Chapter 18 solutions

18.1.1)

$r_1(A)$; $r_1(B)$; $w_1(B)$; $r_1(C)$; $w_1(C)$; $r_1(D)$; $w_1(D)$; $r_1(E)$; $w_1(E)$

18.1.2)

Think of the interleaving as having 10 positions. The 4 actions of the first transaction occupy any 4 of the 10, so the number of interleavings is (10 choose 4), or 10*9*8*7/1*2*3*4 = 210.

18.2.1a)

Assume A=1, B=2

(T_1,T_2)

T_1

Read (A,t)	t=1
T=t+2	t=3
W(A,t)	A=3
R(B,t)	t=2
t=t*3	t=6
W(B,t)	B=6

T_2

R(B,s)	s=6
s=s*2	s=12
W(B,s)	B=12
R(A,s)	s=3
s=s+3	s=6
W(A,s)	A=6

Final values after (T_1,T_2) : **A=6, B=12**

(T_2,T_1)

T_2

R(B,s)	s=2
s=s*2	s=4
W(B,s)	B=4
R(A,s)	s=1
s=s+3	s=4
W(A,s)	A=4

T_1

Read (A,t)	t=4
T=t+2	t=6
W(A,t)	A=6
R(B,t)	t=4
t=t*3	t=12
W(B,t)	B=12

Final values after (T_2,T_1) : **A=6, B=12**

18.2.1b)

Serializable schedule

T_1	T_2
R(A,t)	
t=t*2	
W(A,t)	
	R(B,s)
	s=s*2
	W(B,s)
R(B,t)	
t=t*3	
W(B,t)	
	R(A,s)
	s=s+3
	W(A,s)

Non- serializable schedule

T_1	T_2
R(A,t)	
t=t*2	
W(A,t)	
	R(B,s)
R(B,t)	
	s=s*2
t=t*3	
	W(B,s)
W(B,t)	
	R(A,s)
	s=s+3
	W(A,s)

The update of B by T_2 is lost. The final value of B is 6 (instead of 12 as in 18.2.1a)

18.2.1c)

(T_1,T_2)
(T_2,T_1)

18.2.1d)

The key observation is that in order for an interleaving to be serializable, whichever transaction reads *A* first, must write it before the second reads *A*, and likewise for *B*. There are four possibilities:

1. T_1 reads both *A* and *B* first: Then all the steps of T_1 must precede all the steps of T_2, if the interleaving is serializable. There is 1 order of this type.
2. T_2 reads both *A* and *B* first: Similarly, there is 1 order of this type.
3. T_1 reads *B* first but T_2 reads *A* first. This situation is impossible.

4. T_1 reads *A* first but T_2 reads *B* first. Now, the first three steps of each transaction may interleave in any way, and the last three of each may interleave in any way, but the two groups of six actions must not interleave. The crucial observation is that the fourth step of each transaction (their second reads) must follow the third step of each transaction (their first writes), either to avoid reading *A* or *B* before it is written, or because actions of the same transaction cannot be reordered. The number of serializable orders of this type is (6 choose 3)*(6 choose 3) = 20*20 = 400.

The total number of serial orders is thus 402.

18.2.2a)

If a swap of two adjacent actions is legal, then the two actions can also be swapped back again, legally. Thus, instead of looking for schedules that are conflict equivalent to the given serial order, we can instead consider into what other schedules the serial order can be permuted by legal swaps. However, since in the serial order, the 4th and 5th steps are $w_1(B); r_2(B)$, these cannot be swapped. All other adjacent pairs are of the same transaction, and therefore they surely cannot be swapped. Thus, the only schedule conflict equivalent to the serial order (T_1, T_2) is that order itself; the answer is 1.

18.2.2b)

Only 1

$r_2(B)w_2(B)r_2(A)\underline{w_2(A)r_1(A)}w_1(A)r_1(B)w_1(B)$

Fourth and fifth actions cannot be swapped since they conflict on A (atleast one is write)

18.2.2c)

Similar to 18.2.1 (d), for interleaving schedules to be serializable, a variable must be read and written by one transaction before another transaction reads it.

For serial schedule (T1,T2), the four possibilities are:

1. T1 reads A & B first i.e. the one serial schedule (T1,T2).
2. T2 reads A & B first. This is not possible for serial schedule (T1,T2).
3. T1 reads B first while T2 reads A first. This is impossible since actions within a transaction cannot be reordered.
4. T1 reads A first while T2 reads B first. The first two steps of T1 can interleave in any way with first two steps of T2. Similarly last two steps of T1 can interleave in any way with the last two steps of T2. Thus number of serializable schedules are: (4 choose 2) * (4 choose 2) = 36.

Thus the total number of schedules = 36 + 1 = 37.

18.2.2d)

Number of actions are different.

18.2.3a)

Conflict serializable schedules to (T_1,T_2)

1) $r_1(A)w_1(A)r_1(B)\underline{w_1(B)r_2(A)}w_2(A)r_2(B)w_2(B)$

2) $r_1(A)w_1(A)\underline{r_1(B)r_2(A)}\ \underline{w_1(B)w_2(A)}r_2(B)w_2(B)$ (2 swaps possible)
3) $r_1(A)w_1(A)r_2(A)r_1(B)\underline{w_1(B)w_2(A)}r_2(B)w_2(B)$ (first swap from 2)
4) $r_1(A)w_1(A)r_2(A)\underline{r_1(B)w_2(A)}w_1(B)r_2(B)w_2(B)$
5) $r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$
6) $r_1(A)w_1(A)r_1(B)r_2(A)w_2(A)w_1(B)r_2(B)w_2(B)$ (second swap from 2)

Similarly, there are six conflict-serializable schedules equivalent to serial order (T2,T1)

Hence total schedules = 6 + 6 = 12

18.2.3b)

In addition to the twelve schedules above, interleaving using transaction behavior like 18.2.1 allows for two more possibilities:

1. T1 reads A first while T2 reads B first. Only such one schedule is possible:
r1(A);w1(A);r2(A);w2(A);r2(B);w2(B);r1(B);w1(B);

1. T1 reads B first while T2 reads A first. Only such one schedule is possible:
r2(A);w2(A);r1(A);w1(A);r1(B);w1(B);r2(B);w2(B);

Thus total number of serializable schedules = 12 +1 +1 = 14

18.2.4a)

i)

The precedence graph is:

T3 -----> T2 -----> T1

ii)

The schedule is thus conflict-serializable. The only conflict-equivalent serial schedule is (T_3, T_2, T_1).

iii)

No. Assume, for example, that initially each of *A, B*, and *C* are 10. T_1 sets *A* and *C* to 20, T_2 sets *B* to *A*+*B*+*C*, and T_3 prints *B*. Then if T_3 does not precede T_2, it prints the wrong value, and if T_2 does not precede T_1, it sets *B* to the wrong value.

18.2.4b)

i)

$r_1(A)w_3(A)$ $T_1<T_3$
$w_1(B)r_2(B)$ $T_1<T_2$
$w_2(C)r_3(C)$ $T_2<T_3$
Precedence graph is :

T1 -----> T2 -----> T3

ii)

Yes, schedule is conflict-serializable. Conflict-equivalent serial schedule is (T1,T2,T3).

iii)

No, same as part a iii)

18.2.4c)

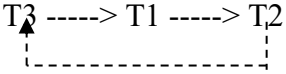
i)

$w_3(A)r_1(A)$ $T_3<T_1$

$w_1(B)r_2(B)$ $T_1<T_2$

$w_2(C)r_3(C)$ $T_2<T_3$

Precedence graph is:



ii)

No, schedule is not conflict-serializable

iii)

NA, since schedule has conflict.

18.2.4d)

i)

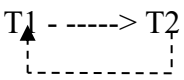
r1(A)r2(A) T1<T2

w1(B)w2(B)r1(B)r2(B) T2<T1

w2(C)

w1(D)

Precendence graph is :



ii)
No, schedule is not conflict-serializable.

iii)
NA, since schedule has conflict.

18.2.4e)
i)
$r_1(A)r_2(A)r_3(A)w_1(A)$ $T_3<T_1$
$r_1(B)r_2(B)r_4(B)w_2(B)$ $T_4<T_2$

ii)
Yes, the schedule is conflict-serializable. The conflict-equivalent serial schedules are:
$T_3T_1T_4T_2$
$T_3T_4T_1T_2$
$T_3T_4T_2T_1$
$T_4T_3T_1T_2$
$T_4T_3T_2T_1$
$T_4T_2T_3T_1$

iii)
NA, since schedule has conflict.

