Transactions

- Serializability
- Properties
  - recoverability, cascading aborts
- Concurrency control via *locks*
  - strict, rigorous
- Deadlocks
- Weakening Guarantees
- Recovery

Recoverability

- Serializability is good for consistency
- 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 violate correctness

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

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

- *Recoverability:*
  - Guaranteed if a transaction has no dirty reads.

- *Cascadeless* schedules guaranteed if:
  - Guaranteed if a transaction has no dirty reads.

- Cascadeless → No cascading rollbacks
  - That's good
  - We will try to guarantee that as well

Recap so far…

- 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 ?
    - Could allow transactions to run, abort them if not serializable
      - Expensive
  - We can instead use schemes to guarantee that the schedule will be conflict-serializable
    - *Hint: locks*

Approach, Assumptions etc..

- Approach
  - *Guarantee* conflict-serializability by limiting concurrency
    - instead of detecting after the fact
    - lock-based
- Assumptions:
  - Still ignoring durability
    - So no crashes
    - Though transactions may still abort
- Goal:
  - Serializability
  - Minimize the bad effect of aborts (cascade-less schedules only)

Lock-based Protocols

- Transactions must *acquire* locks before using data
  - locking usually handled by transaction statements
- Two types:
  - *Shared* (S) locks (*read locks)*
    - Obtained if we want to only read an item
  - *Exclusive* (X) locks (*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 transactions:

Not a 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 transactions:

Not a 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)
	unlock(B)
lock-X(A)	display(A+B)
read(A)	
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
- Assume T1 requests lock held by T2 :

Held lock	**Lock wanted**	**Allow?**
Shared	Shared	YES
Shared	Exclusive	NO
Exclusive	-	NO

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

Lock instructions

Potential schedule

T1	T2
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-S(A)
	read(A)
	unlock(A)
	lock-S(B)
	read(B)
	unlock(B)
	display(A+B)

Good!

Lock instructions

Potential schedule

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

Good!

Lock instructions

Potential schedule

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

Not good!

2-Phase Locking Protocol (2PL)

- Phase 1: Growing phase
  - Transaction may obtain locks
  - But may not release them
- Phase 2: Shrinking phase
  - Only release locks
- 2PL guarantees *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)

not allowed

Lockpoints Intuition

(pseudo-proof by contradiction)

T1	T2
lock(x)	
write(x)	
unlock(x)	
	lock(x)
	read(x)

precedence edge

T1's lockpoint must be before T2's, because T1 already in shrinking phase, T2 still growing phase

But if we *also* have an edge from T2 to T1:
T2's lockpoint must also be before T1's.
Contradiction!
Cycle requires edge and it's reverse, but we just showed this can't happen, so….
conflict-serializable.

Back to locking: 2 Phase Locking

- Guarantees *conflict-serializability*
- Does not guarantee
  - recoverability
  - cascade-less schedules

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
<fail>		

2 Phase Locking

- How to guarantee recoverability:
  - If T2 performs a dirty read from T1, then:
    - T2 can't commit until 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, atomically with 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 aborts>		

Strict 2PL will not allow that

Strict 2PL

T1	T2	T3
lock-X(A), lock-S(B) read(A) read(B) write(A) unlock(A), unlock(B) **commit**		
	lock-X(A) read(A) write(A) unlock(A) **commit**	
		lock-S(A) read(A) **commit**

- Release *exclusive* locks only at the very end
  - Atomically with commit or abort
- *Guarantees recoverable and cascade-less schedules*

Rigorous 2PL

T1	T2
lock-X(A) lock-S(B) read(B) unlock(B)	
…	lock-X(B) write(B) unlock(B) **commit**
write(A) unlock(A) **commit**	

Beginning timestamp order?
- T1 -> T2

Commit order?
- T2 -> T1

Weird.

Rigorous 2PL

T1	T2
lock-X(A)	
lock-S(B)	
read(B)	
…	
write(A)	
unlock(A), unlock(B)	
commit	
	lock-X(B)
	write(B)
	unlock(B)
	commit

Beginning timestamp order?
- T1 -> T2

Commit order?
- T2 -> T1

Weird.

