REMARKS ON OTHER SOLUTIONS:

- for problems dealing with Datalog evaluation please use the notation in lecture notes i.e. {*likes(eve,pie),…*} rather than a table headed *likes*, with pairs underneath it.
- for FD problems, since with n column names there are 2^n possible subsets, in finding keys you might want to try my class suggestion to start with very small sets and grow them into keys, so that you never continue to see superkeys that are not keys.

SOLUTION TO TRANSACTION HOMEWORK + COMMENTS

There are two tuples in Item: ('a',20) and ('b',30). Consider the following two concurrent transactions, each of which runs once and commits. You may assume there are no other transactions in the system and that *individual statements execute atomically*.

```
T1: begin transaction
   S1: insert into Item values ('c',40)
   S2: update Item set price = price+30 where name='c'
   Commit /* Note that T1 creates a new variable and then a Read and Write on it */

T2: begin transaction
   S3: select avg(price) as p1 from Item
   S4: select avg(price) as p2 from Item
   Commit /* note that T2 does 2 sets of Reads only*/
```

Suppose that transaction T1 executes with isolation level _serializable_. What are ALL the possible pairs of values for p1 and p2 returned by T2 under the following isolation levels.

(a)T2 also executes with isolation level *_serializable_*,

The only schedules allowed must be equivalent to T1;T2 or T2;T1; so the effects of schedules is s1,s2,s3,s4 which returns (40,40), or s3,s4,s1,s2, which returns (25,25). ANSWER: {(25,25),(40,40)}

(b) If transaction T2 executes with isolation level *repeatable read*, T2 only holds "tuple locks on existing tuples" so S1 can S2 can "sneak in" between S3 (which only Slocked non-'c' tuples) and S4 (which now Slocks tuple 'c' too). So in addition to T1;T2 and T2;T1 we can definitely also have schedule S3,S1,S2,S4, because T1=S1S2 released all its X-locks by the time S4 got around. Is S3,S1,S4,S2 ok? One can argue 'no' because S4 requires S-lock on all existing tuples, but T1 still holds Xlock on tuple c, which now exists and is no longer a phantom, and this lock won't be released till T1 commits. So, any schedule like S1,S3… is also not allowed. So there are three schedules

schedule 1: s1,s2,s3,s4 (p1,p2)=(40,40) // schedule 2: s3,s4,s1,s2 (p1,p2)=25,25 // schedule 3: s3,s1,s2,s4 (p1,p2)=(25,40)

(c) If transaction T2 executes with isolation level *read committed*, the main difference from (b) is that shared locks are released immediately after the read. The only place this can occur is between S3 and S4, where T2 releases the S-locks on all the old tuples. But T1 does not need any of these. So nothing new happens here w.r.t. (b)

(d) If transaction T2 executes with isolation level *read uncommitted*, then T2 does not respect any (exclusive) locks placed by T1, and hence any of the 6 shuffles of (s1,s2) and (s3,s4) are permitted:

Schedule 1: S1 S2 S3 S4 (p1,p2) = (40,40) // Schedule 2: S1 S3 S2 S4 (p1,p2) = (30, 40) // Schedule 3: S1 S3 S4 S2 (p1,p2) = (30,30) // Schedule 4: S3 S1 S4 S2 (p1,p2) = (25, 30) // Schedule 5: S3 S4 S1 S2 (p1,p2) = (25,25) // Schedule 6: S3 S1 S2 S4 (p1,p2)=(25,40)

Note that in all cases T1 does indeed run in Serializable mode since it is unaffected by T2 in this case.

8(B) CONFLICT SERIALIZABILITY AND LOCKING   Consider the following 2 schedules of *attempted* reads/writes by transactions. In each case, indicate whether the schedule is serial, serializable or not serializable. Annotate the schedule to indicate where each transaction asks for a lock, and then mark the places where a transaction is blocked and where deadlock occurs

Schedule A.

```
T1      T2
R(m)
R(n)
            R(n)
            W(m)
```

This is a serial, and hence serializable schedule. The order is obviously T1;T2. Its annotation with locking could be written as

```
T1                      T2
Sl(m) R(m)
Sl(n) R(n)
Commit
Release locks
                        Sl(n) R(n)
                        Xl(m) W(m)
                        Commit
                        Release locks
```

where

Sl(V) means that that particular transaction gets a shared lock on V
Xl(V) means that that particular transaction gets an exclusive lock on V

An alternate "horizontal notation" for the schedule A would be

```
 r(M,1)    r(N,1)     r(N,2)     w(M,2)      /* r(X,n) stands for "transaction n reads from X" */
```

and its annotation might look like

```
sl(M,1)   sl(M,1)      sl(N,2)    xl(M,2)
 r(M,1)    r(N,1)        r(N,2)     w(M,2)
           release(1)               release(2)
```

where sl(V,j) stands for "Shared lock for V acquired by transaction j", xl(V,j) stands for "Exclusive lock for V acquired by transaction j", and release(j) stands for "transaction j releases all its locks".

Schedule B

T1	T2	T3
R(m)		
	R(y)	
		W(z)
	W(m)	
W(n)		
		abort

Lets annotate with lock acquisition/release:

T1	T2	T3
Sl(m),R(m)		
	Sl(y), R(y)	
		Xl(z),W(z)
	Xl(m) **refused because T1 has s-lock on m blocked/wait**	
Xl(n),W(n)		
Commit		
release locks		
	Xl(m) **granted**	
	W(m)	
	Commit	
	release locks	
		abort
		release locks

Although this is what takes place in practice, note that the actual schedule that took place was not the original one! Because of T2 waiting we actually ran the following schedule, which swapped T1.W(n) with T2.W(m):

T1	T2	T3
R(m)		
	R(y)	
		W(z)
W(n)		
	W(m)	
		abort

So one can argue that the original schedule itself was not equivalent to a serial one, and hence not serializable! And this is what we are going to do in this class for simplicity: we will say that if the schedule leads to blocking anywhere, even if there is no deadlock, the schedule will be called non-serializable. (For grading we will accept both answers: (i) this schedule is non-serializable, because it led to some blocking; or (ii) it is serializable, with one serial order being T1;T2;T3)