18.2.5)
In order that the three conditions are satisfied, one approach can be
:T1 and T2 are not in conflict (better yet, not operating on same variables)
:Another transaction T3 is interleaved into the serial schedule (T1,T2) such that the actions of T3 that conflict with T1 precede it while the actions of T3 that conflict with T2 follow it.

Example:
T1:r1(A);W1(A);
T2:r2(B);w2(B);
T3:r3(A);w3(A);r3(B);w3(B);

Schedule S:
r3(A);w3(A);r1(A);w1(A);r2(B);w2(B);r3(B);w3(B);

The Precedence Graph of schedule S:

T2 -> T3 -> T1

Since T3 writes A before T1 reads A, T3 <s T1
Also, T2 writes B before T3 reads B, T2 <s T3

In schedule S, every action of T1 precedes T2. Hence condition i is satisfied.
There is no cycle in the Precedence Graph. Hence S is conflict serializable(condition ii)
Also from Precedence Graph, T2 precedes T1 for every serial schedule conflict-equivalent to S.

18.2.6)

Consider the precedence graph of the serial schedule of transactions (T1,T2,T3,...) such that T1 has reads for all variables and all other transactions write two of the variables such that T1 is in conflict with all other transactions.
A cycle of length n can be found by reversing the conflicting actions of T1 and Tn. Since no transaction other than T1 and Tn operate on the variable in the conflicting actions, no smaller cycle exists.

18.3.1a)

18.3.1b)

In legal schedules locks on a particular element are only held by one transaction at a given time. Since locks are being acquired by T1 and T2 just before reads and bring released just after writes; we can restate the legality requirement here as:
Between the read and write of an element by one transaction, No other transaction should act on the element. Thus below 32 transactions are illegal:

r2(B);w2(B);r2(A);r1(A);w2(A);w1(A);r1(B);w1(B);
r2(B);w2(B);r2(A);r1(A);w1(A);w2(A);r1(B);w1(B);
r2(B);w2(B);r2(A);r1(A);w1(A);r1(B);w2(A);w1(B);
r2(B);w2(B);r2(A);r1(A);w1(A);r1(B);w1(B);w2(A);
r2(B);w2(B);r1(A);r2(A);w2(A);w1(A);r1(B);w1(B);
r2(B);w2(B);r1(A);r2(A);w1(A);w2(A);r1(B);w1(B);
r2(B);w2(B);r1(A);r2(A);w1(A);r1(B);w2(A);w1(B);
r2(B);w2(B);r1(A);r2(A);w1(A);r1(B);w1(B);w2(A);

r2(B);r1(A);w2(B);r2(A);w2(A);w1(A);r1(B);w1(B);
r2(B);r1(A);w2(B);r2(A);w1(A);w2(A);r1(B);w1(B);
r2(B);r1(A);w2(B);r2(A);w1(A);r1(B);w2(A);w1(B);
r2(B);r1(A);w2(B);r2(A);w1(A);r1(B);w1(B);w2(A);

r2(B);r1(A);w1(A);r1(B);w2(B);r2(A);w2(A);w1(B);

r2(B);r1(A);w1(A);r1(B);w2(B);r2(A);w1(B);w2(A);
r2(B);r1(A);w1(A);r1(B);w2(B);w1(B);r2(A);w2(A);
r2(B);r1(A);w1(A);r1(B);w1(B);w2(B);r2(A);w2(A);
r1(A);r2(B);w2(B);r2(A);w2(A);w1(A);r1(B);w1(B);
r1(A);r2(B);w2(B);r2(A);w1(A);w2(A);r1(B);w1(B);
r1(A);r2(B);w2(B);r2(A);w1(A);r1(B);w2(A);w1(B);
r1(A);r2(B);w2(B);r2(A);w1(A);r1(B);w1(B);w2(A);

r1(A);r2(B);w1(A);r1(B);w2(B);r2(A);w2(A);w1(B);
r1(A);r2(B);w1(A);r1(B);w2(B);r2(A);w1(B);w2(A);
r1(A);r2(B);w1(A);r1(B);w2(B);w1(B);r2(A);w2(A);
r1(A);r2(B);w1(A);r1(B);w1(B);w2(B);r2(A);w2(A);

r1(A);w1(A);r2(B);r1(B);w2(B);r2(A);w2(A);w1(B);
r1(A);w1(A);r2(B);r1(B);w2(B);r2(A);w1(B);w2(A);
r1(A);w1(A);r2(B);r1(B);w2(B);w1(B);r2(A);w2(A);
r1(A);w1(A);r2(B);r1(B);w1(B);w2(B);r2(A);w2(A);
r1(A);w1(A);r1(B);r2(B);w2(B);r2(A);w2(A);w1(B);
r1(A);w1(A);r1(B);r2(B);w2(B);r2(A);w1(B);w2(A);
r1(A);w1(A);r1(B);r2(B);w2(B);w1(B);r2(A);w2(A);
r1(A);w1(A);r1(B);r2(B);w1(B);w2(B);r2(A);w2(A);

While below 38 transactions are legal:

r2(B);w2(B);r2(A);w2(A);r1(A);w1(A);r1(B);w1(B);

r2(B);w2(B);r1(A);w1(A);r2(A);w2(A);r1(B);w1(B);
r2(B);w2(B);r1(A);w1(A);r2(A);r1(B);w2(A);w1(B);
r2(B);w2(B);r1(A);w1(A);r2(A);r1(B);w1(B);w2(A);
r2(B);w2(B);r1(A);w1(A);r1(B);r2(A);w2(A);w1(B);
r2(B);w2(B);r1(A);w1(A);r1(B);r2(A);w1(B);w2(A);
r2(B);w2(B);r1(A);w1(A);r1(B);w1(B);r2(A);w2(A);
r2(B);r1(A);w2(B);w1(A);r2(A);w2(A);r1(B);w1(B);
r2(B);r1(A);w2(B);w1(A);r2(A);r1(B);w2(A);w1(B);
r2(B);r1(A);w2(B);w1(A);r2(A);r1(B);w1(B);w2(A);
r2(B);r1(A);w2(B);w1(A);r1(B);r2(A);w2(A);w1(B);
r2(B);r1(A);w2(B);w1(A);r1(B);r2(A);w1(B);w2(A);
r2(B);r1(A);w2(B);w1(A);r1(B);w1(B);r2(A);w2(A);
r2(B);r1(A);w1(A);w2(B);r2(A);w2(A);r1(B);w1(B);
r2(B);r1(A);w1(A);w2(B);r2(A);r1(B);w2(A);w1(B);
r2(B);r1(A);w1(A);w2(B);r2(A);r1(B);w1(B);w2(A);
r2(B);r1(A);w1(A);w2(B);r1(B);r2(A);w2(A);w1(B);
r2(B);r1(A);w1(A);w2(B);r1(B);r2(A);w1(B);w2(A);

r2(B);r1(A);w1(A);w2(B);r1(B);w1(B);r2(A);w2(A);

r1(A);r2(B);w2(B);w1(A);r2(A);w2(A);r1(B);w1(B);
r1(A);r2(B);w2(B);w1(A);r2(A);r1(B);w2(A);w1(B);
r1(A);r2(B);w2(B);w1(A);r2(A);r1(B);w1(B);w2(A);
r1(A);r2(B);w2(B);w1(A);r1(B);r2(A);w2(A);w1(B);
r1(A);r2(B);w2(B);w1(A);r1(B);r2(A);w1(B);w2(A);
r1(A);r2(B);w2(B);w1(A);r1(B);w1(B);r2(A);w2(A);
r1(A);r2(B);w1(A);w2(B);r2(A);w2(A);r1(B);w1(B);
r1(A);r2(B);w1(A);w2(B);r2(A);r1(B);w2(A);w1(B);
r1(A);r2(B);w1(A);w2(B);r2(A);r1(B);w1(B);w2(A);
r1(A);r2(B);w1(A);w2(B);r1(B);r2(A);w2(A);w1(B);
r1(A);r2(B);w1(A);w2(B);r1(B);r2(A);w1(B);w2(A);
r1(A);r2(B);w1(A);w2(B);r1(B);w1(B);r2(A);w2(A);
r1(A);w1(A);r2(B);w2(B);r2(A);w2(A);r1(B);w1(B);
r1(A);w1(A);r2(B);w2(B);r2(A);r1(B);w2(A);w1(B);
r1(A);w1(A);r2(B);w2(B);r2(A);r1(B);w1(B);w2(A);
r1(A);w1(A);r2(B);w2(B);r1(B);r2(A);w2(A);w1(B);
r1(A);w1(A);r2(B);w2(B);r1(B);r2(A);w1(B);w2(A);
r1(A);w1(A);r2(B);w2(B);r1(B);w1(B);r2(A);w2(A);

r1(A);w1(A);r1(B);w1(B);r2(B);w2(B);r2(A);w2(A);

18.3.1c)

According to semantics of transactions, four possibilities for serializable schedules are:
(i) Serial order (T1,T2) = 1 transaction
(ii) Serial order (T2,T1) = 1 transaction
(iii)T1 reads B first while T2 reads A first. This is impossible.
(iv) T2 reads A first while T1 reads B first. The first 2 actions of T1 can interleave with first two actions of T2. Similarly the last two actions of T1 can interleave with last two actions of T2.