- Also hold *shared* locks until the end
  - serialization order == the commit order
- *More intuitive for users*

Strict 2PL

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

Rigorous 2PL:

- Release both exclusive *and read* locks only at the very end
  - Makes serialization order == commit order
  - More intuitive behavior for the users

Lock Conversion/Upgrading

- 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

Recap so far…

- 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

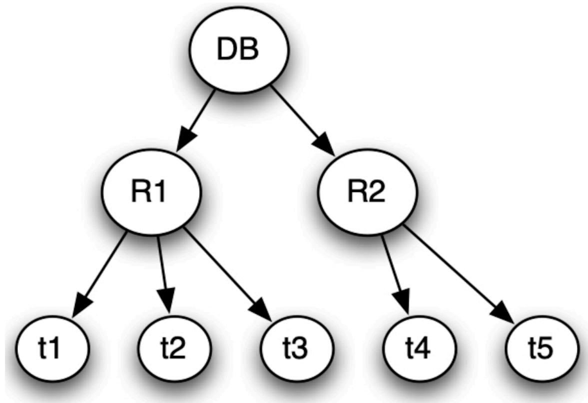
Transactions

- Serializability
- Properties
  - recoverability, cascading aborts
- Concurrency control via locks
  - strict, rigorous, intention
- Deadlocks
- Weakening Guarantees
- Recovery

Locking granularity (not always done)

- Locking granularity
  - What are we taking locks on ? Tables, tuples, attributes ?
- Coarse granularity
  - e.g. take locks on tables
  - less overhead (the number of tables is not that high)
  - very low concurrency
- Fine granularity
  - e.g. take locks on tuples
  - much higher overhead
  - much higher concurrency
  - What if I want to lock 90% of the tuples of a table ?
    - Prefer to lock the whole table in that case

Granularity Hierarchy



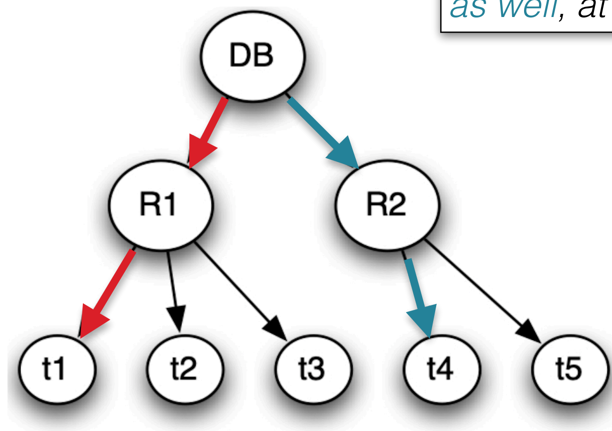
The highest level in the example hierarchy is the entire database.
The levels below are of *relation* and *tuple* in that order.
Can lock at any level in the hierarchy.

Intention Locks

- New lock mode, called *intention* locks
  - Declare an intention to lock parts of the subtree below a node
  - IS: *intention shared*
    - The lower levels below may be locked in the shared mode
  - IX: *intention exclusive*
  - *SIX: shared and intention-exclusive*
    - The entire subtree is locked in the shared mode, but might also want exclusive locks on some nodes below
- Protocol:
  - Before acquiring a lock on a data item, all the ancestors must be locked as well, at least in intention mode
  - Lock acquisition order is from the *root* down to the desired node.

Intention Locks

all the ancestors must be locked as well, at least in intention mode



(1) Want to lock ***t1*** in shared mode, *DB* and then *R1* must be locked in at least IS mode (but IX, SIX, S, X are okay too), then *t1* in S mode.

(2) Want to lock ***t4*** in exclusive mode, *DB* and then *R2* must be locked in at least IX mode (SIX, X are okay too), then *t4* must be locked in X mode.

Compatibility Matrix with Intention Lock Modes

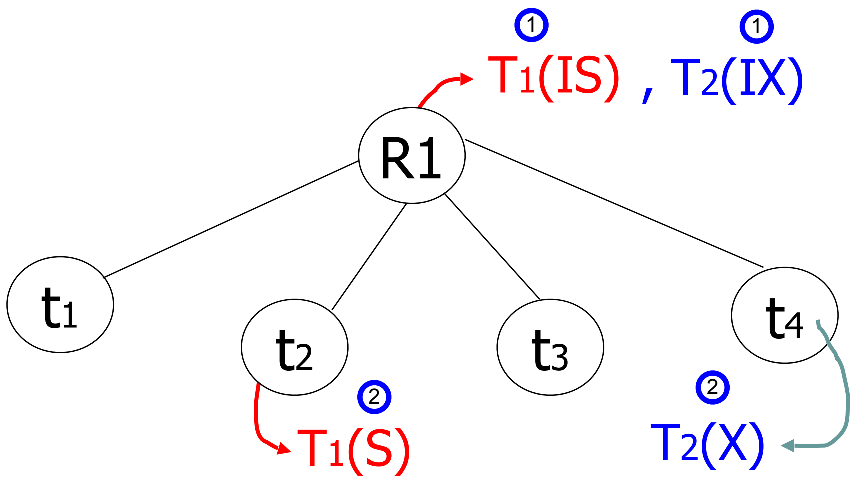
- Locks from different transactions:

