CARNEGIE MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
15-445/645 – DATABASE SYSTEMS (FALL 2022)
PROF. ANDY PAVLO

Homework #4 (by Joyce Liao) – Solutions

Due: **Sunday Nov 13, 2022 @ 11:59pm**

IMPORTANT:

- Enter all of your answers into **Gradescope by 11:59pm on Sunday Nov 13, 2022**.
- **Plagiarism**: Homework may be discussed with other students, but all homework is to be completed **individually**.

For your information:

- Graded out of **100** points; **4** questions total
- Rough time estimate: ≈ 2 - 4 hours (0.5 - 1 hours for each question)

Revision : 2022/11/21 10:47

Question	Points	Score
Serializability and 2PL	27	
Deadlock Detection and Prevention	35	
Hierarchical Locking	20	
Optimistic Concurrency Control	18	
Total:	100	

Question 1: Serializability and 2PL............................[27 points]

(a) True/False Questions:

i. **[2 points]** Cascading aborts do not happen under rigorous 2PL.
■ **True** ☐ False

ii. **[2 points]** A schedule that is view serializable is also conflict serializable.
☐ True ■ **False**

iii. **[2 points]** 2PL is a pessimistic concurrency control protocol.
■ **True** ☐ False

iv. **[2 points]** Deadlocks cannot happen in non-rigorous 2PL.
☐ True ■ **False**

v. **[2 points]** There are not dirty reads in non-rigorous 2PL.
☐ True ■ **False**

vi. **[2 points]** A schedule with an acyclic precedence graph is view serializable.
■ **True** ☐ False

Grading info: -2 for each incorrect answer

(b) Serializability:
Consider the schedule given below in Table 1. R(·) and W(·) stand for 'Read' and 'Write', respectively.

time	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8	t_9	t_{10}
T_1			W(E)		R(D)				W(A)	
T_2		W(B)		R(A)						
T_3								R(E)		W(C)
T_4	W(D)					R(B)	R(C)			

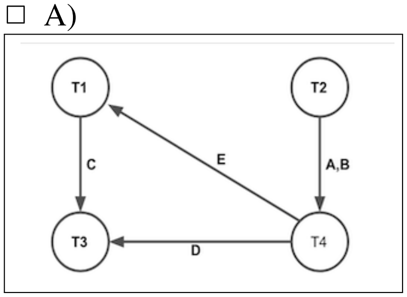
Table 1: A schedule with 4 transactions

i. **[2 points]** Is this schedule serial?
☐ Yes ■ **No**

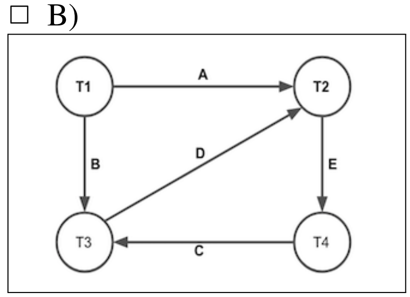
Grading info: -2 for incorrect answer

ii. **[3 points]** Choose the correct dependency graph of the schedule given above. Each edge in the dependency graph looks like this: '$T_x \rightarrow T_y$ with Z on the arrow indicating that there is a conflict on Z where T_x read/wrote on Z before T_y'.

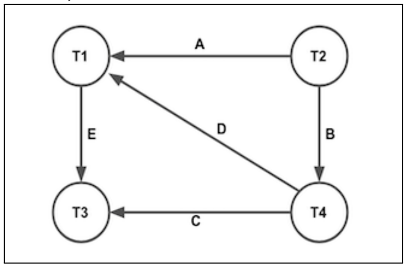
☐ A)



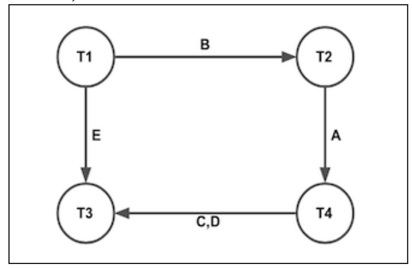
☐ B)



■ **C)**



☐ D)



Grading info: -3 for incorrect answer.

iii. **[2 points]** Is this schedule conflict serializable?