But the two groups of 4 actions cannot interleave. Thus there are (4 choose 2) * (4 choose 2) = 36.

Hence total = 38 serializable schedules.

18.3.1d)

Only two schedules are conflict serializable:the serial orders (T1,T2) and (T2,T1). Both T1 and T2 start with the read of one element and end with write of the other element. Thus no conflict-free swaps are possible in the serial orders.

18.3.1e)
No, there are none. Due to the semantics of the transactions, and the placement of lock and unlock operations (as explained in (b)), all legal schedules are serializable in this exercise.

18.3.2)
Suppose the schedule starts with T_1 locking and reading *A*. If T_2 locks *B* before T_1 reaches its unlocking phase, then there is a deadlock, and the schedule cannot complete. Thus, if T_1 performs an action first, it must perform *all* its actions before T_2 performs any. Likewise, if T_2 starts first, it must complete before T_1 starts, or there is a deadlock. Thus, only the two serial schedules of these transactions are legal.

18.3.3a)
Note that this question refers to the solved Exercise 18.2.4(a). For the sequence of the latter exercise, the second request, *$r_2(A)$*, cannot be granted because of the lock held by T_1. T_2 is thus delayed until after the 4th step of the sequence, *$w_1(A)$*, after which T_1 releases its lock on *A*. No other delays are necessary, so the sequence of scheduled actions would be: $r_1(A)$; $r_3(B)$; $w_1(A)$; $r_2(A)$; $r_2(C)$; $r_2(B)$; $w_2(B)$; $w_1(C)$.

18.3.3b)
$r_1(A)w_1(B)r_2(B)w_2(C)r_3(C)w_3(A)$
No delays necessary.

18.3.3c)
$w_3(A)r_1(A)w_1(B)r_2(B)w_2(C)r_3(C)$
No delays necessary.

18.3.3d)
$r_1(A)r_2(A)w_1(B)$**$\underline{w_2(B)}$**$r_1(B)r_2(B)w_2(C)w_1(D)$
$w_2(B)$ is delayed since $w_1(B)$ is holding a lock to B.
Modified schedule:
$r_1(A)r_2(A)w_1(B)r_1(B)w_2(B)r_2(B)w_2(C)w_1(D)$

18.3.3.e)
$r_1(A)$**$\underline{r_2(A)}$**$r_1(B)r_2(B)$**$\underline{r_3(A)r_4(B)}$**$w_1(A)w_2(B)$
$r_2(A)$ and $r_3(A)$ are delayed since $r_1(A)$ is holding a lock to A.
$r_4(B)$ is delayed since $r_2(B)$ is holding a lock to B.
Modified schedule:
$r_1(A)r_1(B)r_2(B)w_1(A)r_2(A)r_3(A)w_2(B)r_4(B)$

18.3.4)a)
The consistent sequences(i+ii) are interleavings of the sequences

1. l1(A);r1(A);u1(A)
2. l1(B);w1(B);u1(B)

The number of such interleaving are (6 choose 3) = 20.

A sequence is not two-phase locked if the unlock of 1 occurs before lock of 2 or vice versa. Only 2 such interleavings are possible i.e. all of 1 followed by all of 2 or vice versa. Thus number of consistent but not two-phase orders = 2. (ii)

Thus the number of two-phase locked orders are = 20 - 2 = 18. (i)

Next consider number of two-phased locked orders(i+iii). The two locks must occur before the two locks; there are four such permutations:
l2(A);l2(B);u2(A);u2(B)
l2(A);l2(B);u2(B);u2(A)
l2(B);l2(A);u2(A);u2(B)
l2(B);l2(A);u2(B);u2(A)

Action r1(A) can be placed in any of 5 positions combining with above lock-unlock actions i.e. before,after, or in between above lock-unlock actions.
Similarly w1(B) can be placed in any of 6 positions combining above lock-unlock-r1(A) i,e, before,after,in-between.
Thus total two phase lock orders are = 4*5*6 = 120.
Thus number of inconsistent but two phase locked orders = 120 - 18 = 102. (iii)

Total number of orders(i+ii+ii+iv) of 6 actions = 6! = 720

Thus number of inconsistent non-two phase locked orders = 720 - 18 - 2 - 102 = 598 (iv)

Thus the answers are
(i) 18
(ii) 2
(iii) 102
(iv) 598

18.3.4(b)

The consistent sequences(i+ii) are interleavings of the sequences
1. l2(A);r2(A);w2(A);u2(A)
2. l2(B);w2(B);u2(B)

The number of such interleaving are (7 choose 3) = 35. Since in 1 r2(A) and w2(A) can be interchanged, the possible consistent sequences are = 35 * 2=70.

A sequence is not two-phase locked if the unlock of 1 occurs before lock of 2 or vice versa. Only 2 such interleavings are possible i.e. all of 1 followed by all of 2 or vice versa. Again since in 1 r2(A) and w2(A) can be interchanged, total number of consistent but non-two-phase locked orders are =2 * 2 = 4 (ii)

Thus the number of two-phase locked orders are = 70 - 4 = 66. (i)

Next consider number of two-phased locked orders(i+iii). The two locks must occur before the two locks; there are four such permutations:
l2(A);l2(B);u2(A);u2(B)
l2(A);l2(B);u2(B);u2(A)
l2(B);l2(A);u2(A);u2(B)
l2(B);l2(A);u2(B);u2(A)

Action r2(A) can be placed in any of 5 positions combining with above lock-unlock actions i.e. before,after, or in between above lock-unlock actions.
Similarly w2(A) can be placed in any of 6 positions combining above lock-unlock-r2(A) i,e, before,after,in-between.
Similarly w2(B) can be placed in any of 7 positions above. Thus total two phase lock orders are = 4*5*6*7 = 840.
Thus number of inconsistent but two phase locked orders = 840 - 66 = 774. (iii)

Total number of orders(i+ii+ii+iv) of 7 actions = 7! = 5040

Thus number of inconsistent non-two phase locked orders = 5040 - 774 - 66 - 4 = 4196 (iv)

Thus the answers are
(i) 66
(ii) 4
(iii) 774
(iv) 4196

18.4.1a)

i)

$sl_1(A)r_1(A)sl_2(B)r_2(B)sl_3(C)r_3(C)xl_1(B)w_1(B)u_1(B)u_1(A)xl_2(C)w_2(C)u_2(C)u_2(B)$
1 2 3 4 5

$xl_3(D)w_3(D)u_3(D)u_3(C)$
6

ii)

T_1 is delayed. 4th lock request $xl_1(B)$ waits since T_2 is holding shared lock on B.
T_2 is delayed. 5^{th} lock request $xl_2(C)$ waits since T_3 is holding shared lock on C.
T_3 gets $xl_3(D)$. Completes and unlocks C.
T_2 then can get $xl_2(C)$. Completes and unlocks B.
T_1 then can get $xl_1(B)$ and completes.

iii)

Same as i)

iv)
Same as ii)

v)
Same as i)

vi)
Same as ii)

18.4.1b)

i)
$sl_1(A)r_1(A)sl_2(B)r_2(B)sl_3(C)r_3(C)xl_1(B)w_1(B)u_1(B)u_1(A)xl_2(C)w_2(C)u_2(C)u_2(B)$
1 2 3 4 5
$xl_3(A)w_3(A)u_3(A)u_3(C)$
6

ii)
T_1 delayed. 4th lock request $xl_1(B)$ waits since T_2 holds shared lock on B.
T_2 delayed. 5th lock request $xl_2(C)$ waits since T_3 holds shared lock on C.
T_3 delayed. 6th lock request $xl_3(A)$ waits since T_1 holds shared lock on A.
Therefore, deadlock condition.

iii)
Same as i)

iv)
Same as ii)

v)
Same as i)

vi)
Same as ii)

18.4.1c)

i)
$sl_1(A)r_1(A)sl_2(B)r_2(B)sl_3(C)r_3(C)sl_1(B)r_1(B)sl_2(C)r_2(C)sl_3(D)r_3(D)xl_1(C)w_1(C)u_1(C)$
1 2 3 4 5 6 7

$u_1(A)u_1(B)xl_2(D)w_2(D)u_2(D)u_2(B)u_2(C)xl_3(E)w_3(E)u_3(E)u_3(C)u_3(D)$
8 9

ii)
T_1 is delayed. 7th lock request $xl_1(C)$ waits since T_3 and T_2 hold shared lock on C.

