Database System Implementation

Solutions for Chapter 9

Return to Top

Solutions for Section 9.1

Exercise 9.1.1

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).

Exercise 9.1.2

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.

Return to Top

Solutions for Section 9.2

Exercise 9.2.1(d)

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.

Exercise 9.2.2(a)

If a swap of two adjacent actions is legal, then the two actions can alos 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.

Exercise 9.2.4(a)

i. The precedence graph is

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

ii. The schedule is thus conflict-serializable.

iii. The only conflict-equivalent serial schedule is *(T_3, T_2, T_1)*.

Return to Top

Solutions for Section 9.3

Exercise 9.3.1(a)

One example of a prohibited schedule is *r_1(A); r_2(B); w_2(B); r_2(A); w_1(A); r_1(B); w_1(B); w_2(A)*. The problem is that at the fourth step, both transactions must hold a lock on *A*.

Exercise 9.3.2

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.

Exercise 9.3.4(a)

First, of the 24 orders of the four lock and unlock actions, only four are two-phase locked. These are

obtained by ordering the two locks in either order and then the two unlocks in either order.

We must then consider how the read and write actions interleave with the locks and unlocks. To be consistent, r_1(A) must appear after l_1(A) and before u_1(A), while w_1(B) appears between l_1 (B) and u_1(B).

Exercise 9.3.3(a)

Note that this question refers to the solved Exercise 9.2.4(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)*.

Exercise 9.3.4(a)

First, let us count the number of consistent orders. To be consistent, the lock and unlock must surround the read or write. Thus, the only consistent sequences are the interleavings of the two sequences

1. *l_1(A); r_1(A); u_1(A)*

2. *l_1(B); w_1(B); u_1(B)*

The number of these interleavings is (6 choose 3) = 20. Note that 20 is thus the sum of the first two quantities we need to compute; *(i) + (ii)*.

Now, let us count *(ii)*, the number of consistent, non-two-phase-locked sequences. The only way a consistent sequence is not 2PL is if the unlock at the end of subsequence (1) or (2) above precedes the lock at the beginning of the other subsequence. There are only 2 ways to interleave (1) and (2) so that all of one goes in front of all of the other (put either subsequence first). Thus, *(ii)* = 2, so *(i)* = 20 - 2 = 18.

Next, let us count the number of two-phase-locked sequences, which is *(i) + (iii)*. If the sequence is 2PL, then the two locks precede the two unlocks. We can thus pick one of two orders for the locks and one of two orders for the unlocks, for a total of 4 orders of the lock and unlock actions. The *r_1 (A)* action can go anywhere. That is, we can choose any of 5 positions in which to insert the read action among the lock and unlock actions, ranging from before all to after all. Having placed the read action, the *w_1(B)* action can now be placed in any of 6 positions among the other five. The total number of 2PL sequences is thus 4*5*6 = 120.

Since *(i) + (iii)* = 120, and we already know *(i)* = 18, we deduce that *(iii)* = 102.

The last step is to compute the total number of sequences, which is the sum of all four categories. The number of sequences of six actions is 6! = 720. As the number of sequences in the first three categories is 18, 2, and 102, respectively, *(iv)* = 720 - 18 -2 - 102 = 598.

Return to Top

Solutions for Section 9.4

Exercise 9.4.1(d)

(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.

Exercise 9.4.2(a)

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.

Exercise 9.4.6(a)

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).

Exercise 9.4.7(a)

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.

Return to Top

Solutions for Section 9.5

Exercise 9.5.1(b)

If we examine the compatibility matrix from Fig. 9.22 of the text, we see that the only possible sets of locks on a single element are either

1. One exclusive lock.
2. One or more shared locks.
3. 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.

Return to Top

Solutions for Section 9.6

Exercise 9.6.1(a)

At the first step, T_1 puts a IS lock on the extent and on block 1, and an S lock on object O_1.

At step 2, T_2 puts an IX lock on the extent and on block 1, both of which are compatible with the IS locks already there. T_2 also puts an X lock on O_2.

At step 3, T_2 puts an IS lock on the extent and on block 2 and an S lock on O_3, then releases its locks.

At step 4, T_1 puts an IX lock on the extent and on block 2 and an X lock on O_4, then releases its locks.

Return to Top

Solutions for Section 9.7

Exercise 9.7.1(a)

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.

Exercise 9.7.2(a)

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.

Return to Top

Solutions for Section 9.8

Exercise 9.8.1(a)

Since T_1 starts first, it has the lower timestamp. The two read steps execute without problem, and leave *A* and *B* with the read-timestamps of T_1 and T_2, respectively.

When T_2 tries to write *A*, it finds the read-timestamp to be less than the timestamp of T_2, so the write may be performed, and T_2 can commit. However, when T_1 tries to write *B*, it finds the read-timestamp is greater than its own timestamp, so some transaction read another value of *B*, when it should have read T_1's value. Thus, T_1 must abort.

Exercise 9.8.2(a)

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*.

Return to Top

Solutions for Section 9.9

Exercise 9.9.1(a)

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.

Return to Top