CS127 Quiz 3 Prep

Quiz 3: December 4th, 2019 3:00 P.M.

This ungraded handout is to help you prepare for Quiz 3. Answers will be posted later.

HW5 and its solutions are also relevant to the quiz and should be reviewed with this quiz prep.

In answering these questions, if you don't understand the slides, the answers can be found in the textbook (except for the questions about the recovery algorithm, which follows the fourth edition of the textbook).

Topic #1 ACID Properties

1. Transactions
   (a) What is a transaction?
   A collection of operations that form a single logical unit of work.
   (b) How is it indicated in SQL?
   begin transaction ... end transaction

2. Atomicity and Durability
   (a) How does the recovery system create *atomicity*?
   In the event of a transaction or system failure, the recovery system rolls back transactions that have not committed, thus guaranteeing that transactions are either entirely completed or not done at all.
   (b) How does the recovery system create *durability*?
   In the event of a system failure, the recovery system redoes transactions that have committed but which are not on non-volatile storage, thereby preventing them from being lost.

3. Isolation and Consistency
   (a) What is the definition of *isolation* (in the context of databases)?
   For each transaction T_i and T_j, even if they are executing concurrently, it appears to T_i either that it completed before T_j started, or that it started after T_j completed.
   (b) When must isolation be enforced?
   When using concurrency. If concurrency were not used, there would be no need to create the *appearance* of isolation, since transactions would necessarily *be* serial.
   (c) The concurrency control system creates [the appearance of] isolation for concurrent transactions. How does it do this?
   By enforcing serializability on schedules of transactions
   (d) What is the definition of *consistency* (in the context of databases) and who is responsible for it?
   If a transaction is run in isolation starting from a consistent database, the database is again be consistent at the end of the transaction. The programmer is responsible for this, since consistency is specific to each application.
   As a note, the concurrency control system indirectly supports consistency by enforcing isolation. Failure to enforce isolation may cause the database may become inconsistent.

Topic #2 Serial Equivalence

1. Concurrency's Benefits
   How is using concurrency in transaction processing beneficial? Give two reasons.

   (a) It allows better use of computing resources and, consequently, faster response. For example, one transaction many require doing a calculation in the CPU while another may require writing to disk, which uses the I/O system. If these transactions are done concurrently, then, since they are using different resources, they can run in parallel, allowing better throughput.

   (b) It allows transactions to not be held back waiting for long-running transactions, thereby reducing unpredictable delays in running transactions. For example, if a transaction takes 10 minutes to compute, without concurrency all other transactions must wait for it to complete before executing. However, concurrency allows the system to put the long-running transaction on pause and process the other transactions then return the the long one.

2. Serializability
   The schedule in Figure 1 is serializable even though it meets neither the definition of conflict serializable nor the definition of view serializable.

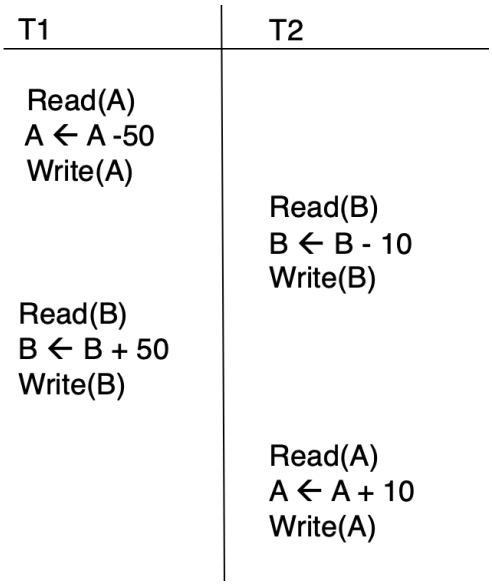


Figure 1: Serializable Schedule

   (a) Why is it not conflict serializable?
   Because the instructions from T1 on B cannot be swapped up to come before the instructions of T2 on B, since there is a read-write conflict between T1 read(B) and T2 write(B).
   Also, the instructions of T1 cannot be swapped down to below the instructions of T2, since there is a write-read conflict between T1 write(A) and T2 read(A). Since it is not possible to swap instructions to make this a serial schedule without a conflict, it is not conflict serializable.

