CS262A: Advanced Topics in Computer Systems
Joe Hellerstein, Spring 2008
UC Berkeley

Concurrency Control: Locking, Optimistic, Degrees of Consistency

Transaction Refresher

Statement of problem:

- **Database:** a fixed set of named resources (e.g. tuples, pages, files, whatever)
- **Consistency Constraints:** must be true for DB to be considered "consistent". Examples:
  - sum(account balances) = sum(assets)
  - P is index pages for R
  - ACCT-BAL > 0
  - each employee has a valid department
- **Transaction:** a sequence of actions bracketed by begin and end statements. Each transaction ("xact") is assumed to maintain consistency (this can be guaranteed by the system).
- **Goal:** Concurrent execution of transactions, with high throughput/utilization, low response time, and fairness.

A transaction schedule:

T1	T2
read(A)	
A = A - 50	
write(A)	
	read(A)
	temp = A*0.1
	A = A - temp
	write(A)
read(B)	
B = B + 50	
write(B)	
	read(B)
	B = B + temp
	write(B)

The system "understands" only reads and writes; cannot assume any semantics of other operations.
Arbitrary interleaving can lead to:
- temporary inconsistencies (ok, unavoidable)
- "permanent" inconsistencies, that is, inconsistencies that remain after transactions have completed.

Some definitions:
- Schedule: A "history" or "audit trail" of all committed actions in the system, the xacts that performed them, and the objects they affected.
- Serial Schedule: A schedule is serial if all the actions of each single xact appear together.
- Equivalence of schedules: Two schedules S1, S2 are considered equivalent if:
    1. The set of transactions that participate in S1 and S2 are the same.
    2. For each data item Q in S1, if transaction Ti executes read(Q) and the value of Q read by Ti was written by Tj, then the same will hold in S2. [reads are all the same]
    3. For each data item Q in S1, if transaction Ti executes the last write(Q) instruction, then the same holds in S2. [the same writers "win"]
- Serializability: A schedule S is serializable if there exists a serial schedule S' such that S and S' are equivalent.

One way to think about concurrency control – in terms of dependencies:
1. T1 reads N ... T2 writes N: a RW dependency
2. T1 writes N ... T2 reads N: a WR dependency
3. T1 writes N ... T2 writes N: a WW dependency

Can construct a "serialization graph" for a schedule S (SG(S)):
- nodes are transactions T1, ..., Tn
- Edges: Ti -> Tj if there is a RW, WR, or WW dependency from Ti to Tj

Theorem: A schedule S is serializable iff SG(S) is acyclic.

Locking

A technique to ensure serializability, but hopefully preserve high concurrency as well. The winner in industry.

Basics:
- A "lock manager" records what entities are locked, by whom, and in what "mode". Also maintains wait queues.
- A well-formed transaction locks entities before using them, and unlocks them some time later.

Multiple lock modes: Some data items can be shared, so not all locks need to be exclusive.
Lock compatibility table 1:
Assume two lock modes: shared (S) and exclusive (X) locks.

	S	**X**
S	T	F
X	F	F

If you request a lock in a mode incompatible with an existing lock, you must wait.
Two-Phase Locking (2PL):
- Growing Phase: A transaction may obtain locks but not release any lock.
- Shrinking Phase: A transaction may release locks, but not obtain any new lock. (in fact, locks are usually all released at once to avoid "cascading aborts".)

Theorem: If all xacts are well-formed and follow 2PL, then any resulting schedule is serializable
(note: this is if, not if and only if!)

Some implementation background
- maintain a lock table as hashed main-mem structure
- lookup by lock name
- lookup by transaction id
- lock/unlock must be atomic operations (protected by critical section)
- typically costs several hundred instructions to lock/unlock an item
- suppose T1 has an S lock on P, T2 is waiting to get X lock on P, and now T3 wants S lock on P. Do we grant T3 an S lock?

Well, starvation, unfair, etc. Could do a little of that, but not much. So...
- Manage FCFS queue for each locked object with outstanding requests
- all xacts that are adjacent and compatible are a compatible group
- The front group is the granted group
- group mode is most restrictive mode amongst group members
- Conversions: often want to convert (e.g. S to X for "test and modify" actions). Should conversions go to back of queue?
- No! Instant deadlock (more notes on deadlock later). So put conversions right after granted group.

Granularity of Locks and Degrees of Consistency

Granularity part is easy. But some people still don't know about it (e.g. MySQL).
Be sure to understand IS, IX, SIX locks. Why do SIX locks come up?

First, a definition: A write is committed when transaction if finished; otherwise, the write is dirty.

A Locking-Based Description of Degrees of Consistency:
This is not actually a description of the degrees, but rather of how to achieve them via locking. But it's better defined.
- Degree 0: set short write locks on updated items ("short" = length of action)
- Degree 1: set long write locks on updated items ("long" = EOT)
- Degree 2: set long write locks on updated items, and short read locks on items read
- Degree 3: set long write and read locks

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.

Examples of Inconsistencies prevented by Various Degrees
Garbage reads:

> T1: write(X); T2: write(X)
> Who knows what value X will end up being?
> *Solution:* set short write locks (degree 0)

Lost Updates:

T1: write(X)
T2: write(X)
T1: abort (physical UNDO restores X to pre-T1 value)
At this point, the update to T2 is lost
Solution: set long write locks (degree 1)

Dirty Reads:

T1: write(X)
T2: read(X)
T1: abort

Now T2's read is bogus.
Solution: set long X locks and short S locks (degree 2)
Many systems do long-running queries at degree 2.

