Discussion 9

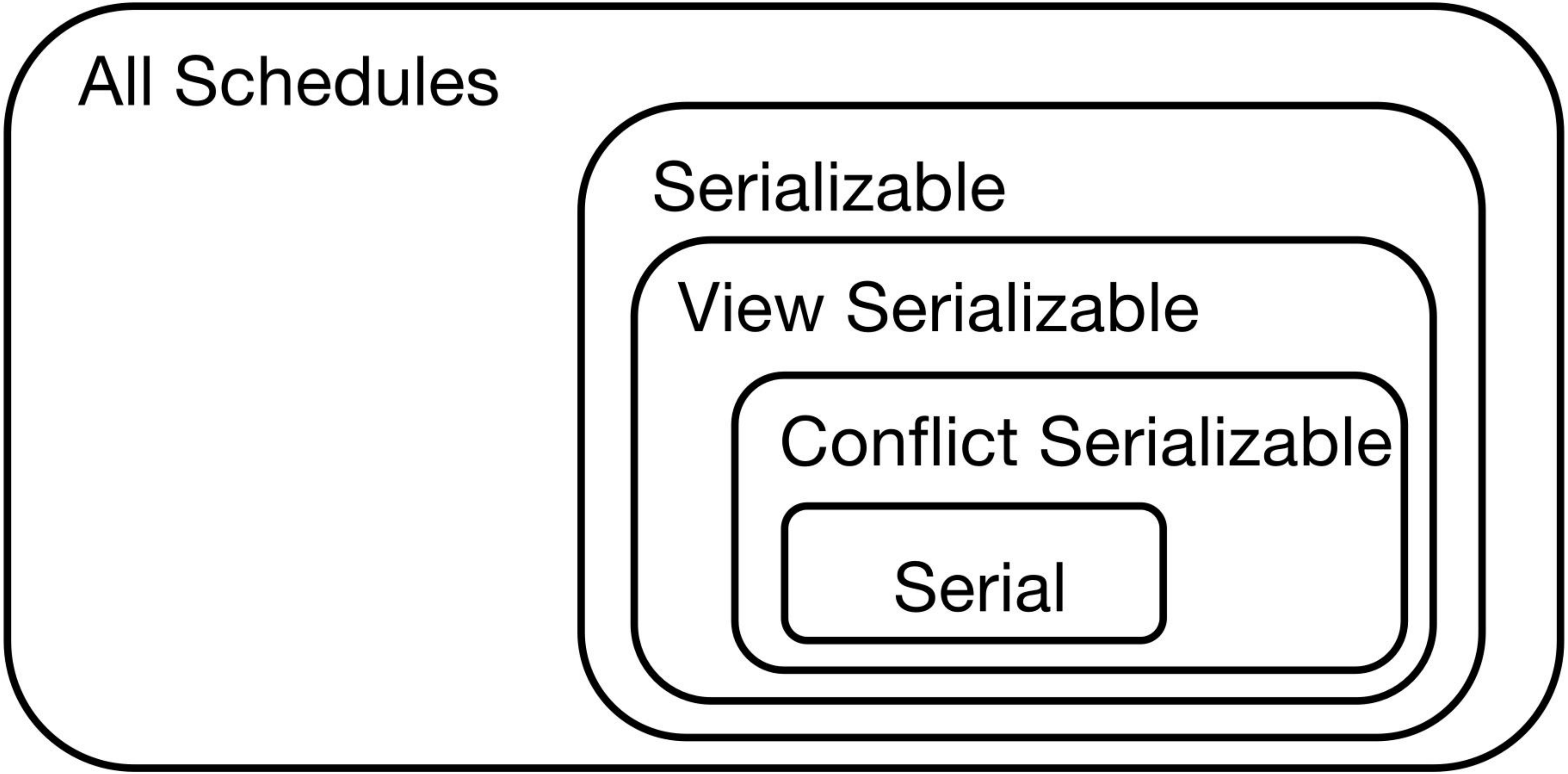
Lock & Recovery

Transactions-ACID

Transactions

- We want transactions to obey ACID
  - atomicity: all operations in a transaction happen, or none of them
  - consistency: database consistency (unique constraints, etc) is maintained
  - isolation: should look like we only run 1 transaction at a time (even if we run multiple concurrently)
  - durability: once a transaction commits, it persists
- commit: indicates successful transaction (save changes)
- abort: indicates unsuccessful transaction (revert changes)

