We have a FINAL coming up…..

Q10 - 1

Q1
4 Points

T1	*T2*
read (A)	
A -= 100	
write(A)	
	read(B)
	B += 10
	write(B)
read(B)	
B += 100	
write(B)	
	read(A)
	A += 10
	write(A)

Choose the most specific answer below:

- (●) serializable
- ( ) conflict serializable
- ( ) view serializable
- ( ) not serializable

Explanation

Not conflict serializable because accesses to A require edge from T1 to T2, while accesses to B require edge from T2 to T1. Hence, cycle in precedence graph, not conflict serialiable.

Not view serializable because T1 has initial read of A, T2 has initial read of B, no view serial schedule could have both of those.

However, the answer IS guaranteed to be the same as either T1,T2 or T2,T1 because the reads and writes do not clobber each other, and the updates are just constants, meaning their order does not matter. Hence, serializable.

Q10 - 2

T1	T2
read (A)	
A -= 100	
write(A)	
	read(A)
	A += 10
	write(A)
read(B)	
B += 100	
write(B)	
	read(B)
	B += 10
	write(B)

Choose the most specific answer below:

- ○ serializable
- ◉ conflict serializable
- ○ view serializable

Explanation

Precedence graph would only have edge from T1 to T2.
No cycle, so conflict serializable. This is most specific as CS is subset of VS which is subset of S.

Q10 - 3

T1	T2
read (A)	
A -= 100	
write(A)	
	read(B)
	tmp = 0.1*B
	B -= tmp
read(B)	
B += 100	
write(B)	
	write(B)
	read(A)
	A += tmp
	write(A)

Choose the most specific answer below:

- ○ serializable
- ○ conflict serializable
- ○ view serializable
- ◉ not serializable

Explanation

Not CS (cycle in graph). Not VS because T2 has both last writes AND initial read of B (no serial sched would have both).

Could show not serializable by plugging in numbers and showing that equiv to neither T1,T2 or T2,T1.

Easier to note that T1's increment of B by 100 is overwritten by T2 (which reads B first and writes last, so T1's add to B is ignored. No serial schedule would do this.

Edge in precedence graph if:

1. T_i executes write(Q) before T_j executes read(Q).
2. T_i executes read(Q) before T_j executes write(Q).
3. T_i executes write(Q) before T_j executes write(Q).

Q10 - 4

Assume that each write outputs a unique value computed using all prior reads by the same transactions.

T2	*T1*	*T3*	*T4*
		read(A)	
read(A)			
	read (A)		
	write(A)		
		read(B)	
			read(C)
write(A)			
	read(B)		
read(B)			
write(B)			
		write(B)	
	read(C)		
			write(C)

Choose the most specific answer below:

- ( ) serializable
- ( ) conflict serializable
- ( ) view serializable
- (•) not serializable

Explanation

A better way to put this might be that the "uniqueness" requirement implies that all reads-from relationships must be the same as in any equivalent serial schedule. Otherwise, written values would change and consistency would presumably be violated.
T2's read of B is before T3's write, so T2 < T3 in the equivalent serial schedule. However, T2's write of A is *after* T3's read of A, meaning T3 < T2 in the equivalent serial schedule. Both of these cannot be true at the same time, so there can be no equivalent serial schedule.

Q10 - 5

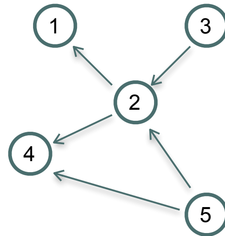
The following schedule is conflict-serializable. List out at least four equivalent serial schedules for this schedule. You can do this by drawing the precedence graph, and then finding sequences of transactions where all the edges go from left to right. As an example: T1, T2, T3, T4, T5 is clearly not an equivalent serial schedule because there is an edge from T3 to T1 (due to A).

T1	*T2*	*T3*	*T4*	*T5*
	read (A)			
		read(A)		
	write(A)			
read(A)				
		read(B)		
		write(B)		
	read(B)			
write(A)				
				read(C)
read(B)				
				write(C)
	read(C)			
			read(C)	
			write(C)	
write(B)				

Explanation

Edges:
$2 \to 1$
$3 \to 2$
$5 \to 2$
$5 \to 4$
$2 \to 4$
3,2,1 must be in that order
5,2,4 must be in that order

53214
53241
35214
35241
might be others?