   (b) Why is this not view serializable?
   One way to answer is that for this to be view serializable, then, by the first rule, it must be view equivalent to a serial schedule in which T1 reads the initial value of A, and by the thrid rule, it must be view equivalent with a serial schedule in which T1 writes the final value of B, which is impossible, since if T1 comes before T2, then T2 writes the final value of B, and if T1 comes after T2, then T1 reads a value of A that was modified by T2.

(c) Why, then, is the schedule serializable?
Because this schedule uses the same transactions and produces the same final values of A and B as a serial schedule produces.

(d) What property of addition and subtraction does the fact that this is a serializable schedlue even though it is neither view nor conflict serializable demonstrate?
It demonstrates the commutative property of addition and subtraction. Thus, updates to data items that only involve addition and subtraction give the same result regardless of their order.

(e) What other arithmetic operations have this same property?
Multiplication and division

3. View Serializability
The schedule in Figure 2 is view serializable.

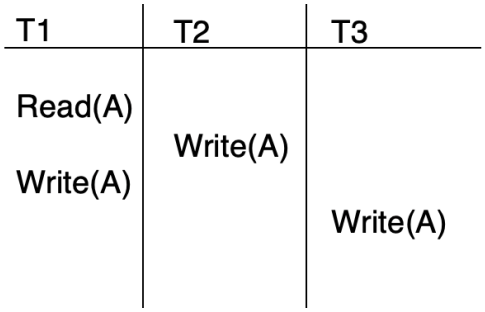
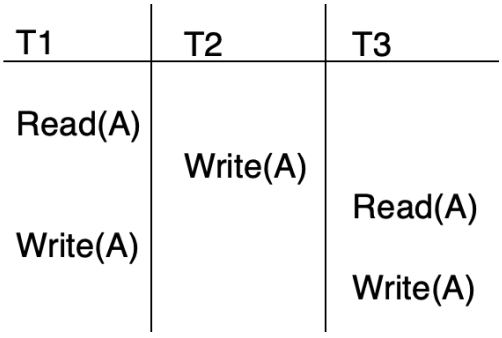


Figure 2: View Serializable Schedule

(a) What serial schedule is it equivalent to?
T1, T2, T3

(b) Why is it not conflict serializable?
Because T1 write(A) cannot be swapped up to be above T2 write(A) since that would cause a write-write conflict. Also, T1 read(A) cannot be swapped down to be below T2 write(A) since that would cause a read-write conflict.

(c) In the above schedule, add a Read(A) instruction to T3 before T3's Write(A) instruction such that it is still view serializable.
read(A) must be added after T2 write(A) and before T1 write(A) since in the view equivalent, where T3 comes after T2, T3 reads the output of T2 write(A). Here is the schedule:



(d) In the above schedule, can a read(A) instruction be added to T2 before T2 write(A) without breaking view serializability?
No. Since in the view equivalent, T2 reads from the output of T1 write A, which in the given schedule is impossible.

(e) Explain how the 3 rules of view serializability ensure that the scheudle is serializable.
In a serial scheudule, the data passes between transactions sequentially, with the final transaction determining the final value of the data. Thus, so long as each transaction takes in the data from the same source as in the serial schedule, and so long as the last transaction to write the data is the same as the last transaction to write the data in the serial schedule, then the outcome of the data must be the same as it is in the serial schedule. The first two rules of view serializability ensure the first property: that each transaction is taking in data from the same source as in the serial schedule. The third rule of view serializability ensures the second property: that the last transaction to manipluate the data is the same as in the serial scheudle. Thus, the rules of view serializability ensure that the outcome of the scheudle is the same as a serial scheudule.