Types of Serializability



lock

TXNs obtain:

- An **X (*exclusive*) lock** on object before **writing**.
  - If a TXN holds, no other TXN can get a lock (S or X) on that object.
- An **S (*shared*) lock** on object before **reading**
  - If a TXN holds, no other TXN can get *an X lock* on that object
- TXNs cannot get new locks after releasing any locks.

Simple Locking

- The lock compatibility matrix looks like:

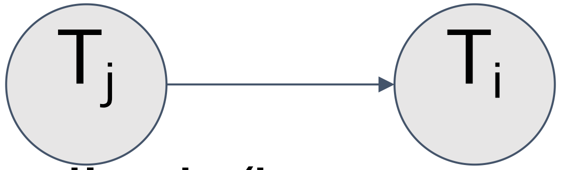
	NL (no lock held)	S	X
NL (no lock held)	✓	✓	✓
S	✓	✓	
X	✓		

Deadlocks

- What if T_1 is waiting for T_2 to release a lock, but T_2 is also waiting for T_1 to release a lock?
  - This is called a deadlock - when a bunch of transactions are waiting on each other in a cycle
- We can either avoid them in the first place (deadlock avoidance) or catch them (deadlock detection) and abort a transaction in the deadlock

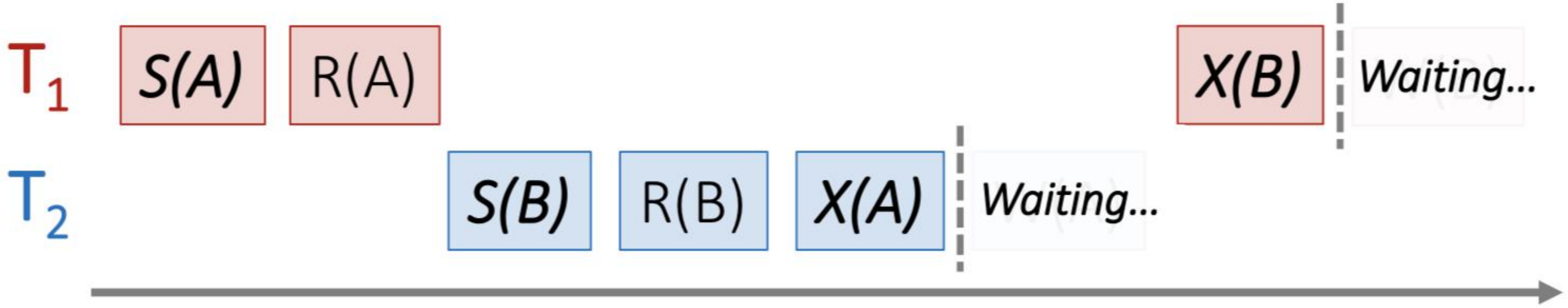
Deadlock Detection

- We draw out a "waits-for" graph
  - One node for each transaction
  - If T_i holds a lock that conflicts with the lock that T_j wants (or T_j "waits for" T_i), we add an edge from T_j to T_i

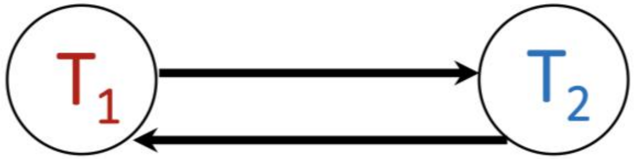


  - A cycle indicates a deadlock (between the transactions in the cycle) - we can abort one to end the deadlock
- Alternative approach (used in some real databases): just kill transactions if they aren't doing anything for a while

Deadlock Detection: Example



Waits-for graph:



