CONCURRENCY

CHAPTER 21-22.1 (6/E)

CHAPTER 17-18.1 (5/E)

LECTURE OUTLINE

- Errors in the absence of concurrency control
  - Need to constrain how transactions interleave
- Serializability
- Two-phase locking

LOST UPDATE PROBLEM

- Problematic interleaving of transactions

DB Values	T1		T2	
X = 80				
	read_item(X);	X = 80		
	X := X – 5;	X = 75		
			read_item(X);	X = 80
			X := X + 10;	X = 90
X = 75	write_item(X);			
X = 90			write_item(X);	

- X should be $X_0 - 5 + 10 = 85$
- Occurs when two transactions update the same data item, but both read the same original value before update

… r1(X);…; r2(X); …; w1(X); …; w2(X)

… r2(X);…; r1(X); …; w1(X); …; w2(X)

DIRTY READ PROBLEM

- *Phantom* update

DB Values	T1		T2	
X = 80				
	read_item(X);	X = 80		
	X := X – 5;	X = 75		
X = 75	write_item(X);			
			read_item(X);	X = 75
			X := X + 10;	X = 85
	X := X / 0;	T1 aborts		
X = 85			write_item(X);	

- X should be as if T1 didn't execute at all: $X_0 + 10 = 90$
- Occurs when one transaction updates a database item, which is read by another transaction but then the first transaction fails

… w1(X);…; r2(X); …; t1 rolled back

INCONSISTENT READS PROBLEM

- Transactions should read consistent values for isolated state of DB

DB Values	T1		T2	
X = <80, 15, 25>				
			read_item(X1);	X1 = 80
			SUM := X1;	SUM = 80
			read_item(X2);	X2 = 15
			SUM := SUM+X2;	SUM = 95
	read_item(X1);	X1 = 80		
	X1 := X1 + 5;	X1 = 85		
X = <85, 15, 25>	write_item(X1);			
	read_item(X3);	X3 = 25		
	X3 := X3 + 5;	X3 = 30		
X = <85, 15, 30>	write_item(X3);			
			read_item(X3);	X3 = 30
			SUM := SUM+X3;	SUM = 125

- SUM should be either 120 (80+15+25, before T1) or 130 (85+15+30, after T1)

… r2(X); …; w1(X); …; w1(Y); …; r2(Y); …

UNREPEATABLE READ PROBLEM

- Even with only one update, might read inconsistent values

DB Values	T1		T2	
X = 80				
			read_item(X);	X = 80
			Y := f(X);	
	read_item(X);	X = 80		
	X := X – 5;	X = 75		
X = 75	write_item(X);			
			read_item(X);	X = 75
			Z := f2(X,Y);	

- Z has a value that depends on two *different* values of X!
- Occurs when one transaction updates a database item, which is read by another transaction both before and after the update

…r2(X); … w1(X);…; r2(X); …

SERIAL SCHEDULES

- A schedule S is **serial** if *no interleaving* of operations from several transactions
  - For every transaction T, all the operations of T are executed consecutively
- Assume consistency preservation (ACID property):
  - Each transaction, if executed on its own (from start to finish), will transform a consistent state of the database into another consistent state.
  - Hence, each transaction is correct on its own.
  - Thus, any serial schedule will produce a correct result.
- Serial schedules are not feasible for performance reasons:
  - Long transactions force other transactions to wait
  - When a transaction is waiting for disk I/O or any other event, system cannot switch to other transaction
  - Solution: allow some interleaving

ACCEPTABLE INTERLEAVINGS

- Need to allow interleaving without sacrificing correctness
- Executing some operations in another order causes a different outcome
  - …r1(X); w2(X)… *vs.* …w2(X); r1(X)…
    - T1 will read a different value for X
  - …w1(Y); w2(Y)… *vs.* …w2(Y); w1(Y)...
    - DB value for Y after both operations will be different
- Two operations **conflict** if:
  1. They access the same data item X
  2. They are from two different transactions
  3. At least one is a write operation
     - Read-Write conflict : … r1(X); …; w2(X); …
     - Write-Write conflict : … w1(Y); …; w2(Y); …
- Note that two read operations do *not* conflict.
  - …r1(Z); r2(Z)… *vs.* …r2(Z); r1(Z)...
    - both transactions read the same values of Z
- Two schedules are **conflict equivalent** if the relative order of any two *conflicting* operations is the same in both schedules.

SERIALIZABLE SCHEDULES

- Although any serial schedule will produce a correct result, they might not all produce the *same* result.
  - If two people try to reserve the last seat on a plane, only one gets it. The serial order determines which one. The two orderings have different results, but either one is correct.
  - There are *n*! serial schedules for *n* transactions; any of them gives a correct result.
- A schedule S with *n* transactions is **serializable** if it is conflict equivalent to *some* serial schedule of the same *n* transactions.
- Serializable schedule "correct" because equivalent to some serial schedule, and any serial schedule acceptable.
  - It will leave the database in a consistent state.
  - Interleaving such that
    - transactions see data as if they were serially executed
    - transactions leave DB state as if they were serially executed
    - efficiency achievable through concurrent execution

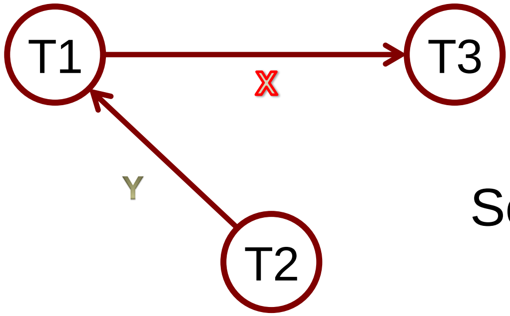
TESTING CONFLICT SERIALIZABILITY

- Consider all read_item and write_item operations in a schedule

1. Construct **serialization** graph
   - Node for each transaction T
   - Directed edge from Ti to Tj if some operation in Ti appears before a conflicting operation in Tj
2. The schedule is serializable if and only if the serialization graph has no cycles.