4. Confict Serializability
While view serializability is flexible, it is hard to test (NP complete), so is not used in practice. Instead, a more rigid requirement for serializability is used: Conflict serializability.

(a) In conflict serializability, when may the position of two instructions be swapped?
When the instructions either are on different items of data or when the instructions are both read instructions.

(b) How is a conflict serializable schedule more similar to a serial schedule than a view serializable schedule?
Conflict serializability requires that the transactions not only read from the same sources and output the same final value as a serial schedule, but also requires that the states of the data are changed in the same order as they are in a serial schedule. Thus, swapping writes causes conflict.

(c) Are the schedules in figures 3 and 4 conflict serializable? If so, which serial schedule are they conflict equivalent to?
The first isn't, the second is. The second is equivalent to a serial schedule of T1, T2.

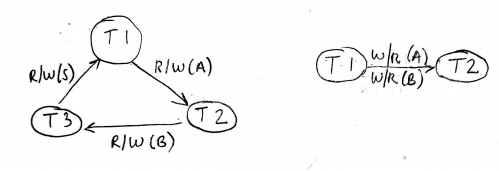
T1	T2	T3
(1) Read(A)		
	(a) Write(A)	
	(b) Read(B)	
		(x) Write(B)
		(y) Read(S)
(2) Write(S)		

Figure 3: Problem 2.4.c, Schedule 1

T1	T2
1. Read(A)	
2. Write(A)	
	a. Read(A)
	b. Write(A)
3. Read(B)	
4. Write(B)	
	c. Read(B)
	d. Write(B)

Figure 4: Problem 2.4.c, Schedule 2

(d) Create a precedence graph for the above two schedules. Explain how this graph can help you to determine conflict serializability.
Answer:



The cycle in the first graph shows that it is not conflict serializable. The lack of cycles in the second graph shows that it is conflict serializable.

Topic #3 Recoverability and Cascadelessness

1. Which of the below schedules is recoverable?

T1	T2
Write(A)	
	Read(A)
	Write(A)
	Commit
Commit	

Figure 5: Problem 3.1, Schedule 1

T1	T2
Write(A)	
	Read(A)
	Write(A)
Commit	
	Commit

Figure 6: Problem 3.1, Schedule 2

Schedule 1 is not recoverable, because T2 reads data which is updated by T1 and commits before T1 commits. Schedule 2 is recoverable, since even though T2 reads data which is modified by T1, it only commits after T1 commits

2. Modify whichever schedule is recoverable to make it cascadeless.
Answer:

T1	T2
Write(A)	
Commit	
	Read(A)
	Write(A)
	Commit

Topic #4 Lock Based Protocols

1. Motivation and basics

(a) Why are lock-based protocols necessary?
To create isolation by preventing other transactions from accessing data that a transaction is modifying until that transaction has completed all modifications to that data.

(b) What are the two types of locks, and what do they allow?
Shared locks and exclusive locks. Shared locks allow the holder to read the data item and multiple transactions may hold shared locks on the same data item. Exclusive locks allow their holder to read and write the data item and prevent other transactions from gaining locks, either shared or exclusive, on the data item.

2. Two-phase locking

(a) What does two-phase locking guarantee?
Serializability.

(b) Does the partial schedule in Figure 7 adhere to two phase locking? If it does not adhere to two phase locking, then change it to adhere to two phase locking, and say whether the result is conflict serializable.

T_1	T_2
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
unlock(B)	
	lock-S(A)
	read(A)
	unlock(A)
	lock-S(B)
	read(B)
	unlock(B)
	display($A + B$)
lock-X(A)	
read(A)	
$A := A - 50$	
write(A)	
unlock(A)	

Figure 7: Problem 4.2.b: Two-phase locking

It does not adhere to two-phase locking. To make it adhere to twp phase locking, swap T1 unlock(B) with T1 lock-X(A) and swap T2 unlock(A) with T2 lock-S(B).

