Concurrency Control

Concurrency Control

- A database must provide a mechanism that will ensure that all possible schedules are
  - either conflict or view serializable, and
  - recoverable and preferably cascadeless
- A policy in which only one transaction can execute at a time generates serial schedules, but provides a poor degree of concurrency
- Testing a schedule for serializability *after* it has executed is a little too late!
- **Goal** – to develop concurrency control protocols that will assure serializability

Why Concurrency Control ?

- Three concurrency problems
  - Lost update
  - Temporary update (dirty read): uncommitted dependency
  - Incorrect summary
  - Unrepeatable read

Lost Update Problem

X=1000, Y=200
N=500, M=300

	T1	T2	
	read(X);		
X'=500	X=X-N;		
		read(X);	X'=1000
		X=X+M;	X'=1300
X=500	write(X);		
	read(Y);		
		write(X);	X=1300
	Y=Y+N;		
	write(Y);		

Temporary Update Problem

X=1000, Y=200
N=500, M=300

	T1	T2	
	read(X);		
X'=500	X=X-N;		
X=500	write(X);		
		read(X);	X'=500
		X=X+M;	
		write(X);	X=800
	read(Y);		
	fails		

Incorrect Summary Problem

X=1000, Y=200
N=500

	T1	T2	
		sum=0	
X'=500	read(X);		
	X=X-N;		
X=500	write(X);		
		read(X);	
		sum=sum+X	sum=500
		read(Y)	
		sum=sum+Y	sum=700
	read(Y);		
	Y=Y+N;		
	write(Y);		

Solution for Lost Update Problem

- Locking techniques

X=1000, Y=200
N=500, M=300

	T1	T2	
	readlock(X)		
	writelock(X)		
	read(X);		
X'=500	X=X-N;		
X=500	write(X)		
	unlock(X)		
		readlock(X)	
		writelock(X)	
		read(X);	X'=500
		X=X+M;	X'=800
		write(X);	X=800
		unlock(X)	
	read(Y);		
	Y=Y+N;		
	write(Y);		

Approach, Assumptions

- Approach
  - Guarantee conflict-serializability by allowing certain types of concurrency
    - Lock-based
- Assumptions:
  - Durability is not a problem
    - 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-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
- If *compatible,* grant the lock. Otherwise T1 waits in a *queue.*

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

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

Assume A=100, B=200

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

lock-S(A), lock-S(B)
Display(A+B)
unlock(A), unlock(B)

A=100, B=150,

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

2-Phase Locking Protocol (2PL)

Phase 1: Growing phase

Transaction may obtain locks

Phase 2: Shrinking phase

Transaction may only release locks

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
Growing phase	lock-X(B)
	read(B)
	B ← B - 50
	write(B)
	lock-X(A)
	read(A)
	A ← A - 50
	write(A)
Shrinking phase	unlock(B)
	unlock(A)

2 Phase Locking

- Can be shown that this achieves *conflict-serializability*
- 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
<action fails>		

2 Phase Locking

- Guaranteeing just recoverability:
  - If T2 reads a dirty data of T1 (i.e., 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
<action 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