Q11 - 2.4

Q2.4
1 Point

Instead of the previous, T3 wants to execute the following SQL query:

```
update R2 set R2.A = 100 where R2.B = 10;
```

Assume that only a few tuples in R2 satisfy the condition. Which of the following is the minimal set of lock requests T3 should make?

- ○ IS(DB), S(R2), X locks on tuples that satisfy condition
- ○ IX(DB), X(R2)
- ○ IX(DB), IX(R2), X locks on tuples that satisfy condition
- ● IX(DB), SIX(R2), X locks on tuples that satisfy condition
- ○ IS(DB), IX(DB), SIX(R2), X locks on tuples that satisfy condition

Explanation

Only needs IX on DB, as it won't be reading R1. Needs shared lock on all of R2 to find specific tuple, then exclusive on the match.

Q11 - 3

Q3
2 Points

Assume a database system is using the timestamp-ordering protocol (as discussed in the slides, no Thomas's Write Rule). Consider 5 transactions with timestamp order: T1, T2, T3, T4, T5 (i.e., T1 was the first transaction, etc).

In the following interleaved schedule, which transactions would be forced to abort? Multiple instructions from the same transaction are grouped together if there is no interleaving, for compactness.

Answer for each transaction independently of what happens to other transactions -- i.e., when considering whether T1 will be aborted, assume all the other transactions are valid.

We are using an alternate notation here for simplicity -- the schedule should still be read top to bottom.

Transaction ID	Instructions
T1	read(A), read(C)
T2	read(B), write(C)
T1	write(C)
T3	read(A), read(B)
T1	write(C)
T4	read(A), read(B), write(A)
T5	read(A), write(A)
T3	read(D)
T2	read(B), write(A)
T4	write(A)

- ☑ T1
- ☑ T2
- ☐ T3
- ☑ T4
- ☐ T5

Explanation

T1 aborts on w(C) because T2 already wrote C
T2 aborts on w(A) because T3 already read it
T4 aborts on w(A) because T5 already read it
T3 is as pure as the driven snow

Q11 - 4

Q4

1 Point

When a transaction is rolled back under timestamp ordering, it is assigned a new timestamp. Clearly explain why can it not simply keep its old timestamp.

Explanation

- If t_i aborted because it read a value for x that it should not have, it's because x was written by a younger transaction t_j. This likely will not change if restarting w/ the same timestamp, as all subsequent transactions are younger than t_j.
- If t_i aborted writing x because because a younger transaction t_j had read x, this also will likely never change for the same reason.
- The "likely" caveat is because it's possible, though not likely, that all younger transactions that would have caused t_i to abort *also* aborted, leaving the field clear.

Q11 - 5

Q5

4 Points

Consider the following schedule under the Validation Protocol where we highlight when transactions enter the different phases.

T1	*T2*	*T3*	*T4*
start, read (A)			
read(B), write(B)			
	start, read(B)		
	write(B)		
validate, **finish**			
			start, read(A)
		start, read(A), read(C)	
			validate, **finish**
		write(B), write(C)	
	validate, **finish**		
		validate, **finish**	

Q5.1

2 Points

Which of the transactions will successfully pass the validation? Answer for each transaction in isolation -- i.e., assume all other transactions are successful when deciding whether any specific transaction will validate or not.

- [x] T1
- [ ] T2
- [x] T3
- [x] T4

Explanation

Order is T1, T4, T2, T3 (= validate order)
T1 validates because no prior trans.
T4 validates because only prior trans is T1, which finished before T4 started.
T2 fails because prior trans T1 didn't finish first, and wrote B, which T2 reads.
T3 validates because only reads A,C, never written by preceding trans.

Q5.2

2 Points

What is the serialization order?

- ( ) $T_1 < T_2 < T_3 < T_4$
- ( ) $T_2 < T_1 < T_3 < T_4$
- ( ) $T_1 < T_3 < T_4 < T_2$
- (•) $T_1 < T_4 < T_2 < T_3$

Q11 - 6

Looking at T_i, if for all T_k with TS(T_k) < TS(T_i) then validation succeeds if:

- finish(T_i) < start(T_k)

or:

- the set of data items written by T_k does not intersect with the set of data items read by T_i and
- T_k completes its write phase before T_i starts validation:

  start(T_i) < finish(T_k) < validation(T_i)

- [x] T1
- [ ] T2
- [ ] T3
- [x] T4
- [x] T5

Q6
1 Point

Which transactions will correctly validate under the Validation Protocol in this schedule?