The result is serializable since T1 has exclusive locks to both A and B before T2 gets any locks, so either T2 will wait until T1 completes then T2 will start, which is in serial order, or T1 will be rolled back and T2 will get a shared lock on A and then T1 will start, getting an exclusive lock on B and when T1 requests an exclusive lock on A there will be deadlock, which will be dealt with by whatever your deadlock policy is, and which will necessarily mean that one schedule completes all its conflicting instructions before the other schedule starts its conflicting instructions, which means that it will be conflict serializable.

(c) What does strict two phase locking guarantee?
Cascadelessness.

(d) Adjust the result of whatever you answered to part (c) to adhere to strict two phase locking.
In T1, add a commit at the end, and then move both unlock(A) and unlock(B) to after the commit.

3. Starvation

What protocol must be adhered to to prevent starvation of a transaction?
The concurrency control manager should only issue a lock on a data item to a transaction if there is no other transaction waiting for a lock on that data item that made its lock request before this transaction.

4. Deadlock avoidance protocols

(a) The following two partial schedules attempt to use two phase locking but run into a lock conflict.

T1	T2
Lock-S(A)	
Read(A)	
	Lock-X(B)
	Read(B)
Lock-S(B)	
Unlock(A)	
Read(B)	
	Write(B)
	Unlock(B)
Unlock(B)	

Figure 8: Problem 4.4.a, Schedule 1

T1	T2
Lock-S(A)	
Read(A)	
	Lock-X(B)
	Read(B)
Lock-S(B)	
Unlock(A)	
Read(B)	
	Write(B)
	Lock-X(A)
	Unlock(B)
	Read(A)
	Write(A)
	Unlock(A)
Unlock(B)	

Figure 9: Problem 4.4.a, Schedule 2

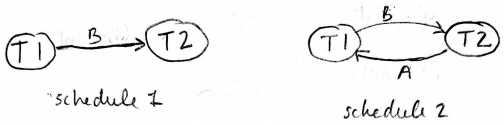
i. If using the wait die protocol, how is this resolved?
Schedule 1: When T1 requests a shared lock on B, which is currently under an exclusive lock of T2, T1 waits for T2 to complete, and then completes.
Schedule 2: When T1 requests a shared lock on B, which is currently under an exclusive lock of T2, it waits for T2 to complete. As T2 is completing, T2 requests an exclusive lock on A, which is still held by T1. Since T2 is the younger schedule, it "dies" and is rolled back. T1 then gets its shared lock on B and completes, after which T2 starts and completes.

ii. If using the wound-wait protocol, how is it resolved?
Schedule 1: When T1 requests a shared lock on B, which is currently under an exclusive lock of T2, T2 is "wounded" and rolled back. T1 gets a shared lock on B and completes. Then T2 begins and completes.
Schedule 2: Same as for schedule 1.

iii. The above protocols have the disadvantage of potentially causing unnecessary rollbacks. To avoid this, you might want to only issue a rollback in the case of deadlock. To determine when you have reached deadlock, you draw a dynamic wait-for graph and run cycle detection

on it.

Draw the wait for graphs of the Schedule 1 at the time of the instruction T2 write(B), and the wait-for graph of Schedule 2 at the time of the instruction T2 lock-X(A). Determine from these graphs which of the above schedules has deadlock. (Note: In a wait-for graph, T1 points to T2 if T1 requires a lock that T2 holds).
Answer:



The lack of cycles in the the first graph shows that it is not in deadlock. The cycle in the second graph shows that it is in deadlock.

Topic #5 Timestamp Based Protocols

Timestamp ordering is another way of guaranteeing serializability. The serial equivalent of a given schedule is a serial schedule that is in the order of the timestamps assigned to each transaction. Transactions that enter the system at the beginning of the schedule are arbitrarily assigned a unique timestamp, based on a logical counter or the system clock. Transactions that enter the system after the schedule has begun are issued timestamps when they enter which are greater than the timestamps of transactions already in the system.