■ **Yes** ☐ No

Grading info: -2 for incorrect answer

iv. **[4 points]** What is the minimum number of transactions that need to be removed to produce a conflict serializable schedule?

☐ Two (`T1 and T4`)
☐ One (`T1`)
☐ One (`T4`)
■ **Zero (`original schedule is already serializable`)**

Grading info: -2 for incorrect answer

v. **[4 points]** Which of the following serial schedules is conflict equivalent to the given schedule? Mark all that apply.

☐ `T1,T2,T3,T4`
■ **`T2,T4,T1,T3`**
☐ `T2,T3,T1,T4`
☐ None of the above

Grading info: +4 for correct answer, -2 for each incorrect option

Question 2: Deadlock Detection and Prevention ....[35 points]

(a) **Deadlock Detection:**
Consider the following transactions and note that

- S(·) and X(·) stand for 'shared lock' and 'exclusive lock', respectively.
- T_1 and T_2 represent two transactions.
- Transactions will never release a granted lock until the last operation shown for the transaction is completed.

T_1: (a) read(A); (b) write(B); (c) read(C); (d) read(B); (e) read(A); (f) write(C);

i. For each position in T_1 above, which lock should be requested?

α) **[1 point]** At (a): ■ **S(A)** □ X(A) □ No lock needs to be requested

β) **[1 point]** At (b): □ S(B) ■ **X(B)** □ No lock needs to be requested

γ) **[1 point]** At (c): ■ **S(C)** □ X(C) □ No lock needs to be requested

δ) **[1 point]** At (d): □ S(B) □ X(B) ■ **No lock needs to be requested**

ϵ) **[1 point]** At (e): □ S(A) □ X(A) ■ **No lock needs to be requested**

ζ) **[1 point]** At (f): □ S(C) ■ **X(C)** □ No lock needs to be requested

ii. **[4 points]** Which of the following schedules can cause a deadlock? Mark all that apply.

□

T_1	S(A)	S(B)				S(C)
T_2			S(A)	X(B)	X(C)	

□

T_1	S(A)	X(C)		X(A)		
T_2			S(B)		X(B)	X(A)

■

T_1	S(A)			S(B)	S(C)	
T_2		X(B)	S(C)			X(A)

(b) Consider the following lock requests in Table 2. And note that

- S(·) and X(·) stand for 'shared lock' and 'exclusive lock', respectively.
- T_1, T_2, and T_3 represent three transactions.
- LM stands for 'lock manager'.
- Transactions will never release a granted lock.

time	t_1	t_2	t_3	t_4	t_5	t_6
T_1	S(A)					S(C)
T_2			S(B)			
T_3		X(B)		X(C)	X(A)	
LM	g					

Table 2: Lock requests of three transactions

i. For the lock requests in Table 2, determine which lock will be granted or blocked by the lock manager. Please write 'g' in the LM row to indicate the lock is granted and 'b' to indicate the lock is blocked. For example, in the table, the first lock (S(A) at time t_1) is marked as granted.

α) **[1 point]** At t_2: ■ **g** □ b

Grading info: -1 for incorrect answer

β) **[1 point]** At t_3: □ g ■ **b**

Solution: There is already an exclusive lock on B by T3.

γ) **[1 point]** At t_4: ■ **g** □ b

Grading info: -1 for incorrect answer

δ) **[1 point]** At t_5: □ g ■ **b**

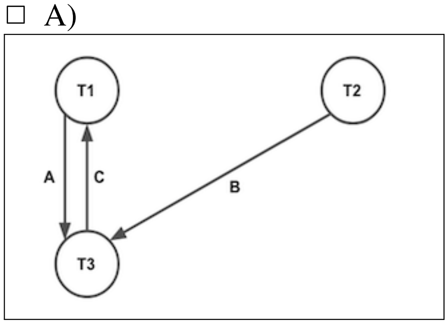
Solution: There is already a shared lock on A by T1.