Cycle = DEADLOCK

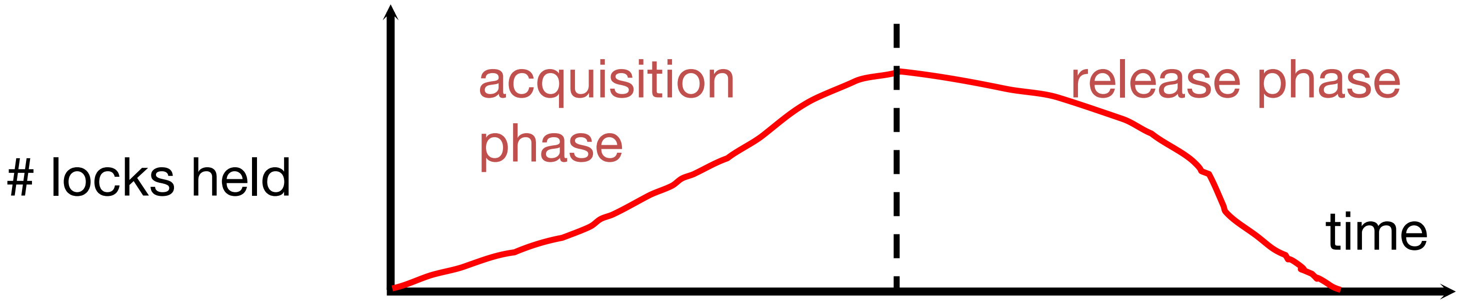
Finally, T_1 requests an exclusive lock on B to write to it- **now T_1 is waiting on T_2... DEADLOCK!**

2-Phase Locking (2PL)

- One way to enforce conflict serializability
- In 2-phase locking,
  - a transaction may not acquire a lock after it has released any lock
  - two “phases”

    from start to until a lock is released, the transaction is only acquiring locks

    then until the end of the transaction, it is only releasing locks



T1	T2
Lock_X(A)	
Read(A)	
	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Unlock(A)	
	Read(A)
	Lock_S(B)
Read(B)	
B := B +50	
Write(B)	
Unlock(B)	
	Unlock(A)
	Read(B)
	Unlock(B)
	PRINT(A+B)

Cascading Aborts

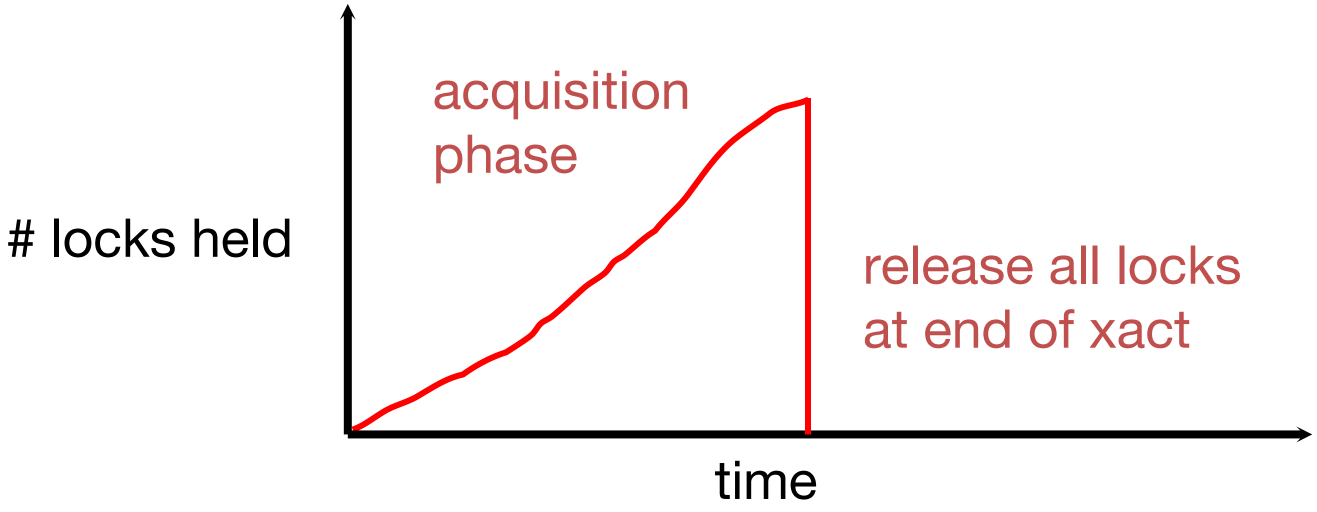
- Consider the following (+ for acquire, - for release):