In the following schedules, T1 starts with an initial timestamp smaller than T2. Attempt to implement the following schedules following the timestamp protocol outlined in the slides and in the textbook (p. 683). If a transaction gets rolled back and restarted at a later timestamp, retain it's original identifier (i.e. "T1" is still called "T1" even after it's been rolled back and restarted). When relevant, use Thomas's write rule.

What serial schedule do each of the following partial schedules end up being equivalent to (assuming that the only instructions not shown here are commits)?

T1	T2
read(B) display(B) read(A)	 write(A)

Figure 10: Problem 5, Schedule 1

T1	T2
read(B) write(A)	 read(A) write(A)

Figure 11: Problem 5, Schedule 2

T1	T2
read(B) write(A)	 write(A)

Figure 12: Problem 5, Schedule 3

Answer

1. Schedule 1: T2, T1
2. Schedule 2: T2, T1
3. Schedule 3: T1, T2

Topic #6 Recovery

1. Commits
   What defines a commit?
   When the log entry for the commit is written to non-volatile storage
2. Immediate and Deferred Database Modification
   (a) In immediate database modification, what happens when a write command is issued?
   The update is written to the disk buffer in memory, which is a part of memory holding data that will be eventually written back to disk, or it is written to disk.
   (b) How does this explain why it is necessary to undo or redo writes when there is a system failure?
   Since the update might not have been written to non-volatile memory, it can be lost if the system fails, so to guarantee atomicity and durability, instructions after a checkpoint must be either undone or redone (depending on whether the transaction has committed yet).
   (c) If we use deferred database modification, is it necessary to undo writes?
   No, it is unnecessary to undo writes since they aren't written to the disk buffer until commit, (after which, if there is a system failure, they must be redone rather than undone.)
3. Logging
   There are 4 types of log entries used in a database log that we studied in this course:
   1. $< T_i$ start$>$
   2. $< T_i, X_j, V_1, V_2 >$ (this is an update entry)
   3. $< T_i$ commit$>$
   4. $< T_i$ checkpoint$>$

   Why is V_1 in an update entry?
   V1 indicates the original value, and is kept so that in the case of rollback, the original value can be restored.
4. Recovery algorithm:
   The lecture slides slides give the following algorithm for recovering from system failures.
   (1) Initialize an undo list and a redo list
   (2) Scan the log from the end backward until the first encountered checkpoint, adding a transaction's ID to the redo list when a transaction commit is encountered, and adding a transaction ID to the undo list when a transaction start is encountered and the transaction is not in the redo list.
   (3) Pass backward from the end of the log undoing updates by transactions in the undo list. When a start entry for a transaction in the undo list is encountered, remove the transaction ID from the undo list. Continue until the undo list is empty.
   (4) Pass forward from the last checkpoint until the end of the log and redo (in memory) each log entry from a transaction in the redo list.

   (a) Given the following log:

Beginning of log
<T_0 start>
<T_0, B, 2000, 2050>
<T_1 start>
<checkpoint {T_0, T_1}>
<T_1, C, 700, 600>
<T_1 commit>
<T_2 start>
<T_2, A, 500, 400>

Figure 13: Problem 6.4.a, Database Log

i. After the analysis pass, which transaction IDs are in the undo list and which are in the redo list?
Undo list: [T0, T2]
Redo list: [T1]

ii. What are the values of A, B, and C after the algorithm has finished?
A: 500, B: 2000, C: 600

(b) Suppose that A's initial value is 0, and you have a partial schedule that adheres to two phase locking and looks like the schedule shown in Figure 14:

T1	T2
lock-X(A) X := 10 write(A) unlock(A)	
	lock-X(A) X := 20 write(A) unlock(A) commit