ϵ) **[1 point]** At t_6: □ g ■ **b**

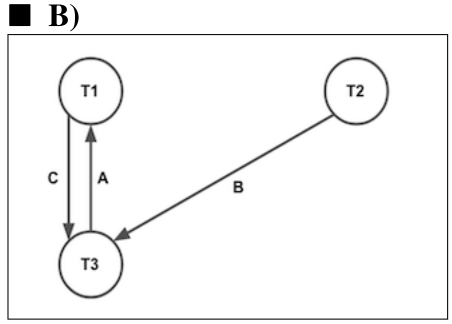
Solution: There is already an exclusive lock on C by T3

ii. **[4 points]** Mark the correct wait-for graph for the lock requests in Table 2. Each edge in the wait-for graph looks like this: $T_x \rightarrow T_y$ because of Z. Z is denoted in the arrow in the figure. (i.e., T_x is waiting for T_y to release its lock on resource Z).

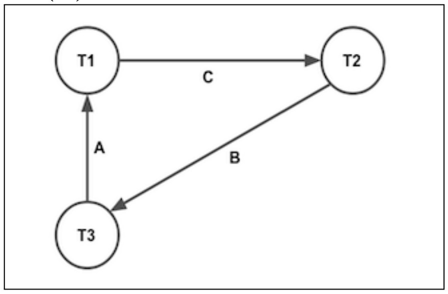
□ A)



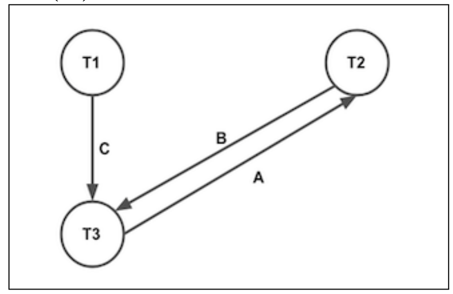
■ **B)**



□ (C)



□ (D)



Grading info: -4 for incorrect answer.

iii. **[2 points]** Determine whether there exists a deadlock in the lock requests in Table 2.

□ There is no deadlock

□ Cycle ($T_1 \rightarrow T_2 \rightarrow T_3$) exists and schedule deadlocks

■ **Cycle ($T_1 \rightarrow T_3 \rightarrow T_1$) exists and schedule deadlocks**

□ Cycle ($T_2 \rightarrow T_3 \rightarrow T_2$) exists and schedule deadlocks

Grading info: -2 for incorrect answer

(c) **Deadlock Prevention:**
Consider the following lock requests in Table 3.
Like before,

- S(·) and X(·) stand for 'shared lock' and 'exclusive lock', respectively.
- T_1, T_2, T_3, T_4, and T_5 represent five transactions.
- LM represents a 'lock manager'.
- Transactions will never release a granted lock.

time	t_1	t_2	t_3	t_4	t_5	t_6	t_7	t_8
T_1							X(C)	
T_2	S(A)		S(B)			S(C)		
T_3		X(A)		X(B)				
T_4					X(C)			X(B)
LM	g							

Table 3: Lock requests of four transactions

i. To prevent deadlock, we use a lock manager (LM) that adopts the Wait-Die policy. We assume that in terms of priority: $T_1 > T_2 > T_3 > T_4$. Here, $T_1 > T_2$ because T_1 is older than T_2 (i.e., older transactions have higher priority). *Determine whether the lock request is granted ('g'), blocked ('b'), aborted ('a'), or already dead('–').*

α) **[1 point]** At t_2: ☐ g ☐ b ■ **a** ☐ –

Solution: T_2 already holds shared lock on A. T_3 is younger than T_2 so it aborts.

β) **[1 point]** At t_3: ■ **g** ☐ b ☐ a ☐ –

γ) **[1 point]** At t_4: ☐ g ☐ b ☐ a ■ **–**

Solution: T_3 aborted at t_2.

δ) **[1 point]** At t_5: ■ **g** ☐ b ☐ a ☐ –

ϵ) **[1 point]** At t_6: ☐ g ■ **b** ☐ a ☐ –

Solution: T_4 already holds exclusive lock on C. T_2 is older than T_4 so it waits.