T_2 is delayed. 8th lock request $xl_2(D)$ waits since T_3 holds shared lock on D.
T_3 gets 9th lock request $xl_3(E)$. Completes. Unlocks D.
T_2 then gets $xl_2(D)$ and completes. Unlocks C and D.
T_1 proceeds to completion.

iii)
Same as i)

iv)
Same as ii)

v)
Same as i)

vi)
Same as ii)

18.4.1d)

i): $xl_1(A); r_1(A); xl_2(B); r_2(B); xl_3(C); r_3(C); sl_1(B); r_1(B); sl_2(C); r_2(C); sl_3(D); r_3(D); w_1(A); u_1(A); u_1(B); w_2(B); u_2(B); u_2(C); w_3(C); u_3(C); u_3(D)$

(ii): The first three locks and reads are allowed. However, T_1 cannot get a shared lock on *B*, nor can T_2 get a shared lock on *C*. When T_3 requests a shared lock on *D* it is granted. Thus, T_3 can proceed and eventually unlocks *C* and *D*.

As soon as *C* is unlocked, T_2 can get its shared lock on *C*. Thus, T_2 completes and releases its locks. As soon as its lock on *B* is released, T_1 can get its lock on *A*, so it completes.

(iii): $sl_1(A); r_1(A); sl_2(B); r_2(B); sl_3(C); r_3(C); sl_1(B); r_1(B); sl_2(C); r_2(C); sl_3(D); r_3(D); xl_1(A); w_1(A); u_1(A); u_1(B); xl_2(B); w_2(B); u_2(B); u_2(C); xl_3(C); w_3(C); u_3(C); u_3(D)$

(iv): First, all six shared-lock requests are granted. When T_1 requests an exclusive lock on *A*, it gets it, because it is the only transaction holding a lock on *A*. T_1 completes and releases its locks. Thus, when T_2 asks for an exclusive lock on *B*, it is the only transaction still holding a shared lock on *B*, so that lock too may be granted, and T_2 completes. After T_2 releases its locks, T_3 is able to upgrade its shared lock on *C* to exclusive, and it too proceeds. The actions are thus executed in exactly the same order as they are requested; i.e., there are no delays by the scheduler.

(v): $ul_1(A); r_1(A); ul_2(B); r_2(B); ul_3(C); r_3(C); sl_1(B); r_1(B); sl_2(C); r_2(C); sl_3(D); r_3(D); xl_1(A); w_1(A); u_1(A); u_1(B); xl_2(B); w_2(B); u_2(B); u_2(C); xl_3(C); w_3(C); u_3(C); u_3(D)$

(vi): The update locks prevent the fourth and fifth shared-lock requests, $sl_1(B)$ and $sl_2(C)$, so T_1 and T_2 are delayed, while T_3 is allowed to proceed. The situation is exactly as for part *(ii)*, where the initial lock requests are for exclusive locks.

18.4.1e)

i)

$xl_1(A)r_1(A)xl_2(B)r_2(B)xl_3(C)r_3(C)sl_1(B)r_1(B)sl_2(C)r_2(C)sl_3(A)r_3(A)w_1(A)u_1(A)u_1(B)w_2(B)$
1 2 3 4 5 6
$u_2(B)u_2(C)w_3(C)u_3(C)u_3(A)$

ii)
T1 is delayed. 4th lock request $sl_1(B)$ waits since T2 has exclusive lock on B.
T2 is delayed. 5th lock request $sl_2(C)$ waits since T3 has exclusive lock on C.
T3 is delayed. 6th lock request $sl_3(A)$ waits since T1 has exclusive lock on A.
Therefore, deadlock.

iii)
$sl_1(A)r_1(A)sl_2(B)r_2(B)sl_3(C)r_3(C)sl_1(B)r_1(B)sl_2(C)r_2(C)sl_3(A)r_3(A)xl_1(A)w_1(A)u_1(A)u_1(B)$
1 2 3 4 5 6 7
$xl_2(B)w_2(B)u_2(B)u_2(C)xl_3(C)w_3(C)u_3(C)u_3(A)$
8 9

iv)
T_1 is delayed. 7th lock request $xl_1(A)$ waits since T_3 has shared lock on A.
T_2 is delayed. 8th lock request $xl_2(B)$ waits since T_1 has shared lock on B.
T_3 is delayed. 9th lock request $xl_3(C)$ waits since T_2 has shared lock on C.
Therefore, deadlock.

v)
$ul_1(A)r_1(A)ul_2(B)r_2(B)ul_3(C)r_3(C)sl_1(B)r_1(B)sl_2(C)r_2(C)sl_3(A)r_3(A)xl_1(A)w_1(A)u_1(A)$
1 2 3 4 5 6 7
$u_1(B)xl_2(B)w_2(B)u_2(B)u_2(C)xl_3(C)w_3(C)u_3(C)u_3(A)$
8 9

vi)
T1 is delayed. 4th lock request $sl_1(B)$ waits since T2 has update lock on B.
T2 is delayed. 5th lock request $sl_2(C)$ waits since T3 has update lock on C.
T3 is delayed. 6th lock request $sl_3(A)$ waits since T1 has update lock on A.
Therefore, deadlock.

18.4.2a)
First, let us count the number of serializable orders that are conflict-equivalent to the order (T_1, T_2). By symmetry, there will be the same number equivalent to the opposite

order, (T_2, T_1). In order for a schedule to be equivalent to (T_1, T_2), *$inc_1(A)$* must precede *$r_2(A)$*. Thus, the first three steps of T_1 must be first in the schedule.

Also, *$inc_1(B)$* must come before *$r_2(B)$*, so the former can either be the fourth action of the schedule, or it can be fifth, coming after *$r_2(A)$*. Thus, there are two serializable schedules equivalent to (T_1, T_2), and four serializable schedules altogether.

18.4.2b)

Possible serializable schedules are
(i) serial order (T1,T2)
(ii)serial order (T2,T1)
(iii) Serializable order equivalent to serial order (T1,T2) obtained by non-conflicting swaps.
Consider serial order (T2,T1). For equivalent serializable schedule, the only non-conflicting swap allowed is between inc1(B) and r2(A).
(iv) Serializable order equivalent to serial order (T2,T1) obtained by non-conflicting swaps.
For serial order (T2,T1), no non-conflicting swaps are possible since the last action of T2 i.e. inc2(A) and first action of T1 i.e. r1(A) are in conflict.

Thus total only three possible interleavings exist.

18.4.3a)

$sl_1(A)r_1(A)sl_2(B)r_2(B)il_1(B)inc_1(B)il_2(C)inc_2(C)xl_1(C)w_1(C)u_1(A)u_1(B)u_1(C)xl_2(D)$
1 2 3 4 5 6
$w_2(D)u_2(B)u_2(C)u_2(D)$

T_1 is delayed. 3rd lock request $il_1(B)$ waits since T_2 has shared lock on B.
T_2 gets $inc_2(C)$ and $)xl_2(D)$. T_2 completes and unlocks B,C and D.
T_1 then continues and gets $il_1(B)$ and $xl_1(C)$ and completes.

18.4.3b)

$sl_1(A)r_1(A)sl_2(B)r_2(B)il_1(B)inc_1(B)il_2(A)inc_2(A)xl_1(C)w_1(C)u_1(C)u_1(A)u_1(B)xl_2(D)w_2(D)$
1 2 3 4 5 6
$u_2(D)u_2(B)u_2(A)$

T_1 is delayed. 3rd lock request $il_1(B)$ waits since T_2 has shared lock on B.
T_2 is delayed. 4th lock request $il_2(A)$ waits since T_1 has shared lock on A.
Therefore, deadlock.

18.4.3c)

$il_1(A)inc_1(A)il_2(B)inc_2(B)il_1(B)inc_1(B)il_2(C)inc_2(C)xl_1(C)w_1(C)u_1(C)u_1(B)u_1(A)xl_2(D)$

1 2 3 4 5 6
$w_2(D)u_2(D)u_2(B)u_2(C)$

$il_1(B)$ does not conflict with $il_2(B)$.
T_1 is delayed. 5th lock request $xl_1(C)$ waits since T_2 holds increment lock on C.
T_2 completes and unlocks C.
T_1 then gets $xl_1(C)$ and completes.

