Database System Implementation

Solutions for Chapter 10

Return to Top

Solutions for Section 10.1

Exercise 10.1.1

The difference between strict and nonstrict locking for this example is only in the matter of whether the lock on *A* is released prior to the write of *B*. That is, the lock on *A* must be taken before *r_1(A)*. The lock on *B* can occur in any of the three positions prior to *r_1(B)* (3 choices).

If locking is strict, then the two unlocks must occur, in either order, after the second write action. Thus, there are 6 orders for strict, 2-phase locking.

If locking is not strict, the unlock of *B* must occur after *w_1(B)*, but then the unlock of *A* can occur in any of the three positions after *w_1(A)*. Thus, there are 9 orders if strictness is not required, and 3 of these must be nonstrict.

Exercise 10.1.2(a)

T_1 wrote only *B*, but that value was later read by T_3. Thus, T_3 must be rolled back. T_3 wrote *D*, but no transaction has read *D*, so no further rollbacks are needed.

Return to Top

Solutions for Section 10.2

Exercise 10.2.1(a)

```
         ___________________________
        |                          |
 A      |                          v      B
```

```
T_0 ----> T_1       T_2 ----> T_3 ----> T_f
| |          A        ^         ^
| |__________________|         |
|              A               |
|______________________________|
```

Above is the polygraph. We have labeled with *A* the arcs due to the reading of *A*, and likewise for *B*. The two unlabeled arcs are second-phase requirements that the other writers of *B* --- T_1 and T_2 --- not interfere with the reading of *B*. by the final ``transaction'' T_f.

Clearly, T_3 must come last in an view-serializable schedule, but there is no required ordering of T_1 and T_2. Notice that conflict-serializability would require T_1 to precede T_2, because their writes of *B* cannot swap places.

Exercise 10.2.2(a)

For conflict-equivalent schedules, the three writes must remain in order, but the three reads may be moved anywhere, as long as they precede the write by the same transaction.

Start with the three writes. We can only place *r_1(A)* prior to *w_1(B)*. Next, we can place *r_2(A)* in any of the three positions prior to *w_2(B)*. At the last step, there are now five places where we can put *r_3(A)* prior to *w_3(B)*. Thus, there are 15 conflict-equivalent schedules.

For view-serializable orders, there are the above 15, and another 15 in which *w_1(B)* is placed after *w_2(B)*. Intuitively, we can do so because neither T_1 nor T_2 have their value of *B* read, so the order in which they write doesn't matter.

Return to Top

Solutions for Section 10.3

Exercise 10.3.1(a)

All locks are granted until *w_3(B)*. Now, T_3 has to wait for T_1, which has a shared lock on *B*, and the waits-for graph is

```
T_3 ----> T_2
```

At the next step, T_2 must wait for T_1, which has a lock on *C*. The waits-for graph is:

```
T_3 ----> T_2 ----> T_1
```

Next, *w_4(A)* causes T_4 to wait for T_1. There is still no cycle, and the waits-for graph is:

```
T_3 ----> T_2 ----> T_1 <---- T_4
```

Finally, T_1 tries to write *D*. It cannot get the exclusive lock on *D* that it needs, because T_3 holds a shared lock on *D*. If we allowed T_1 to wait, the graph would be:

```
T_3 ----> T_2 ----> T_1 <---- T_4
 ^                   |
 |___________________|
```

Thus, T_1 must abort and relinquish its locks. At that time, T_2 and T_4 can each get the locks they need. When T_2 finishes, T_3 can proceed.

Exercise 10.3.7

Suppose T_1 is a transaction that tries to lock elements *A* and *B* in that order. There are also an indefinitely long sequence of transactions T_2, T_3,... that lock *B* and then *A*.

Initially, T_1 locks *A* and T_2 locks *B* and then requests a lock on *A*. T_2 therefore waits for T_1. Meanwhile, T_3 starts and requests a lock on *B*. Thus, T_3 waits for T_2, and the waits-for graph is:

```
T_3 ----> T_2 ----> T_1
```

When T_1 requests a lock on *A*, it would cause a cycle, so T_1 is rolled back. Now, T_2 completes and releases its locks. The lock on *A* is given to T_1 and the lock on *B* is given to T_3. Next, T_3 requests a lock on *A*, but has to wait for T_1. Also, T_4 starts up and requests a lock on *B*, thus being forced to wait for T_3. The waits-for graph is now:

```
T_4 ----> T_3 ----> T_1
```

The above graph is just like the first, except T_4 and T_3 have replaced T_3 and T_2, respectively. As long as there is a supply of transactions like T_2, T_3, and T_4, they can prevent T_1 from finishing ever.

Notice that the transactions in the example above request locks on *A* and *B* in different orders. Perhaps forcing transactions to request locks in a fixed order will solve the problem. Indeed, if we follow the policy that when a lock becomes available, it is given to a waiting transaction on a first-come-first-served basis, then any transaction that is waiting for a lock will eventually become the transaction that has been waiting for the lock the longest. At that point, the next time the lock becomes available, it will be granted to that transaction, which thus makes some progress. The above observation is the germ of an inductive proof that requesting locks in order plus first-come-first-served will allow every transaction eventually to complete.

However, if the scheduler is able to give locks to any transaction it wishes, then it is easy to make a transaction starve. Suppose T_1 wants a lock on *A* when some other transaction has that lock. As long as there is always another transaction besides T_1 waiting for a lock on *A*, and the scheduler always chooses to give the lock to some transaction other than T_1 when the lock becomes available, T_1 never makes any progress.

Last, let us consider deadlock prevention by timeout. If the duration before a timeout can vary even slightly, then the example given initially for deadlock prevention by avoiding cycles will work. That is, even though T_2 has been waiting slightly longer for a lock than T_1, if we allow that T_1 might timeout first, then the sequence of events described above could also occur with timeout-based deadlock prevention.

