CMSC 424 – Database design
Lecture 21
Concurrency/recovery

Mihai Pop

Admin

- Office hours tomorrow @ 10am in AVW 3223

Serializability

- Not possible to look at all n! serial schedules to check if the effect is the same
  - Instead we ensure serializability by allowing or not allowing certain schedules

- Conflict serializability

- View serializability

- View serializability allows more schedules

Conflict Serializability

- Two read/write instructions “conflict” if
  - They are by different transactions
  - They operate on the same data item
  - At least one is a “write” instruction

- Why do we care ?
  - If two read/write instructions don’t conflict, they can be “swapped” without any change in the final effect
  - However, if they conflict they CAN’T be swapped without changing the final effect

Equivalence by Swapping

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

Effect:	Before	After
A	100	45
B	50	105

==

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

Effect:	Before	After
A	100	45
B	50	105

Equivalence by Swapping

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

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

Effect:	Before	After
A	100	45
B	50	105

! ==

Effect:	Before	After
A	100	45
B	50	55

Conflict Serializability

- Conflict-equivalent schedules:
  - If S can be transformed into S′ through a series of swaps, S and S′ are called *conflict-equivalent*
  - *conflict-equivalent guarantees same final effect on the database*

- A schedule S is conflict-serializable if it is conflict-equivalent to a serial schedule

Equivalence by Swapping

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

Effect:	Before	After
A	100	45
B	50	105

==

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

Effect:	Before	After
A	100	45
B	50	105

Equivalence by Swapping

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

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

Effect:	Before	After
A	100	45
B	50	105

==

Effect:	Before	After
A	100	45
B	50	105

Example Schedules (Cont.)

A "bad" schedule

T1	T2
read(A) A = A -50	
	Y read(A) tmp = A*0.1 A = A – tmp write(A) read(B)
X write(A) read(B) B=B+50 write(B)	
	B = B+ tmp write(B)

Can't move Y below X
read(B) and write(B) conflict

Other options don't work either

So: Not Conflict Serializable

Serializability

- In essence, following set of instructions is not conflict-serializable:

T_3	T_4
read(Q)	
	write(Q)
write(Q)	

View-Serializability

- Similarly, following not conflict-serializable

T_3	T_4	T_6
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

- BUT, it is serializable
  – Intuitively, this is because the *conflicting write instructions* don't matter
  – The final write is the only one that matters

- View-serializability allows these
  – Read up (chap. 15)

Other notions of serializability

- Not conflict-serializable or view-serializable, but serializable
- Mainly because of the +/- only operations
  - Requires analysis of the actual operations, not just read/write operations
- Most high-performance transaction systems will allow these

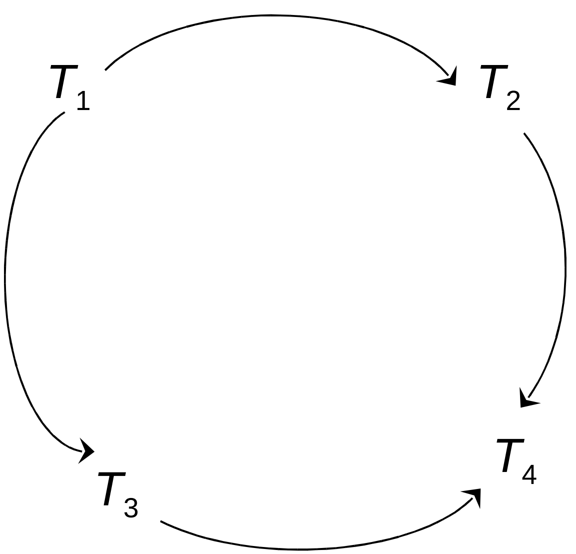
T_1	T_5
read(A)	
$A := A - 50$	
write(A)	
	read(B)
	$B := B - 10$
	write(B)
read(B)	
$B := B + 50$	
write(B)	
	read(A)
	$A := A + 10$
	write(A)

Testing for conflict-serializability

- Given a schedule, determine if it is conflict-serializable
- Draw a *precedence-graph* over the transactions
  - A directed edge from T1 and T2, if they have conflicting instructions, and T1's conflicting instruction comes first
- If there is a cycle in the graph, not conflict-serializable
  - Can be checked in at most $O(n+e)$ time, where n is the number of vertices, and e is the number of edges
- If there is none, conflict-serializable
- Testing for view-serializability is NP-hard.

