Database Management Systems

Madhavan Mukund

https://www.cmi.ac.in/~madhavan

Sai University

Lecture 22, 17 November 2023

Concurrent execution and schedules

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

Concurrent execution and schedules

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Serial schedule 1

Concurrent execution and schedules

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

T_1	T_2
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	

Isolation

Serial schedule 2

Concurrent execution and schedules

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Serial schedule 1

T_1	T_2
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)	
commit	
	read(B)
	$B := B + temp$
	write(B)
	commit

Consistent concurrent schedule

Concurrent execution and schedules

T_1: read(A);
$A := A - 50$;
write(A);
read(B);
$B := B + 50$;
write(B).

T_2: read(A);
$temp := A * 0.1$;
$A := A - temp$;
write(A);
read(B);
$B := B + temp$;
write(B).

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B)
	commit

Serial schedule 1

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
commit	
	$B := B + temp$
	write(B)
	commit

Inconsistent concurrent schedule

Serializability

- Serial schedule — each transaction executes as a block, no interleaving
- Serializable schedule — equivalent to *some* serial schedule
- Conflicting operations — two operations on the *same* value where *at least one is a write*
- Conflict equivalence — one schedule can be transformed into the other by reordering non-conflicting operations
- Conflict serializable — can be reordered to a conflict-equivalent serial schedule

Testing for conflict serializability

- Start with a schedule — interleaved sequence of operations from multiple transactions
- Build a graph, with transactions as nodes
- Edge $T_i \rightarrow T_j$ if an earlier operation in T_i conflicts with a later operation in T_j
- If this conflict graph has cycles, there is a circular dependency, not conflict serializable
- If the conflict graph is acyclic, use topological sort to order the transactions into a serial schedule.

Concurrency control

- Ensure that only serializable schedules are generated
- Allow concurrency
- Control access to data to avoid conflicts
- Mechanisms
  - Locking
  - Timestamps
  - Multiple versions — snapshot isolation

Concurrency control using locks

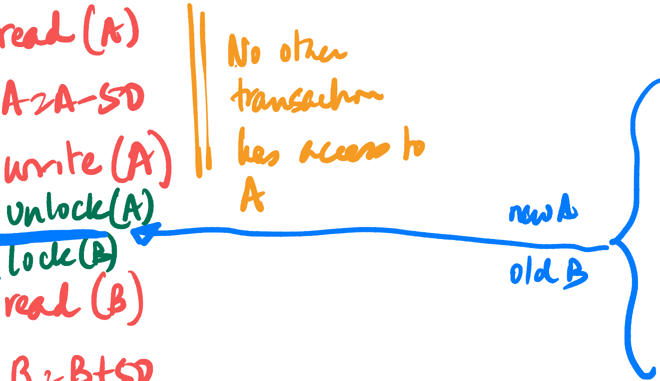
Lock – securing access to data

T_i wants to access item A, it "locks" A

If another T_j has locked A, T_i has to wait

After finishing with A, T_i "unlocks" A

Concurrency control using locks



Hold locks to ensure isolation/serializability

T1 before T2
T2 before T1

T_1	T_2
lock(A)	lock(A)
read(A)	old(A) – read(A) – new(A)
A = A - 50	⋮
write(A)	
lock(B)	lock(B)
read(B)	old(B) – read(B) – new(B)
B = B + 50	⋮
write(B)	
unlock(A)	unlock(A)
unlock(B)	unlock(B)

– stuck

after

Concurrency control using locks

- Locks
- Locking protocol – how to acquire & release locks

  Legal schedule – follows the protocol

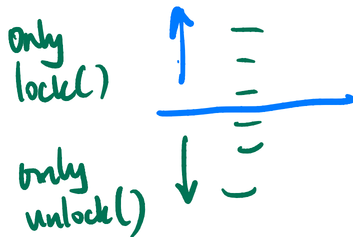
  Will every legal schedule be serializable?

