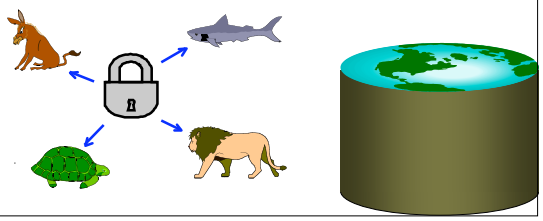
Alternative Concurrency Control Methods

R&G - Chapter 17



Roadmap

- So far:
  - Correctness criterion: serializability
  - Lock-based CC to enforce serializability
    - Strict 2PL
    - Deadlocks
    - Locking granularities
    - Tree locking protocols
    - Phantoms
- Today:
  - Alternative CC mechanisms

Optimistic CC (Kung-Robinson)

Locking is a conservative approach in which conflicts are prevented.

- Disadvantages:
  - Lock management overhead.
  - Deadlock detection/resolution.
  - Lock contention for heavily used objects.
- Locking is "pessimistic" because it assumes that conflicts will happen.
- What if conflicts are rare?
  - We might get better performance by not locking, and instead checking for conflicts at commit time.

Kung-Robinson Model

- Xacts have three phases:
  - READ: Xacts read from the database, but make changes to private copies of objects.
  - VALIDATE: Check for conflicts.
  - WRITE: Make local copies of changes public.

Validation

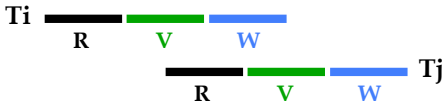
- *Idea:* test conditions that are sufficient to ensure that no conflict occurred.
- Each Xact assigned a numeric id.
  - Just use a **timestamp**.
  - Assigned at end of READ phase.
- ReadSet(Ti): Set of objects read by Xact Ti.
- WriteSet(Ti): Set of objects modified by Ti.

Test 1

- For all i and j such that Ti < Tj, check that Ti completes before Tj begins.

Test 2

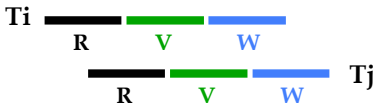
- For all i and j such that Ti < Tj, check that:
  - Ti completes before Tj begins its Write phase **AND**
  - WriteSet(Ti) ⋂ ReadSet(Tj) is empty.



Does Tj read dirty data? Does Ti overwrite Tj's writes?

Test 3

- For all i and j such that Ti < Tj, check that:
  - Ti completes Read phase before Tj does **AND**
  - WriteSet(Ti) ⋂ ReadSet(Tj) is empty **AND**
  - WriteSet(Ti) ⋂ WriteSet(Tj) is empty.



Does Tj read dirty data? Does Ti overwrite Tj's writes?

Applying Tests 1 & 2: Serial Validation

- To validate Xact T:

```
valid = true;
// S = set of Xacts that committed after Begin(T)
//    (above defn implements Test 1)
//The following is done in critical section
< foreach  Ts in S do {
   if ReadSet(T) intersects WriteSet(Ts)
        then valid = false;
   }
   if valid then { install updates; // Write phase
                   Commit T } >
              else Restart T
```

start of critical section

end of critical section

Comments on Serial Validation

- Applies Test 2, with T playing the role of Tj and each Xact in Ts (in turn) being Ti.
- Assignment of Xact id, validation, and the Write phase are inside a **critical section**!
  - Nothing else goes on concurrently.
  - So, no need to check for Test 3 --- can't happen.
  - If Write phase is long, major drawback.
- Optimization for Read-only Xacts:
  - Don't need critical section (because there is no Write phase).

Overheads in Optimistic CC

- Record xact activity in ReadSet and WriteSet
  - Bookkeeping overhead.
- Check for conflicts during validation
  - Critical section can reduce concurrency.
- Make validated writes "global"
  - Can reduce clustering of objects.
- Restart xacts that fail validation.
  - Work done so far is wasted; requires clean-up.

Optimistic CC vs. Locking

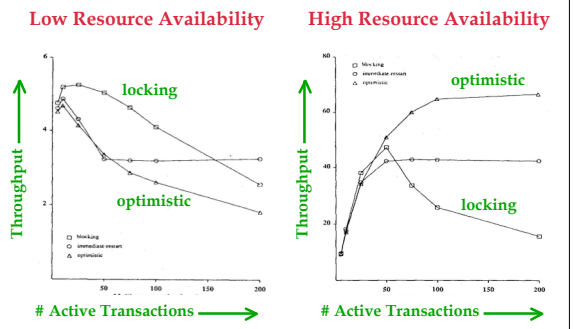
- Despite its own overheads, Optimistic CC can be better if conflicts are rare
  - Special case: mostly read-only xacts
- What about the case in which conflicts are not rare?
  - The choice is less obvious …

Optimistic CC vs. Locking (for xacts that tend to conflict)