ζ) **[1 point]** At t_7: ☐ g ■ **b** ☐ a ☐ –

Solution: T_4 already holds exclusive lock on C. T_1 is older than T_4 so it waits.

η) **[1 point]** At t_8: ☐ g ☐ b ■ **a** ☐ –

Solution: T_2 already holds shared lock on B. T_4 is younger than T_2 so it aborts.

Grading info: -1 for each incorrect answer

ii. Now we use a lock manager (LM) that adopts the Wound-Wait policy. We assume that in terms of priority: $T_1 > T_2 > T_3 > T_4$. Here, $T_1 > T_2$ because T_1 is older than T_2 (i.e., older transactions have higher priority). *Determine whether the lock request is granted ('g'), blocked at the current timestamp('b_{curr}'), blocked at a* *previous timestamp ('b_{prev}'), granted by aborting another transaction ('k'), or the requester is already dead('–').* Follow the same format as the previous question.

α) **[1 point]** At t_2: ☐ g ■ b_{curr} ☐ b_{prev} ☐ k ☐ –

Solution: T_2 already holds shared lock on A. T_3 is younger than T_2 so it waits.

β) **[1 point]** At t_3: ■ **g** ☐ b_{curr} ☐ b_{prev} ☐ k ☐ –

γ) **[1 point]** At t_4: ☐ g ☐ b_{curr} ■ b_{prev} ☐ k ☐ –

Solution: T_3 was blocked at t_4.

δ) **[1 point]** At t_5: ■ **g** ☐ b_{curr} ☐ b_{prev} ☐ k ☐ –

ϵ) **[1 point]** At t_6: ☐ g ☐ b_{curr} ☐ b_{prev} ■ **k** ☐ –

Solution: T_4 already holds exclusive lock on C. T_2 is older than T_4 so it causes T_4 to abort.

ζ) **[1 point]** At t_7: ☐ g ☐ b_{curr} ☐ b_{prev} ■ **k** ☐ –

Solution: T_2 already holds exclusive lock on C. T_1 is older than T_2 so it causes T_2 to abort.

η) **[1 point]** At t_8: ☐ g ☐ b_{curr} ☐ b_{prev} ☐ k ■ **–**

Solution: T_4 was aborted at t_6.

Question 3: Hierarchical Locking .... [20 points]

Consider a database (D) consisting of two tables, Release (R) and Artists (A). Specifically,

- `Release(rid, name, artist_credit, language, status, genre, year, number_sold)`, spans 1000 pages, namely R_1 to R_{1000}
- `Artists(id, name, type, area, gender, begin_date_year)`, spans 50 pages, namely A_1 to A_{50}

Further, **each page contains 100 records**, and we use the notation $R_3 : 20$ to represent the 20^{th} record on the third page of the Release table. Similarly, $A_5 : 10$ represents the 10^{th} record on the fifth page of the Artists table.

We use Multiple-granularity locking, with **S, X, IS, IX** and **SIX** locks, and **four levels of granularity**: (1) *database-level (D)*, (2) *table-level (R, A)*, (3) *page-level ($R_1 - R_{1000}$, $A_1 - A_{50}$)*, (4) *record-level ($R_1 : 1 - R_{1000} : 100$, $A_1 : 1 - A_{50} : 100$)*.

For each of the following operations on the database, check all the sequence of lock requests based on intention locks that should be generated by a transaction that wants to efficiently carry out these operations by maximizing concurrency. Please take care of efficiency for e.g., share vs. exclusive lock and granularity.

Please follow the format of the examples listed below:

- mark **"IS(D)"** for a request of **database-level IS lock**
- mark **"X(A_2 : 30)"** for a request of **record-level X lock for the 30^{th} record on the second page of the Artists table**
- mark **"S($A_2 : 30 - A_3 : 100$)"** for a request of **record-level S lock from the 30^{th} record on the second page of the Artists table to the 100^{th} record on the third page of the Artists table**.