18.4.4
Schedule is $r_1(A)r_1(B)w_1(B)r_1(C)w_1(C)r_1(D)w_1(D)r_1(E)w_1(E)$
With locks:
$Ul_1(A)r_1(A)ul_1(B)r_1(B)xl_1(B)w_1(B)ul_1(C)r_1(C)xl_1(C)w_1(C)ul_1(D)r_1(D)xl_1(D)w_1(D)ul_1(E)r_1(E)xl_1(E)w_1(E)$

(Quest: shared or update lock better for concurrency?)

18.4.5a)

Held \ Requested	**S**	**X**	**M**
S	Yes	No	No
X	No	No	No
M	No	No	Yes

18.4.5b)

Held \ Requested	**S**	**X**	**I**	**M**
S	Yes	No	No	No
X	No	No	No	No
I	No	No	Yes	No
M	No	No	No	Yes

18.4.6a)
The only actions that commute are (i)-(ii) and (iii)-(iv). That is, it doesn't matter whether we first set the x-axis and then the y-axis, or vice-versa. Likewise, it doesn't matter whether we first set the angle and then the magnitude or vice-versa. However, all other pairs of actions matter.

For example, consider the actions (set angle = 90)(set x = 5) applied to the vector (10,0). The first step gives us (0,10), and the second step gives us (5,10).

On the other hand, the reverse order of the actions: (set x = 5)(set angle = 90) starting with (10,0) gives us (0,5), and the second step gives us (5,5).

18.4.6b)

Held \ Requested	X	Y	A	M
X	No	Yes	No	No
Y	Yes	No	No	No
A	No	No	No	Yes
M	No	No	Yes	No

18.4.6c)

Held \ Requested	X	Y	A	M
X	Yes	Yes	No	No
Y	Yes	Yes	No	No
A	No	No	Yes	Yes
M	No	No	Yes	Yes

18.4.7a)

The only constraint on a serial order so far is that T_1 must precede T_2, because T_1 reads *A* before T_2 writes *A*. The only way we could get a cycle in the precedence graph is if ??? required T_2 to precede T_1. A read of anything by T_1 will not introduce any constraints, but $r_2(C)$ will cause an arc from T_2 to T_1 and lead to a cycle.

18.4.7b)

$w_1(B)$ will require T_2 to precede T_1. Given that T_1 is required to precede T_1 since T_1 reads A before T_2 writes A, a cycle is introduced in precedence graph.

18.4.7c)

$r_1(C)$ or $r_2(A)$ will cause update locks.
Both of these actions do not lead to a cycle. So, no schedule is non-serializable.

18.4.7d)

$inc_2(C)$ will conflict with $w_1(C)$. It will require T_2 to precede T_1 leading to a cycle.
Also, $incr_1(B)$ wil conflict with $r_2(B)$ requiring T_2 to precede T_1, thus causing a cycle.

18.5.1a)

Based on the compatibility matrix of shared and exclusive locks, the only possible sets of locks on a single element are either:

i) One exclusive lock
ii) One or more shared locks

Thus, the only needed lock modes are exclusive and shared, which correctly sum up the limitations on access to a given element in the above two situations, respectively.

18.5.1b)

Based on the compatibility matrix of shared, exclusive and increment locks, we see that the only possible sets of locks on a single element are either

i) One exclusive lock.
ii) One or more shared locks.
iii) One or more increment locks.

Thus, the only needed lock modes are exclusive, shared, and increment, which correctly sum up the limitations on access to a given element in the above three situations, respectively.

18.5.1c)

Based on the compatibility matrix drawn up in 18.4.6b, the only possible sets of locks on a single element are either:

i) One X-Lock and one Y-lock
ii) One A-lock and one M-lock
iii) One X-lock
iv) One Y-lock
v) One M lock
vi) One A lock

Each of the above six situations represent a lock mode.

18.5.2a)

$r_1(A)r_2(A)r_3(B)w_1(A)r_2(C)r_2(B)w_2(B)w_1(C)$

The lock scheduler described in this section inserts shared-exclusive-update locks. Thus the sequence of lock requests and actions for the above schedule is:

$u_1(A)r_1(A)sl_2(A)r_2(A)sl_3(B)r_3B)xl_1(A)w_1(A)sl_2(C)r_2(C)ul_2(B)r_2(B)xl_2(B)w_2(B)xl_1(C)w_1(C)$

This is how the lock scheduler would execute:

- T_1 issues $r_1(A)$. Part1 of scheduler inserts ul1(A) since there is a future upgrade to exclusive lock needed. Part 2 of scheduler consults the lock table and grants $ul_1(A)$ since there is no other lock on A.

- Next T_2 issues $r_2(A)$. Part 1 inserts $sl_2(A)$ since there is no future upgrading to exclusive lock. Part 2 consults lock table and finds that it cannot grant $sl_2(A)$ since T_1 holds an update lock to A. So, this and the subsequent actions from T_2 are delayed.
- T_3 then issues $r_3(B)$. Part 1 insert $sl_3(B)$. Part 2 grants the shared lock on B to T_3. T_3 completes action and commits. Part1 then releases lock on B.
- T_1 issues $w_1(A)$. Part1 inserts an exclusive lock $xl_1(A)$. Part 2 upgrades the lock to exclusive.
- T_1 issues $w_1(C)$. Part 1 issues $xl_1(C)$. Part 2 grants the lock. T_1 commits and Part1 releases locks on A and C.
- Its then found out that T is waiting for lock on A. Part 2 is notified and it grants $sl_2(A)$. T_2 then completes.

18.5.2b)

$r_1(A)w_1(B)r_2(B)w_2(C)r_3(C)w_3(A)$
The sequence of lock requests in the schedule (assuming no delays) would be:
$Sl_1(A)r_1(A)xl_1(B)w_1(B)sl_2(B)r_2(B)xl_2(C)w_2(C)sl_3(C)r_3(C)xl_3(A)w_3(A)$
The lock scheduler would execute as follows:

- T_1 issues $r_1(A)$. Part 1 inserts $sl_1(A)$ since no future upgrades are required. Part 2 grants $sl_1(A)$ after consulting the lock table.
- T_1 issues $w_1(B)$. Part1 inserts $xl_1(B)$. Part 2 grants $xl_1(B)$ and updates the lock table.
- T_1 commits and Part 1 then unlocks A and B.
- T_2 issues $r_2(B)$. Part 1 inserts $sl_2(B)$. Part 2 grants.
- T_2 issues $w_2(C)$. Part1 inserts $xl_2(C)$. Part 2 grants.
- T_2 commits. Part 1 releases locks on B and C.
- Similarly, T_3 completes. So this schedule completes without any delays.

18.5.2c)

$w_3(A)r_1(A)w_1(B)r_2(B)w_2(C)r_3(C)$
Here we show the sequence of lock requests inserted by Part 1 and granted by Part 2 after consulting the lock table. We do not provide detailed explanation since there are no delays in this schedule.
$xl_3(A)w_3(A)sl_1(A)r_1(A)xl_1(B)w_1(B)T_1$ commits. $ul_1(A)ul_1(B)sl_2(B)r_2(B)sl_2(C)w_2(C)$. T_2 commits. $ul_2(C)$. $ul_2(B)sl_3(C)r_3(C)$. T_3 commits. $Ul_3(C)$.

18.5.2d)

$r_1(A)r_2(A)w_1(B)w_2(B)r_1(B)r_2(B)w_2(C)w_1(D)$
Here we show the sequence of lock requests (assuming no delays):
$sl_1(A)r_1(A)sl_2(A)r_2(A)xl_1(B)w_1(B)xl_2(B)w_2(B)r_1(B)sl_2(B)r_2(B)xl_2(C)w_2(C)ul_2(A)ul_2(B)ul_2(C)xl_1(D)w_1(D)ul_1(A)ul_1(B)ul_1(D)$
Here is how the lock scheduler would execute:

- T_1 issues $r_1(A)$. Part 1 issues and Part 2 grants $sl_1(A)$. Lock table is updated.

