CSE 444 Practice Problems

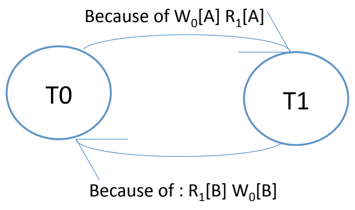
Transactions: Concurrency Control

1. **Schedules, Serializability, and Locking**

   (a) Consider the following two transactions and schedule (time goes from top to bottom). Is this schedule conflict-serializable? Explain why or why not.

Transaction T_0	Transaction T_1
$r_0[A]$	
$w_0[A]$	
	$r_1[A]$
	$r_1[B]$
	c_1
$r_0[B]$	
$w_0[B]$	
c_0	

Solution:
The schedule is not conflict serializable because the precedence graph contains a cycle. The graph has an edge $T_0 \rightarrow T_1$ because the schedule contains $w_0[A] \rightarrow r_1[A]$. The graph has an edge $T_1 \rightarrow T_0$ because the schedule contains $r_1[B] \rightarrow w_o[B]$.



(b) Show how 2PL can ensure a conflict-serializable schedule for the same transactions above. Use the notation $L_i[A]$ to indicate that transaction i acquires the lock on element A and $U_i[A]$ to indicate that transaction i releases its lock on A.

Solution:

Multiple solutions are possible.

Transaction T_0	Transaction T_1
$L_0[A]$	
$r_0[A]$	
$w_0[A]$	
	$L_1[A] \rightarrow blocks$
$L_0[B]$	
$r_0[B]$	
$w_0[B]$	
$U_0[A]$	
$U_0[B]$	
c_0	
	$L_1[A] \rightarrow granted$
	$r_1[A]$
	$L_1[B]$
	$r_1[B]$
	$U_1[A]$
	$U_1[B]$
	c_1

(c) Show how the use of locks without 2PL can lead to a schedule that is NOT conflict serializable.

Solution:

Transaction T_0	Transaction T_1
$L_0[A]$	
$r_0[A]$	
$w_0[A]$	
$U_0[A]$	
	$L_1[A]$
	$r_1[A]$
	$U_1[A]$
	$L_1[B]$
	$r_1[B]$
	$U_1[B]$
	c_1
$L_0[B]$	
$r_0[B]$	
$w_0[B]$	
$U_0[B]$	
c_0	

2. **More serializability and locking**

Consider a database with objects X and Y and assume that there are two transactions T_1 and T_2. T_1 first reads X and Y and then writes X and Y. T_2 reads and writes X then reads and writes Y.

(a) Give an example schedule that is not serializable. Explain why your schedule is not serializable.
Solution:
$R_1[X] \rightarrow R_1[Y] \rightarrow R_2[X] \rightarrow W_2[X] \rightarrow R_2[Y] \rightarrow W_1[X] \rightarrow W_1[Y] \rightarrow W_2[Y] \rightarrow C_1 \rightarrow C_2$

A schedule is serializable if it contains the same transactions and operations as a serial schedule *and* the order of all conflicting operations (read/writes to the same objects by different transactions) is also the same. In the above schedule, T_1 reads X before T_2 writes X. However, T_1 writes X after T_2 reads and writes it. The schedule is thus clearly not serializable. Additionally, according to the above schedule, the final content of object X is written by T_1 and the final content of object Y is written by T_2. Such a result is not possible in any serial execution, where transactions execute one after the other in sequence.

(b) Show that strict 2PL disallows this schedule.
Solution:

Strict 2PL has two two rules:

i. If a transaction T wants to read (respectively, modify) an object, it first requests a shared (respectively, exclusive) lock on the object.
ii. All locks held by a transaction are released when the transaction is completed.

With strict 2PL, the above schedule is not possible. Indeed, T_1 first acquires shared locks on X and Y. When T_2 runs, it also acquires a shared lock on X. When it tries to acquire an exclusive lock before writing X, however, it blocks, waiting for T_1 to release its lock, which will happen only when T_1 commits. The above schedule is thus impossible with strict 2PL. In fact, the schedule leads to a deadlock: when T_1 tries to write X, it also blocks waiting for T_2 to release its shared lock. Now, both transactions are waiting for each other. We have a deadlock. When the deadlock is detected, the DBMS will abort one of the transactions, allowing the other one to commit and release its locks.