(a) **[4 points]** Fetch the record that represents the artist with the earliest `begin_date_year`. Assume there are no ties.

☐ IX(D), X(A)

■ **IS(D), S(A)**

☐ S(D)

☐ IS(D), SIX(A), X(A_1)

Grading info: -4 for incorrect answer

(b) **[4 points]** Modify the 4^{th} record on R_{45}.

■ **IX(D), IX(R), IX(R_{45}), X($R_{45} : 4$)**

☐ IX(D), SIX(R), X(R_{45})

☐ IX(D), IX(R), SIX(R_{45}), X($R_{45} : 4$)

☐ IS(D), IS(R), IX(R_{45}), X($R_{45} : 4$)

Grading info: -4 for incorrect answer

(c) **[4 points]** Find the average `number_sold` across all releases.

☐ S(D)

- ☐ `S(D), S(R)`
- ☐ `IS(D), X(R)`
- ■ **`IS(D), S(R)`**

Grading info: -4 for incorrect answer

(d) **[4 points]** Scan all records in `R` and modify the 1^{st} record on R_5

- ■ **IX(D), SIX(R), IX(R_5), X($R_5 : 1$)**
- ☐ IS(D), S(R), IX($R_5 : 1$)
- ☐ S(D), IX(R), X($R_5 : 1$)
- ☐ IS(D), SIX(R), X(R_5)

Grading info: -4 for incorrect answer

(e) **[4 points]** Update the `name` of all artists to be in uppercase letters only.

- ☐ `SIX(D), X(A)`
- ☐ `SIX(D), S(A)`
- ■ **`IX(D), X(A)`**
- ☐ `IX(D), SIX(A)`

Grading info: -4 for incorrect answer

Question 4: Optimistic Concurrency Control .... [18 points]

Consider the following set of transactions accessing a database with object *A, B, C, D*. The questions below assume that the transaction manager is using **optimistic concurrency control** (OCC). Assume that a transaction switches from the READ phase immediately into the VALIDATION phase after its last operation executes.

Note: VALIDATION may or may not succeed for each transaction. If validation fails, the transaction will get immediately aborted.

You can assume that the DBMS is using the serial validation protocol discussed in class where only one transaction can be in the validation phase at a time, and each transaction is doing backward validation (i.e. Each transaction, when validating, checks whether it intersects its read/write sets with any transactions that have already committed. You may assume there are no other transactions in addition to the ones shown below.)

time	T_1	T_2	T_3
1	READ(A)		
2	READ(C)		
3		READ(B)	
4	WRITE(A)		
5			READ(B)
6	WRITE(C)		
7	VALIDATE?		
8		READ(D)	
9	WRITE?		
10		WRITE(D)	
11		WRITE(B)	
12		VALIDATE?	
13			READ(A)
14		WRITE?	
15			WRITE(A)
16			WRITE(B)
17			VALIDATE?
18			WRITE?

Figure 1: An execution schedule

(a) **[4 points]** Will T1 abort?
- ☐ Yes
- ■ **No**

Solution: T_1 does not need to abort because no other transactions have commited yet.

Grading info: -4 for incorrect answer

(b) **[4 points]** Will T2 abort?
- ☐ Yes
- ■ **No**

Solution: T_2's write-set does not intersect with T_1's write-set.

Grading info: -4 for incorrect answer

(c) **[4 points]** Will T3 abort?
■ **Yes**
☐ No

Solution: T_3's write-set intersects with T_1's write-set and T_2's write-set, so it will fail the VALIDATE phase.

Grading info: -4 for incorrect answer

(d) **[2 points]** OCC works best when concurrent transactions access the same subset of data in a database.
☐ True ■ **False**

Solution: OCC is good to use when the number of conflicts is low.

Grading info: -2 for incorrect answer

(e) **[2 points]** Transactions can suffer from *phantom reads* in OCC.
■ **True** ☐ False

Grading info: -2 for incorrect answer

(f) **[2 points]** Aborts due to OCC are wasteful because they happen after a transaction has already finished executing.
■ **True** ☐ False

Grading info: -2 for incorrect answer