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

Homework #5 (by William) – Solutions

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: $\approx$ 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**

Solution: False. Strict 2PL prevents cascading aborts by holding all the locks until a transaction reaches its commit point, ensuring that other transactions do not see the intermediate, uncommitted data.

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

Solution: True. Using regular 2PL guarantees a conflict-serializable schedule because it generates schedules whose precedence graph is acyclic. This is because this ordering of acquiring resources (via the locks) ensures that there is a order of acquisition precedence.

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

Solution: True. Using regular 2PL guarantees a conflict-serializable schedule, and a schedule provided by strong strict 2PL has an even stronger guarantee which means it also avoids the formation of cycles in the dependency graph.

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

Solution: False. There exists view-serializable schedules that are not conflict-serializable.

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

Solution: True. Conflict-serializability ensures that the schedule is conflict equivalent to a serial schedule. However, dirty reads can still occur in conflict-serializable schedules, especially when considering cascading aborts. If a transaction reads data written by another transaction which later gets aborted, then the first transaction has effectively read "dirty" data.

(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**

Solution: This schedule isn't serial because this schedule interleaves the actions of different transactions.

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

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

Solution: The answer is:

- $T_1 \rightarrow T_2(A), T_1 \rightarrow T_3(A)$
- $T_3 \rightarrow T_1(B), T_3 \rightarrow T_2(A, C)$
- $T_4 \rightarrow T_1(B), T_4 \rightarrow T_2(D)$

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

Solution: This schedule is *not* conflict-serializable because there are cycles in its data dependency graph.

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

Solution: This schedule is not possible under 2PL because it is not conflict serializable, and 2PL is guaranteed to produce conflict serializable schedules.

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

Solution: This schedule isn't view serializable because this schedule is not view equivalent to any serial schedule of transaction execution.

(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, $T_1 > T_2$ 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_3 already holds exclusive lock on A. T_1 is older than T_3 so it waits.

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

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

Solution: T_3 already holds shared lock on C. T_2 is older than T_3 so it waits.

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

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

Solution: T_3 already holds exclusive lock on B. T_4 is younger than T_3 so it aborts.

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

Solution: T_4 is already dead since it got aborted at t_6.

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, $T_1 > T_2$ 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** ☐ –

Solution: T_3 already holds exclusive lock on A. T_1 is older than T_3 so it causes T_3 to abort.

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 ■ **–**

Solution: T_3 was aborted at t_2.

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

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

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

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

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

Solution: True. A txn is always executed with the same timestamp it was originally assigned. Otherwise it could get starved out by other txns that keep arriving.

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)

Solution: The correct choice is IX(D), IX(A), IX(A40), X(A40.24). This choice is correct because it accesses all intention locks and the exclusive lock on A40.24, so it can perform both a read and write while holding this exclusive lock.

- IX(D), IX(A), SIX(A40), X(A40.24) is incorrect because it gains a shared intention lock for A40 when it only needs to read A40.24, thus limiting potential for concurrency.
- IS(D), IS(A), IS(A40), S(A40.24) is incorrect because we plan on modifying A40.24, but we are only gaining a shared lock on A40.24, which isn't sufficient.
- IX(D), IX(A), S(A40), X(A40.24) is incorrect because while it gains a intention-exclusive locks for R and the database, it then gains a shared lock on A40 which is not sufficient to gain the exclusive lock on 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)

Solution: The correct answer choice is `IS(D), S(R)`. We need to scan records in table R to find records where `genre = 'Metal'`. This choice is correct because it accesses the intended shared parent lock to get the shared lock on table `R`.

- `SIX(D), S(R)` is incorrect because it gains a shared+intention-exclusive lock on the database `D` when it only needs to read from `R` and has no intention of modifying any records.
- `S(D)` is incorrect because it gains a shared lock on the entire database `D` when it only needs to fetch rows in table `R`.
- `IX(D), S(R)` is incorrect because it gains an intention-exclusive lock when it has no intention of modifying.

(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)`

Solution: The correct answer choice is `IX(D), X(R)`. This choice is correct because it accesses all intended locks and the exclusive lock on `R`, since we potentially need to modify all records in `R`.

- `IX(D), IX(R)` is incorrect because it does not gain an exclusive lock for the records of `R` it needs to delete.
- `IX(D), SIX(R)` is incorrect because it does not gain an exclusive lock for the records of `R` it needs to delete.
- `SIX(D), X(R)` is incorrect because it gains a shared intention parent lock for `D`, when it only needs to read from `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)`

Solution: The correct choice is `IX(D), IX(R), IX(R42), X(R42.29)`. This choice is correct because it accesses all intended exclusive locks for all parent levels necessary, and then accesses the exclusive lock for the particular record.