requestor

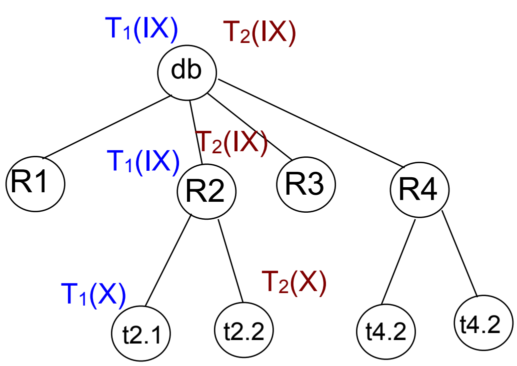
holder	IS	IX	S	SIX	X
IS	✓	✓	✓	✓	×
IX	✓	✓	×	×	×
S	✓	×	✓	×	×
SIX	✓	×	×	×	×
X	×	×	×	×	×

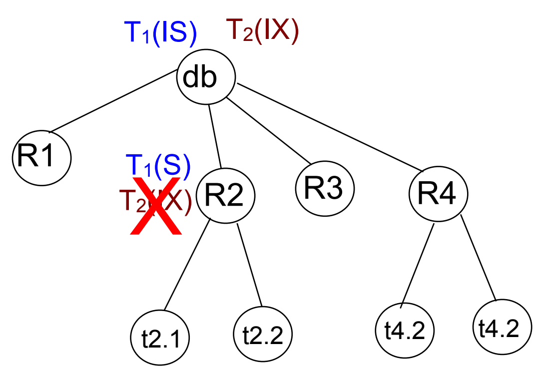
Example

- Assume:
  - T_1 wants *shared* lock on t2
  - T_2 wants *exclusive* lock on t4



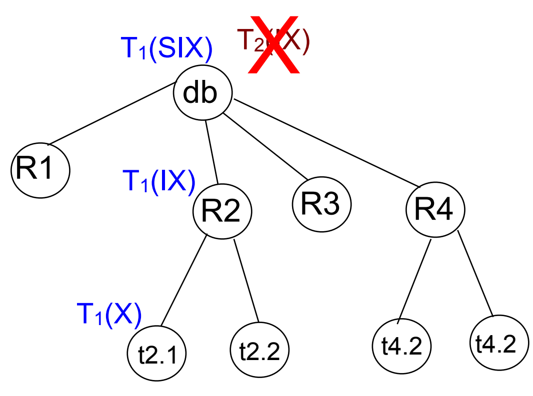
T_2 Needs Locks…But T_1 already there…





Can T_2 access object t2.2 in X mode?

What locks will T_2 get?



Transactions

- Serializability
- Properties
  - recoverability, cascading aborts
- Concurrency control via locks
  - strict, rigorous, intention
- Deadlocks
- Weakening Guarantees
- Recovery

More Locking Issues: Deadlocks

No xction proceeds

T1	T2
lock-X(B)	
read(B)	
B ← B-50	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B) block
block lock-X(A)	

Deadlock:

- T1 waits for T2 to unlock A
- T2 waits for T1 to unlock B

Rolling back transactions can be costly...

Deadlocks

- 2PL does not prevent deadlock
  - Strict doesn't either

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

Rolling back transactions can be costly...

Preventing deadlocks

- Graph-based protocols
  - Acquire locks only in a well-known order

bad

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

good

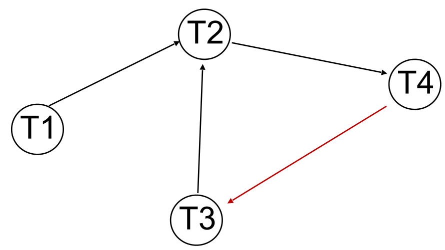
T1	T2
lock-X(A)	
lock-X(B)	
read(B)	
B ← B-50	
write(B)	
…	lock-S(A)
	read(A)
	lock-S(B)

- But might not know locks in advance

Detecting existing deadlocks

- Timeouts (local information)
- cycles in *waits-for graph* (global information):
  - edge $T_i \rightarrow T_j$ when T_i waiting for T_j on locks

T1	T2	T3	T4
		X(Z)	
	X(V)		
			X(W)
S(V)			
	S(W)		
		S(V)	



Suppose T4 requests lock-S(Z)....

Dealing with Deadlocks

- Deadlock detected, now what ?
  - Will need to abort some transaction
- Victim selection
  - Use time-stamps; say T1 is *older* than T2
  - *wait-die scheme:*
    - T1 will wait for T2 if T2 has a lock T1 needs.
    - T2 immediately aborts if needs a lock held by T1
  - ***wound-wait*** *scheme:*
    - T1 will *wound* T2 (force it to abort) if T2 has a lock that T2 needs.
    - T2 waits for T1 if it needs a lock held by T1.
- Issues
  - Prefer to prefer transactions with the most work done
  - Possibility of starvation
    - If a transaction is aborted too many times, it may be given priority in continuing