Figure 14: Problem 6.4.b, A Schedule

i. Does this schedule meet the requirements of recoverabiity and cascadelessness?
Yes, since T2 does not read data written by T1, it does not need to commit after T1 has committed to adhere to recoverabiity and cascadelessness.

ii. Does it meet the requirements of 2 phase locking?
Yes, since each transaction obrains all locks before releasing locks.

iii. Now suppose that the system fails at the end of the above schedule. T1 must be undone, and T2 redone.
If step 4 in the above recovery algorithm came before step 3, what error would result from this and how is this avoided by undoing before redoing?
After redoing T2, T1 would be undone and A would be set to 0, which is incorrect, because the final value of A should be 20. By undoing first and then redoing, we avoid this problem.

iv. Now, undoing before redoing is in accordance with the 4^{th} edition of the textbook (from 2004), p 659, however, according to the $5th$ and 6^{th} editions of the textbook, ($6th$ edition, pp. 753-755), redoing comes before undoing. If so, then to avoid the above problem, a recovery algorithm must require that if a data item has been modified by a transaction, no other transaction may modify that data item until the first transaction either is undone or is committed. What locking protocol may be used to ensure this?
Strict two-phase locking

v. [Note: The following questions further examine the discrepancies between the textbook's recovery algorithm and the slides' recovery algorithm, and will afford you a broader understanding of recovery, but will not be tested on in quiz 3.]

Even if the above requirement is met, if redo comes before undo, an error may arise. Consider the schedule shown in figure 15.

$<T_i,\ A,\ 10,\ 20>$
$<T_j,\ A,\ 10,\ 30>$
$<T_j$ commit$>$

Figure 15: Problem 6.4.b.v, A Schedule

This schedule implies that T_i was rolled back before T_j modified A. If the system now fails, then T_i must be undone a second time, and T_j redone.

If redo comes before undo, what problem does this cause?
After A is set to 30 in the redo pass, A is then set to 10 in the undo pass, effectively overriding T_j's modification of A.

One way to solve this is to keep track of transactions that have already been undone, so that they will not be undone a second time. The later versions of the textbook implement this by logging a $< T_i$ abort $>$ entry when a transaction has been aborted. During the analysis phase, transactions that have a $< T_i$ abort $>$ entry are omitted from the undo list.

vi. Another key difference between the recovery algorithm given in the later versions of the textbook and the recovery algorithm given in the slides is that the algorithm in the later editions redoes everything in the log from the last checkpoint forward so as to simplify the recovery process, rather than implementing a redo list. This is called "redoing history." However, this would seemingly cause an error when transactions in the log have been undone. Explain that error.
If everything is redone, then transactions that have been rolled back are also redone, so modify the database, even though they shouldn't.

vii. To resolve this, the later editions say to log a "redo-only" log entry of the form $< T_i, X_j, V >$, indicating that data item X_j has been restored to value V, after undoing an update. This is called a "redo-only" entry because it cannot be undone, since no original value is given. For example, in the schedule in figure 15, $< T_i, A, 10 >$ is entered into the log when T_i is rolled back. The log at the time of the system crash looks like this:

$< T_i, A, 10, 20 >$
$< T_i, A, 10 >$
$< T_i$ abort$>$
$< T_j, A, 10, 30 >$
$< T_j$ commit$>$

Explain how this resolves the problem mentioned above.
When redoing history, even if updates of aborted transactions are redone, the database is not corrupted, since updates of aborted transactions are always followed by redo-only log entries which undo the updates.
A complete outline of the 6*th* edition's recovery algorithm can be found on pp. 736-37.

Topic #7 Further resources for quiz 3 study

Once you have completed the above questions, the following textbook exercises will help you to absorb the material even better. It is less important that you do the exercises than that review the correct answers and understand them. The solutions can be found at https://www.db-book.com/db6/practice-exer-dir/index.html

1. Ch. 14: 1, 5, 6, 7, 11
2. Ch 15: 1, 2, 11
3. Ch. 16: 1, 2, 5, 7