T1	*T2*	*T3*	*T4*	*T5
start, read (A)				
		start, read(A)		
	start, read(A)			
write(A)				
		read(B), write(B)		
	write(A)			
read(B)				
				start, read(C)
	read(B)			
read(C)				
validate/finish				
		validate/finish		
				write(C)
				validate/finish
			start, read(C)	
			write(C)	
			validate/finish	
	write(B)			
	validate/finish			

Q11 - 8

Q8.1
1 Point

T1	T2	T3	T4	T5
Start R(B)→ 2 R(C) → 2				
	Start R(A) → 1 R(B)→ 2			
		Start		
			Start R(A) → 1 W(A := 3) Commit-Req Commit	
		R(A) → ????? W(A:=2) Commit-Req		
				Start R(C) → 2 W(C := 4) Commit-Req Commit
	R(C) → ?????? Commit-Req			
W(B) := 3 Commit-Req				

What values of "A" and "C" would T3 and T2 read respectively? See the "??????" for the specific instructions.

- (•) T2 will read C = 2, T3 will read A = 1
- ( ) T2 will read C = 2, T3 will read A = 3
- ( ) T2 will read C = 4, T3 will read A = 1
- ( ) T2 will read C = 4, T3 will read A = 3

Explanation

Snapshot reads are always satisfied by either the snapshot values (values at transaction start or by values written by the same transaction.

Q8.2
1 Point

Of the three transactions T1, T2, and T3, that have made a Commit-Req, which ones will be allowed to commit?

- (•) T1 and T2 will be allowed to commit.
- ( ) T1 and T3 will be allowed to commit.
- ( ) T2 and T3 will be allowed to commit.
- ( ) All three will be allowed to commit.

Explanation

All read transactions (T2) commit in snapshot isolation.

T1 succeeds because it's writeset (B) does not intersect with that of any other committed transaction.

T3 fails because during the course of its execution (between its start and commit), one of the items in its writeset (A) was written back by another transaction (T4).

Q12 - 2

Q2
2 Points

The main difference between strict 2PL and rigorous 2PL is that -- for the latter, the serializability order is the same as the commit order (i.e., if you were to draw a precedence graph and find the serial schedule -- it would be consistent with the order in which the transactions committed).

Show through an example how this is not true for Strict 2PL. More specifically, provide an example schedule using two transactions that follows Strict 2PL, but the serializability order is different from the commit order.

(show lock acquisitions (and type), releases, commits, reads, writes)

Explanation

t1	t2
	S(A)
	read(A)
	release(A)
X(A)	
write(A)	
release(A)	
commit	
	commit

Q12 - 3

Q5.1
2 Points

Which rule(s) of log-based recovery does the following schedule violate ?

Transaction ID	Instruction	Log Record and Number
T1	B = B + 100	1: <T1, UPDATE, B, 100 --> 200>
T2	C = C - 900	2: <T2, UPDATE, C, 1000 --> 900>
	output(C)	
T1	A = A - 100	3: <T1, UPDATE, A, 400 --> 300>
	output(Log Record 1)	
	output(Log Record 2)	

- [ ] Write-ahead logging: log record 1 needs to be output before B
- [x] Write-ahead logging: log record 2 needs to be output before C
- [ ] In-order logging: log record 2 must be output before log record 1
- [ ] No steal: B should not be output before T1 has committed

Q5.2
1 Point

Which rule(s) of log-based recovery does the following schedule violate ?

Transaction ID	Instruction	Log Record and Number
(a) T1	A = A + 100	1: <T1, UPDATE, A, 100 --> 200>
(b) T2	B = B + 100	2: <T2, UPDATE, B, 300 --> 400>
(c)	output(B)	
(d) T2	undo	3: <T2, CLR, B, 300>
(e)	output(Log record 2)	
(f) T1	COMMIT	4: <T1, COMMIT>
(g)	output(Log Record 1)	
(h)	output(Log Record 3)	
(i)	output(Log Record 4)	

- [x] Write-ahead logging: log record 2 needs to be output before B
- [x] In-order logging: log record 1 must be output before log record 2
- [ ] No steal: B should not be output before T2 has committed/aborted

Q12 - 6

Q6
2 Points

Consider the following log trace. What should be the values of the two attributes after restart recovery? The update log records list the old value first, and the new value second. CLR stands for Compensation Log Record -- these are log records written out during UNDO.