- T_2 issues $r_2(A)$. Part 1 inserts $sl_2(A)$. Part 2 consults the lock table and it sees that the only lock held on A is a shared lock and thus it grants $sl_2(A)$.
- T_1 issues $w_1(B)$. Part1 inserts $xl_1(B)$ and Part 2 issues it since there are no locks on B.
- T_2 issues $w_2(B)$. Part 2 issues $xl_2(B)$. Part 2 consults the lock table for B and finds out that T_1 already holds an exclusive lock on B. T_2 is thus delayed.
- T_1 then issues $r_1(B)$. Since T_1 already holds and exclusive lock on B, no new lock request is inserted and T_1 performs $r_1(B)$.
- T_1 issues $w_1(D)$. Part 1 inserts ad Part 2 grants $xl_1(D)$.
- T_1 then commits and Part 1 releases locks on A, B and D.
- Part 2 is then notified that T_2 is waiting on $xl_2(B)$. It grants $xl_2(B)$.
- T_2 then completes.

18.5.2e)

$r_1(A)r_2(A)r_1(B)r_2(B)r_3(A)r_4(B)w_1(A)w_2(B)$

Here we show the sequence of lock requests (assuming no delays):

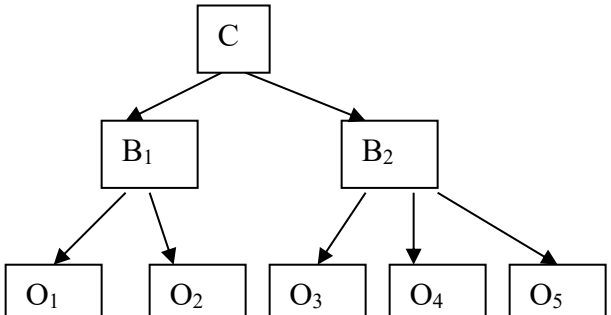
$ul_1(A)r_1(A)sl_2(A)r_2(A)sl_1(B)r_1(B)ul_2(B)r_2(B)sl_3(A)r_3(A)sl_4(B)r_4(B)xl_1(A)w_1(A)ul_1(A)ul_1(B)$

Here is how the lock scheduler would execute:

- T_1 issues $r_1(A)$. Part 1 inserts $ul_1(A)$. Part 2 grants $ul_1(A)$.
- T_2 issues $r_2(S)$. Part 1 inserts $sl_2(A)$. Part 2 consults the lock table and denies $sl_2(A)$ since lock table shows T_1 holds an update lock to A. The actions of T_2 are thus delayed.
- T_1 issues $r_1(B)$. Part 2 issues $sl_1(B)$.
- T_3 issues $r_3(A)$. Part 1 inserts $sl_3(A)$. Part 2 consults the lock table and denies $sl_3(A)$ since T_1 still holds an update lock to A. T_3 is thus delayed.
- T_4 issues $r_4(B)$. Part 1 inserts $sl_4(B)$. Part 2 consults the lock table for B and denies $sl_4(B)$ since T_1 holds $sl_1(B)$. T_4 is thus delayed.
- T_1 issues $w_1(A)$. Part 2 upgrades T_1's lock on A to $xl_1(A)$.
- T_1 commits and Part 1 releases locks on A and B.
- Part 2 is notified that T_2 is waiting on $Sl_2(A)$. Part 2 grants $sl_2(A)$.
- T_2 issues $r_2(B)$. Part 1 issues and Part 2 grants $ul_2(B)$.
- T_3 and T_4 still wait.
- T_2 issues $w_2(B)$. Part 2 upgrades T_2's lock to $xl_2(B)$.
- T_2 then completes and Part 1 releases locks on A and B.
- T_3 and T_4, which are waiting on A and B respectively, get the locks and complete.

18.6.1a)

Refer to the following hierarchy for all four sub-parts:



i) T_1 puts IS lock on C and B_1, and S lock on O_1
ii) T_2 puts IX lock on C, IX lock on B_1 and X lock on O_2
iii) T_2 puts IS lock on C, B_2 and S lock on O_3. T_2 releases its locks.
iv) T_1 puts IX lock on C, B_2 and S lock on O_4. T_1 releases its locks.

18.6.1b)

i) T1 puts IS lock on C, B2 and S lock on O5.
ii) T2 puts IX lock on C, B2. It attempts X lock on O5, which is rejected since T1 already holds an S lock on O5. T2 is delayed.
iii) T1 puts IX lock on C, B2 and X lock on O4. T1 releases all locks
iv) T2 then puts X lock on O5.
v) T2 puts IS lock on C, B2 and S lock on O3. T2 releases all locks.

18.6.1c)

i) T_1 puts IS lock on C, B_1 and S lock on O_1
ii) T_1 puts IS lock on C, B_2 and S lock on O_3. T_1 releases locks.
iii) T_2 puts IS lock on C, B_1 and S lock on O_1
iv) T_2 puts IX lock on C, B_2 and X lock on O_4
v) T_2 puts IX lock on C, B_2 and X lock on O_5. T_2 releases locks.

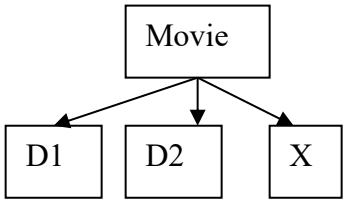
18.6.2d)

i) T_1 puts IS lock on C, B_1 and S lock on O_1
ii) T_2 puts IS lock on C, B_1 and S lock on O_2
iii) T_3 puts IS lock on C, B_1 and S lock on O_1
iv) T_1 puts IX lock on C, B_2 and X lock on O_3
v) T_2 puts IX lock on C, B_2 and X lock on O_4. T_2 releases locks.
vi) T_3 puts IX lock on C, B_2 and X lock on O_5. T_3 releases locks.
vii) T_1 puts IX lock on C, B_1 and X lock on O_2. T_1 releases locks.

18.6.2

By replacing w4(D3) by w4(Movie), the schedule becomes:

R3(D1)r3(D2)w4(Movie)w4(X)w3(L)w3(X)
The hierarchy of elements:



The sequence of actions of warning protocol scheduler are:

i) T3 puts IS lock on Movie and S lock on D1.
ii) T3 puts IS lock on Movie and S lock on D2.
iii) T4 attempts X lock on Movie, but is rejected since X-lock and IS lock are not compatible. T4 is delayed.
iv) T3 computes L=sum of lengths of D1 and D2.
v) T3 puts IX lock on Movie and X lock on X. T3 releases locks.
vi) T4 then gets X lock on Movie. Updates movie. Then gets X lock on X and updates X. T4 releases locks.

The sequence of actions is equivalent to serial order (T3,T4).

18.6.3

With increment locks, the steps of the warning protocol scheduler stay the same, except the compatibility matrix is updated to include II (intention increment) and I (increment) locks as following:

	IS	**IX**	**II**	**S**	**X**	**I**
IS	Y	Y	Y	Y	N	N
IX	Y	Y	Y	N	N	N
II	Y	Y	Y	N	N	Y
S	Y	N	N	Y	N	N
X	N	N	N	N	N	N
I	N	N	Y	N	N	Y

18.7.1a)

As soon as we reach the left child of the root, we see that node is not full. Thus, the insert of 10 cannot cause that node to split, and there will be no reason to rewrite the root. We can now release the exclusive lock on the root.

We are then directed to the second leaf. Since that leaf is not full either, we know the insertion will not cause the left child of the root to be changed, so we can release the exclusive lock on that child. As soon as we have rewritten the second leaf to reflect the insertion of 10, we can release the lock on the leaf.

18.7.1b)

Lookup of 20 leads to right child of root node. Since right child is full, we might need to rewrite root, so do not release exclusive lock on root. Now, 20<23, so get directed to third leaf, which is full. Therefore, it needs to be spilt. We also do not release the exclusive lock on right child of root since the split of leaf will cause the right child to be updated. First the exclusive lock on third leaf gets released after rewrite, followed by right child and followed by root node.

18.7.1c)

For delete of 5, lookup of 5 leads to left child of root node. We then get directed to first leaf since 5<7. We see that delete of 5 would not cause unbalanced leaf. So we can release the exclusive lock on left child and root node before doing the delete operation.

18.7.1d)

Lookup of 23 leads us to the right child of root node. We can release the exclusive lock on root node since delete of 23 would not cause root to be updated. We follow the second pointer in the right child and reach the fourth leaf. Delete of 23 would cause the fourth leaf to have only 1 key, which would cause 19 to be moved from third to fourth leaf. We thus hold on to the exclusive lock on second child of root until after updating the leafs since the second child needs to be updated as well.

18.7.2a)

Note: we shall assume only a single type of lock, so that even though the operations are all reads (and therefore, any interleaving could in principle be considered serializable), it is not possible for two transactions to access the same element until one has relinquished its lock.

