CARNEGIE MELLON UNIVERSITY
COMPUTER SCIENCE DEPARTMENT
15-445/645 – DATABASE SYSTEMS (SPRING 2025)
PROF. JIGNESH PATEL

Homework #5 (by Aditya Bhatnagar) – Solutions

Due: **Sunday April 06, 2025 @ 11:59pm**

IMPORTANT:

- Enter all of your answers into **Gradescope by 11:59pm on Sunday April 06, 2025**.
- **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 : 2025/03/25 10:50

Question	Points	Score
Serializability and 2PL	34	
Hierarchical Locking	30	
Optimistic Concurrency Control	36	
Total:	100	

Question 1: Serializability and 2PL............................[34 points]

(a) True/False Questions:

i. **[3 points]** Strong strict Two-Phase Locking (2PL) prevents the occurrence of cascading aborts and inherently avoids deadlocks without the need for additional prevention or detection techniques.

☐ True [✓ **False**]

Solution: False. While 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, it does not inherently resolve deadlocks. Deadlocks, which are situations where two or more transactions prevent each other from progressing by holding locks that the other needs, still require specific detection or prevention mechanisms in strict 2PL.

ii. **[3 points]** Using regular (i.e., not strong strict) 2PL does not guarantee a conflict serializable schedule.

☐ True [✓ **False**]

Solution: False. 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. **[3 points]** Conflict-serializable schedules prevent unrepeatable reads and dirty reads.

☐ True [✓ **False**]

Solution: False. Conflict-serializability ensures that the schedule is conflict equivalent to a serial schedule, thus protecting against unrepeatable reads. 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.

iv. **[3 points]** 2PL is a pessimistic concurrency control protocol.

[✓ **True**] ☐ False

Solution: True. 2PL is considered a pessimistic concurrency control protocol because it uses locks to prevent conflicts before they can occur. By ensuring transactions follow a strict locking protocol, 2PL aims to prevent any possible conflicts, in contrast to optimistic protocols which allow conflicts but then resolve them.

v. **[3 points]** Every conflict-serializable schedule is always conflict-equivalent to at least one view-serializable schedule.

[✓ **True**] ☐ False

Solution: True. Every conflict-serializable schedule is conflict-equivalent to a serial schedule. Since all serial schedules are inherently view-serializable, the statement holds true.

(b) Serializability:

Question 1 continues…

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	W(B)					W(C)				R(D)
T_2			W(A)	W(B)						
T_3		R(C)			R(A)					
T_4							W(B)	R(D)	W(A)	

Table 1: A schedule with 4 transactions

i. **[3 points]** Is this schedule serial?
☐ Yes [✓ **No**]

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

ii. **[3 points]** Is this schedule conflict serializable?
☐ Yes [✓ **No**]

Solution: This schedule isn't conflict serializable because there is a cycle in its data dependency graph. Moreover, this schedule is not conflict equivalent (every pair of conflicting operations is ordered in the same way) to any serial schedule of transaction execution.

iii. **[3 points]** Is this schedule view serializable?
[✓ **Yes**] ☐ No

Solution: To determine if the schedule is view serializable, we need to check it against three criteria for view equivalence for original schedule S_1 and serial schedule S_2:

1. If T_3 reads the initial value of A in S_1, then T_3 must also read the initial value of A in S_2.
2. If T_3 reads a value of A written by T_4 in S_1, then T_3 must also read that value of A written by T_4 in S_2.
3. If T_3 writes the final value of A in S_1, then T_3 must also write the final value of A in S_2.

From our analysis, we found a possible serial schedule that is view-equivalent to the given schedule: $T_2 \rightarrow T_3 \rightarrow T_1 \rightarrow T_4$. This serial schedule fulfills the criteria for view equivalence:

- T_2 writes A and B.
- T_3 reads the initial value of C and the value of A written by T_2.

Question 1 continues…

- T_1 writes B, writes the final version of C, and reads the initial value of D.
- T_4 writes the final version of B, reads the initial version of D, and writes the final version of A.