- Is the following schedule serializable?

b1; **r1(X)**; b2; **r2(Y)**; **w1(X)**; b3; **w2(Y)**; e2; **r1(Y)**; **r3(X)**; e3; **w1(Y)**; e1;



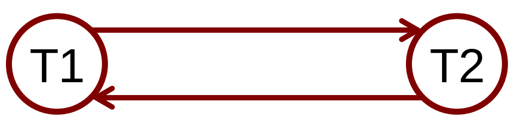
Serializable; equivalent to: T2; T1; T3

b2; **r2(Y)**; **w2(Y)**; e2; b1; **r1(X)**; **w1(X)**; **r1(Y)**; **w1(Y)**; e1; b3; **r3(X)**; e3;

TESTING CONFLICT SERIALIZABILITY

- Is the following schedule serializable?

DB Values	T1		T2	
X = 80				
	read_item(X);	X = 80		
	X := X – 5;	X = 75		
			read_item(X);	X = 80
			X := X + 10;	X = 90
X = 75	write_item(X);			
X = 90			write_item(X);	



DATABASE LOCKS

- Use **locks** to ensure that conflicting operations cannot occur
  - **exclusive** lock for writing; **shared** lock for reading
  - cannot read item with first getting shared or exclusive lock on it
  - cannot write item with first getting write (exclusive) lock on it
- Request for lock might cause transaction to **block** (wait)
  - No lock granted on X if some transaction holds write lock on X
    - write lock is exclusive
  - Write lock cannot be granted on X if some transaction holds any lock on X

T1 \ T2	holds read (shared) lock	holds write (exclusive) lock
requests read lock	OK	block T1
requests write lock	block T1	block T1

- Blocked transactions are unblocked and granted the requested lock when conflicting transaction(s) release their lock(s)
  - Like passing a microphone (but two types: one allows sharing)

ENFORCING CONFLICT SERIALIZABILITY

- **Rigorous two-phase locking (2PL)**:
  - Obtain read lock on X if transaction will read X
  - Obtain write lock on X (or promote read lock to write lock) if transaction will write X
  - Release all locks at end of transaction
    - whether commit or abort
  - This is SQL’s protocol.
- Rigourous 2PL ensures conflict serializability
- Potential problems:
  - **Deadlock**: T1 waits for T2 waits for … waits for Tn waits for T1
    - Requires assassin
  - **Starvation**: T waits for write lock and other transactions repeatedly grab read locks before all read locks released
    - Requires scheduler

T1	T2
request_read(A);	
read_lock(A);	
read_item(A);	
A := A + 100;	
request_write(A);	
write_lock(A);	
write_item(A);	
	request_read(A);
request_read(B);	
read_lock(B);	
read_item(B);	
B := B -10;	
request_write(B);	
write_lock(B);	
write_item(B);	
commit; /*unlock(A,B)*/	
	read_lock(A);
	read_item(A);
	…

OTHER TYPES OF EQUIVALENCE

- Rigorous two-phase locking is quite constraining.
- Under special semantic constraints, schedules that are not serializable may work correctly.
  - Consider transactions using commutative operations
  - Consider the following schedule S for the two transactions:

    b1; r1(X); w1(X); b2; r2(Y); w2(Y); r1(Y); w1(Y); e1; r2(X); w2(X); e2;

    - Not (conflict) serializable
    - However, results are correct if it came from following update sequence:
      - r1(X); X := X – 10; w1(X);
      - r2(Y); Y := Y – 20; w2(Y);
      - r1(Y); Y := Y + 30; w1(Y);
      - r2(X); X := X + 40; w2(X);
    - Known as *debit-credit transactions*
      - Sequence explanation: debit, debit, credit, credit
- Specialized transaction processing may be conducted under more liberal constraints to allow more interleavings.

LECTURE SUMMARY

- Characterizing schedules based on serializability
  - Serial and non-serial schedules
  - Conflict equivalence of schedules
  - Serialization graph
- Rigorous two-phase locking
  - Guarantees conflict serializability
  - Deadlock and starvation
- Weaker forms of “correctness”

SAMPLE QUESTION

- Determine whether or not each of the following four transaction schedules is conflict serializable. If a schedule is serializable, specify a serial order of transaction execution to which it is equivalent.

  H1 = r1[x]; r2[y]; w2[x]; r1[z]; r3[z]; w3[z]; w1[z];
  H2 = w1[x]; w1[y]; r2[u]; w2[x]; r2[y]; w2[y]; w1[z];
  H3 = w1[x]; w1[y]; r2[u]; w1[z]; w2[x]; r2[y]; w1[u];
  H4 = w1[x]; w2[u]; w2[y]; w1[y]; w3[x]; w3[u]; w1[z];