# CS-245 Database System Principles – Winter 2004 Assignment 6

# Solution

*Due at the beginning of class on Thursday, February 26th*

- State all assumptions and show all work.
- You can email questions to cs245-staff@cs.stanford.edu

---

**Problem 1 (20 points)**

For each of the following schedules:

a) $S_a = r_1(A); r_2(B); w_1(B); w_2(C); r_3(C); w_3(A);$
b) $S_b = r_1(A); r_2(A); r_1(B); r_2(B); r_3(A); r_4(B); w_1(A); w_2(B);$

Answer the following questions:

i. What is the precedence graph for the schedule?
ii. Is the schedule conflict-serializable? If so, what are all the equivalent serial schedules?

*Answer:*

*a) i. T2 → T1, T2 → T3, T1 → T3.*
*ii. Yes – equivalent schedules: T2 → T1 → T3.*
*Common mistakes:*
*Omitting T2→T3 as redundant without showing that it existed (no points were taken off since the professor said in class it is possible). While you can reduce the graph, the arc is based on different variables from the T1/T2 dependency arcs, and thus you should have showed your understanding by identifying the arc before reducing the graph.*

*b) i. T2 → T1, T3 → T1, T1 → T2, T4 → T2*
*ii. No – there are cycles in the precedence graph (T2 → T1, T1 → T2)*

**Problem 2 (15 points)**

In a lock table, the system keeps a "group mode" that records the "strongest" lock type of the transactions that have currently locked an object. In particular, say object *O* is locked in modes $M_1, \ldots M_j$ and let the group mode of *O* be *GM(O)*. Then, for any possible lock mode N,

- *GM(O) and N are not compatible if and only if there is an $M_i$ element of $\{M_1, \ldots, M_j\}$ such that N and $M_i$ are not compatible;*

- *GM(O) and N are compatible if and only if for all $M_i$ element of $\{M_1, ..., M_j\}$, N and $M_i$ are compatible.*

When a new lock request arrives, for lock mode *N*, the system can simply check if *N* is compatible with *GM(O),* instead of checking *N* against all locks currently held on object *O*.
Consider the multiple-granularity locking mechanism (Section 9.6 [18.6]). In each of sub-problems (a) through (e) below, the modes of currently held locks on an object *O* are given. (For instance, in case (b), object *O* is locked in mode *IX* by two transactions and in mode *IS* by two transactions.) For each case, give the group mode, if there is one. (Be careful, some of the cases below are impossible! In those cases, just say there is no group mode and explain why it is so).

a) SIX, IS
b) IX, IS, IX, IS
c) S, IS, IX, SIX
d) S, IS
e) IX, S, IS

*Answer:*

*a) SIX*
*b) IX*
*c) No group mode (impossible S, IX and SIX can't coexist)*
*d) S*
*e) No group mode (impossible S, IX can't coexist)*

**Problem 3 (24 points)**

Consider the following two transactions:
T1 = w1(C) r1(A) w1(A) r1(B) w1(B);
T2 = r2(B) w2(B) r2(A) w2(A)
Say our scheduler performs exclusive locking only (i.e., no shared locks). For each of the following three instances of transactions T1 and T2 annotated with lock and unlock actions, say whether the annotated transactions:

1. obey two-phase locking,
2. will necessarily result in a conflict serializable schedule (if no deadlock occurs),
3. will necessarily result in a recoverable schedule (if no deadlock occurs),
4. will necessarily result in a schedule that avoids cascading rollback (if no deadlock occurs),
5. will necessarily result in a strict schedule (if no deadlock occurs),
6. will necessarily result in a serial schedule (if no deadlock occurs), and
7. may result in a deadlock.

a)
T1 = **L1**(B) **L1**(C) w1(C) **L1**(A) r1(A) w1(A) r1(B) w1(B) Commit **U1**(A) **U1**(C) **U1**(B)
T2 = **L2**(B) r2(B) w2(B) **L2**(A) r2(A) w2(A) Commit **U2**(A) **U2**(B)

b)
T1 = **L1**(C) **L1**(A) r1(A) w1(C) w1(A) **L1**(B) r1(B) w1(B) **U1**(A) **U1**(C) **U1**(B) Commit
T2 = **L2**(B) r2(B) w2(B) **L2**(A) r2(A) w2(A) Commit **U2**(A) **U2**(B)
c)
T1 = **L1**(C) w1(C) **L1**(A) r1(A) w1(A) **L1**(B) r1(B) w1(B) Commit **U1**(A) **U1**(C) **U1**(B)
T2 = **L2**(B) r2(B) w2(B) **L2**(A) r2(A) w2(A) Commit **U2**(A) **U2**(B)

Format your answer in a table with Yes/No entries.