T_1	+X(A)	-X(A)						abort!
T_2			+S(A)					
T_3			+S(A)	+X(B)	-X(B)			
T_4						+S(B)		
T_5							+S(B)	

- T_1 aborting means all of the transactions have to be rolled back
  - Even though T_4 and T_5 didn't read A at all
  - This is a cascading abort

Strict 2-Phase Locking (Strict 2PL)

- The problem is that 2PL lets another transaction read new values before the transaction commits (since locks can be released long before commit)
- Strict 2PL avoids cascading aborts
  - Same as 2PL, except only allow releasing locks at end of transaction



T1	T2
Lock_X(A)	
Read(A)	
	Lock_S(A)
A: = A-50	
Write(A)	
Lock_X(B)	
Read(B)	
B := B +50	
Write(B)	
Unlock(A)	
Unlock(B)	
	Read(A)
	Lock_S(B)
	Read(B)
	PRINT(A+B)
	Unlock(A)
	Unlock(B)

Strict 2PL

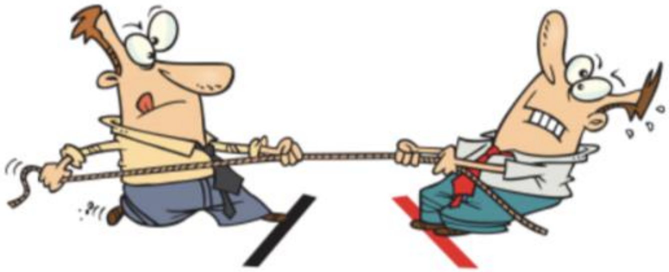
Theorem: Strict 2PL allows only schedules whose dependency graph is acyclic

Proof Intuition: In strict 2PL, if there is an edge $T_i \rightarrow T_j$ (i.e. T_i and T_j conflict) then T_j needs to wait until T_i is finished – so *cannot* have an edge $T_j \rightarrow T_i$

Therefore, Strict 2PL only allows conflict serializable ⇒ serializable schedules

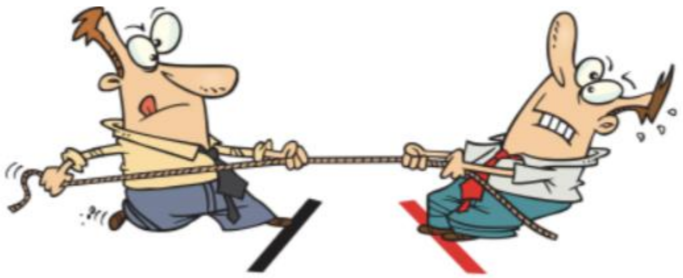
logging

Two Important Logging Decisions



- **Decision 1: STEAL or NO-STEAL**
  - Impacts ATOMICITY and UNDO
  - Steal: allow the buffer pool (or another txn) to “steal” a pinned page of an uncommitted txn by flushing to disk
  - No-steal: disallow
  - If we allow “Steal”, then need to deal with uncommitted txn edits appearing on disk
    - To ensure Atomicity we need to support UNDO of uncommitted txns
  - OTOH “No-steal” has poor performance (pinned pages limit buffer replacement)
    - But no UNDO required. Atomicity for free.

Two Important Logging Decisions



- **Decision 2: FORCE or NO-FORCE**
    - Impacts DURABILITY and REDO

    - Force: ensure that all updates of a transaction is “forced” to disk prior to commit
    - No-force: no need to ensure

    - If we allow “No-force”, then need to deal with committed txns not being durable
        - To ensure Durability we need to support REDO of committed txns
    - OTOH, “Force” has poor performance (lots of random I/O to commit)
        - But no REDO required, Durability for free.

