ECE356: Database Systems Winter 2018

Lecture 22 — Concurrency Control

Jeff Zarnett

Concurrency Control

Having discussed in some detail the idea of transaction isolation, we now need to turn our attention to the question of how the database will control the various concurrent elements to make sure that execution goes as planned. There are a variety of schemes, but at this point the ones we will examine will always assume that failure does not happen. Once we understand the concurrency control mechanisms we can then build on them to include failure recovery.

No Concurrency Control

It is possible from the point of view of the database to simply choose not to implement concurrency control of any sort. This is easy for the database designers and effectively means putting the problem of concurrency management in the hands of the users of the database (application developers).

This is not as insane as it sounds, since the application itself could have its own sort of concurrency control mechanisms and it would be fine to just let the application manage that. The solution is, however, inadequate for commercial databases since they can allow multiple applications to run at once and do not trust that application developers do things the "right" way at all times.

Lock-Based Protocols

As you will recall from learning about concurrency and synchronization, our typical "go-to" strategy for dealing with concurrency problems is locking of some sort. You are surely familiar with locks and locking so we will not need to repeat any of the material on that subject from earlier.

For performance reasons we will assume that we use reader/writer type locks, meaning that there are accordingly two modes in which a data element may be locked: shared mode, and exclusive mode. In keeping with [**?**] the shared mode lock is denoted by S and exclusive by X. If transaction T_i has a shared lock on some data element e then it may read but not write it; if the transaction has an exclusive lock then it may both read and write. Exclusive is exactly what it sounds like in that no other transaction may have any sort of lock if one transaction has an exclusive lock.

Every transaction must request a lock before accessing a data item, and the sort of lock it will request depends on what it would like to do. The request is made of the concurrency control system and it will either grant the request or delay that transaction until the request is granted. This you will recognize as being the same sort of behaviour of the mutex and semaphore: a requestor may proceed or will be blocked for some period of time.

As for whether the request can be granted, this is pretty simple. If there are no locks for that item, then the request may be granted. If the request is exclusive and there exists any lock on that data item, the request must wait. If the request is for a read and there exists an exclusive lock on the item, the request must wait. If the request is for a read and there exist only read locks on the item, the read may proceed. If the request can be granted, that might not guarantee it will be granted immediately.

If there are more lock modes (which we will not consider in this course, at least) then we would need to check to see whether a new request's locking mode is compatible with the existing ones. The compatibility rule for reader-writer locks is pretty simple as above, but in the more general approach it is necessary to check all the locks currently held for that item and see if they are compatible. If they are all compatible, the lock may be granted.

As with the kinds of locks we are accustomed to from C-like languages, there is also an unlock operation to, well, unlock that item. Unlocking an item will allow some other waiting transaction (or transactions) to be able to then lock that item.

The standard advice in concurrency is to unlock items as soon as possible (but no sooner). The sooner a transaction unlocks things it is finished with, the sooner another transaction may be unblocked and may proceed. But there is such a thing as unlocking too soon. Why? Serializability may not be ensured: it is possible that an early unlocking allows some intermediate state to be read. Let's do an example from [**?**].

Let us suppose there are two concurrent transactions going on. T_1 is to transfer \$50 from chequing (account C) to savings (account S). The second is to show your total net worth which is computed by summing C and S (and we will ignore any other details like credit card debts, loans, investments, etc). Let's look at the statements needed to get this done:

Transaction T_1

```
T1.1 Exclusive lock C
T1.2 Read C
T1.3 C = C - 50
T1.4 Write C
T1.5 Unlock C
T1.6 Exclusive Lock S
T1.7 Read S
T1.8 S = S + 50
T1.9 Write S
T1.10 Unlock S
```

Transaction T_2

```
T2.1 Shared lock C
T2.2 Read C
T2.3 Unlock C
T2.4 Shared Lock S
T2.5 Read S
T2.6 Unlock S
T2.7 TEMP = A + B
T2.8 Print TEMP
```

If executed serially we have no problem; we will always get consistent results. No matter what order we execute the transactions, the result will be the same since any amount moved from C to S does not affect the total. If they are, respectively \$100 and \$200 then the total must remain \$300. This is fine. But if the transactions are executed incorrectly, we will print a result of \$250. How? Think back to earlier courses on concurrency: try to find an interleaving of statements that could lead to this inconsistent result.

A quick analysis says this can happen if T_2 reads the values of `C` and `S` after the decrease has been completed but before the increase. One such possible ordering places all of T_2 between statements `T1.5` and `T1.6`, but you can likely find other orders that reproduce this problem. As we know, any such order with a problem means we need to make some changes to prevent this problem.

How can we fix this? In short, the unlocking of elements needs to be delayed appropriately to ensure serializability: that is to say that partial state should not become visible. A modified version of T_1 and T_2 shown below that avoids the problem altogether.

Transaction T_1'

```
T1.1 Exclusive lock C
T1.2 Read C
T1.3 C = C - 50
T1.4 Write C
T1.5 Exclusive Lock S
T1.6 Read S
T1.7 S = S + 50
T1.8 Write S
T1.9 Unlock C
T1.10 Unlock S
```

Transaction T_2'

```
T2.1 Shared lock C
T2.2 Read C
T2.3 Shared Lock S
T2.4 Read S
T2.5 TEMP = A + B
T2.6 Print TEMP
T2.7 Unlock C
T2.8 Unlock S
```

There are no free lunches, though: the risk of longer-held locks is that there can be, as you guessed... *deadlock*! Deadlock, as you will recall, is what happens when two or more transactions (formerly processes) are permanently blocked waiting for one another. As before, an unfortunate interleaving of lock and unlock statements can mean that the transactions get stuck. Deadlock is bad, but not as bad (we think) as inconsistent states being shown (or worse, saved!). If a deadlock occurs, we can detect it and then can do something about it; if an inconsistent state is shown then we may never know about it.

There are often rules in the system for how transactions behave with respect to locks called the *locking protocol*. The locking protocol sets up some rules about when items may be locked and unlocked which reduces the number of possible schedules such that the only allowed schedules are conflict-serializable schedules [**?**].