Concurrency control using locks

Two phase locking

Initially – acquire locks ("growing")

Later – release locks ("shrinking")

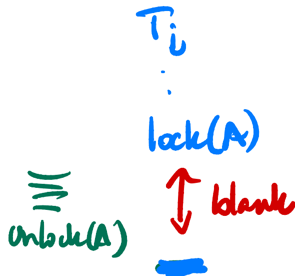


lock(A)
unlock(A)
lock(B)
unlock(B)

Violates 2 phase

Concurrency control using locks

Theorem If all transactions follow 2 phase locking, then any concurrent schedule generated is serializable

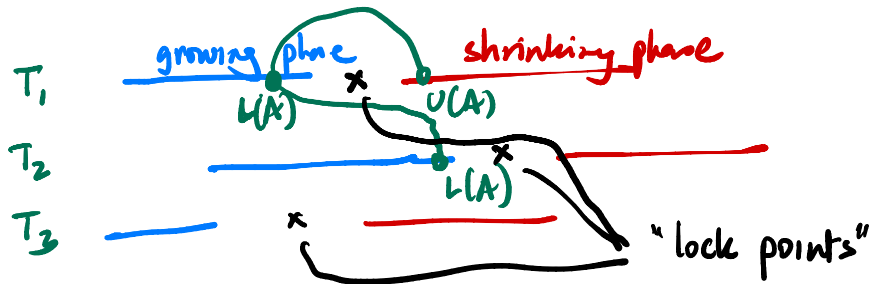


Concurrency control using locks

Prove by induction — not in textbook, look it up

If true, what is the witness serial schedule

In what serial order did transactions happen?



No locks in common

Any order is OK

Concurrency control using locks

Any 2 phase schedule can be serialized in order of lock points

- Any 2 phase schedule is serializable
- Equivalent serial schedule is given by order of lock points

Concurrency control using locks

Specific case of 2phase

Lock all data you need

Execute all updates

Unlock all data

Forcibly serializes the schedule

Allows no concurrency

Concurrency control using locks

2 types of locks

- R(A) & R(A) are compatible
- To allow concurrent reads – "shared lock"

  read, not write

lock-S(A)

read(A) ← not write(A)

unlock(A)

Concurrency control using locks

If T_1 has lock-S(A) and T_2 requests lock-S(A), it gets it — compatible "shared" lock

To write A — exclusive lock — only one transaction at a time

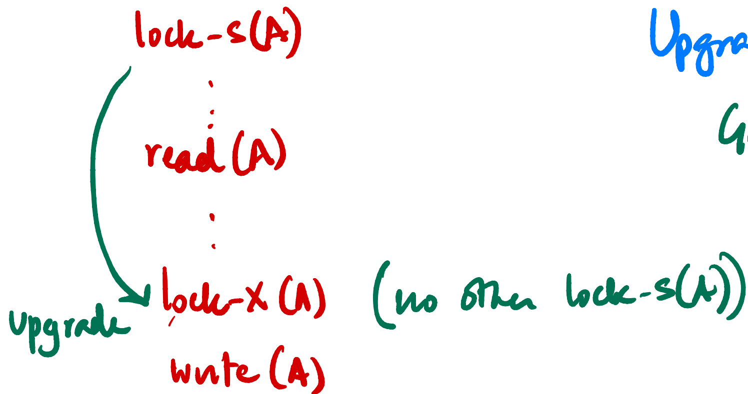
lock-X(A)

write(A)

unlock(A)