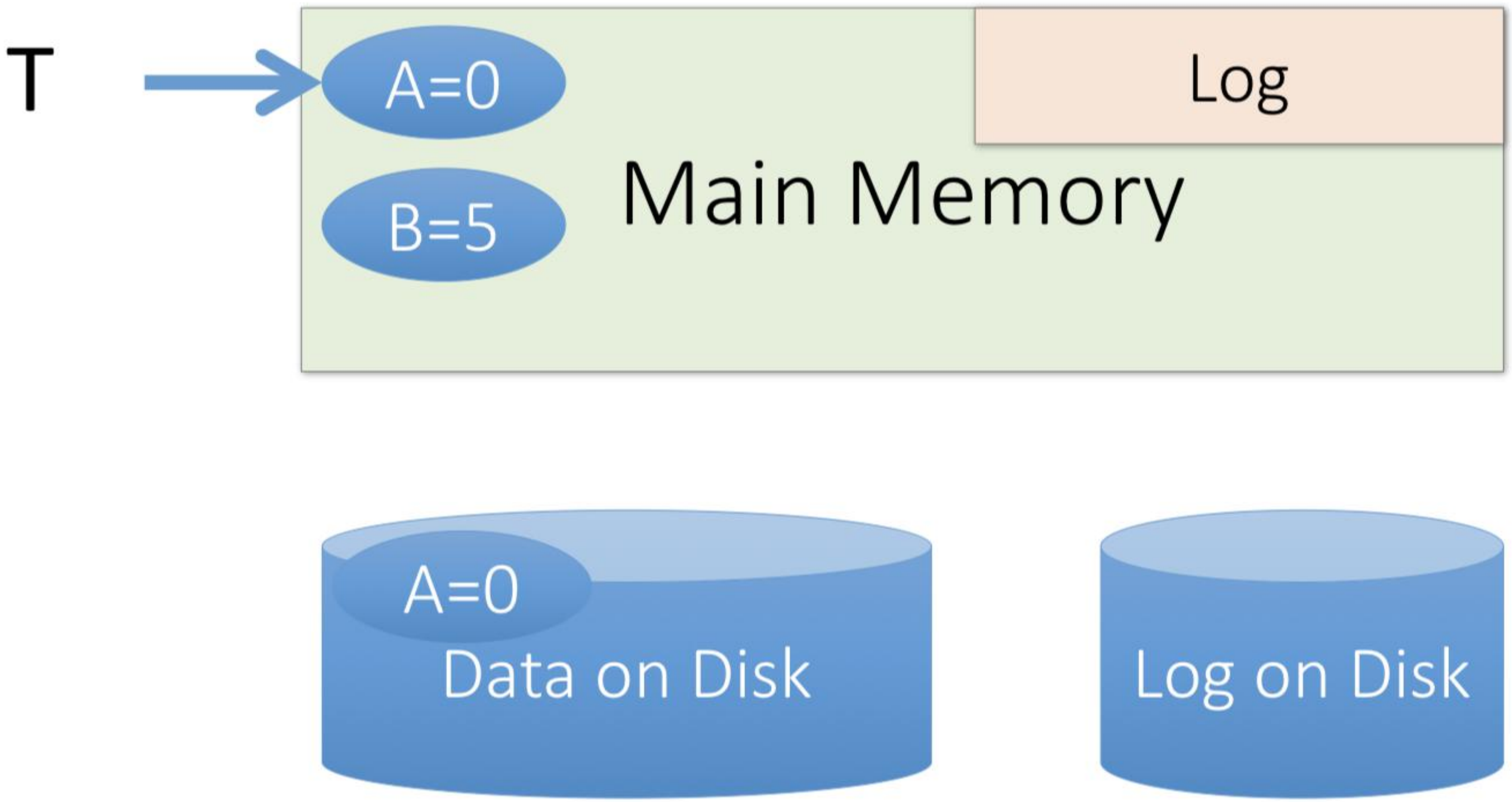
- The **log** consists of **an ordered list of actions**
  - Log record contains:

    <XID, location, old data, new data>

This is sufficient to UNDO any transaction!

