Weekly exercises IN3020&4020 (2026)
Week 9: Transaction Management Concepts and Basic Concurrency Protocols

1 Concepts Review

1. Remind yourself what is a transaction, schedule and conflict, and what is ACID principles.

   Solution: See the solutions to the previous week.

2. What is a serial schedule? Does two serial schedules of the same transactions always result in the same database state?

   Solution: A schedule is serial when it is always only one active (not committed and not aborted) transaction. Two serial schedules may result in different databases (i.e., they may not be result equivalent).

3. What is (conflict-)serialisability and how can serialisability be used as a way to ensure isolation?

   Solution: A schedule S is serialisable if it is conflict-equivalent to some serial schedule S'. If S is serialisable it is considered correct, same as a serial schedule. It may not, however, be serial by itself. Thus a serialisable schedule gives the benefit of allowing concurrent transactions, but at the same time being correct.

4. What does it mean for two schedules to be conflict-equivalent and what is the tool to check it?

   Solution: The two schedules are of the same transactions and the relative order of every two conflicting operations is the same in both schedules. We can check it by constructing the precedence graph.

5. What does it mean for two schedules to be view-equivalent and what is the relations between conflict-equivalent and view-equivalent schedules?

   Solution: The two schedules are of the same transactions, each read

is from the same place, and the last write of each item is the same. Each conflict equivalent schedule is view equivalent, but not other way round (i.e., view equivalence for defining serialisability allow for more concurrency).

6. What is recoverable, cascadeless and strict schedules, what is their purpose and relations?

   Solution: Intuitively, all these ensure durability. Each strict schedule is cascadeless (they are easy to recover), and all those are recoverable (but may be difficult). A schedule is recoverable if no transaction T commits before all transactions from which T read have committed. I is cascadeless if every transaction in the schedule reads only from committed transactions. It is strict if additionally every write is to a place where the previous write is committed.

7. Discuss the four levels of isolation in SQL.

   Solution:

   - Read uncommitted: allows for dirty reads, nonrepeatable reads, and phantoms
   - Read committed: disallows dirty reads
   - Repeatable read: disallows dirty reads and nonrepeatable reads
   - Serialisable: disallows all three

   Real DBMSs usually disallow more than required, including dirty writes.

2 Serialisability

1. Let T_1 be the transaction $r_1(X), w_1(X), r_1(Y), w_1(Y)$ and T_2 the transaction $r_2(X), w_2(X)$. List several (at least 5) possible schedules of T_1 and T_2 and decide which are (conflict-)serialisable.

   Solution:

   - $r_1(X), w_1(X), r_1(Y), w_1(Y), r_2(X), w_2(X)$ Serializable, it is serial
   - $r_1(X), w_1(X), r_1(Y), r_2(X), w_1(Y), w_2(X)$ Serializable
   - $r_1(X), w_1(X), r_2(X), r_1(Y), w_1(Y), w_2(X)$ Serializable
   - $r_1(X), r_2(X), w_1(X), r_1(Y), w_1(Y), w_2(X)$ Not serialisable
   - $r_2(X), r_1(X), w_1(X), r_1(Y), w_1(Y), w_2(X)$ Not serialisable
   - $r_2(X), r_1(X), w_1(X), r_1(Y), w_2(X), w_1(Y)$ Not serialisable

T_1
`read`(X);
$X := X - 1$;
`write`(X);
`read`(Y);
$Y := Y + 1$;
`write`(Y);

T_2
`read`(X);
$X := X + 1$;
`write`(X);

Figure 1: Transactions T_1 and T_2

- $r_2(X), r_1(X), w_1(X), w_2(X), r_1(Y), w_1(Y)$ Not serialisable
- $r_2(X), r_1(X), w_2(X), w_1(X), r_1(Y), w_1(Y)$ Not serialisable
- $r_2(X), w_2(X), r_1(X), w_1(X), r_1(Y), w_1(Y)$ serialisable, it is serial
- etc.

2. Below are two transactions, described in terms of their effect on the database elements A and B, which we may assume are integers.

T_1
$r_1(A)$
$A := A + 1$
$w_1(A)$
$r_1(B)$
$B := B * 3$
$w_1(B)$

T_2
$r_2(B)$
$B := B * 2$
$w_2(B)$
$r_2(A)$
$A := A + 3$
$w_2(A)$

   (a) Give examples of a serialisable schedule and a non-serialisable schedule of the these transactions

   (b) It turns out that both serial schedules have the same effect on any database, that is, (T_1, T_2) and (T_2, T_1) are semantically equivalent. Demonstrate this fact by showing the effect of the two transactions on an arbitrary initial database state (i.e., on a database where the values of items A and B are A_{init} and B_{init}. Modify the transactions so that semantic equivalence does not hold any more.