- `IS(D), IS(R), IS(R42), X(R42.29)` is incorrect because the DBMS intends to write to a tuple, so it should not grab the shared intention parent locks.
- `SIX(D), IX(R), IX(R42), X(R42.29)` is incorrect because the DBMS only

intends to write to a tuple, not read any data. Therefore it should not grab the shared-exclusive intention lock.

- `IX(D), IX(R), IX(R42), IX(R42.29)` is incorrect because it only gets the intention exclusive lock for the record.

(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)`

Solution: The correct choice is `IX(D), SIX(R), IX(R17), X(R17.12)`. This choice is correct because it accesses all intended locks and the exclusive lock `X(R17.12)`. It also gains a shared lock on `R`, so it can read pages `R1` to `R10`.

- `IX(D), S(R), X(R17)` is incorrect because it fails to gain a intention-exclusive lock for `R`.
- `SIX(D), IX(R), IX(R17), X(R17.12)` is incorrect because it gains a shared intention lock for the database `D` when it only needs to read from `R`.
- `IX(D), IX(R), IX(R1-R10), IX(R17), X(R17.12)` is incorrect because it fails to gain shared locks to read from `R1-R10`.

(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 27^{th} 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)`**

Solution: The correct choice is `User A: IS(D), S(A)` and `User B: IX(D), IX(R), IX(R3), X(R3.27)`. This choice is correct because:

- User A only intends to read all records in `R`. Thus, he acquires an intention shared lock on the database `D` and a shared lock on the table `R`.
- User B intends to modify a specific record in table `A`. He acquires an intention exclusive lock on the database `D`, an intention exclusive lock on the table `R`, and then an exclusive lock on the specific record `R3.27`.

Other choices are incorrect because:

- `User A: SIX(D), S(A)` and `User B: SIX(D), IX(R), IX(R3), X(R3.27)` is incorrect because User A doesn't need to acquire shared intention exclusive

locks for a mere read operation, and neither does User B for a specific record modification.

- `User A: S(D) and User B: X(D)` is problematic as it locks the entire database either in shared mode or exclusive mode, preventing concurrent operations.
- `User A: IS(D), S(A) and User B: SIX(D), IX(R), IX(R3), X(R3.27)` is incorrect because while User A's locks are appropriate for his read operation, User B does not need a shared intention exclusive lock on the database for modifying a specific record.

(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)`

Solution: The correct choice is `IX(D), X(A)`. This choice is correct because it accesses all intended locks and the exclusive lock on `A`, since we potentially need to modify all records in `A`.

- `IX(D), IX(A)` is incorrect because it does not gain an exclusive lock for the records of `A` it needs to delete.
- `SIX(D), X(A)` is incorrect because it gains a shared lock for the database, when it does not need to read any contents.
- `SIX(D), SIX(A)` is incorrect because it gains a shared intention parent lock for both `D` and `A`, when it does not need to read any contents, and it does not gain the exclusive lock for the records of `A` it needs to delete.

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.

Solution: Each transaction's timestamp is assigned at the beginning of the validation phase.

(b) **[4 points]** When time = 5, will T_2 read C written by T_1?

☐ Yes ■ **No**

Solution: In OCC, each transaction maintains a private workspace that is invisible to other transactions until its write phase is completed. Only transactions that start after T_1's write phase can see the value of C written by T_2, provided T_1 commits successfully. Hence, T_2 at time = 5 will read the original C, that isn't written by T_1.

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

Solution: T_1 will abort since it should end up reading the value written by T_2.

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

Solution: T_2 does not need to abort because no other transactions have been committed yet.

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

Solution: T_3 will not abort since it does not read anything in T_2's write set.

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

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

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

Solution: Considering the following sequence of events for table A and transactions T_1 and T_2:

- T_1 reads all tuples of A.
- T_2 inserts a new tuple into A.
- T_2 enters validation phase. Note T_2's write set contains the new tuple whereas T_2's read set only contains the initial existing tuples. So T_2's write set does not intersect T_1's read set, and T_2 validation succeeds.
- T_2 completes the write phase.
- T_1 reads all tuples of A again, now including the tuple added by T_2.

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

Solution: Under OCC, a transaction only aborts after the read and validation phase. In contrast, a transaction can abort much earlier in 2PL.

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

Solution: One txn cannot see the private workspace of another concurrent txns, so therefore it can **never** see dirty reads.