CARNEGIE MELLON UNIVERSITY
DEPARTMENT OF COMPUTER SCIENCE
15-445/645 – DATABASE SYSTEMS (FALL 2018)
PROF. ANDY PAVLO

Homework 4 (by David Gershuni) – Solutions

Due: **Monday Nov 12, 2018 @ 11:59pm**

IMPORTANT:

- **Upload this PDF** with your answers to **Gradescope by 11:59pm on Monday Nov 12, 2018**.
- **Plagiarism**: Homework may be discussed with other students, but all homework is to be completed **individually**.
- **You have to use this PDF for all of your answers.**

For your information:

- Graded out of **100** points; **4** questions total

Revision : 2018/11/22 11:07

Question	Points	Score
Serializability and 2PL	20	
Deadlock Detection and Prevention	30	
Hierarchical Locking	30	
Optimistic Concurrency Control	20	
Total:	100	

Question 1: Serializability and 2PL............................[20 points]

(a) Yes/No questions:

i. **[2 points]** In Strict 2PL, a transaction does not release any locks until it commits.
☐ Yes ■ **No**

ii. **[2 points]** A schedule generated by Strict 2PL will never cause a deadlock.
☐ Yes ■ **No**

iii. **[2 points]** A schedule generated by 2PL is always view serializable.
☐ Yes ■ **No**

iv. **[2 points]** A conflict serializable schedule will never contain a cycle in its precedence graph.
■ **Yes** ☐ No

v. **[2 points]** Every view serializable schedule is conflict serializable.
☐ Yes ■ **No**

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(B)		R(H)	R(I)	W(A)		R(K)	
T_2	R(A)		W(D)	W(E)		W(F)		W(J)		
T_3		R(C)			W(G)	R(D)	W(H)	R(B)	R(E)	R(G)

Table 1: A schedule with 3 transactions

i. **[1 point]** Is this schedule serial?
☐ Yes ■ **No**

Grading info: -1 for incorrect answer

ii. **[3 points]** Give the dependency graph of this schedule. List each edge in the dependency graph like this: '$T_x \rightarrow T_y$ because of Z'. This notation signifies that T_x precedes T_y because Z was last read/written by T_x before it was read/written by T_y. Order the edges in ascending order with respect to x.

Solution:

- $T_1 \rightarrow T_3$ because of H, B
- $T_2 \rightarrow T_1$ because of A
- $T_2 \rightarrow T_3$ because of D, E

Grading info: -1 for each missing/incorrect edge.

iii. **[1 point]** Is this schedule conflict serializable?
■ **Yes** ☐ No

Grading info: -1 for incorrect answer

iv. **[3 points]** If you answer "yes" to (iii), provide the equivalent serial schedule. If you answer "no", briefly explain why.

Solution: Yes, the equivalent serial schedule is T_2, T_1, T_3.

Grading info: -3 for a justification that does not agree with previous part

v. **[2 points]** Is this schedule possible under 2PL?

■ **Yes** □ No

Grading info: -1 for incorrect answer

Question 2: Deadlock Detection and Prevention .... [30 points]

(a) **Deadlock Detection:**
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_7
T_1			X(A)	S(B)			S(C)
T_2	S(B)				S(A)		
T_3		X(C)				X(B)	
LM	g						

Table 2: Lock requests of three transactions

i. **[3 points]** 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(D) at time t_1) is marked as granted.

Solution:

- X(C) at t_2: g
- X(A) at t_3: g
- S(B) at t_4: g
- S(A) at t_5: b
- X(B) at t_6: b
- S(C) at t_7: b

Grading info: Half points for one mistake in the schedule, no points >1 mistake.

ii. **[4 points]** Give the wait-for graph for the lock requests in Table 2. List each edge in the graph like this: $T_x \rightarrow T_y$ because of Z (i.e., T_x is waiting for T_y to release its lock on resource Z). Order the edges in ascending order with respect to x.

Solution:

- $T_1 \rightarrow T_3$ because of C
- $T_2 \rightarrow T_1$ because of A
- $T_3 \rightarrow T_1$ because of B

- $T_3 \rightarrow T_2$ because of B

Grading info: Half points for 1 missing directed edge, no points if missing > 1.

iii. **[3 points]** Determine whether there exists a deadlock in the lock requests in Table 2, and briefly explain why.

Solution: Deadlock exists because there is a cycle ($T_1 \rightarrow T_3 \rightarrow T_1$) in the dependency graph.
OR: Deadlock exists because there is a cycle ($T_1 \rightarrow T_3 \rightarrow T_2 \rightarrow T_1$) in the dependency graph.

Grading info: -2 points for not explaining why there is a deadlock

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

- S($\cdot$) and X($\cdot$) 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			S(B)	X(A)			X(B)	
T_2	X(A)							
T_3					X(C)			
T_4		S(B)						X(B)
T_5						X(C)		
LM	g							

Table 3: Lock requests of four transactions

i. **[3 points]** For the lock requests in Table 3, determine which lock request will be granted, blocked or aborted by the lock manager (LM), if it has no deadlock prevention policy. *Please write 'g' for grant, 'b' for block, 'a' for abort, and '$-$'if the transaction has already died.* Again, example is given in the first column.

Solution:

- S(B) at t_2: g
- S(B) at t_3: g
- X(A) at t_4: b
- X(C) at t_5: g

- X(C) at t_6: b
- X(B) at t_7: b
- X(B) at t_8: b

Grading info: Half points for one mistake in the schedule, no points > 1 mistake.

ii. **[4 points]** Give the wait-for graph for the lock requests in Table 3. List each edge in the graph like this: $T_x \to T_y$ because of Z (i.e., T_x is waiting for T_y to release its lock on resource Z). Order the edges in ascending order with respect to x.