Suppose that T_1 locks *A* first. Then T_2 cannot perform any steps until T_1 relinquishes the lock. Therefore, T_1 must also read *B*, at the second step of the schedule. Now, T_1 can relinquish the lock on *A*. The remaining step of T_1, $r_1(E)$, can occur either before all of T_2, or after the first or second step of T_2. It cannot occur after the last step of T_2, $r_2(B)$, because T_1 can't relinquish its lock on *B*, until it has locked *E*. Thus, there are three interleavings in which T_1 starts first.

Now, consider what happens if T_2 starts first. T_2 cannot relinquish its lock on *A* until it has locked both children *B* and *C*. Thus, if T_2 starts first, it must finish before T_1 starts. We conclude that there are four legal schedules.

18.7.2b)

T1 can lock A anytime. Next T1 can get a lock on B if T3 didn't lock B yet. If T1 gets a lock on B first it will get a lock on E first also. Now T1 can release a lock on B and T3 can get it but T3 also need to wait to get a lock on E until T1 releases it. After getting a lock on E, T3 can lock F. Thus, if T1 locks B first, T3 can start after T1 gets a lock and E and release a lock on B and T3 can finish all steps after after T1 finishes all steps. The same situation occurs when T3 locks B first. Therefore, there are two interleavings.

1) T1 -> T3:

$r_1(A)$->$r_1(B)$->$r_1(E)$->
$r_3(B)$->$r_3(E)$->$r_3(F)$,

If T1 reads B first, T2 needs to wait to get a lock on B until T1 gets a lock on E and releases a lock on B.

2) T3 -> T1:

$r_1(A)$-> $r_1(B)$->$r_1(E)$
$r_3(B)$->$r_3(E)$->$r_3(F)$

If T2 locks B first, T1 needs to get a lock on B until T2 gets a lock on E and releases a lock on B. *$r_1(A)$* can start in anytime either before all steps, between steps, or after all steps (4 cases) and $r_3(F)$ can start either before or after $r_1(B)$ but not after $r_1(E)$ (2 cases). Therefore, all (3 cases $r_1(A)$ * 2 cases of $r_1(E)$) + 1 case of $r_1(A)$ starts after $r_3(F)$ = 7 interleavings can occur.

Thus, all 8 possible interleavings here.

18.7.2c)
1) T1 – T2 – T3 : … The sequence of Ts get locks on a common element.

$r_1(A)$->$r_1(B)$->$r_1(E)$
$r_2(A)$->$r_2(C)$-> $r_2(B)$
$r_3(B)$->$r_3(E)$->$r_3(F)$

If T3 gets a lock on B first, T3 can start either before or after $r_2(C)$(2 cases) and $r_2(B)$ can start before or after $r_3(F)$(2 cases) (2 * 2 = 4 cases). Thus, there are all 5 interleavings here.

2)T1 – T3 – T2 :

$r_1(A)$->$r_1(B)$->$r_1(E)$
$r_2(A)$->$r_2(C)$-> $r_2(B)$
$r_3(B)$->$r_3(E)$->$r_3(F)$

If T2 gets a lock on B first, T3 needs wait until T2 finishes all steps.
Thus, there is only one interleaving here.

3) T2 – T1 – T3 :

$r_1(A)$->$r_1(B)$->$r_1(E)$

$r_2(A)$->$r_2(C)$->$r_2(B)$
$r_3(B)$->$r_3(E)$->$r_3(F)$

There's only one interleaving.

4) T2 – T3 – T1 :

$r_1(A)$-> $r_1(B)$->$r_1(E)$
$r_2(A)$->$r_2(C)$->$r_2(B)$
$r_3(B)$->$r_3(E)$->$r_3(F)$

$r_1(A)$ can start before $r_2(B)$, after $r_3(F)$, or between those two (5 cases). $r_1(B)$can start before or after $r_3(F)$(2 cases). (4 cases of $r_1(A)$ * 2 cases of $r_1(B)$) + 1 cases that $r_1(A)$ starts after $r_3(F)$. Thus, 9 interleavings here.

5) T3 – T1 – T2 :

$r_1(A)$-> $r_1(B)$->$r_1(E)$
$r_2(A)$->$r_2(C)$->$r_2(B)$
$r_3(B)$->$r_3(E)$->$r_3(F)$

$r_1(A)$ can start anytime before, between, or after all steps of T3 (4 cases) and $r_3(F)$ can start before or after $r_1(B)$ but not after $r_1(E)$ (2 cases). $r_1(E)$ can start anytime in T2 but not after $r_2(B)$ (3 cases). Thus, there are 24 interleavings from this sequence.

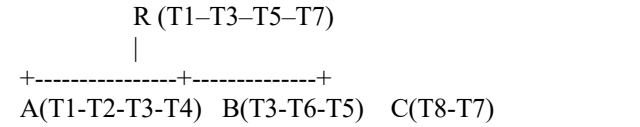
6) T3 – T2 – T1:

$r_1(A)$->$r_1(B)$->$r_1(E)$
$r_2(A)$->$r_2(C)$->$r_2(B)$
$r_3(B)$->$r_3(E)$->$r_3(F)$

$r_2(A)$ can start before, after, or between all steps of T3 (4 cases). So can $r_2(C)$. But $r_2(B)$ can start only before or after $r_3(F)$. $r_1(A)$ can only start before or after $r_2(B)$. (4 * 4 – 1(the case both $r_2(A)$ and $r_2(C)$starts after $r_3(F)$ * 2) = 30 interleavings.

Therefore, all possible interleavings are
5 + 1 + 1 + 9 + 16 + 24 + 30 = 86

18.7.3



T1 T2 T3 T4 T5 T6 T7 T8

```
l1(R); r1(R);
l8(C); r8(C);
l1(A); r1(A);
u8(C);
u1(R);                          l3(R); r3(R);
u1(A)          l2(A); r2(A);    l3(B); r3(B);
               u2(A);           l3(A); r3(A);
                                u3(B); u3(R);                  l5(R); r5(R);   l6(B); r6(B);
                                u3(A);          l4(A); r4(A);                  u6(B);
                                                u4(A);          l5(B); r5(B);
                                                               u5(R); u5(B);                  l7(R); r7(R);
                                                                                              l7(C); r7(C);
                                                                                              u7(R); u7(C);
```

T1 < T2 < T3 < T4
T3 < T6 < T5 < T7
T8 < T7

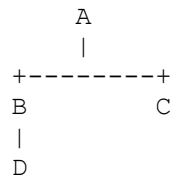
T4 and T8 are not dependent on any other transaction after T3 and before T7 respectively. 4 possible serial order based on T4's position (before T6, after T7, or between those) * 6 possible serial order based on T8's position (before T1, after T5, or between those) = 24 serial orders are consistent with the statement.

18.7.4

Let's consider the following serializable transaction.

S: w1(C); r1(B); r2(B); w2(D)

Suppose we have the following tree.



Based on Rule(2) of the tree protocol, each transaction needs to get a lock on a parent node to get a shared or exclusive lock of the subsequent node. It can lead S', which is not serializable, when T2 gets a exclusive lock to get a exclusive lock on D before T1 whether T1 starts r2(B) first or w2(B) first.

S': w1(A); w1(C); w2(B); r2(B); w2(D); r1(B)

Change Rule (2) to ""Subsequent shared or exclusive lock may be acquired if the transaction has a **shared lock** on its parent node" can prevent unserializable behavior and it leads S'' based on the changed Rule(2).

S'': r1(A); w1(C); r2(B); w2(D); r1(B),

which can be converted into

S'': r1(A); w1(C); r1(B); r2(B); w2(D);

18.8.1a)

T_1	**T_2**	**A**	**B**
		RT = 0 WT=0	RT=0 WT=0
st_1 Assume TS(T_1)=1			
	st_2 Assume TS(T_2)=2		
r_1(A)		RT=1	
	r_2(B)		RT=2
	w_2(A)	WT=2	
w1(B) **T_1 aborts since TS(t1)<RT(B)**			

18.8.1b)

T_1	**T_2**	**A**	**B**
		RT=0 WT=0	RT=0 WT=0
st_1 Assume TS(T_1)=1			
r_1(A)		RT=1	
	st_2 Assume TS(T_2)=2		
	w_2(B)		WT=2
	r_2(A)	RT=2	
w_1(B) TS(T_1)<WT(B) i.e. later trans has already written B. So w_1(B) ignored.			

