SYSTEM LOG EXAMPLE

[start-trans T_1]

...

[commit T_1]

[start-trans T_2]

....

A:

[checkpoint] ⇒ commit before this point ⇒ update to database

B: ....

[commit T_2]

C:

SYSTEM LOG EXAMPLE
Recovery Scheme

	IMMEDIATE	DEFERRED
A	REDO T_1, UNDO T_2	REDO T_1
B	UNDO T_2	—
C	REDO T_2	REDO T_2

Immediate:
1. Commited but not update to database ⇒ Redo
2. Commited and update to database ⇒ Nothing
3. Not commited ⇒ Undo

Deferred:
1. Undo are ignored
2. Redo must be redone

SYSTEM LOG

- Records all transaction operations that affect the value of database items to be able to recover from failure.
- Log is kept on disk & periodically backed up to archival storage.

Let T be a unique transaction ID, X be a database item.

The definition of T

LOG ENTRIES
- [start-transaction, T]
- [write-item, T, X, new_value], or
  [write-item, T, X, old_value, new_value]
- [read-item, T, X]
- [commit, T]

INCREMENTAL LOG WITH IMMEDIATE UPDATES

This protocol uses

[start-transaction, T]
[read-item, T, X]
[write-item, T, X, old_value, new_value] ← deferred
[commit, T] ← deferred writes are executed

Log records must be written to "stable storage" before executing an immediate update.

INCREMENTAL LOG WITH DEFERRED UPDATES

- During the execution of a transaction, all the write operations are deferred, recorded on a log and transaction workspace, until the transaction partially commits.
- At commit point, log is force written to disk and deferred writes are executed.

This protocol uses entries:

[start-transaction, T]
[write-item, T, X, new_value]
[commit, T]

CHECKPOINTS

Log entry used to improve performance of recovery

[checkpoint] indicates that the system has written to the disk the effect of all write operations of committed transactions.

⇒ Transactions that have a commit entry before a checkpoint entry will **NOT** require their write operations to be redone in case of a system failure.

A checkpoint may contain additional info
- list of active IDs
- locations of first and most recent records in the log

SYSTEM LOG ASSUMPTIONS

- Transactions are not nested
- All permanent changes to the DB occur within transactions; so using the log, operations can be undone or redone based on transaction state.

INCREMENTAL LOG WITH IMMEDIATE UPDATES

Recovery scheme:

Undo(T_i) where T_i is uncommitted transaction
Redo(T_j) where T_j is committed transaction

Redo & Undo must be idempotent – "executing it several times is equivalent to executing it once."

INCREMENTAL LOG WITH DEFERRED UPDATES

Recovery scheme

Undo
- Log entries for uncommitted transactions are ignored because writes are deferred.
- Committed transactions must be redone (in order). Redo must be idempotent.

Redo

Consider the following schedule for three transaction: T1, T2, and T3.

R3(Y) R3(Z) R1(X) W1(X) W3(Y) W3(Z) R2(Z) R1(Y) W1(Y) R2(Y) W2(Y) R2(X) W2(X)

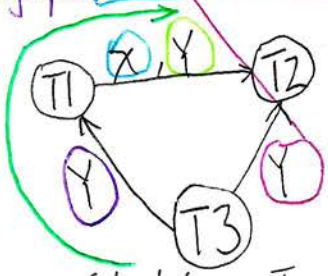
(a) Draw a precedence graph to show that the above schedule is serializable. Also, show the serialization order of the transactions as indicated by the graph.

key: Precedence Graph

1. Create a Node for every transaction T_i
2. If [read(X), T_i] executed before [write(X), T_j], then draw $T_i \rightarrow T_j$.

Serializable

If the graph contains a cycle, it is not serializable. Otherwise, it is serializable.



∴ Serialization Schedule: T_3, T_1, T_2

[18 pts.] 5. Concurrency Control.

Assume the following interleaved, serializable transaction execution schedule:

T1	T2	T3
LX(X)		LX(Z)
R(X)		R(Z)
W(X)		W(Z)
LX(Y)		
UN(X)		
	LS(X)	
	R(X)	
		LS(X)
		R(X)
		UN(X)
		UN(Z)
R(Y)		
W(Y)		
	LX(Z)	
	R(Z)	
	W(Z)	
	UN(X)	
	UN(Z)	
UN(Y)		

[6 pts.] a. Does the locking scheme conform to two-phase locking requirements? Why or why not?

Yes, because all 3 transactions complete their locking operations before their first unlocking operation.

[6 pts.] b. What is the concept of isolation in interleaved transaction execution? Identify an advantage and a disadvantage of isolation.

Isolation means that a transaction should not reveal its uncommitted results to other transactions.

Adv: No cascading rollbacks

Dis: The system executing each operation of a transaction achieves less concurrency.

[6 pts.] c. Does the above schedule support isolation? If yes, explain why. If not, is it possible to change the lock and unlock sequence without changing the interleaved sequence of reads and writes so that isolation is supported? Explain your answer.

No it does not, since T2 reads uncommitted update value of X by T1.

No, since T1 should not release its exclusive lock on X until its commit point, which is after its W(Y).

R3(Y) R3(Z) R1(X) W1(X) W3(Y) W3(Z) R2 (Z) R1(Y) W1(Y) R2 (Y) W2 (Y) R2 (X) W2 (X)

[14 pts.] b. Is the above schedule accepted by the timestamping protocol? Prove or disprove your answer by showing the read and write timestamps of each variable during the transaction execution.

Operation	X		Y		Z	
	R	W	R	W	R	W
R3(Y)	0	0	0	0	0	0
R3(Z)	0	0	0	0	0	0
R1(X)	1	0	0	0	0	0
W1(X)	1	1	0	0	0	0
W3(Y)	1	1	0	0	0	0
W3(Z)	1	1	0	0	0	0
R2(Z)	1	1	0	0	2	0
R1(Y)	1	1	1	0	2	0
W1(Y)	1	1	1	1	2	0
R2(Y)	1	1	2	1	2	0
W2(Y)	1	1	2	2	2	0
R2(X)	2	1	2	2	2	0
W2(X)	2	2	2	2	2	0

Let TS: timestamp of transaction requesting operation

READ(x,TS)
if TS < write_TS(x)
reject & restart
else
execute read
read_TS(x) ← max(read_TS(x), TS)

WRITE(x,TS) /* assume read before write */
if TS < read_TS(x)
reject & restart
else
execute write
write_TS ← TS

TS(Ti): unique fixed timestamp for trans Ti; assigned by the db system before Ti starts execution

If a trans Ti has timestamp TS(Ti) and a new trans Tj enters the system, then TS(Ti) < TS(Tj)

write_TS(X):

the largest timestamp of any trans that successfully executes write(X)

read_TS(X):

the largest timestamp of any trans that successfully executes read(X)

Timestamp