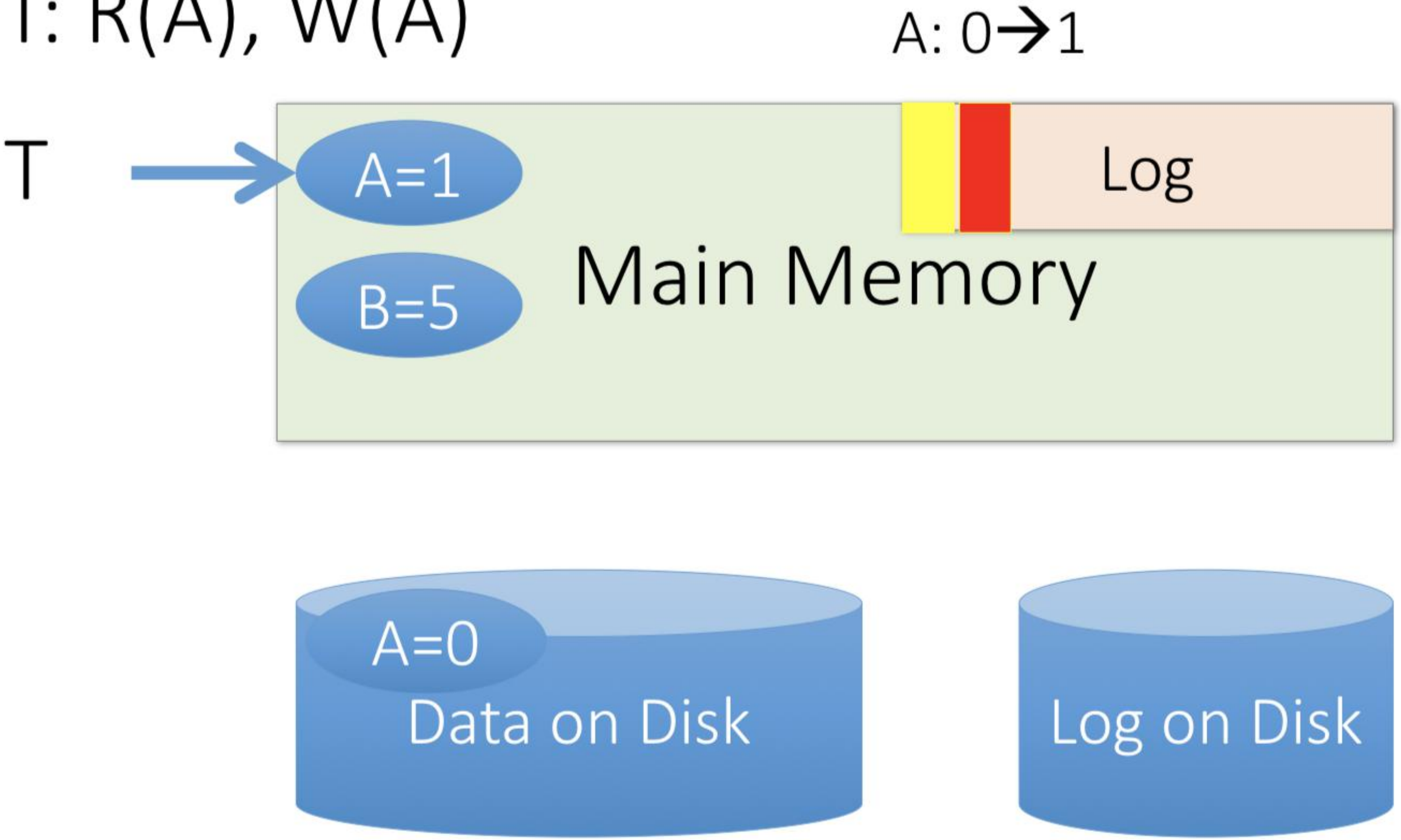
A picture of logging

T: R(A), W(A)



Incorrect Commit Protocol #1

T: R(A), W(A)



Let's try committing *before* we've written either data or log to disk...

OK, Commit!