*Answer:*

| | *2PL* | *Necessarily conflict Serializable* | *Necessarily recoverable* | *Necessarily ACR* | *Necessarily Strict Schedule* | *Necessarily Serial Schedule* | *May Result in Deadlock* |
|---|---|---|---|---|---|---|---|
| *a)* | *Y* | *Y* | *Y* | *Y* | *Y* | *Y* | *N* |
| *b)* | *Y* | *Y* | *N* | *N* | *N* | *Y*[*] | *Y* |
| *c)* | *Y* | *Y* | *Y* | *Y* | *Y* | *N* | *Y* |

[*]*Any non-serial schedule will result in deadlock. Notice that a schedule like <L1(C); L2(B); …L2 executes to the end; L1(A); …L1 executes to the end> is (of course) legal* ***but also serial*** *since the actions of T1 never started. The locks are not part of the transaction, only the scheduler.*
*The schedule*
*<L1(C); … ; U1(B); L2(B); … ;U2(B); CommitT1> (T1 executes but does not commit until after T2 is done) was considered for this question to be* ***serial*** *for a similar reason - we only asked you to look at the reads/write actions (i.e., un-committed reads were allowed), so a commit does not change the serializeability of the transactions.*

**Problem 4 (20 points)**

In the following sequences of events, we use $R_i(X)$ to mean "transaction $T_i$ starts, and its read set is the list of the database elements $X$." Also, $V_i$ means "$T_i$ attempts to validate," and $W_i(X)$ means that "$T_i$ **finishes**, and its write set was $X$.". State what happens when each sequence is processed by a validation-based scheduler. In particular, state which set intersections are performed for each $V_j$ action and indicate if the validation is successful or not.

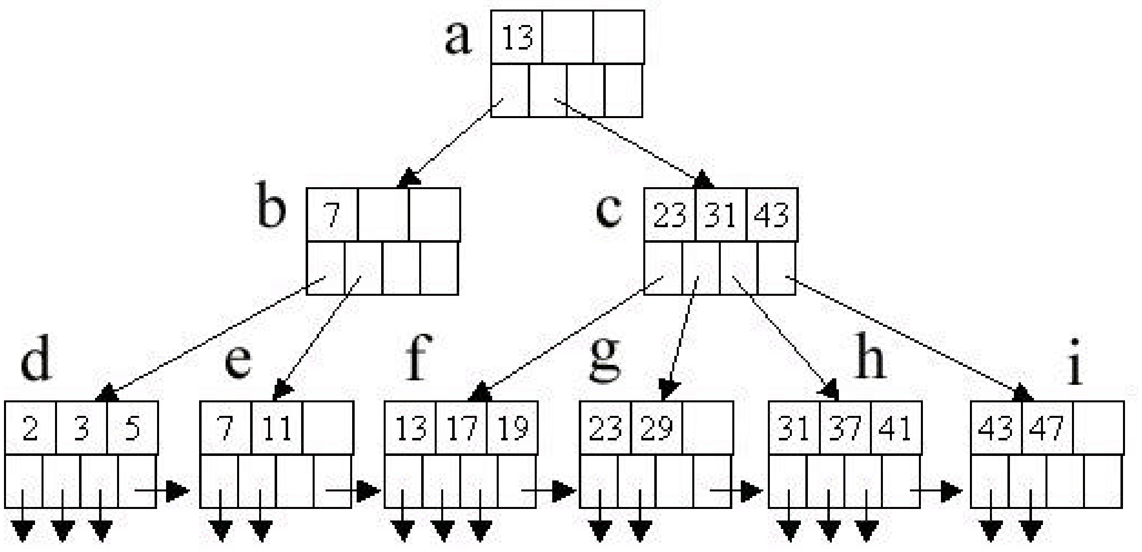
a) $R_1$(A,B); $R_2$(B,C); $R_3$(C); $V_1$; $V_2$; $V_3$; $W_1$(A); $W_2$(B); $W_3$(C);
b) $R_1$(A,B); $R_2$(B,C); $R_3$(B); $V_1$; $V_2$; $W_1$(C); $V_3$; $W_2$(B); $W_3$(C);

*Answer:*
*a)*
<u>*Validation of T1*</u>
*There are no other validated transactions, so* ***T1 validates****.*
<u>*Validation of T2*</u>
*T1 is validated but not finished. FIN(T1) > START(T2), so we need to check*
*RS(T2) Ç WS(T1) = {B,C} Ç {A} = {}*