Example Schedule (Schedule A) + Precedence Graph

T_1	T_2	T_3	T_4	T_5
	read(X)			
read(Y)				
read(Z)				
				read(V)
				read(W)
				read(W)
	read(Y)			
	write(Y)			
		write(Z)		
read(U)				
			read(Y)	
			write(Y)	
			read(Z)	
			write(Z)	
read(U)				
write(U)				



Recap

- We discussed:
  - Serial schedules, serializability
  - Conflict-serializability, view-serializability
  - How to check for conflict-serializability

- We haven't discussed:
  - How to guarantee serializability ?
    - Allowing transactions to run, and then aborting them if the schedules wasn't serializable is clearly not the way to go
  - We instead use schemes to guarantee that the schedule will be conflict-serializable
  - Also, ***recoverability ?***

Recoverability

- Serializability is good for consistency

- But what if transactions fail ?
  - T2 has already committed
    - A user might have been notified
  - Now T1 abort creates a problem
    - T2 has seen its effect, so just aborting T1 is not enough. T2 must be aborted as well (and possibly restarted)
    - But T2 is *committed*

T1	T2
read(A)	
A = A -50	
write(A)	
	read(A)
	tmp = A*0.1
	A = A – tmp
	write(A)
	COMMIT
read(B)	
B=B+50	
write(B)	
ABORT	

Recoverability

- Recoverable schedule: If T1 has read something T2 has written, T2 must commit before T1
  - Otherwise, if T1 commits, and T2 aborts, we have a problem

- Cascading rollbacks: If T10 aborts, T11 must abort, and hence T12 must abort and so on.

T_{10}	T_{11}	T_{12}
read(A)		
read(B)		
write(A)		
	read(A)	
	write(A)	
		read(A)

Recoverability

- *Dirty read*: Reading a value written by a transaction that hasn’t committed yet
- Cascadeless schedules:
  - A transaction only reads *committed* values.
  - So if T1 has written A, but not committed it, T2 can’t read it.
    - *No dirty reads*
- Cascadeless → No cascading rollbacks
  - That’s good
  - We will try to guarantee that as well

Recap

- We discussed:
  - Serial schedules, serializability
  - Conflict-serializability, view-serializability
  - How to check for conflict-serializability
  - Recoverability, cascade-less schedules

- We haven't discussed:
  - How to guarantee serializability ?
    - Allowing transactions to run, and then aborting them if the schedules wasn't serializable is clearly not the way to go
  - We instead use schemes to guarantee that the schedule will be conflict-serializable

Concurrency control

Approach, Assumptions etc..

- Approach
  - Guarantee conflict-serializability by allowing certain types of concurrency
    - Lock-based
- Assumptions:
  - Durability is not a problem
    - So no crashes
    - Though transactions may still abort
- Goal:
  - Serializability
  - Minimize the bad effect of aborts (cascade-less schedules only)

Lock-based Protocols

- A transaction *must* get a *lock* before operating on the data

- Two types of locks:
  - *Shared* (S) locks (also called *read locks)*
    - Obtained if we want to only read an item
  - *Exclusive* (X) locks (also called *write locks)*
    - Obtained for updating a data item

Lock instructions

- New instructions
  - lock-S: shared lock request
  - lock-X: exclusive lock request
  - unlock: release previously held lock

Example schedule:

T1	T2
read(B)	read(A)
B ← B-50	read(B)
write(B)	display(A+B)
read(A)	
A ← A + 50	
write(A)	

Lock instructions

- New instructions
  - lock-S: shared lock request
  - lock-X: exclusive lock request
  - unlock: release previously held lock

Example schedule:

T1	T2
lock-X(B)	lock-S(A)
read(B)	read(A)
B ←B-50	unlock(A)
write(B)	lock-S(B)
unlock(B)	read(B)
lock-X(A)	unlock(B)
read(A)	display(A+B)
A ←A + 50	
write(A)	
unlock(A)	

Lock-based Protocols

- Lock requests are made to the *concurrency control manager*
  - It decides whether to *grant* a lock request
- T1 asks for a lock on data item A, and T2 currently has a lock on it ?
  - Depends

T2 lock type	**T1 lock type**	**Should allow ?**
Shared	Shared	YES
Shared	Exclusive	NO
Exclusive	-	NO

- If *compatible,* grant the lock, otherwise T1 waits in a *queue*.

