Exercise 7 Solution

1. Assume the following two transactions start at nearly the same time and there is no other concurrent transaction. The 2nd operation of both transactions is Xlock(B). Is there a potential problem if Transaction 1 performs the operation first? What if Transaction 2 performs the operation first?

Transaction 1		Transaction 2	
1.1	Slock(A)	2.1	Slock(A)
1.2	Xlock(B)	2.2	Xlock(B)
1.3	Read(A)	2.3	Write(B)
1.4	Read(B)	2.4	Unlock(B)
1.5	Write(B)	2.5	Read(A)
1.6	Unlock(A)	2.6	Xlock(B)
1.7	Unlock(B)	2.7	Write(B)
		2.8	Unlock(A)
		2.9	Unlock(B)

Solution:

If Transaction 1 executes Step 1.2 before Transaction 2 executes Step 2.2, there would be no problem. This is because T1 releases the lock at the end. T2 has to wait till then. However, if Transaction 2 executes Step 2.2 first, Transaction 1 may have a dirty read of B if it tries to run Step 1.2 immediately after Transaction 2 acquires the Xlock on B. This is because when Transaction 2 releases the lock on B (Step 2.4), Transaction 1 will be granted the Xlock on B (Step 1.2). After that, Transaction 1 will read B (Step 1.4). After Transaction 1 completes, Transaction 2 will acquire another Xlock on B (Step 2.6) and then modify the object. In other words, the reading of B by Transaction 1 happens between two writes of B by Transaction 2.

2. What degree of isolation does the following transaction provide?

Slock(A)
Xlock(B)
Read(A)
Write(B)
Read(C)
Unlock(A)
Unlock(B)

Solution:

Degree 1 as there is one read operation 'Read(C)' without taking any lock. The only write operation has an exclusive lock associated with it. The transaction is two-phase with respect to exclusive lock.

3. The following operations are given with Degree 2 isolation locking principles in place. Convert the locking sequence to Degree 3.

Degree 2

Slock(A)
Read(A)
Unlock(A)
Xlock(C)
Xlock(B)
Write(B)
Slock(A)
Read(A)
Unlock(A)
Write(C)
Unlock(B)
Unlock(C)

Solution:

Degree 3

Slock(A)
Read(A)
Xlock(C)
Xlock(B)
Write(B)
Read(A)
Unlock(A)
Write(C)
Unlock(B)
Unlock(C)

4. The following transactions are issued in a system at the same time. Answer for both scenarios.
   (a) **Scenario 1:** When the value of A is 3, which of the following transactions can run concurrently from the beginning till commit (that is, all operations and locks are compatible to run concurrently with another one) and which ones need to be delayed? Please give an explanation for the delayed transactions. Note that, the order of start of transactions can be deducted from the beginning positions of the transactions in the table given.
   (b) **Scenario 2:** When the value of A is 2, which of the following transactions can run concurrently from the beginning till commit (that is, all operations and locks are compatible to run concurrently with another one) and which ones need to be delayed? Please give an explanation for the delayed transactions. Let's assume T1 starts slightly earlier than others for this case. [1]

	T2	T3
	Lock (U,A)	Lock (IX,A)
T1	Read A	Read A
Lock (S,A)	if(A ==3) {	if(A ==3){
Read A	Lock(X,A)	Lock(X,A)
Unlock A	Write A	Write A
	}	}
	Unlock A	Unlock A

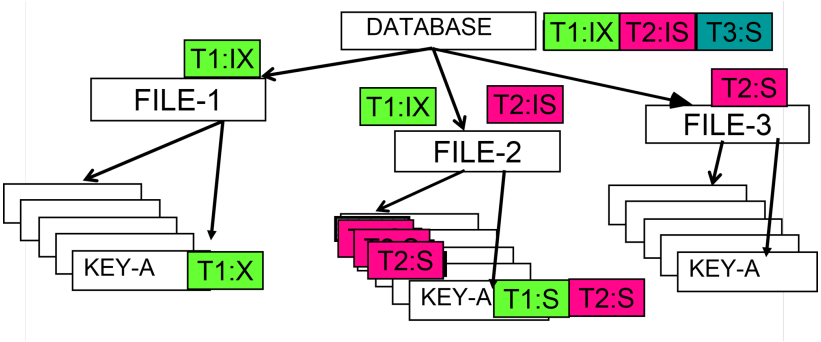
[1] *There can be different versions of the compatibility matrix. Please use the compatibility matrix from the lecture slides for this exercise.

Solution:

Scenario 1: None of the transactions can run concurrently. T2's update lock and T3's IX conflict with each other, and the subsequent X locks for A==3 will conflict. T3's IX lock conflicts with T1's S lock. Although T1's shared lock and T2's update lock are compatible if T1 gets the lock first, but T1 gets the lock later which does not work as the compatibility matrix we saw in class shows. So only one can run at a time while the other transactions must be delayed.

Scenario 2: When A is 2, T2 and T3 will not request exclusive lock. Hence, T1 and T2 can run concurrently as T2's update lock can be granted if T1 gets the shared lock on A first. T1 and T3 cannot run concurrently - one of them must be delayed as the locks are not compatible. T2 and T3 cannot run concurrently - one of them must be delayed as the locks are not compatible.

5. Review the concepts of granular locks then answer the following question. Given the hierarchy of database objects and the corresponding granular locks in the following picture, which transactions can run if the transactions arrive in the order T1-T2-T3? What if the order is T3-T2-T1? Note that locks from the same transaction are in the same colour. We assume that the transactions need to take the locks when they start to run.



Solution:

If the order of the arrival of transactions is T1-T2-T3, then T1 and T2 can run in parallel while T3 waits. This is because T1's IX lock at the root node is not compatible with T3's S lock at the same node.

If the order is T3-T2-T1, then T3 and T2 can run while T1 waits. This is due to a similar reason as above. This example shows that the order of transactions can be a deciding factor in the set of parallel-running transactions. We should also note that granular locks can lead to the delay of transactions at any level in the hierarchy below the root node, e.g., a transaction may need to wait for a lock at the FILE-3 node or a KEY-A node due to lock compatibility issues.

6. With two-phase locking we have already seen a successful strategy that will solve concurrency problems for DBMSs. Then discuss why someone may want to invent something like Optimistic Concurrency control in addition to that locking mechanism.

Solution:

Two-phase locking or in general locking assumes the worst, i.e. there are many updates in the system and most of them will lead to conflicts in access to objects. This means there is a good rationale to pay the overhead of locking and stop problems from occurring in the first place. But what if the DBMS is one such that people tend to work on different parts of data, or most of the operations are read operations, and as such there aren't many conflicts at all? Then there is no need to pay the overhead of lock management but rather it may be better to allow transactions to run freely and have a simple check when they finish whether there was any conflict with concurrent transactions. Most of the time there will not be so one will get increased concurrency with less overheads. Obviously, the reverse is also true, i.e., if there were really many conflicting writes then doing optimistic concurrency control means many problems would be observed only after running the transactions and a lot of work will need to be wasted to preserve the consistency of the data. So there is no clear answer but depending on the situation a strategy may be good or bad.