(c) What are the differences between the four levels of isolation?

Solution:

- **Read uncommitted:** Transactions do not need to acquire any locks before reading data. Transactions may thus read data written by other transactions that have not yet committed. The value read may thus later be changed further or rolled-back. This problem is called the *dirty read* problem. This level of isolation also suffers from all the problems of the more restrictive isolation levels below.
- **Read committed:** Transactions must acquire shared locks before reading data. They may release these locks as soon as they read the data (short duration read locks). This level of isolation guarantees that the transaction never reads uncommitted data by other transactions. However, it doesn't ensure the data will not change until the end of the transaction. If a transaction reads the same data item twice, it can see two different values. This problem is called the *non-repeatable read* problem. This level of isolation also suffers from all the problems of the more restrictive isolation levels below.
- **Repeatable read:** Transactions must acquire long duration read locks on the individual data items that they read. This level of isolation provides all the guarantees of the read committed level. It also ensures that data seen by a transaction does not change until the end of the transaction: i.e., it provides repeatable reads. However, because locks are held on individual data items, transactions may experience the *phantom problem.* If a transaction reads *twice* a set of tuples that satisfy a predicate, it only locks the individual data items that match the predicate. If another transaction inserts a tuple that matches the predicate between the two read operations, that new tuple will appear as a result of the second read.
- **Serializable:** Transactions must acquire long duration read locks on *predicates* as well as on individual data items. This level of isolation protects against all the problems of the less restrictive levels. It ensures serializability.

3. **Optimistic Concurrency Control**

   Consider a concurrency control manager by timestamps. Below are several sequences of events, including start events, where sti means that transaction Ti starts and coi means Ti commits. These sequences represent real time, and the timestamp-based scheduler will allocate timestamps to transactions in the order of their starts. In each case below, say what happens with the last request.

   You have to choose between one of the following four possible answers:

   (a) the request is accepted,
   (b) the request is ignored,
   (c) the transaction is delayed,
   (d) the transaction is rolled back.

   (a) st1; st2; r1(A); r2(A); w1(B); w2(B);
   Solution:
   The system will perform the following action for w2(B): accepted.

   (b) st1; st2; r2(A); co2; r1(A); w1(A)
   Solution:
   The system will perform the following action for w1(A): rolled back.

   (c) st1; st2; st3; r1(A); w3(A); co3; r2(B); w2(A)
   Solution:
   The system will perform the following action for w2(A): ignored.

   (d) st1; st2; st3; r1(A); w1(A); r2(A);
   Solution:
   The system will perform the following action for r2(A): delayed.

   (e) st1; st2; st3; r1(A); w2(A); w3(A); r2(A);
   Solution:
   The system will perform the following action for r2(A): rolled back.

4. **Miscellaneous**

For each statement below, indicate if it is true or false.

(a) Serializability is the property that a (possibly interleaved) execution of a group of transactions has the same effect on the database and produces the same output as some serial execution of those transactions.

TRUE FALSE

(b) The following schedule is serializable:
$r_0[A] \rightarrow w_0[A] \rightarrow r_1[B] \rightarrow w_1[B] \rightarrow r_1[A] \rightarrow w_1[A] \rightarrow r_0[C] \rightarrow w_0[C] \rightarrow c_0 \rightarrow c_1$

TRUE FALSE

(c) A NO-STEAL buffer manager policy means that all pages modified by a transaction are forced to disk before the transaction commits.

TRUE FALSE

(d) Strict two-phase locking (2PL) ensures that transactions never deadlock.

TRUE FALSE

(e) Strict two-phase locking (2PL) ensures serializability.
Note: This question can be confusing. Be careful! To get serializability, one needs both strict 2PL and predicate locking. Strict 2PL alone guarantees only conflict serializability. Please see book and notes for more details.

TRUE FALSE

(f) In the ARIES protocol, at the end of the analysis phase, the Dirty Page Table contains the exact list of all pages dirty at the moment of the crash.

TRUE FALSE

(g) The ARIES protocol uses the "repeating history" paradigm, which means that updates for all transactions (committed or otherwise) are redone during the REDO phase.

TRUE FALSE

Solution:
T, T, F, F, F, F, T