Compatibility

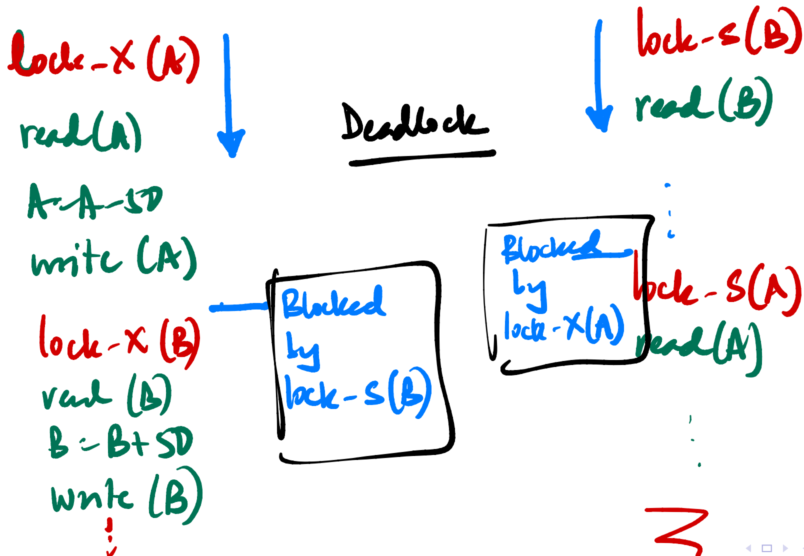
T_j \ T_i	lock-S()	lock-X()
lock-S()	✓	✗
lock-X()	✗	✗



Upgrade = "Growing"

Generalize 2 phase to allow upgrading in growing phase

Concurrency control using locks

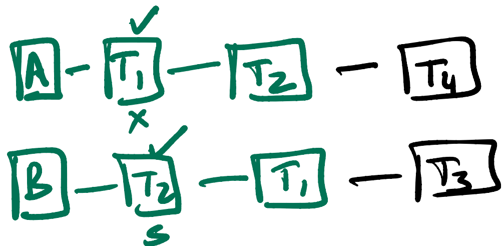


Concurrency control using ~~timestamps~~ locks

How are locks handed out?

Lock Manager

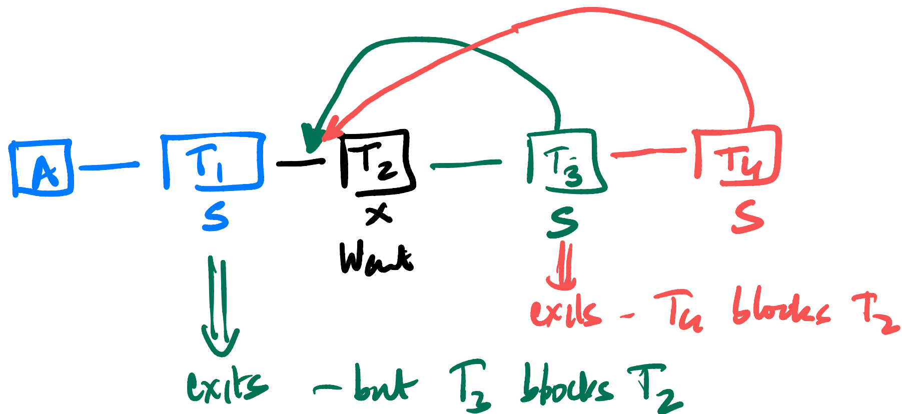
For each item — list of pending lock requests



Hand over locks based on request order

Concurrency control using ~~timestamps~~ locks

Subtlety



STARVATION - To avoid, respect queue even for lock-S()

Concurrency control using locks

Deadlocks — Dealing with them

- If deadlock, some transaction must roll back
- Prevent deadlocks
  - Two transactions request same locks in diff. order
  - Impose an order on all data items & enforce this order for locking — lock A before B before C

~~Multi-version concurrency control~~ Locks

- Detect & fix

Graph T_i is waiting for T_j

want lock (under T_i) — holds lock (under T_j)

Break a cycle — abort some transaction (rollback)

- Estimate "cost" of rollback

Pre-empt deadlocks

Timestamp — Assign each T_i a timestamp $TS(T_i)$ when it starts - smaller timestamps $\Rightarrow$ older

If T_i needs a lock held by T_j, avoid waiting

restart, keep timestamp

- Wait for younger — if $T_i < T_j$, wait else rollback T_i
- Kill younger — if $T_i < T_j$, kill T_j else rollback T_i