Concurrency Control

CS262, Fall 2011.

Joe Hellerstein

I. Concurrency & controlling order.

- Order in a traditional programming abstraction.
- Why is concurrency hard?
- Why does order matter? What orders matter? These are deep questions.
- Step back: ignore shared state, side-channels, etc. Assume no sharing, explicit communication,

II. Inter-Agent Communication: Rendezvous/Join

The Ancient Communication Scenario: 1 sender, 1 receiver.

1. **SpatioTemporal Rendezvous**: Timed Smoke Signals on 2 mountaintops. Agent 1 has instructions to generate a puff at certain time (and place). Agent 2 has instructions to watch at same time (and place).
2. **Receiver Persist**: Smoke Signal and Watchtower. Agent 2 *waits* in watchtower for smoke-signal in the agreed-upon place. Agent 1 puffs at will. (What if too early?)
3. **Sender Persist**: WatchFires. Agent 1 can light a watchfire. Agent 2 checks for the watchfire when convenient. (What if too early?)
4. **Both Persist**: Watchfire and Watchtower. Both persist.

Upshot: (1) one-sided persistence allows asynchrony on the other side. (2) BUT persistent party must span the time of the transient party. (3) Without any temporal coordination, persistence of both sides is the way to guarantee rendezvous: second arrival necessarily overlaps first.

This is *very* much like a choice of join algorithm in a database engine!

Next Question: Multiple communications. Can we guarantee order? With temporal rendezvous, the agents coordinate ("share a clock"). With receiver persist, receiver can observe/maintain sender order. Sender persist much more common, but doesn't (organically) preserve order!

Beware: In most computing settings, this reasoning is flipped. Storage is a given, and ends up being an *implicit* comm channel. Must control operations on it across agents.

III. On State and Persistence

What is state?

- all values of memory, registers, stack, PC, etc.

What is persistent state?

- persistent state: state that is communicated across *time*. E.g. across PC settings. Write is a Send into the future, Read is Receive.
- how does DRAM persist state? sender-persist! And recall: Sender-Persist doesn't guarantee order!!
- (aside: how does OS do disk persistence?)

IV. How to control communication?

- Prevention: Change the space or the time of rendezvous!
    - writer uses private space (shadow copies, copy-on-write, multiversion, functional

programming, etc.)
    - and/or a *protocol* arranges appropriate "time" for rendezvous (e.g. agree on timing scheme via locks, or on a publication location scheme).
        - these protocols typically rely recursively on controlled communication! (e.g. callback or continuation on lock release, e.g. publication of a fwding pointer in a well-known location followed by polling, etc.)
- Detection: Identify an undesirable communication (one that violates ordering constraints), and somehow *undo* an agent.
    - Ensure that all communication it performed was invisible, or recursively undo.

V. Acceptable orders of communication

1. What are the basic operations on state? Communication — i.e. data dependencies.
2. What orders matter for these operations? It depends!
3. A client's view of the acceptable orderings: serial schedules, serializability.
4. Conflicts. Assume two clients, one server. What interleavings do not preserve client views? R-W and W-W conflicts. *Conflict Serializability is a conservative test for Serializability.*

VI. Locking as the "antijoin antechamber"

- Idea: Since communication is rendezvous, let's prevent inappropriate rendezvous — no conflicts! Result: equiv. to a serial schedule based on time-of-commit. (Dynamic!)
- consider a rendezvous in space: e.g. two spies passing a message in a locked room.
- before you can "enter" the rendezvous point:
    - check that no conflicting action is currently granted access to the rendezvous point (not-in = "antijoin").
        - if yes conflict with the granted actions (join), wait in the antechamber
        - if no conflict, mark yourself (persistently until EOT) as having been granted access. Enter the rendezvous point to do your business
    - when granted transactions depart, choose a mutually compatible group of transactions to let into the granted group.
- some games we can play with space
    - could "move" the rendezvous point in space (locking proxy? lock caching?)
    - could further delay rendezvous in time based on other considerations
- You should know the multi-granularity locks from Gray's paper! (Took the MySQL guys years to learn this.)

VII. Timestamp ordering (T/O)

Streaming symmetric join of reads and writes, outputting data and aborts.

