CARNEGIE MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
15-445/645 – DATABASE SYSTEMS (FALL 2024)
PROF. ANDY PAVLO

Homework #5 (by William)

Due: **Sunday November 17, 2024 @ 11:59pm**

IMPORTANT:

- Enter all of your answers into **Gradescope by 11:59pm on Sunday November 17, 2024**.
- **Plagiarism**: Homework may be discussed with other students, but all homework is to be completed **individually**.

For your information:

- Graded out of **100** points; **3** questions total
- Rough time estimate: ≈ 2 - 4 hours (0.5 - 1 hours for each question)

Revision : 2024/11/16 13:07

Question	Points	Score
Serializability, 2PL, Deadlock Prevention	42	
Hierarchical Locking	28	
Optimistic Concurrency Control	30	
Total:	100	

Question 1: Serializability, 2PL, Deadlock Prevention . . . . [42 points]

(a) True/False Questions:

i. **[2 points]** Cascading aborts is possible under Strong strict Two-Phase Locking (2PL).
☐ True ☐ False

ii. **[2 points]** Using 2PL guarantees a conflict-serializable schedule.
☐ True ☐ False

iii. **[2 points]** For a schedule following strong strict 2PL, the dependency graph is guaranteed to be acyclic.
☐ True ☐ False

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

v. **[2 points]** Dirty reads are possible under conflict-serializable schedules.
☐ True ☐ False

Question 1 continues. . .

(b) Serializability:

Consider the schedule of 4 transactions in Table 1. R(·) and W(·) stand for 'Read' and 'Write', respectively, and time increases from left to right. (This is in contrast to the diagrams in class, where time proceeded downward.)

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

Table 1: A schedule with 4 transactions

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

ii. **[6 points]** Compute the conflict dependency graph for the schedule in Table 1, selecting all edges that appear in the graph.

☐ $T_1 \rightarrow T_2$	☐ $T_2 \rightarrow T_1$	☐ $T_3 \rightarrow T_1$	☐ $T_4 \rightarrow T_1$
☐ $T_1 \rightarrow T_3$	☐ $T_2 \rightarrow T_3$	☐ $T_3 \rightarrow T_2$	☐ $T_4 \rightarrow T_2$
☐ $T_1 \rightarrow T_4$	☐ $T_2 \rightarrow T_4$	☐ $T_3 \rightarrow T_4$	☐ $T_4 \rightarrow T_3$

iii. **[2 points]** Is this schedule conflict-serializable?
☐ Yes ☐ No

iv. **[2 points]** Is this schedule possible under regular 2PL?
☐ Yes ☐ No

v. **[4 points]** Is this schedule view-serializable?
☐ Yes ☐ No

Question 1 continues…

(c) **Deadlock Prevention:**
Consider the following lock requests in Table 2.
Like before,

- S(·) and X(·) stand for 'shared lock' and 'exclusive lock', respectively.
- T_1, T_2, T_3, and T_4 represent four 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(A)						
T_2				X(C)				
T_3	X(A)		S(C)		X(B)			X(D)
T_4						S(B)	X(C)	
LM	g							

Table 2: 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, $\overline{T_1 > T_2}$ 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 ☐ –
β) **[1 point]** At t_3: ☐ g ☐ b ☐ a ☐ –
γ) **[1 point]** At t_4: ☐ g ☐ b ☐ a ☐ –
δ) **[1 point]** At t_5: ☐ g ☐ b ☐ a ☐ –
ϵ) **[1 point]** At t_6: ☐ g ☐ b ☐ a ☐ –
ζ) **[1 point]** At t_7: ☐ g ☐ b ☐ a ☐ –
η) **[1 point]** At t_8: ☐ g ☐ b ☐ a ☐ –

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, $\overline{T_1 > T_2}$ 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'), 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 ☐ k ☐ –
β) **[1 point]** At t_3: ☐ g ☐ b ☐ k ☐ –
γ) **[1 point]** At t_4: ☐ g ☐ b ☐ k ☐ –
δ) **[1 point]** At t_5: ☐ g ☐ b ☐ k ☐ –
ϵ) **[1 point]** At t_6: ☐ g ☐ b ☐ k ☐ –
ζ) **[1 point]** At t_7: ☐ g ☐ b ☐ k ☐ –
η) **[1 point]** At t_8: ☐ g ☐ b ☐ k ☐ –

Question 1 continues…

iii. **[2 points]** If a transaction is aborted because of the DBMS's deadlock prevention policy, then that transaction keeps its *original timestamp* when restarting.
☐ True ☐ False

Homework #5 continues…

Question 2: Hierarchical Locking .... [28 points]

Consider a database `D` consisting of two tables `A` (which stores information about musical artists) and `R` (which stores information about the artists' releases). Specifically:

- `R(rid, name, artist_credit, language, status, genre, year, number_sold)`
- `A(id, name, type, area, gender, begin_date_year)`

Table `R` spans 1000 pages, which we denote `R1` to `R1000`. Table `A` spans 50 pages, which we denote `A1` to `A50`. Each page contains 100 records. We use the notation `R3.20` to denote the twentieth record on the third page of table `R`. There are no indexes on these tables.