Therefore, this serial schedule is view-equivalent to the original one, so it is proved to be view-serializable. Importantly, compared to conflict serializability, view serializability allows all conflict serializable schedules along with ”blind writes”, ensuring database consistency after all transactions are committed.

iv. **[7 points]** Compute the conflict dependency graph for the schedule in Table 1, selecting all edges (and the object that caused the dependency) that appear in the graph.

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

Solution: The answer is:

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

For A, there are W-R conflict from T_2 to T_3, R-W conflict from T_3 to T_4, and W-W conflict from T_2 to T_4;
For B, there are W-W conflict from T_1 to T_4, W-W conflict from T_2 to T_4 and W-W conflict from T_1 to T_2;
For C, there is R-W conflict from T_3 to T_1;
For D, there is no conflict.

v. **[3 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.

Homework #5 continues…

Question 2: Hierarchical Locking .... [30 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 2000 pages, which we denote R1 to R2000. Table A spans 100 pages, which we denote A1 to A100. Each page contains 200 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]** Update the type of all musical artists in table A with the name = 'Eminem' to 'Slim Shady'

☐ X(D)
■ IX(D), X(A)
☐ SIX(D), X(A)
☐ IX(D), IX(A)

Solution: The correct answer choice is IX(D), X(A). This is because we potentially need to modify records in table A where the artist's name is 'Eminem', and this choice accesses the intended exclusive parent lock to get the exclusive lock on those specific rows in table A.

- X(D) is incorrect because it gains an exclusive lock on the entire database D when it only needs to modify specific rows in table A.
- SIX(D), X(A) is incorrect because it gains a shared intention lock on the database D when it has no need to read the contents of D.
- IX(D), IX(A) is incorrect because it does not gain the exclusive lock on the specific rows in table A that match the condition (artist name is 'Eminem'), which it needs to modify the type of those artists.

(b) **[5 points]** Find the release record whose year equals the oldest year in all releases.

☐ S(D)
☐ IS(D), IS(R)
☐ IX(D), X(R)
■ IS(D), S(R)

Question 2 continues...

Solution: The correct choice is `IS(D), S(R)`. The query needs to scan all records in `R` to find the release with the oldest year and scan all records again to find the release records that satisfy the equation condition. So, it requires a shared lock on `R`.

- `S(D)` is incorrect because it doesn't acquire the intended parent locks necessary to obtain the `S(R)` lock for reading the table `R`.
- `IS(D), IS(R)` is incorrect because it only states the intention to read-access the table R but doesn't actually acquire the lock to perform the read.
- `IX(D), X(R)` is incorrect because it accesses an intended exclusive lock, which is not needed for a read-only query.

(c) **[5 points]** Increment the number_sold for the 10th record on `R1999`.

☐ `IX(D), IX(R), IX(R1999), IX(R1999.10)`
■ `IX(D), IX(R), IX(R1999), X(R1999.10)`
☐ `SIX(D), SIX(R), SIX(R1999), X(R1999.10)`
☐ `IS(D), IS(R), IS(R1999), X(R1999.10)`

Solution: The correct choice is `IX(D), IX(R), IX(R1999), X(R1999.10))`. 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.

- `IX(D), IX(R), IX(R1999), IX(R1999.10)` is incorrect because it only gets the intention exclusive lock for the record.
- `SIX(D), SIX(R), SIX(R1999), X(R1999.10)` 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 locks.
- `IS(D), IS(R), IS(R1999), X(R1999.10)` is incorrect because the DBMS intends to write to a tuple, so it should not grab the shared intention parent locks.

(d) **[5 points]** Scan all records between `R10` and `R100` and modify the 2nd record on `R50`

■ `IX(D), SIX(R), IX(R50), X(R50.2)`
☐ `IS(D), S(R), IX(R50.2)`
☐ `IX(D), IX(R), IX(R10-R100), IX(R50), X(R50.2)`
☐ `IS(D), SIX(R), X(R50)`

Solution: The correct choice is `IX(D), SIX(R), IX(R50), X(R50.2)`. This choice is correct because it accesses all intended locks and the exclusive lock `X(R50.2)`. It also gains a shared intention lock on `R`, so it can read and modify records in `R`.

