CS 370 Concurrency worksheet

Name____

Student ID____

1) Apply the appropriate locks and show the resulting schedule for the following sequence of operations using strict 2PL. Assume locks can be upgraded.

   T1:R(X); T2:W(Y); T3:R(X); T2:R(X); T2:R(Z); T2:Commit; T3:W(X); T3:Commit; T1:W(Y); Commit

T1	T2	T3

2) Is the following schedule conflict serializable? Why?

T8:R(X); T9:R(Y); T8:W(Y); T9:W(X);T8:Commit; T9:Commit;

3) Using the timestamp method of creating conflict serialization, fill in the following chart. Which (if any) transactions are rolled back? T3's timestamp is 3, T4's timestamp is 4.

Schedule	Action	R-TS (X)	W-TS(X)	R-TS(Y)	W-TS(Y)
Initial Values		1	2	2	1
T3	R(Y)				
T4	R(X)				
T4	R(Y)				
T3	R(X)				
T4	W(Y)				
T3	W(X)				
T4	W(X)				
T3	R(X)				

Serializable schedules

Example 1

T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
R(C)	
W(C)	
Commit	
	Commit

In this situation, the two transactions do not have operations on the same data item, so the schedule is serializable. A serializable schedule is a schedule whose effect on any consistent database instance is guaranteed to be identical to that of some complete serial schedule over S.

Example 2.

T1	T2
R(A)	
W(A)	
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

Is this schedule serializable?

Yes – because it is equivalent to the serial schedule T2, T1. The results are equivalent to T2 running first in its entirety, then T1 running.

Example 3

T1	T2
R(A)	
	R(B)
W(A)	
	W(B)
	Commit
R(B)	
W(B)	
Commit	

Is this schedule serializable? Yes, this is still equivalent to the serial schedule T2, T1.

Example 4.

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
	Commit
R(B)	
W(B)	
Commit	

Is the schedule serializable? Let's look at the 2 possibilities. The schedule cannot be T2, T1 because T2 is reading the result of T1's write. The schedule cannot be T1, T2 because T1 is reading the result of T2's write. So the schedule is not serializable.

This is an example of reading uncommitted data (called a dirty read).

Locking

Although requests to insert and release locks are automatically inserted into transactions by the operating system, we look at the mechanisms to that it does to do so.

There are 2 types of locks, shared (read) and exclusive (write) for each data item. If you are going to do nothing but read an item, you acquire a shared lock. If your intent is to modify the data item, you acquire a write lock. It is not necessary to get both a read and a write lock to read and write a data item, a write lock is sufficient.

2-phase locking protocol

2-phase locking protocol is one in which there are 2 phases that a transaction goes through. The first is the growing phase in which it is acquiring locks, the second is one in which it is releasing locks. Once you have released a lock, you cannot acquire any more locks.

This protocol ensures a serializable schedule.

Let's look at a couple of examples:

T1	T2
RL(A)	
R(A)	
RL(B)	
R(B)	
	RL(A)
	R(A)
	UL(A)
UL(A)	
UL(B)	

In this example, read locks for both A and B were acquired. Since both transactions did nothing but read, this is easily identifiable as a serializable schedule.

T1	T2
WL(A)	
R(A)	
	Attempt to RL(A) fails
W(A)	
UL(A)	
	RL(A)
	R(A)
	UL(A)
Commit	
	Commit

In this case, we stopped the conflicting operations from occurring by not allowing the locks to be granted until the write lock had been freed. How does this ensure a serializable schedule? Let's look at the previous unserializable schedule example:

T1	T2
R(A)	
W(A)	
	R(A)
	W(A)
	R(B)
	W(B)
R(B)	
W(B)	
Commit	
	Commit

Note the change in our commit order from the previous example. This is done to make the schedule recoverable (i.e. transactions that have dependencies on previous transactions must not commit until the previous transactions commit).

Using the 2-phase locking protocol, we get the following:

T1	T2
WL(A)	
R(A)	
W(A)	
	WL(A) Not granted
WL(B)	
R(B)	
W(B)	
UL(A)	
UL(B)	
	WL(A)
	R(A)
	W(A)
	WL(B)
	R(B)
	W(B)
	UL(A)
	UL(B)
Commit	
	Commit

We see that the locking protocol keeps the schedule serializable.

Strict 2-phase locking protocol

Let's look at the previous example:

T1	T2
WL(A)	
R(A)	
W(A)	
	WL(A) Not granted
WL(B)	
R(B)	
W(B)	
UL(A)	
UL(B)	
	WL(A)
	R(A)
	W(A)
	WL(B)
	R(B)
	W(B)
	UL(A)
	UL(B)
Commit	
	Commit

We saw that 2-phase locking protocol ensured a serializable schedule. But what happens if T1 aborts rather than commits? There is a dependency of T2 on T1 – so we get a cascading abort (i.e. when T1 aborts, T2 must abort as well).

This is resolved by using a strict 2-phase locking protocol as opposed to the 2-phase locking protocol. In 2-phase locking protocol – all locks are held until the transaction commits or aborts.

T1	T2
WL(A)	
R(A)	
W(A)	
	WL(A) Not granted
WL(B)	
R(B)	
W(B)	
Commit	
UL(A)	
UL(B)	
	WL(A)
	R(A)
	W(A)
	WL(B)
	R(B)
	W(B)
	Commit
	UL(A)
	UL(B)

Strict 2-phase locking protocol ensures both a serializable schedule and one that avoids cascading aborts.

Looking at the schedule – we have ended up with a serial schedule and you might ask why go to all that work just to end with a serial schedule. Remember we are using conflicting operations to illustrate the point. If the operations were not conflicting, there would be no issues interleaving operations.

T1	T2
WL(A)	
R(A)	
W(A)	
	WL(C)
	W(C)
RL(X)	
	WL(Y)

Time-stamp protocol – ensures that conflicting read and write operations occur in timestamp order. Each data item has the following timestamps associated with them:

R-timestamp(Q) – largest timestamp of a transaction to read Q

W-timestamp(Q) – largest timestamp of a transaction to write Q

If a read (Q) is issued by Ti:

1) If TS(Ti) < W-timestamp(Q) then Ti needs to read a value of Q that was already overwritten. The read operation is rejected and Ti is rolled back.
2) If TS(Ii) >= W-timestamp(Q) then the read operation is executed and R-timestamp (Q) is set to the maximum of R-timestamp(Q) and TS(Ti).

If a write(Q) is issued:

1) If TS(Ti) < R-timestamp(Q) then the value of Q that Ti is producing was needed previously, and the system assumed that the value would never be produced. The write is rejected and Ti is rolled back.
2) If TS(Ti) < W-timestamp(Q) then Ti is attempting to write an obsolete value of Q. The write is rejected and Ti is rolled back.
3) Otherwise the write is accepted and W-timestamp(Q) is set to TS(Ti).

Time-stamp protocol ensures conflict serializability – because conflicting operations are processed in timestamp order. It also ensures freedom from deadlock since no transaction ever waits. There is the possibility of starvation if a sequence of short transactions continue to cause repeated restarting of a long transaction.

The protocol can generate schedules that are not recoverable. It can be extended to include recoverability by:

1) (Recoverable and cascadeless) Forcing all the writes together at the end of the transaction. The writes are atomic in that while the writes are in progress, no transaction is permitted to access any of the data items being written.
2) (Recoverable and cascadeless) Using a limited form of locking where the reads of uncommitted items are postponed until the transaction that updated the item commits.
3) (Recoverable) Tracking uncommitted writes and allowing Ti to commit only after the commit of any transaction that write a value the Ti read.