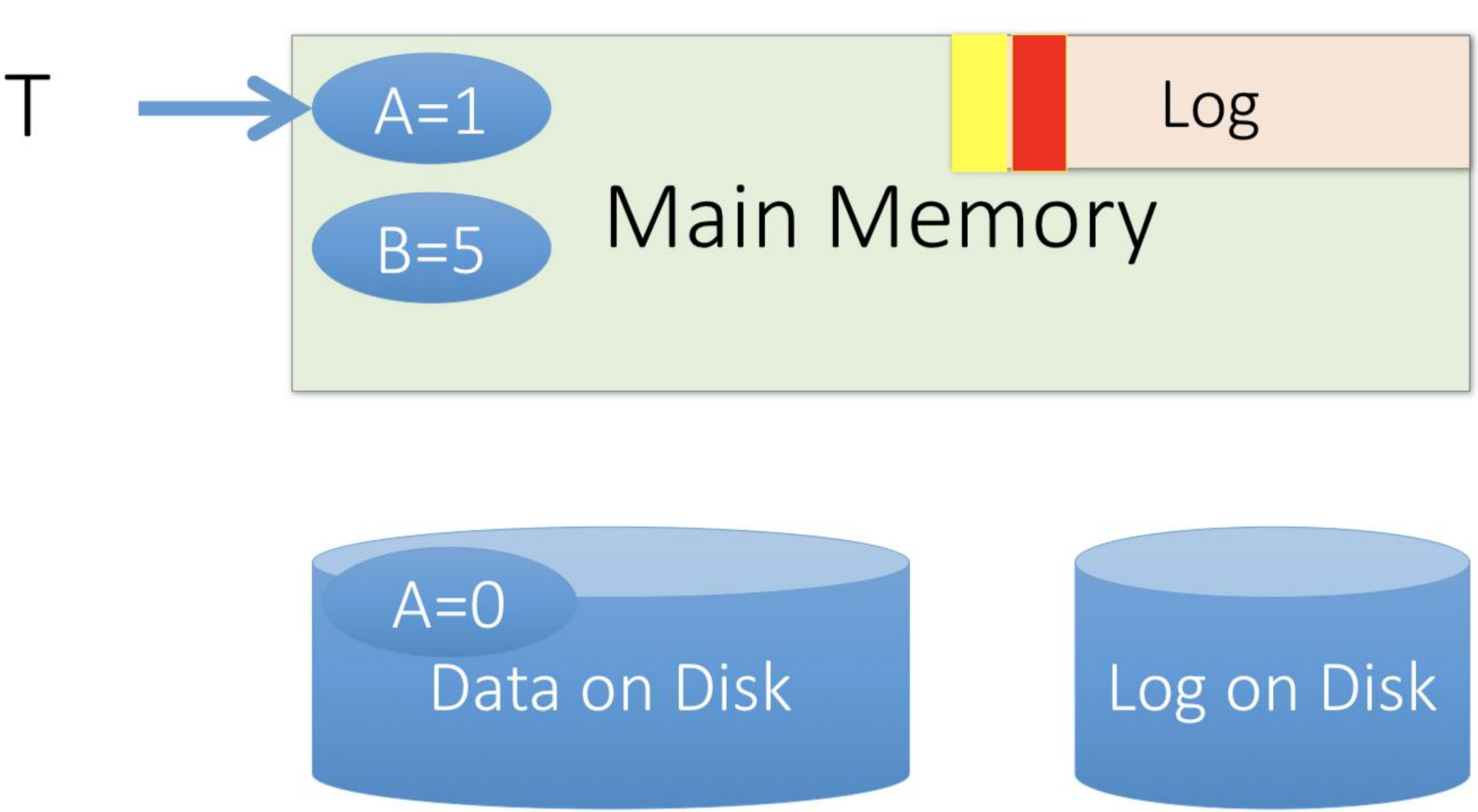
If we crash now, is T durable?

Lost T's update!