Unrepeatable reads:

T1: read(X)
T2: write(X)
T2: end transaction
T1: read(X)

Now T2 has read two different values for X.
Solution: long read locks (degree 3)

Phantoms:

T1: read range [x - y]
T2: insert z, $x < z < y$
T2: end transaction
T1: read range [x - y]

Z is a "phantom" data item (eek!)
Solution: ??

NOTE: if everybody is at least degree 1, than different transactions can CHOOSE what degree they wish to "see" without worry. I.e. can have a mixture of levels of consistency.

Oracle's Snapshot Isolation
A.K.A. "SERIALIZABLE" in Oracle. Orwellian!

Idea: Give each transaction a timestamp, and a "snapshot" of the DBMS at transaction begin. Then install their writes at commit time. Read-only transactions never block or get rolled back!

Caveat: to avoid Lost Updates, ensure that "older" transactions don't overwrite newer, committed transactions.

Technique: "archive on write". I.e. move old versions of tuples out of the way, but don't throw them out.

- On write, if there's space on the page, they can move to a different "slot". Else move to a "rollback segment"
- Readers see the appropriate versions (snapshot plus their own updates). Non-trivial: needs to work with indexes as well as filescans.
- Writes become visible at commit time.
- "First committer wins": At commit time, if T1 and T2 have a WW conflict, and T1 commits, then T2 is aborted. Oracle enforces this by setting locks. Pros and Cons??

A snapshot isolation schedule:

```
T1: R(A0), R(B0),                          W(A1), C
T2:                R(A0), R(B0), W(B2), C
```

Now, B2 = f(A0, B0); A2 = g(A0, B0). Is this schedule serializable? Example:

- A is checking, B is savings. Constraint: Sum of balances > $0.
- Initially, A=70, B = 80.
- T1 debits $100 from checking.
- T2 debits $100 from savings.

"Write Skew"! There are subtler problems as well, see O'Neil paper.
Still, despite IBM complaining that these anomalies mean the technique is "broken", Snapshot Isolation is popular and works "most of the time" (and certainly on leading benchmarks.)

Question: How do you extend Snapshot Isolation to give true serializable schedules? Cost?

Optimistic Concurrency Control

Attractive, simple idea: optimize case where conflict is rare.
Basic idea: all transactions consist of three phases:

1. Read. Here, all writes are to private storage (shadow copies).
2. Validation. Make sure no conflicts have occurred.
3. Write. If Validation was successful, make writes public. (If not, abort!)

When might this make sense? Three examples:

1. All transactions are readers.
2. Lots of transactions, each accessing/modifying only a small amount of data, large total amount of data.
3. Fraction of transaction execution in which conflicts "really take place" is small compared to total pathlength.

The Validation Phase

- Goal: to guarantee that only serializable schedules result.
- Technique: actually find an equivalent serializable schedule. That is,
  1. Assign each transaction a TN during execution.
  2. Ensure that if you run transactions in order induced by "<" on TNs, you get an equivalent serial schedule.
  3.

Suppose TN(Ti) < TN(Tj). Then if one of the following three conditions holds, it's serializable:

1. Ti completes its write phase before Tj starts its read phase.
2. WS(Ti) ∩ RS(Tj) = ∅ and Ti completes its write phase before Tj starts its write phase.
3. WS(Ti) ∩ RS(Tj) = ∅ and WS(Ti) ∩ WS(Tj) = ∅ and Ti completes its read phase before Tj completes its read phase.

Is this correct? Each condition guarantees that the three possible classes of conflicts (W-R, R-W, W-W) go "one way" only: higher transaction id (j) depends on lower (i), but not vice versa. (When we speak of conflicts below, they are implicitly ordered i then j.)

1. For condition 1 this is obvious (true serial execution!)
2. Interleave Wi/Rj, but ensure no WR conflict. RW and forward WW allowed.
3. interleave Wi/Rj or Wi/Wj. I.e. forbid only Ri/Wj interleaving, AND ensure all conflicts are RW

Assigning TN's: at beginning of transactions is not optimistic; do it at end of read phase. Note: this satisfies second half of condition (3).

Note: a transaction T with a very long read phase must check write sets of all transactions begun and finished while T was active. This could require unbounded buffer space.
Solution: bound buffer space, toss out when full, abort transactions that could be affected.

- Gives rise to starvation. Solve by having starving transaction write-lock the whole DB!

Serial Validation
Only checks properties (1) and (2), since writes are not going to be interleaved.
Simple technique: make a critical section around <get xactno; validate (1) or (2) for everybody from your start to finish; write>. Not great if:

- write takes a long time
- SMP – might want to validate 2 things at once if there's not enough reading to do

Improvement to speed up validation:

```
repeat as often as you want {
      get current xactno.
      Check if you're valid with everything up to that xactno.
}
<get xactno; validate with new xacts; write>.
```

Note: read-only xacts don't need to get xactnos! Just need to validate up to highest xactno at end of read phase (without critical section!)

Parallel Validation
Want to allow interleaved writes.
Need to be able to check condition (3).

- Save active xacts (those which have finished reading but not writing).
- Active xacts can't intersect your read or write set.
- Validation:
  <get xactno; copy active; add yourself to active>
  check (1) or (2) against everything from start to finish;
  check (3) against all xacts in active copy
  If all's clear, go ahead and write.
  <bump xact counter, remove yourself from active>.

Small critical section.
Problems:

- a member of active that causes you to abort may have aborted
  - can add even more bookkeeping to handle this
  - can make active short with improvement analogous to that of serial validation

Does this make any sense?
Ask yourself:

- what's the running state of the approach, what bounds that state?
- what happens on conflict and how does that affect performance?