If timeouts occur for any transaction at exactly *t* seconds from when it first started to wait, then we need a simple modification of the above example. Assume that T_1 is faster at requesting its second lock than any of the other transactions. Then, after T_1 gets a lock on *A* and T_2 gets a lock on *B*, T_1 next requests its lock on *B* and starts to wait. Slightly later, T_2 requests a lock on *A*. T_3 requests its lock on *B* just before T_1 times out, so when that timeout occurs, T_2 completes, gives its *A*-lock to T_1 and its *B*-lock to T_3. When T_2 next requests a lock on *B* and starts to wait, this cycle can repeat with T_3 and a new T_4 in place of T_2 and T_3.

Return to Top

Solutions for Section 10.4

Exercise 10.4.1

Define U to be the sum over all sites i of $10du\_i$. That is, U is the cost of sending one second's worth of updates to one site other than the site at which the update originated.

Now, consider the value of creating a copy of R at site i. On the positive side, the queries generated at site i that used to have to be answered remotely, at a cost of $10cq\_i$, can now be answered for cq_i, a positive benefit of $9cq\_i$.

However, the additional copy of R has to be updated. The cost is du_i for the updates generated at site i and $U - 10du\_i$ for the updates generated at all other sites. Recall that U represents the cost of a remote update for all generated updates, so if we subtract $10du\_i$ we get just the cost of the updates that do not originate at i.

In order for the benefit of creating a copy at i to outweigh the additional update cost, we must have

$$9cq\_i + 9du\_i >= U$$

Thus, the optimal selection of sites at which to maintain copies of R is all sites for which the above inequality holds. Roughly, we favor those sites at which there is a lot of activity, but the larger U is (i.e., the greater the update rate), the less likely we are to want to create many copies.

There is one special case: what if the inequality fails for all i; that is, it is not cost effective to create *any* copies. As long as all the q_i's are zero, that's OK. The proverbial ``write-only database'' needs no copies! However, in realistic cases with a heavy update rate, we shall be forced to create one copy. The correct choice is whichever i has the largest value of $cq\_i + du\_i$.

Return to Top

Solutions for Section 10.5

Exercise 10.5.1(a)

There is a component, call it A at the home computer, and components at each of the banks --- components B and C. Component B receives the directive from A. B checks that there is adequate money in the account, and aborts if not. Otherwise, B subtracts \$10,000 from the account at B and signals C to deposit \$10,000 into the designated account at C. Component C checks that the account exists, and aborts if not. Otherwise, C adds \$10,000 to the account and informs A of a successful conclusion.

Exercise 10.5.2(a)

$(0,1,P)$, $(0,2,P)$, $(1,0,R)$, $(2,0,D)$, $(0,1,A)$, $(0,2,A)$

Exercise 10.5.2(b)

In the first phase, the coordinator exchanges the messages $(0,1,P)$, $(1,0,R)$ with site 1 and the messages $(0,2,P)$, $(2,0,R)$ with site 2. These messages may occur in any of $(4 \text{ choose } 2) = 6$ possible interleavings. That is, there are four positions in which these messages occur, and we can pick any two of them to devote to the messages involving site 1.

In the second phase, the coordinator can send commit messages to the other two sites in either order. There are thus 6 * 2 = 12 possible message sequences.

Return to Top

Solutions for Section 10.6

Exercise 10.6.1(a)

Suppose that *s*, *x*, and *i* are the numbers of local shared, exclusive, and intcrement locks that a transaction needs to have a global lock of that type. Since no two transactions can hold an exclusive lock on the same element, we need:

$$2x > n$$

just as for the shared + increment system discussed in the section. We also cannot allow a shared and exclusive lock at the same time, which tells us:

$$x + s > n$$

Likewise, we cannot have an exclusive and increment lock at the same time, so:

$$x + i > n$$

Finally, we cannot have a shared and increment lock at the same time, so:

$$s + i > n$$

For example, we could require that any type of lock requires a majority. We cannot allow both *s* and *i* to be *n/2* or less, because of the last inequality. But we could let one of *s* and *i* be 1, while the other, and *x*, are both *n*.

Exercise 10.6.2(a)

The 90% of the accesses that are read-only require no messages, since there is a lock table and a copy at each site. The remaining 10% of the accesses require exclusive locks. Thus, each requires three messages between the originating site and each of the four other sites, a total of 12 messages. The average number of messages is 1.2.

Return to Top

Solutions for Section 10.7

Exercise 10.7.1

The task of the compensating transaction is first to determine whether the file *f* is still the present one with that name, or whether it has been replaced with a later version.

1. If *f* is still in place, then the compensating transaction must
   1. Restore *f'* if the latter existed, or
   2. Delete *f* if not.

2. If *f* has been overwritten, then there is no need to change the file with that name.

To see that this compensation works, we can consider all the possible cases above, and check that the file system, after compensation, is the same as would be the case had *f* never been written. If *f* was overwritten before compensation, then we surely leave the file system in the same state as if *f* were never written. If *f* is not overwritten, then we have correctly left the file with that name being either *f'*, if it existed, or left the file system with no file of that name.

There are two interesting issues. First, in order for there to be a compensating transaction, it seems we must both remember the time at which the installation occurred (so we can tell whether the file *f* has been overwritten when the uninstallation occurs), and we must also remember the file *f'*, at least until such time as the file is overwritten, and the new value is itself guaranteed never to be uninstalled (i.e., the saga that wrote it has finished). These features are not commonly available.

Second, note that file systems do not support serializability. Thus, it is entirely possible that the uninstalled file *f* has been read, and its contents have influenced other files or the behavior of the system.

Return to Top