18.8.1c)

T_1	**T_2**	**T_3**	**A**	**B**	**C**

			RT=0 WT=0	RT=0 WT=0	RT=0 WT=0
st_1. Assume $TS(T_1) = 1$					
	st_2.Assume $TS(T_2)=2$				
		st_3. Assume $TS(T_3)=3$			
$r_1(A)$			RT=1		
	$r_2(B)$			RT=2	
$w_1(C)$					WT=1
		$r_3(B)$		RT=3	
		$r_3(C)$			RT=3
	$w_2(B)$ $TS(T_2)<RT(B)$ **T2 aborts**.				
		$w_3(A)$	WT=3		

18.8.1d)

T1	**T2**	**T3**	**A**	**B**	**C**
			RT=0 WT=0	RT=0 WT=0	RT=0 WT=0
st_1.Assume $TS(T_1)=1$					
		st_3. Assume $TS(T_3)=2$			
	st_2. Assume $TS(T_2)=3$				
$r_1(A)$			RT=1		
	$r_2(B)$			RT=3	
$w_1(C)$					WT=1
		$r_3(B)$ No change in RT(B) since $RT(B)>TS(T_3)$		RT=3	
		$r_3(C)$			RT=2
	$w_2(B)$ Since $TS(T_2)=RT(B)$, unchanged			WT=3	
		$W_3(A)$	WT=2		

18.8.2a)

The three writes create three versions of *A*. When T_2 tries to rad *A*, it is given the value that it itself wrote, since that is the version with the greatest timestamp that does not

exceed the timestamp of T_2. That makes sense, although in practice, we doubt that a well written transaction would read its own value through the database storage system.

When T_4 tries to read *A* the system finds that T_4's timestamp is larger than that of any version of *A* written. Thus, T_4 gets the version with the largest of the timestamps, the one written by T_3. That makes sense, because in the hypothetical serial order based on the timestamps of the transactions, T_3 would be the last to write *A*.

18.8.2b)

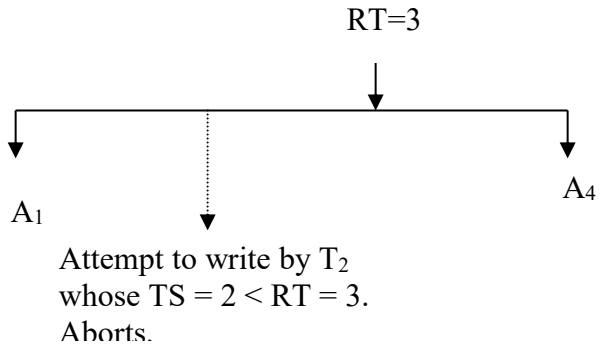
T_1	**T_2**	**T_3**	**T_4**	**A_1**	**A_3**
st1. Assume $TS(T_1)=1$	st_2. Assume $TS(T_2)=2$	st3. Assume $TS(T3)=3$	st4. Assume $TS(T4)=4$		
$w_1(A)$				Create	
		$w_3(A)$			Create
			$r_4(A)$		Read
	$r_2(A)$			Read	

Without multi-version timestamp scheduler, T_2 would have aborted on issuing $r_2(A)$ since $TS(T_2)$ [2]would be less than write time of A [3].

18.8.2c)

T_1	**T_2**	**T_3**	**T_4**	**A_1**	**A_4**
st1. Assume $TS(T_1)=1$	st2. Assume $TS(T_2)=2$	st3. Assume $TS(T_3)=3$	st4. Assume $TS(T_4)=4$		
$w_1(A)$				Create	
			$w_4(A)$		Create
		$r_3(A)$		Read	
	$w_2(A)$ **T2 aborts.**				

On w2(A), following occurs:



Without multi-version timestamp scheduler, T_3 would have aborted on $r_3(A)$ since A would have already been written by a later transaction T_4. Also, $w_2(A)$ would have been ignored since T_4 with a later timestamp would have written A.

18.8.3

When a younger transaction T2 requests an item X held by an older transaction T1, T2 waits until the commit bit, C(X), becomes true. T1 never waits for T2. Wait-die strategy is for T2, where the scheduler let T2 be aborted and started again when T2 waits too long.

When an older transaction T1 requests an item X held by a younger transaction T2, T1 waits for T2. T2 never waits for T1. Wound-wait strategy is for T1, where the scheduler preempts T2 by aborting it.

In either situation, no cycle is introduced between T1 and T2. Thus, no deadlock situation occurs.

18.9.1a)

As T_1 is the first to validate, there is nothing to check; T_1 validates successfully.

T_3 validates next. The only other validated transaction is T_1, and T_1 has not yet finished. Thus, both the read- and write-sets of T_3 must be compared with the write-set of T_1. However, T_1 writes only *A*, and T_3 neither reads nor writes *A*, so T_3's validation succeeds.

Last, T_2 validates. Both T_1 and T_3 finish after T_2 started, so we must compare the read-set of T_2 with the write-sets of both T_1 and T_3. Since *B* is in both W_3 and R_2, we cannot validate T_2. Note that since T_3 (but not T_1) finishes after T_2 validates, we would also compare the write set of T_2 with the write set of T_3, had we not already found a reason not to validate T_2.

18.9.1b)

The sequence of validation steps are explained below:

- v_1
  - No transaction has validated yet. So, T_1 validates.
- v_3 (T_1 has already validated)
  - $FIN(T_1)>START(T_3)$ since when T_3 started, T_1 has not finished. $WS(T_1)\cap RS(T_3)$. (A) $\cap$(C,D)=Ø.
  - $FIN(T_1)>VAL(T_3)$. $WS(T_1) \cap WS(T_3)$. (A) $\cap$(D)= Ø. T_3 validates.
- v_2 (T_1 and T_3 have validated)
  - $FIN(T_1)>START(T_2).RS(T_2) \cap WS(T_1).(B,C) \cap A=$ Ø.
  - $FIN(T_3)>START(T_2).RS(T_2) \cap WS(T_3).(B,C) \cap D=$ Ø.
  - $FIN(T_3)>VAL(T_2).WS(T_2) \cap WS(T_3).A \cap D=$ Ø. T_2 validates.

18.9.1c)

- v_1
  - No transaction validated yet. So T_1 validates.

- v_3 (T_1 has validated)
    - $FIN(T_1)>START(T_3).RS(T_3) \cap WS(T_1).(C,D) \cap (C)=C$. T_3 rolled back.
- v_2 (T_1 has validated)
    - $FIN(T_1)>START(T_2).RS(T_2) \cap WS(T_1).(B,C) \cap (C)=C$. T_2 also rolled back.

18.9.1.d)

- v_1
    - No transaction validated yet. So T_1 validates.
- v_2 (T_1 already validated)
    - $FIN(T_1)>START(T_2).RS(T_2) \cap WS(T_1).(B,C) \cap A= \emptyset$.
    - $FIN(T_1)>VAL(T_2).WS(T_1) \cap WS(T_2).A \cap B= \emptyset$. T_2 validates.
- v_3 (T_1 and T_2 have validated)
    - $FIN(T_1)>START(T_3).RS(T_3) \cap WS(T_1).C \cap A= \emptyset$.
    - $FIN(T_1)>VAL(T_3).WS(T_3) \cap WS(T_1).A \cap C= \emptyset$.
    - $FIN(T_2)>START(T_3).RS(T_3) \cap WS(T_2).C \cap B= \emptyset$.
    - $FIN(T_2)>VAL(T_3).WS(T_3) \cap WS(T_2).B \cap C= \emptyset$. T_3 validates.

18.9.1e)

- v1
    - No transaction validated yet. T_1 validates.
- v_2 (T_1 validated)
    - $FIN(T_1)>START(T_2).RS(T_2) \cap WS(T_1).(B,C) \cap (C)=C$.$T_2$ rolled back.
- v_3 (T_1 validated)
    - $FIN(T_1)>START(T_3).RS(T_3) \cap WS(T_1).(C) \cap (C)=C$. T_3 rolled back.

18.9.1f)

- v_1
    - No transaction validated yet. T_1 validates.
- v_2 (T_1 is validated)
    - $FIN(T_1)>START(T_2).RS(T_2) \cap WS(T_1).(B,C) \cap (A)= \emptyset$.
    - $FIN(T_1)>VAL(T_2).WS(T_2) \cap WS(T_1).A \cap C= \emptyset$. T_2 validates.
- V3 (T_1 and T_2 validated)
    - $FIN(T_1)>START(T_3).RS(T_3) \cap WS(T_1).C \cap A= \emptyset$.
    - $FIN(T_1)>VAL(T_3).WS(T_3) \cap WS(T_1).B \cap A= \emptyset$.
    - $FIN(T_2)>START(T_3).RS(T_3) \cap WS(T_2).(C) \cap (C)=C$. T_3 rolled back.