the w lock

- If that transaction is not aborted (e.g. deadlock) it will get to its locked point and issue all its dm-writes at commit time
- Seems to solve ww
  - Better than Basic 2PL in that you can proceed without getting all copies to respond
- 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."*
  - This seems to assume that quorums are write-all/read-one (propagate write to all copies)

Centralized 2PL

- LaaS -- locking as a service!

Deadlocks

- Main issue with locking schemes is deadlocks
- Can either prevent or detect(and fix)

Deadlock Prevention

- CS186 stuff: Wound-Wait and Wait-Die.
  - remember it's `<low> - <hi>` priority
  - think low priority is youngsters, hi priority is elderly
- Wound-Wait:
  - Preemptive technique
  - if youngster arrives 2nd, they kill the elderly; if elder arrives 2nd, waits.
- Wait-Die:
  - Non-preemptive
  - if youngster arrives 2nd, waits; if elder arrives 2nd, aborts
- Restart with same age (increasing priority over time)!
- 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
- One issue: deadlocks may be "stale"

Timestamp Ordering

Timestamps

- Each TM assigns a unique TS to every entering transaction.
- This dictates the serialization order
- One scheme proposed:
  - Take local clock, append unique TM identifier
  - Q: What can go wrong with this?
    - Clock skew
  - Some modern solutions: Lamport clocks, Hybrid Logical Clocks, Centralized allocators

Basic Single-Site T/O

- Every data item x has a `R-ts(x)` and a `W-ts(x)` which are the timestamps of the highest-numbered transactions that Read and Wrote x respectively.
- 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
- If using Basic T/O need both conditions to be satisfied for a dm-write

Basic Distributed T/O

As above but:

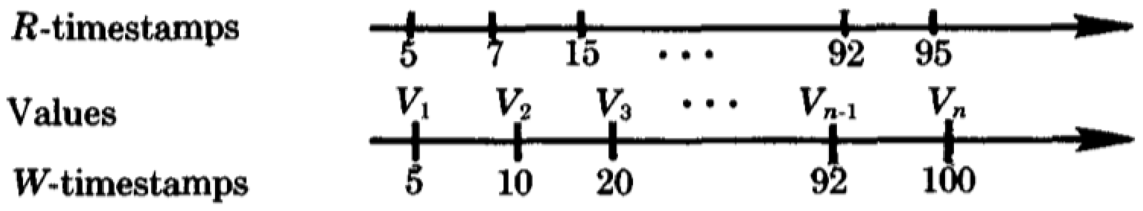
- Accept/reject on prewrite (not on dm-write)
  - Accepting a prewrite is a promise to accept the dm-write
  - Essentially a write lock until commit!
- dm-read, dm-write and prewrite are buffered by the scheduler instead of immediately output
  - `min-R-ts(x), min-W-ts(x), min-P-ts(x)`
  - Can release 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.
- Modern influences: some eventual consistency systems

MultiVersion T/O (from Reed'78)

- Easiest to explain with a picture:
- For each item there are a set of R-ts's (all read timestamps) and a set of <W-ts, value> pairs - the *versions* (all write timestamps and versions)



- Process dm-read(x) with TS=95.
  - Simply use TS=92 - largest timestamp less than 95
  - In general reads are never rejected
- Process dm-write(x) with TS=93.
  - Here, you cannot install a write at 93 because at 95 you read V_{n-1}
  - This is 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.
    - Said another way, if in the interval between (93, 100), there is a read, then we have to reject
- 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
- Writes are never rejected in ww -- hence no need for 2-phase commit!
- Q: Pros? Cons?
  - No read blocking/aborts
  - No deadlocks
  - No ww rejection
  - Version storage overhead
  - Timestamp management complexity
  - Garbage collection needed
  - Potential write aborts due to read history
- Modern context: many systems use MVCC

Conservative TO

- Pretty complicated and impractical - safely ignore

Timestamp Management

- Representation? Where do we store them?
- Forgetting/Garbage Collection/Compaction?

Combinations

- All theoretically possible! Devil in the details
- “Interface” to get a serialization order mixing 2PL and T/O?
  - Assign timestamps to 2PL at locked points!
  - Implementation:
    - L-ts for each lock request
    - TS is assigned to be bigger than any L-ts for the transaction
- One mixed method: Basic 2PL, TWR
  - each item has an L-ts and a W-ts
  - Read:
    - `dm-read(xi)`
    - read-lock requested (implicitly)
    - `L-ts(xi)` returned
  - Write:

        - prewrite all replicas
        - acquire write-locks
        - for each such write-lock, collect `L–ts`
    - Assign ts(T) > all L-ts
    - dm-writes processed via TWR:
        - If `ts(T) > W–ts(xj)` → apply
        - Else ignore
- From paper: *The interesting property of this method is that writelocks never conflict with writelocks. The writelocks obtained by prewrites are only used for rw synchronization, and only conflict with readlocks. This permits transactions to execute concurrently to completion even if their writesets intersect. Such concurrency is never possible in a pure 2PL method.*
- Any problems with this?

Takeaways

- Can intermix (at least in theory) different methods for rwr and ww synchronization
- Formalize distributed serializability
- Serializability playing nice with logging - through notion of prewrites
- TWR helps make ww synchronization easy
- MVTO helps make rwr synchronization easy
- In modern practice:
    - Systems implicitly mix ideas from Bernstein Goodman, e.g., MVCC for rwr, locking for ww
    - MVCC not necessarily done via MVTO (as we will see)
    - OCC is used instead of TO-based schemes for limited overhead (more next time!)

Exercise: 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?