Incorrect Commit Protocol #2

T: R(A), W(A)

A: 0→1



Let's try committing *after* we've written data but *before* we've written log to disk…

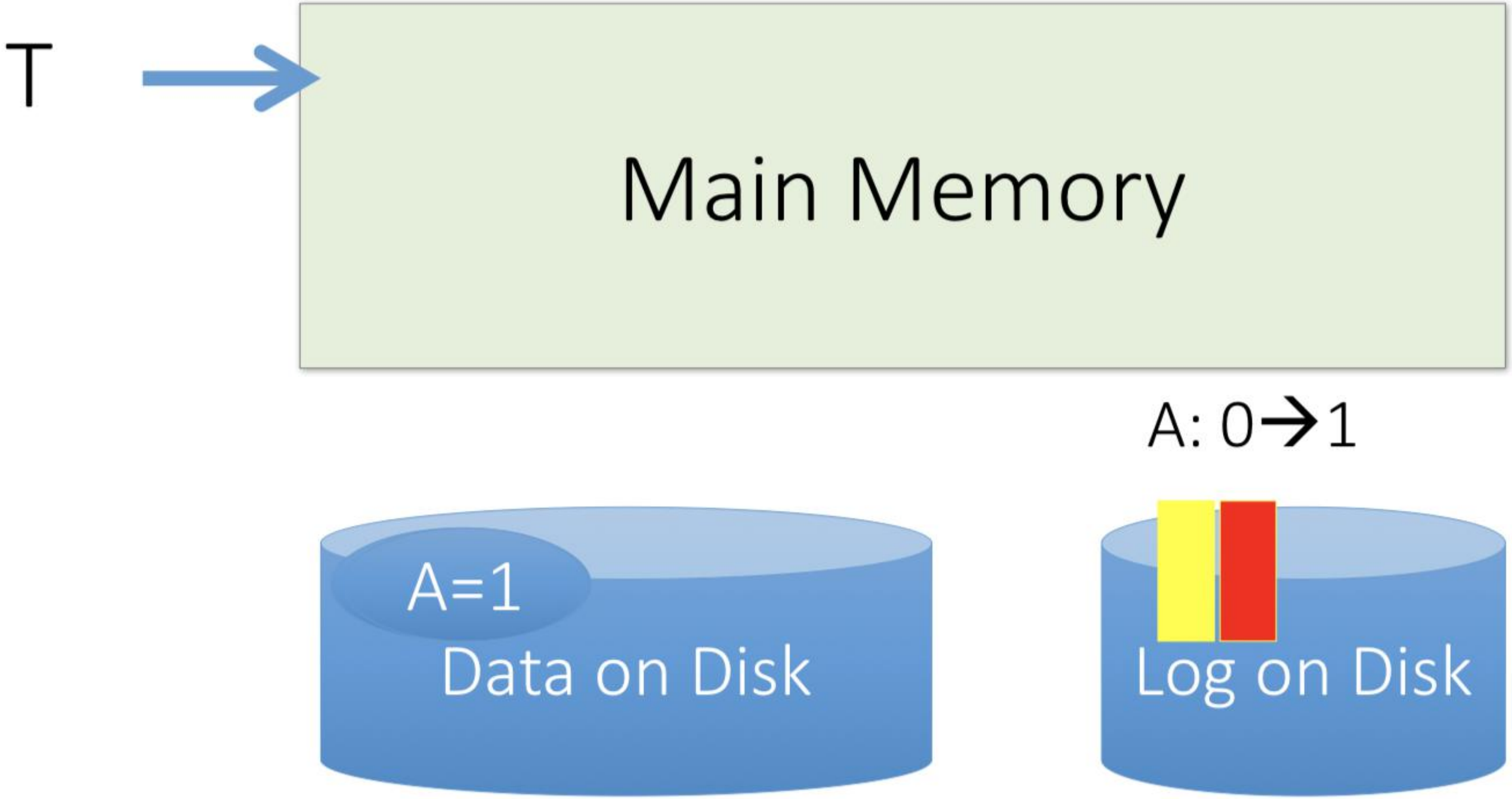
OK, Commit!

If we crash now, is T durable? Yes! Except…

How do we know whether T was committed??

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)



This time, let's try committing ***after*** **we've written log to disk but** ***before*** **we've written data to disk**... this is WAL!

OK, Commit!

If we crash now, is T durable?

USE THE LOG!

WAL

- The **Write-Ahead Logging Protocol** :
  1. Must **force** the **log record** for an update **before** the corresponding **data page** gets to the DB disk.
  2. Must **force all log records** for a Xact **before commit** .
     - I.e. transaction is not committed until
       all of its log records including its "commit" record are on the stable log.