- <T1, START>, <T2, START>, <T3, START>
- <T1, UPDATE, A, 1000 --> 900>
- <T3, UPDATE, A, 900 --> 800>
- <T3, CLR, A, 900>
- <T1, COMMIT>
- <T2, UPDATE, A, 900 --> 700>
- <T2, UPDATE, B, 900 --> 1000>
- <T3, ABORT>
- -------- SYSTEM CRASH -------

- ○ A = 700, B = 900
- ◉ A = 900, B = 900
- ○ A = 800, B = 1000
- ○ A = 900, B = 1000

Q12 - 7

Q7
2 Points

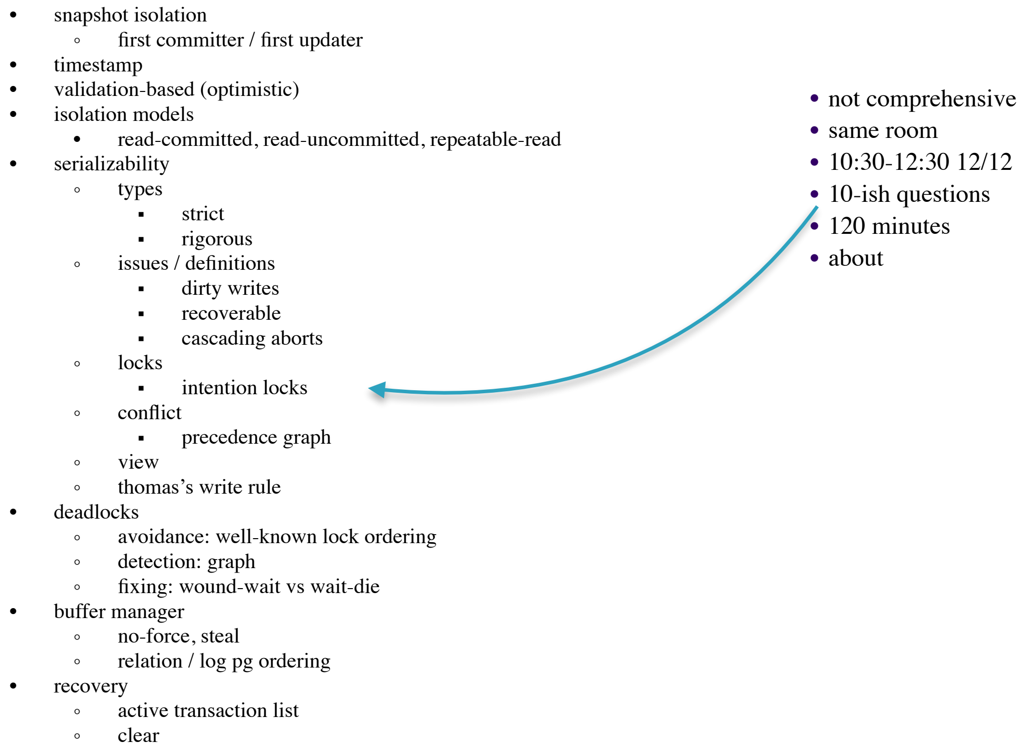
Consider the following log trace. The update log records list the old value first, and the new value second. CLR stands for Compensation Log Record -- these are log records written out during UNDO.

What should be the values of the two attributes after restart recovery?

- <T1, START>, <T2, START>, <T3, START>, <T4, START>
- <T1, UPDATE, B, 800 --> 700>
- <T2, UPDATE, A, 800 --> 700>
- <T1, COMMIT>
- <T3, UPDATE, B, 700 --> 600>
- <T3, CLR, B, 700>
- <T4, UPDATE, C, 800 --> 500>
- -------- SYSTEM CRASH -------

- ○ A = 500 and B = 700
- ○ A = 500 and B = 800
- ◉ A = 800 and B = 700
- ○ A = 800 and B = 800
- ○ A = 700 and B = 600

Final



Class grades

- updated assignment weights
- hw11 now reflected

Minimum	39.55
Maximum	70.71
Mean	57.46
Std Dev	6.64

out of 74 pts on 12/5

- Syllabus:

  Final course grades will be **curved** as necessary (roughly the avg/median is a B-, w/ A- and C- one stdev up and down. The curve is based the entirety of the the coursework at the end of the semester.

current 12/5 w/ 74% of grade

	PTS	PCT
stdev	6.64	9.0%
avg	57.46	77.6%
A-	64.1	86.6%
B-	57.5	77.6%
C-	50.8	68.7%