Locking

Although requests to insert and release locks are automatically inserted into transactions by the DBMS, 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)

Deadlocks

When you introduce a locking protocol, deadlocks always become a possibility.

T1	T2
WL(A)	
	WL(B)
R(A)	
	R(B)
WL(B) not granted	
	WL(A) not granted

There are many ways to recognize deadlock – precedence graphs. The most common is to put a timeout on the transaction – if the locks haven't been granted in X amount of time, roll the transaction back to the beginning.