- Locking:
  - Delay xacts involved in conflicts
  - Restart xacts involved in deadlocks
- Optimistic CC:
  - Delay other xacts during critical section (validation+write)
  - Restart xacts involved in conflicts
- Observations:
  - Locking tends to delay xacts longer (duration of X locks usually longer than critical section for validation+write)
    → could decrease *throughput*
  - Optimistic CC tends to restart xacts more often
    → more "wasted" resources
    → decreased throughput if resources are scarce

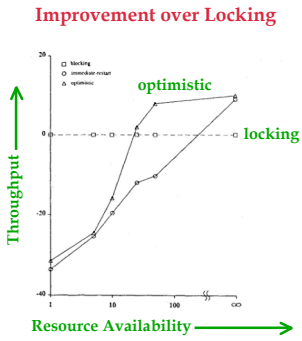
Choice should depend on resource availability

Choice Depends on Resource Availability



Source: [Agrawal, Carey, Livny]

Choice Depends on Resource Availability



Source: [Agrawal, Carey, Livny]

Two Other CC Techniques

Timestamp CC:

- Give each **object** a read-timestamp (RTS) and a write-timestamp (WTS)
- Give each **xact** a timestamp (TS) when it begins
  - Check that conflicting actions on an object always occur in order of xact timestamp.
  - If a xact tries to violate this condition, restart it.

Multiversion CC:

- Let writers make a "new" copy while readers use an appropriate "old" copy.
  - Advantage is that readers don't need to get locks
  - Oracle uses a form of Multiversion CC.

Timestamp CC: When Xact T wants to read Object O

- If TS(T) < WTS(O), this violates timestamp order of T w.r.t. writer of O.
  - Abort T and restart it with a new, larger TS.
  - *(Why assign new TS ?)*
- If TS(T) > WTS(O):
  - Allow T to read O.
  - Reset RTS(O) to max(RTS(O), TS(T))
- Change to RTS(O) on reads must be written to disk! This and restarts represent overheads.

Timestamp CC: When Xact T wants to Write Object O

- If TS(T) < RTS(O), this violates timestamp order of T w.r.t. reader of O; abort and restart T.
- If TS(T) < WTS(O), violates timestamp order of T w.r.t. writer of O.
  - **Thomas Write Rule: We can safely ignore such outdated writes; need not restart T! (T's write is effectively followed by another write, with no intervening reads.)**
    Allows some view serializable but non conflict serializable schedules:
- Else, allow T to write O.

T1	T2
R(A)	
	W(A)
	Commit
Commit	

Timestamp CC and Recoverability

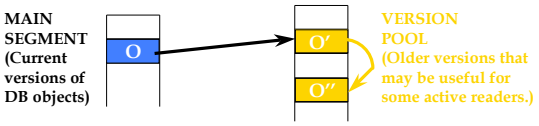
- Recoverable schedule: xacts commit only after (and if) all xacts whose changes they read commit
  - A weaker condition than Avoid Cascading Rollback

T1	T2
W(A)	
	R(A)
	W(B)
	Commit

- ❖ Unrecoverable schedules are allowed by Timestamp CC !
- Timestamp CC can be modified to allow only recoverable schedules:
  - Block readers T (where TS(T) > WTS(O)) until writer of O commits.
- Similar to writers holding X locks until commit, but still not quite 2PL.

Multiversion Timestamp CC

- **Idea:** Let writers make a "new" copy while readers use an appropriate "old" copy:

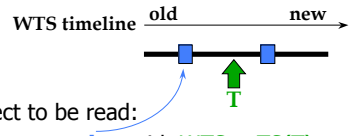


- ❖ Readers are always allowed to proceed.
  - But may be blocked until writer commits.

Multiversion CC (Contd.)

- Each version of an object has its writer's TS as its WTS, and the TS of the Xact that most recently read this version as its RTS.
- Versions are chained backward; we can discard versions that are "too old to be of interest".
- Each Xact is classified as Reader or Writer.
  - Writer *may* write some object; Reader never will.
  - Xact declares whether it is a Reader when it begins.

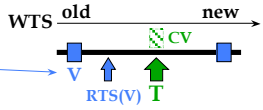
Reader Xact



- For each object to be read:
  - Finds **newest version** with WTS < TS(T).
- Reader Xacts are never restarted.
  - However, might block until writer of the appropriate version commits.

Writer Xact

- To read an object, follows reader protocol.
- To write an object:
  - Finds **newest version V** s.t. WTS < TS(T).
  - If RTS(V) < TS(T), T makes a copy CV of V, with WTS(CV) RTS(CV) = TS(T). (Readers are blocked until T commits.)
  - Else, reject write.



Summary

Optimistic CC using end-of-xact "validation"

- Good if:
  - Read-dominated workload
  - System has lots of extra resources
- Most real systems use locking

Summary (Contd.)

Another alternative: Timestamp CC

- Decide logical xact execution order when xacts enter system
- Enforce by comparing xact timestamps with object timestamps

Variant: Multiversion CC

- Keep out-of-date versions of objects, so "old" readers don't have to restart (they can run "in the past")
- *Oracle* uses a flavor of multiversion CC