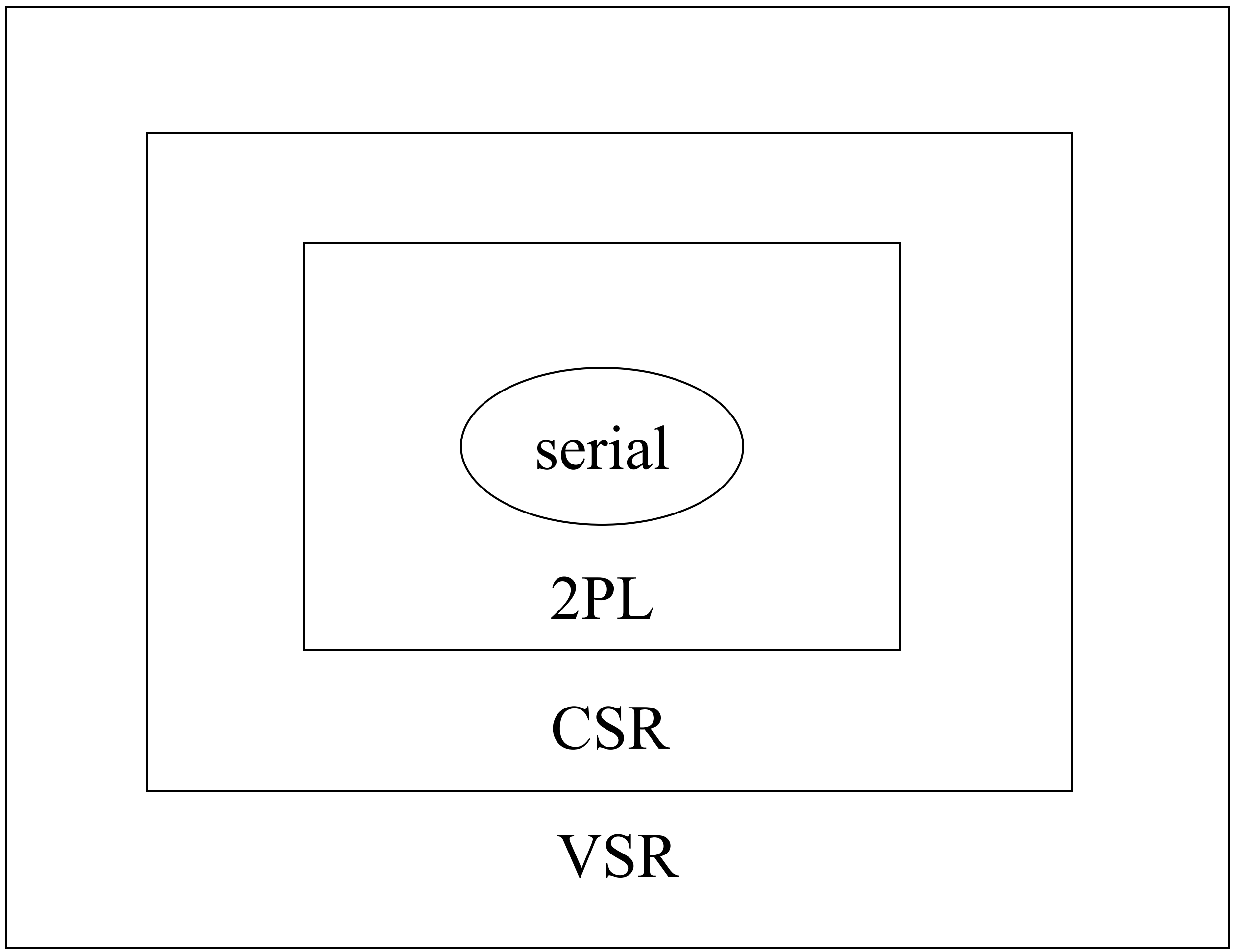
Strict 2PL

- 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

Recap

- Concurrency Control Scheme
  - A way to guarantee serializability, recoverability
- 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*

Hierarchy of Serializable Schedules



Other CC Schemes

- Time-stamp based
  - Transactions are issued time-stamps when they enter the system
  - The time-stamps determine the *serializability* order
  - If T1 entered before T2, then T1 should be before T2 in the serializability order
    - *timestamp(T1) < timestamp(T2)*
  - If T1 wants to read data item A
    - If any transaction with larger time-stamp wrote that data item, then this operation is not permitted, and T1 is *aborted*
  - If T1 wants to write data item A
    - If a transaction with larger time-stamp already read that data item or written it, then the write is *rejected* and T1 is aborted
  - Aborted transaction are restarted with a new timestamp
    - Possibility of *starvation*

Other CC Schemes

- Time-stamp based
  - Example (Timestamps T1<T2<T3<T4<T5)

T_1	T_2	T_3	T_4	T_5
				write(X)
	read(Y)			
read(Y)				
		write(Y)		
		write(Z)		
				read(Z)
	read(X)			
	abort			
		write(Z)		
		abort		
				write(Y)
				write(Z)

read(X)

?