   Solution:
   Schedule $T_1 : T_2$ gives $A_{new} := (A_{init} + 1) + 3 = A_{init} + 4$ and $B_{new} := (B_{init} * 3) * 2 = 6B_{init}$
   Schedule $T_2 : T_1$ gives $A_{new} := (A_{init} + 3) + 1 = A_{init} + 4$ and $B_{new} := (B_{init} * 2) * 3 = 6B_{init}$

For example, we can replace $B := B * 2$ by $B := B + 2$ to loose equivalence.

3. Consider the following two schedules:

   S_1: $w_3(A); r_1(A); w_1(B); r_2(B); w_2(C); r_3(C)$

   S_2: $r_1(A); r_2(A); r_3(B); w_1(A); r_2(C); r_2(B); w_2(B); w_1(C)$

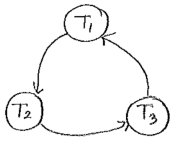
   Answer the following questions:

   (a) What is the precedence graph for each of the schedules?

   (b) Is the schedule serialisable? If so, what are the equivalent serial schedules?

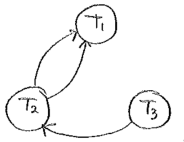
   Solution:

   Schedule S_1 has a cyclic precedence graph:

   This means that the schedule is not serialisable. Indeed, it is not conflict-equivalent to any of $T_1; T_2; T_3 \mid T_1; T_3; T_2 \mid T_2; T_3; T_1$.

   Schedule S_2 has precedence graph without cycles:

   This means the schedule is conflict serialisable. It is equivalent with $T_3; T_2; T_1$.

4. (*) Does view-equivalence to a serial schedule guarantee no dirty writes? No dirty reads? No non-repeatable reads? Justify your answers.

   Solution:

   No dirty writes is not guaranteed: $r_1(X), w_2(X), w_1(X), w_3(X)$ has a dirty write, but is view-equvalent to $r_1(X), w_1(X), w_2(X), w_3(X)$. No dirty reads is not guaranteed either, same as for conflict-equivalence (it is the business of recovery control). No non-repeatable reads is guaranteed, because view-equivalence ensures that all reads are from the same place.

3 Recoverability concepts

1. For each of the following schedules, classify them recoverable, cascadeless, strict or non-recoverable:

   - $S_1 : r_1(X), w_1(X), c_1, r_2(X), w_2(X), c_2$
   - $S_2 : r_1(X), w_1(X), a_1, r_2(X), w_2(X), c_2$
   - $S_3 : r_1(X), r_2(X), w_1(X), c_1, w_2(X), c_2$
   - $S_4 : r_1(X), r_2(X), w_2(X), c_2, w_1(X), c_1$
   - $S_5 : r_1(X), w_1(X), r_2(X), w_2(X), c_2, c_1$
   - $S_6 : r_1(X), w_1(X), r_2(X), c_1, w_2(X), c_2$

   Solution:

   - Schedule S_1 is recoverable, cascadeless and strict.
   - Schedule S_2 is recoverable, cascadeless and strict; note, however, that after the abort of T_1 all its changes should be rolled back, and T_2 reads an original value of X.
   - Schedule S_3 is recoverable, cascadeless and strict.
   - Schedule S_4 is recoverable, cascadeless and strict.
   - Schedule S_5 is non-recoverable because T_2 reads from T_1 and T_2 commits first.
   - Schedule S_6 is recoverable because all transactions that have written the X that T_2 reads are committed before T_2 commits. It is, however not cascadeless or strict because T_2 reads from uncommitted write from T_1.

4 Locks

1. What is the two-phase locking protocol?

   Solution: The two-phase locking protocol guarantees that all locking operations precede the first unlock operation in the transaction.

2. If you are using transactions with binary locks, can you be sure that a schedule is serialisable? What if shared/exclusive locks?

   Solution: Locks be themselves do not ensure anything, but they facilitate protocols (e.g., two-phase locking) that guarantee serialisation.

3. If you are using the two-phase locking protocol, can you be sure that a schedule is serialisable?

Solution: The two-phase locking protocol can be proven to always give a serialisable schedule (if this schedule is complete).