Suppose the database supports shared and exclusive hierarchical intention locks (`S`, `X`, `IS`, `IX` and `SIX`) at four levels of granularity: database-level (`D`), table-level (`R` and `A`), page-level (e.g., `R10`), and record-level (e.g., `R10.42`). We use the notation `IS(D)` to mean a shared database-level intention lock, and `X(A2.20-A3.80)` to mean a set of exclusive locks on the records from the 20th record on the second page to the 80th record on the third page of table `A`.

For each of the following operations below, what sequence of lock requests should be generated to **maximize the potential for concurrency** while guaranteeing correctness?

(a) **[4 points]** Edit the `begin_date_year` for the 24^{th} record on `A40`.
☐ `IX(D), IX(A), IX(A40), X(A40.24)`
☐ `IX(D), IX(A), SIX(A40), X(A40.24)`
☐ `IS(D), IS(A), IS(A40), S(A40.24)`
☐ `IX(D), IX(A), S(A40), X(A40.24)`

(b) **[4 points]** Fetch the records of all releases in `R` with `genre = 'Metal'`.
☐ `SIX(D), S(R)`
☐ `IS(D), S(R)`
☐ `S(D)`
☐ `IX(D), S(R)`

(c) **[4 points]** Update the `status` for all release records with `year = 2023` to `'Finished'`.
☐ `IX(D), IX(R)`
☐ `IX(D), SIX(R)`
☐ `IX(D), X(R)`
☐ `SIX(D), X(R)`

(d) **[4 points]** Modify the 29^{th} record on `R42`.
☐ `IS(D), IS(R), IS(R42), X(R42.29)`
☐ `SIX(D), IX(R), IX(R42), X(R42.29)`
☐ `IX(D), IX(R), IX(R42), X(R42.29)`
☐ `IX(D), IX(R), IX(R42), IX(R42.29)`

(e) **[4 points]** Scan all records on pages `R1` to `R10` and modify the 12^{th} record on `R17`.
☐ `IX(D), S(R), X(R17)`
☐ `SIX(D), IX(R), IX(R17), X(R17.12)`
☐ `IX(D), IX(R), IX(R1-R10), IX(R17), X(R17.12)`
☐ `IX(D), SIX(R), IX(R17), X(R17.12)`

Question 2 continues...

(f) **[4 points]** Two users are trying to access data. User A is scanning all the records in `A` to read, while User B is trying to modify the 27th record in `R3`. Which of the following sets of locks are most suitable for this scenario?

☐ `User A: SIX(D), S(A), User B: SIX(D), IX(R), IX(R3), X(R3.27)`
☐ `User A: S(D), User B: X(D)`
☐ `User A: IS(D), S(A), User B: SIX(D), IX(R), IX(R3), X(R3.27)`
☐ `User A: IS(D), S(A), User B: IX(D), IX(R), IX(R3), X(R3.27)`

(g) **[4 points]** Delete records in `A` if `type='Orchestra'`.
☐ `IX(D), X(A)`
☐ `IX(D), IX(A)`
☐ `SIX(D), X(A)`
☐ `SIX(D), SIX(A)`

Question 3: Optimistic Concurrency Control . . . . [30 points]

Consider the following set of transactions accessing a database with object *A, B, C, D*. You should make the following assumptions:

- The transaction manager is using **optimistic concurrency control** (OCC).
- A transaction begins its read phase with its first operation and switches from the READ phase immediately into the VALIDATION phase after its last operation executes.
- The DBMS is using the serial validation protocol discussed in class where only one transaction can be in the validation phase at a time.
- Each transaction is doing **backwards validation** (i.e. Each transaction, when validating, checks whether it intersects its read/write sets with any transactions that have already committed).
- There are no other transactions in addition to the ones shown below.

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

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

Figure 1: An execution schedule

(a) **[4 points]** When is each transaction's timestamp assigned in the transaction process?
☐ The start of the read phase.
☐ The start of the validation phase.
☐ The start of the write phase.
☐ Timestamps are not necessary for OCC.

(b) **[4 points]** When time = 5, will T_2 read C written by T_1?
☐ Yes ☐ No

(c) **[4 points]** Will T1 abort?
☐ Yes
☐ No

Question 3 continues. . .

(d) **[4 points]** Will T2 abort?
☐ Yes
☐ No

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

(f) **[3 points]** OCC works best when concurrent transactions access disjoint sets of data in a database.
☐ True ☐ False

(g) **[3 points]** Transactions can suffer from *phantom reads* in OCC.
☐ True ☐ False

(h) **[2 points]** Aborts are less wasteful in OCC than under 2PL.
☐ True ☐ False

(i) **[2 points]** Transactions can perform *dirty reads* in OCC.
☐ True ☐ False