More on Transactions and Concurrency

Serializability

- **Fallacy: being serializable is the same as being serial**
- Being serializable implies that the schedule is a correct schedule
  - It will leave the database in a consistent state
  - The interleaving is appropriate and will result in a state the equivalent of at least one schedule where the transactions are serially-executed.
  - It will achieve efficiency due to concurrent execution

Checking for Serializability

T_1	T_2
read_item(X); X:=X-N;	
	read_item(X); X:=X+M;
write_item(X); read_item(Y);	
	write_item(X);
Y:=Y+N; write_item(Y);	

- This schedule suffers from the lost update problem. It is not serializable as the final state of X = 7 which is not possible with any sequential execution (either T1:T2 or T2:T1) of these transactions.

Checking for Serializability (contd)

- **<u>Serializability is hard to guarantee</u>**
- Interleaving of operations happens at runtime through some scheduler
- Difficult to determine/predict how the operations of a schedule will be interleaved
- Practical Approach
  - Protocols that ensure serializability
    - Tradeoff (constrain how transactions are written, and their performance for serializability)
    - Reduce the problem of checking the whole schedule to checking only a committed projection of the schedule (I.

Locks and Transactions

- Shared Locks(multiple outstanding locks)
  - Read Locks (many transactions can read at same time
- Exclusive Locks (one transaction at a time)
  - Write Locks
- Rules (for transaction T)
  - T must obtain a read or write lock on X before reading X.
  - T must obtain a write lock before writing to X
  - T cannot obtain locks on items it already has locked
  - T must issue the unlock(X) operation after all read and write operations on X within T are finished.

Using locks to guarantee serializability

- Initial values X=20,Y=30.
- Result of T1→T2
  - X=50, X=80
- Result of T2→T1
  - X=70, Y=50
- Does this work?
  - Is locking sufficient

(a)

T_1	T_2
read_lock(*Y*);	read_lock(*X*);
read_item(*Y*);	read_item(*X*);
unlock(*Y*);	unlock(*X*);
write_lock(*X*);	write_lock(*Y*);
read_item(*X*);	read_item(*Y*);
X:=*X*+*Y*;	*Y*:=*X*+*Y*;
write_item(*X*);	write_item(*Y*);
unlock(*X*);	unlock(*Y*);

NO!

- Locking rules hold in this schedule
- Result of this schedule
  - X=50, Y=50
  - It does not match T1→T2, or T2→T1
- So just locking does not suffice

Time	T_1	T_2
	read_lock(Y);	
	read_item(Y);	
	unlock(Y);	
		read_lock(X);
		read_item(X);
		unlock(X);
		write_lock(Y);
		read_item(Y);
		Y:=X+Y;
		write_item(Y);
		unlock(Y);
	write_lock(X);	
	read_item(X);	
	X:=X+Y;	
	write_item(X);	
	unlock(X);	

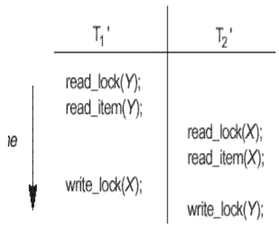
2-phase locking

- Basic criteria (2 **separable** phases)
  - Locking Phase: Obtain all locks needed in the transaction [growing/expanding phase]
    - New locks may be obtained, none may be released.
  - Unlocking Phase: Release existing locks [shrinking phase]
    - Locks may be released, none may be obtained

	T_1'	T_2'
Phase1 locking	read_lock (Y);	read_lock (X);
	read_item (Y);	read_item (X);
	write_lock (X);	write_lock (Y);
Phase 2 unlocking	unlock (Y);	unlock (X);
	read_item (X);	read_item (Y);
	X:=X+Y;	Y:=X+Y;
	write_item (X);	write_item (Y);
	unlock (X);	unlock (Y);

All problems solved?

- Deadlock Problem
  - T1 acquires Y, T2 acquires X. Both want the others lock and are unwilling to release their locks.