- `IS(D), S(R), IX(R50.2)` is incorrect because it accesses an intended shared parent lock for both `D` and `R`, when we plan to modify a record in `R`.
- `IX(D), IX(R), IX(R10-R100), IX(R50), X(R50.2)` is incorrect because we do

Question 2 continues…

plan on reading and modifying in relation `R`, but we are only gaining a exclusive lock on the whole `R` table, which is too much.

- `IS(D), SIX(R), X(R50)` is incorrect because while it gains a shared intention lock for `R` and the database, it tries to exclusively lock the whole page `R4` instead of just the specific record.

(e) **[5 points]** Delete records in `A` if type = 'Band'.

☐ `SIX(D), SIX(A)`
■ `IX(D), X(A)`
☐ `IX(D), IX(A)`
☐ `SIX(D), X(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`.

- `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..
- `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.

(f) **[6 points]** Two users are trying to access data. User A is scanning all the records in `R` to read, while User B is trying to modify the 15^{th} record in `A10`. Which of the following sets of locks are most suitable for this scenario?

☐ `User A: SIX(D), S(R), User B: SIX(D), IX(A), X(A10.15)`
☐ `User A: S(D), User B: X(D)`
■ **`User A: IS(D), S(R), User B: IX(D), IX(A), X(A10.15)`**
☐ `User A: IS(D), S(R), User B: SIX(D), IX(A), X(A10.15)`

Solution: The correct choice is `User A: IS(D), S(R), User B: IX(D), IX(A), X(A10.15)`. 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 `A`, and then an exclusive lock on the specific record `A10.15`.

Other choices are incorrect because:

Question 2 continues…

- `User A: SIX(D), S(R) and User B: SIX(D), IX(A), X(A10.15)` 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(R) and User B: SIX(D), IX(A), X(A10.15)` 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.

Homework #5 continues…

Question 3: Optimistic Concurrency Control . . . . [36 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 begins its read phase with its first operation and 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(A)		
3			
4	WRITE(A)	WRITE(A)	
5		WRITE(B)	
6		WRITE(C)	
7		VALIDATE?	
8	READ(B)	WRITE?	READ(C)
9	VALIDATE?		
10	WRITE?		
11			
12			
13			WRITE(C)
14			
15			READ(D)
16			WRITE(D)
17			VALIDATE?
18			WRITE?

Figure 1: An execution schedule

(a) **[5 points]** When is each transaction's timestamp assigned in the transaction process?
- ☐ The start of the write phase.
- ☐ Timestamps are not necessary for OCC.
- ■ **The start of the validation phase.**
- ☐ The start of the read phase.

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

Question 3 continues. . .

(b) **[5 points]** When time = 8, will T_3 read C written by T_2?
[✓ **Yes**] ☐ No

Solution: Yes. In OCC, when a transaction commits then the write set is written to the global database and when a transaction wants to read and bring the value of a object in its private workspace it picks the value up from the global database at the time of reading. So, T_3 will read C at 8 which was written by T_2.

(c) **[5 points]** Will T1 abort?
[✓ **Yes**]
☐ No

Solution: T_1 has not read the value of A written by T_2 at timestamp = 4. Hence, the validation phase of T_1 fails and hence it aborts.

(d) **[5 points]** Will T2 abort?
☐ Yes
[✓ **No**]

Solution: T_2 will not abort because T_2's read set does not intersect with any other transactions read-write set before T_2 commits in backward validation.

(e) **[5 points]** Will T3 abort?
☐ Yes
[✓ **No**]

Solution: T_3 won't abort because T_3's read and write set doesn't intersect with T_2's read and write set.

(f) **[3 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.

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

(h) **[3 points]** Aborts due to OCC are wasteful because they happen after a transaction has already finished executing.
[✓ **True**] ☐ False

(i) **[2 points]** All Objects in the read-set of a transaction are read as a whole when the transaction starts.
☐ True [✓ **False**]