Other CC Schemes

- Time-stamp based
  - As discussed here, has too many problems
    - Starvation
    - Non-recoverable
    - Cascading rollbacks required
    - Remember: We can always put more and more restrictions on what the transactions can do to ensure these things
    - The goal is to find the minimal set of restrictions to as to not hinder concurrency

Other CC Schemes

- Optimistic concurrency control
  - Also called validation-based
  - Intuition
    - Let the transactions execute as they wish
    - At the very end when they are about to commit, check if there might be any problems/conflicts etc
      - If no, let it commit
      - If yes, abort and restart
  - Optimistic: The hope is that there won't be too many problems/aborts

- Rarely used any more

More Locking Issues: Deadlocks

- No action proceeds:

Deadlock

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

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

Rollback transactions
Can be costly

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

Preventing deadlocks

- Solution 1: A transaction must acquire all locks before it begins
  - Not acceptable in most cases
- Solution 2: A transaction must acquire locks in a particular order over the data items
  - Also called *graph-based protocols*
- Solution 3: Use time-stamps; say T1 is older than T2
  - *wait-die scheme:* T1 will wait for T2. T2 will not wait for T1; instead it will abort and restart
  - *wound-wait scheme:* T1 will *wound* T2 (force it to abort) if it needs a lock that T2 currently has; T2 will wait for T1.
- Solution 4: Timeout based
  - Transaction waits a certain time for a lock; aborts if it doesn't get it by then

Deadlock Prevention

- Assign priorities based on timestamps.
- small timestamp --- high priority

T1:	S(A), R(A),		S(B) wait!				
T2:		X(B),W(B)			X(C) wait!		
T3:				S(C), R(C)			X(A) abort!
T4:						X(B) abort!	

- Assume Ti wants a lock that Tj holds.
- Wait-Die: If Ti has higher priority, Ti waits for Tj; otherwise Ti aborts (commits suicide)
- **Lower priority never waits for higher priority.**

Deadlock Prevention

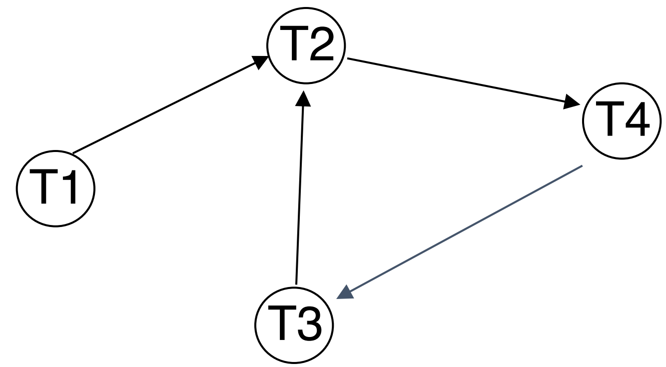
- Assign priorities based on timestamps.
- small timestamp --- high priority

T1: S(A), R(A), S(B) abort T2! X(C) abort T3!
T2: X(B),W(B)
T3: S(C), R(C)
T4: X(C) wait!

- Assume Ti wants a lock that Tj holds.
- Wound-wait: If Ti has higher priority, Tj aborts; otherwise Ti waits
- Higher priority never waits for lower priority.

Deadlock detection and recovery

- Instead of trying to prevent deadlocks, let them happen and deal with them if they happen
- How do you detect a deadlock?
  - Wait-for graph
  - Directed edge from Ti to Tj
    - Ti waiting for Tj



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
  - Prefer to abort the one with the minimum work done so far
  - Possibility of starvation
    - If a transaction is aborted too many times, it may be given priority in continuing