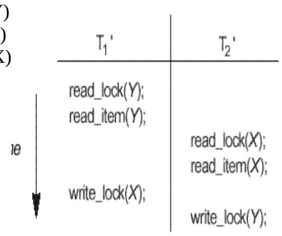


2-phase hierarchical locking

- Basic criteria (2 **<u>separable</u>** phases)
  - Locking Phase: Obtain all locks needed in the transaction [growing/expanding phase]
    - New locks may be obtained, <u>none may be released</u>.
  - Order locks, and obtain locks in predefined order.
    - Trade off deadlock avoidance for performance (less concurrency)
  - Unlocking Phase: Release existing lock [shrinking phase]
    - Locks may be released, <u>none may be obtained</u>

Solved? Yes!

T1	T2
Read_lock(y)	Write_lock(Y)
Read_item(y)	Read_lock(X)
Write_lock(x)	Read_Item(X)
Unlock(y)	Unlock (x)
Unlock(X)	Unlock (y)



With the above transactions, schedule on right can never happen.
Unlock can happen in any order.

Conflict Equivalence

- Two schedules are said to be conflict equivalent if the order of any two **<u>conflicting operations</u>** is the same in both schedules
- Conflicting Operations (<u>different transactions</u>)
  - Read after Write (RAW)
    - W<u>1</u>(X), R<u>2</u>(X)
  - Write after Read (WAR)
    - R<u>2</u>(Y), W<u>1</u>(Y)
  - Write after Write (WAW)
    - W<u>1</u>(Y), W<u>2</u>(Y)

Conflict Serializability

(a)

T_1	T_2

read_item(*X*);
X:=*X*-*N*;
write_item(*X*);	

	read_item(*X*);
	X:=*X*+*M*;
	write_item(*X*);

read_item(*Y*);
Y:=*Y*+*N*;
write_item(*Y*);	

Time

T_1	T_2

read_item(*X*);
X:=*X*-*N*;
write_item(*X*);
read_item(*Y*);
Y:=*Y*+*N*;
write_item(*Y*);	

	read_item(*X*);
	X:=*X*+*M*;
	write_item(*X*);

<u>LHS is Conflict Equivalent to RHS.</u>
<u>RHS is a serial schedule.</u>
<u>Therefore, LHS is conflict serializable</u>

Testing Conflict Serializability

- Create a precedence graph.
  - Create a node for each transaction.
  - Create a dependency (line from one node to another) for every conflict.
- Test for cycles in precedence graph.
- No cycles => conflict serializable
- Cycles => no serial schedule exists that is conflict equivalent to original schedule.

transaction T_1	transaction T_2	transaction T_3
	read_item (Z); read_item (Y); write_item (Y);	
		read_item (Y); read_item (Z);
read_item (X); write_item (X);		
		write_item (Y); write_item (Z);
	read_item (X);	
read_item (Y); write_item (Y);		
	write_item (X);	

Schedule E

transaction T_1	transaction T_2	transaction T_3
		read_item (Y); read_item (Z);
read_item (X); write_item (X);		
		write_item (Y); write_item (Z);
	read_item (Z);	
read_item (Y); write_item (Y);		
	read_item (Y); write_item (Y); read_item (X); write_item (X);	

Schedule F

View Serializable

- Slightly weaker notion of serializability when compared to conflict-serializability.
- Premise
  - Each read operation of a transaction reads the result of the same write operation in both schedules
  - The write operation of each transaction must prodcue same results

Relationship between View and Conflict Serializability

- The two are the same under the "constrained write assumption" which assumes any transaction T that writes a value X (in other words **no blind writes**)
  - Reads OLD VALUE OF X
  - New X = F(OLD VALUE OF X)
- Example
  - T1: r1(X), w1(X)
  - T2: w2(X)
  - T3: w3(X)
  - The schedule r1(X);w2(X);w1(X);w3(X); is view equivalent but not conflict serializable