Lock-based Protocols

- How do we actually use this to guarantee serializability/recoverability ?
  - Not enough just to take locks when you need to read/write something

T1

lock-X(B)
read(B)
B ← B-50
write(B)
unlock(B)

lock-X(A)
read(A)
A ← A + 50
write(A)
unlock(A)

→

lock-X(A), lock-X(B)
A = A-50
B = B+50
unlock(A), unlock(B)

2-Phase Locking Protocol (2PL)

- Phase 1: Growing phase
  - Transaction may obtain locks
  - But may not release them
- Phase 2: Shrinking phase
  - Transaction may only release locks
- Can be shown that this achieves *conflict-serializability*
  - *lock-point*: the time at which a transaction acquired last lock
  - if *lock-point*(T1) < *lock-point*(T2), there can't be an edge from T2 to T1 in the *precedence graph*

T1

lock-X(B)
read(B)
B ← B-50
write(B)
unlock(B)

lock-X(A)
read(A)
A ← A + 50
write(A)
unlock(A)

2 Phase Locking

- Example: T1 in 2PL

T1
lock-X(B)
read(B)
B $\leftarrow$ B - 50
write(B)
lock-X(A)
read(A)
A $\leftarrow$ A - 50
write(A)
unlock(B)
unlock(A)

Growing phase

Shrinking phase

2 Phase Locking

- Guarantees *conflict-serializability*, but not cascade-less recoverability

T1	T2	T3
lock-X(A), lock-S(B)		
read(A)		
read(B)		
write(A)		
unlock(A), unlock(B)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
	Commit	lock-S(A)
		read(A)
		Commit
<xction fails>		

2 Phase Locking

- Guarantees *conflict-serializability*, but not cascade-less recoverability

- Guaranteeing just recoverability:
  - If T2 reads a dirty data of T1 (ie, T1 has not committed), then T2 can't commit unless T1 either commits or aborts
  - If T1 commits, T2 can proceed with committing
  - If T1 aborts, T2 must abort
    - So cascades still happen

Strict 2PL

- Release *exclusive* locks only at the very end, just before commit or abort

T1	T2	T3
lock-X(A), lock-S(B)		
read(A)		
read(B)		
write(A)		
unlock(A), unlock(B)		
	lock-X(A)	
	read(A)	
	write(A)	
	unlock(A)	
	Commit	lock-S(A)
		read(A)
		Commit
<xction fails>		

Strict 2PL will not allow that

Works. Guarantees cascade-less and recoverable schedules.

Strict 2PL

- Release *exclusive* locks only at the very end, just before commit or abort
  - Read locks are not important

- Rigorous 2PL: Release both *exclusive and read* locks only at the very end
  - The serializability order === the commit order
  - More intuitive behavior for the users
    - No difference for the system

Strict 2PL

- Lock conversion:
  - Transaction might not be sure what it needs a write lock on
  - Start with a S lock
  - *Upgrade* to an X lock later if needed
  - Doesn’t change any of the other properties of the protocol

Implementation of Locking

- A separate process, or a separate module
- Uses a *lock table* to keep track of currently assigned locks and the requests for locks
  - Read up in the book (chap. 16)

Recap

- Concurrency Control Scheme
  - A way to guarantee serializability, recoverability etc
- Lock-based protocols
  - Use *locks* to prevent multiple transactions accessing the same data items
- 2 Phase Locking
  - Locks acquired during *growing phase,* released during *shrinking phase*
- Strict 2PL, Rigorous 2PL