T/O. Predeclare the schedule via timestamp (wall-clock? sequence? out-of-sequence?) at transaction birth. No antijoin = no waiting! But restart when reordering detected…

- Keep track of r-ts and w-ts for each object: *r-ts: max counter rather than persistence.*
- reject (abort) stale reads.
- reject (abort) delayed writes to objects that have later r-ts (write was already missed). Can allow (ignore) delayed writes to objects that have a later w-ts tho. (Thomas Write Rule.)

Multiversion T/O: Even fewer restart scenarios

- Keep sets of r-ts, and . This is kind of like symmetric persistence.
- Reads never rejected! (more liberal than plain T/O)
- Write rule on x: interval(W(x)) = [ts(W), mw-ts] where mw-ts is the next write after W(x) — i.e. Min_x w-ts(x) > ts(W). If any R-ts(x) exists in that interval, must reject write. I.e. that read shoulda read this write. Note more liberal than plain T/O since subsequent writes may "mask" this one for even later reads.
- GC of persistent state: anything older than Min TS of live transactions.

Note: no waiting (blocking, antijoin) in Multiversion T/O.

VIII. Optimistic concurrency: antijoin with history

Schedule predeclared by acquiring timestamp before validation.

- go through **basic OCC**
- idea: persist read history, copy on write, and try to antijoin over time window on commit to ensure "conflicts in a particular order"
- OCC antijoin predicates are complicated — transaction numbers, timestamps for read/write phases, read/write sets:
    - Valdating Tj. Suppose TN(Ti) < TN(Tj). Serializable if *for all uncommitted Ti* one of the following holds ("for all" is like anti join — i.e. notin(set-of-uncommited-Ti, NONE of the following hold))
    - Ti completes writes before Tj starts reads (prevents rw and ww conflicts out of order).
    - WS(Ti) \cap RS(Tj) = empty, Ti completes writes before Tj starts writes (no rw conflicts in order, prevent ww conflicts out of order)
    - WS(Ti) \cap RS(Tj) = empty, WS(Ti) \cap WS(Tj) = empty, Ti finishes read before Tj starts read (no ww conflicts, prevent backward rw conflicts).
    - GC?

Note: the antijoin in OCC is over all uncommitted transactions. That means that during validation, the set of uncommitted transactions must not change. I.e. only one transactions can be validating at a time. I.e. implicitly there's an X lock on the "system validating" resource.

IX. ACID

Not an axiom system. Just a mnemonic. Don't over-interpret. Remember serializability, and guarantee commit/abort.

- Atomicity: visibility of all effects at one time
- Consistency: based on state constraints
- Isolation: application writer need not reason about concurrency
- Durability: commit is a contract

X. What is all this discussion about NoSQL and Loose Consistency?

Move some coordination out of read/write storage, and into application logic.

- *ACID*: build application logic on a foundation of theory+system for controlling order.
- *Loose Consistency*: build application logic with controlled order over a loose foundation. I.e. Abandon Isolation, require programmer to worry about concurrency issues that they might care about.
- Missing from loose consistency: a foundation of theory+SW to ensure app logic meets desired constraints.
- What about Gray's transactional "degrees of consistency"?
    - A Dirty-Data Description of Degrees of Consistency Transaction T sees degree X consistency if…
        - Degree 0: T does not overwrite dirty data of other transactions
        - Degree 1: T sees degree 0 consistency, and T does not commit any writes before EOT
        - Degree 2: T sees degree 1 consistency, and T does not read dirty data of other transactions
        - Degree 3: T sees degree 2 consistency, and other transactions do not dirty any data read by T before T completes.
    - Note there were long-unresolved problems with Gray's definitions. See the Adya paper in the reading list.

    - **NOTE**: if everybody is at least degree 1, than different transactions can CHOOSE what degree they wish to "see" without worry. I.e. true isolation is an option for all.

- Eventual Consistency?
    - Typically in regard to R/W reasoning on replicated state. A good reference is **Terry, et al. PDIS 1994**
    - At higher levels of abstraction than R/W, what orders matter?

XI. What orders really matter? CALM Theorem.

CALM Theorem: Consistency and Logical Monotonicity. No coordination (i.e. order control) needed for Monotonic code. Non-monotonic code needs to be protected by coordination.