Solution:

- $T_1 \to T_2$ because of A
- $T_1 \to T_4$ because of B
- $T_4 \to T_1$ because of B
- $T_5 \to T_3$ because of C

Grading info: Half points for 1 missing directed edge, no points if missing > 1.

iii. **[3 points]** Determine whether there exists a deadlock in the lock requests in Table 3, and briefly explain why.

Solution: Deadlock exists because there is a cycle ($T_1 \to T_4 \to T_1$) in the dependency graph.

Grading info: -2 points for not explaining why there is a deadlock

iv. **[5 points]** To prevent deadlock, we use the 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('-')*. Follow the same format as the previous question.

Solution:

- S(B) at t_2: g
- S(B) at t_3: g
- X(A) at t_4: b
- X(C) at t_5: g
- X(C) at t_6: a (T_5 dies because T_3 holds the lock)
- X(B) at t_7: b

- X(B) at t_8: a (T_4 dies because T_1 is waiting for the lock)

Grading info: -2 *points for one mistake in the schedule, no points* > 1 *mistake.*

v. **[5 points]** Now we use the 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'), aborted ('a'), or already dead('-').* Follow the same format as the previous question.

Solution:

- S(B) at t_2: g
- S(B) at t_3: g
- X(A) at t_4: g (T_1 wounds T_2)
- X(C) at t_5: g
- X(C) at t_6: b
- X(B) at t_7: g (T_1 wounds T_4)
- X(B) at t_8: $-$ (T_4 is already dead from the wound by T_1)

Grading info: -2 *points for one mistake in the schedule, no points* > 1 *mistake.*

Question 3: Hierarchical Locking .... [30 points]

Consider a database (D) consisting of two tables, Cars (C) and Authors (A). Specifically,

- `Cars(cid, aid, make, model, year, review)`, spans 500 pages, namely C_1 to C_{500}
- `Authors(aid, first_name, last_name)`, spans 20 pages, namely A_1 to A_{20}

Further, **each page contains 100 records**, and we use the notation $C_3 : 20$ to represent the 20^{th} record on the third page of the Cars table. Similarly, $A_5 : 10$ represents the 10^{th} record on the fifth page of the Authors 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 (A, C)*, (3) *page-level ($A_1 - A_{20}$, $C_1 - C_{500}$)*, (4) *record-level ($A_1 : 1 - A_{20} : 100$, $C_1 : 1 - C_{500} : 100$)*.

For each of the following operations on the database, please determine the sequence of lock requests that should be generated by a transaction that wants to efficiently carry out these operations by maximizing concurrency.

Please follow the format of the examples listed below:

- write **"IS(D)"** for a request of **database-level IS lock**
- write **"X(C_2 : 30)"** for a request of **record-level X lock for the 30^{th} record on the second page of the Cars table**
- write **"S($C_2 : 30 - C_3 : 100$)"** for a request of **record-level S lock from the 30^{th} record on the second page of the Cars table to the 100^{th} record on the third page of the Cars table**.

(a) **[6 points]** Fetch the 5^{th} record on page A_{15}.

> **Solution:** IS(D), IS(A), IS(A_{15}), S($A_{15} : 5$)
> *Grading info:* -2 *for each missing/incorrect mistake*

(b) **[6 points]** Scan all the records on pages C_1 through C_{10}, and modify the record $C_9 : 3$.

> **Solution:** IX(D), SIX(C), IX(C_9), X($C_9 : 3$);
> also acceptable: IX(D), IX(C), S($C_1 - C_8$), S(C_{10}), SIX(C_9), X($C_9 : 3$)
> *Grading info:* -2 *for each missing/incorrect mistake*

(c) **[6 points]** Count the number of cars with 'year' > 1999.

> **Solution:** IS(D), S(C);
> *Grading info:* -2 *for each missing/incorrect mistake*

(d) **[6 points]** Increase the year of all cars by 1.

> **Solution:** IX(D), X(C)
> also acceptable: IX(D), IX(C), IX($C_1 - C_{500}$), X($C_1 : 1 - C_{500} : 100$)
> *Grading info:* -2 *for each missing/incorrect mistake*

(e) **[6 points]** Capitalize the 'first_name' of ALL authors and capitalize the make of ALL cars.

Solution: X(D)

Grading info: -2 *for each missing/incorrect mistake*

Question 4: Optimistic Concurrency Control .... [20 points]

Consider the following set of transactions accessing a database with object A. 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 it succeeds, then the transaction will immediately switch into the WRITE phase. 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.

time	T_1	T_2	T_3
1	READ(A)		
2		READ(A)	
3	WRITE(A)		
4	VALIDATE/WRITE?		
5			READ(A)
6			VALIDATE/WRITE?
7		WRITE(A)	
8		VALIDATE/WRITE?	

Figure 1: An execution schedule

(a) **[8 points]** The result of this execution is the following. Mark all that apply:
■ **T1 aborts**
□ T2 aborts
□ T3 aborts
□ None of the transactions abort

Solution: T1's write-set intersects with T2's read-set, so it will fail the VALIDATION phase.
This is the example from "Validation Condition #2" from the lecture slides.

Grading info: Full points if they only select T1

Grading info: -2 points if they select T1 and any other transaction

Grading info: -4 points if they do not select T1 at all

(b) **[6 points]** In general, transactions can suffer from *dirty reads* in OCC.
□ Always
□ Sometimes
■ **Never**

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

Grading info: Full points only if they select "Never"

(c) **[6 points]** In general, transactions can suffer from *unrepeatable reads* in OCC.
□ Always

☐ Sometimes
■ **Never**

Solution: If one txn writes to an object read by another txn, then the modifying txn will not be allowed to commit. Thus, it can **never** have unrepeatable reads.

Grading info: Full points only if they select "Never"