4. In what cases can we have a deadlock? What are the approaches to this problem?

   Solution: We have a deadlock when several transactions are waiting for unlocking some items in a circular manner. Overall, there are deadlock prevention, deadlock detection, and timeout approaches to deal with deadlocks.

5. Consider two schedules:

   (a) $w_3(A); r_1(A); w_1(B); r_2(B); w_2(C); r_3(C)$

   (b) $r_1(A); r_2(A); r_3(B); w_1(A); r_2(C); r_2(B); w_2(B); w_1(C)$

   Assume that at transaction takes a (binary) lock on each database element immediately before it reads or writes the element, and that each transaction releases its locks immediately after the last time it accesses an element. Tell what the locking scheduler would do with each of these schedules—that is, transform these schedules to schedules that follow locking rules (i.e., show what request would get delayed, and when would they be allowed to resume)?

   Solution:

   (a) The transactions with locks will be as follows, and the schedule is given in Fig. 2 on the left:

   - $T_1 : l_1(A), r_1(A), l_1(B), w_1(B), u_1(A), u_1(B)$
   - $T_2 : l_2(B), r_2(B), l_2(C), w_2(C), u_2(B), u_2(C)$
   - $T_3 : l_3(A), w_3(A), l_3(C), r_3(C), u_3(A), u_3(C)$

   (b) The transactions with locks will be as follows, and the schedule is given in Fig. 2 on the right:

   - $T_1 : l_1(A), r_1(A), w_1(A), l_1(c), w_1(c), u_1(A), u_1(C)$
   - $T_2 : l_2(A), r_2(A), l_2(C), r_2(C), l_2(B), w_2(B), u_2(A), u_2(C), u_2(B)$
   - $T_3 : l_3(B), r_3(B), u_3(B)$

6. (*) Prove that two-phase locking always gives serialisable schedules (Hint: if a serialisability graph for a schedule has a cycle, at least one of the transactions in the schedule does not obey the two-phase locking protocol).

   Solution: Let T be a set of transactions $T_1, T_2, \ldots, T_n$ that all obey the basic two-phase locking rule. Let S be a non-serialisable schedule

T_1	T_2	T_3
		$l_3(A)$
		$w_3(A)$
$l_1(A)$ – wait		
	$l_2(B)$	
	$r_2(B)$	
	$l_2(C)$	
	$w_2(C)$	
	$u_2(B)$	
	$u_2(C)$	
		$l_3(C)$
		$r_3(C)$
		$u_3(A)$
($l_1(A)$ acquired)		
		$u_3(C)$
$r_1(A)$		
$l_1(B)$		
$w_1(B)$		
$u_1(A)$		
$u_1(B)$		

T_1	T_2	T_3
$l_1(A)$		
$r_1(A)$		
	$l_2(A)$ – wait	
		$l_3(B)$
		$4_3(B)$
		$u_3(B)$
$w_1(A)$		
$l_1(C)$		
$w_1(C)$		
$u_1(A)$		
$u_1(C)$		
	($l_2(A)$ acquired)	
	$r_2(A)$	
	$l_2(C)$	
	$r_2(C)$	
	$l_2(B)$	
	$w_2(B)$	
	$u_2(A)$	
	$u_2(C)$	
	$u_2(B)$	

Figure 2: Schedule A (left) and Schedule B (right)

for T. Then, the serialisation graph for S must have a cycle. Thus, there must be some sequence within the schedule on the form:

$$S \ldots; [o_1(X); ...; o_2(X)] \ldots; [o_2(Y); ...; o_3(Y)] \ldots; [o_n(Z); ...; o_1(Z)] \ldots,$$

where each pair of operations between square brackets $[o, ..., o]$ are conflicting (either $[w, ..., w]$, or $[w, ..., r]$ or $[r, ..., w]$) in order to create an arc in the serialisation graph. This implies that in transaction T_1 a sequence on the following form occurs: $T_1; \ldots; o_1(X); \ldots; o_1(Z) \ldots$. Furthermore, T_1 has to unlock item X (so T_2 can lock it before applying $o_2(X)$ to follow the rules of locking) and has to lock item Z (before applying $o_1(Z)$, but this must occur after T_n has unlocked it). Hence, a sequence in T_1 of the following form occurs: $T_1 : \ldots o_1(X); \ldots; unlock(X); \ldots; lock(Z); \ldots; o_1(Z); \ldots$. This implies that T_1 does not obey the two-phase locking protocol since $lock(Z)$ follows $unlock(Z)$, contradicting our assumption that all transactions in S follow the two-phase locking protocol.