*FIN(T1) > VAL(T2), so we need to check*
*WS(T2) Ç WS(T1) = {B} Ç{A}  = {}*
*Intersections are empty, so* ***T2 validates****.*
*<u>Validation of T3</u>*
*T1 is validated before but not finished. FIN(T1) > START(T3), so we need to check*
*RS(T3) Ç WS(T1) = {C} Ç {A}  = {}*
*FIN(T1) > VAL(T3), so we need to check:*
*WS(T3) Ç WS(T1) = {C} Ç {A}  = {}*
*T2 is validated before but not finished. FIN(T2) > START(T3), so we need to check*
*RS(T3)  Ç WS(T2) = {C} Ç{B}  = {}*
*FIN(T2) > VAL(T3), so we need to check*
*WS(T3) Ç WS(T2) = {C} Ç{B}  = {}*
*All intersections are empty, so* ***T3 validates****.*

*b)*
*<u>Validation of T1</u>*
*There are no other validated transactions, so* ***T1 validates****.*
*<u>Validation of T2</u>*
*T1 is validated before but not finished. FIN(T1) > START(T2), so we need to check*
*RS(T2) Ç WS(T1) = {B,C} Ç  {C}  = {C}*
*Since the intersection is not empty, validation of T2 fails. So* ***T2 is rolled back*** *and does not write the value for B.*
*<u>Validation of T3</u>*
*T1 is validated and finished. FIN(T1) > START(T3) so we need to check*
*RS(T3) Ç WS(T1) = {B} Ç {C}  = {}*
*FIN(T1) < VAL(T3) so we do not need to check the write sets. Notice that if you do, you get an erroneous rollback of T3.*
***T3 validates****.*

**Problem 5 (20 points)**



Suppose we perform the following actions on the B-tree given in the figure. Assume there is only one kind of lock available (i.e., no read/write/update locks). If we use the tree protocol, when can we release the lock on each of the nodes accessed? Note that we would like to unlock a node as early as possible to maximize concurrency. We also would like to maximize throughput; i.e., releasing a higher-level node has priority over releasing a lower level node. Use the notation L(node), UL(node), R(node), W(node) to indicate locking, unlocking, reading and writing a node respectively. Use Create(node) to indicate creation of a new node, if necessary. List the actions in the order they occur, and add short explanations if necessary. E.g., start part a. with the sequence
Lock and read root a: L(a) R(a)
Lock and read node b: L(b) R(b)
…
The actions to be performed:

a. Delete 5 (assume records are borrowed from right sibling if node underfull)
b. Insert 39 (assume overfull nodes are split)

*Answer:*
*a.*

| *Explanation* | *Actions* |
|---|---|
| *Lock and read Root a* | *L(a) R(a)* |
| *Lock and read node b* | *L(b) R(b)* |
| *Lock and read leaf d* | *L(d) R(d)* |
| *If we delete 5 from node d, there are enough records left in the node. An update to higher levels is not necessary.* | *UL(a) UL(b)* |
| *Update node d and release its lock.* | *W(d) UL(d)* |

*b.*

| *Explanation* | *Actions* |
|---|---|
| *Lock and read Root a* | *L(a) R(a)* |
| *Lead and read node c* | *L(c) R(c)* |
| *Lock and read node h* | *L(h) R(h)* |
| *Since h is full, we have to split it and create a new leaf j, which will hold 39 and 41. The writing of j can be performed after updating parent node c, so we can release the lock on node c as early as possible. Node j will be buffered in memory.* | *Create(j) L(j)* |
| *New entry has to be inserted in node c. Since it's full, it needs to be split and a new node k has to be created. Now node c will contain keys 23,31 and node k will contain key 43. 39 will be moved up to the root.* | *Create(k) L(k)* |
| *Update root node a to contain 13 and 39 and release its lock* | *W(a) UL(a)* |
| *Update node c and release its lock* | *W(c) UL(c)* |
| *Update new node k and release its lock* | *W(k) UL(k)* |
| *Update node h and release its lock* | *W(h) UL(h)* |
| *Update node j and release its lock* | *W(j) UL(j)* |

*It is possible to reorder this schedule slightly and produce an answer that was acceptable. The main key is that the root node (a) should be written and unlocked before any other node is written. Similarly, the internal nodes (c) and (k) should be written and unlocked before the leaf nodes (h) and (j) are written. You can place the Create(j) and Create(k) actions in several different places to achieve this as long as the schedule is correct; the important thing is that you create and lock (j) and (k) but do not write to them until the end. This of course assumes that Create(x) is cheap compared to W(x). If Create(x) is expensive, then it is important to defer block creation (specifically Create(j)) for as long as possible.*

**Problem 6 (1 point)**
Name a movie star mentioned in the book, or any other movie star.
*Answer:*

*Clint Eastwood. See www.imdb.com for more possible movie stars. Hector Garcia-Molina was also an accepted answers.*