CS 6400 A

Database Systems Concepts and Design

Lecture 20

11/09/26

Announcements

Rank	Submission Name	Time
1	(⌒.⌒)♡	335.0154047348159
2	Akshar Ravichandran	681.5265325501618
3	Surabhi	708.8635344557608

Quiz 2 logistics

- Wednesday Nov 11, 30min
- Quiz will start promptly at 5PM. Please arrive a few minutes early.
- Format: Open book and notes, closed Internet (same as midterm)
- Contents covered: lec 14 (joins) – lec 16 (qo)

Assignment 3 due next Wednesday (Nov 18)

Desirable Properties of Transactions: ACID

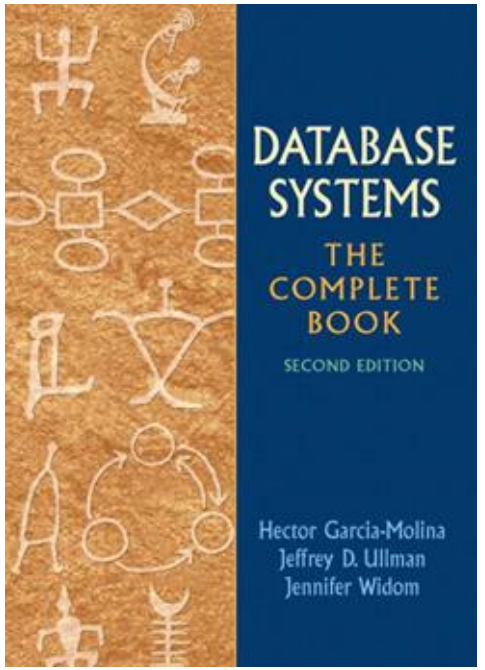
- **Atomicity**: A transaction is an atomic unit of processing; it is either performed in its entirety or not performed at all.
- **Consistency**: A correct execution of the transaction must take the database from one consistent state to another.
- **Isolation**: A transaction should not make its updates visible to other transactions until it is committed.
- **Durability**: Once a transaction changes the database and the changes are committed, these changes must never be lost because of subsequent failure.

This class: ensuring atomicity and durability with logging and recovery manager

Reading Materials

Database Systems: The Complete Book (2nd edition)

- Chapter 17 - Copying with System Failures



Supplementary materials

Fundamental of Database Systems (7th Edition)

- Chapter 22 - Database Recovery Techniques

Agenda

1. WAL Protocol
2. Undo Logging
3. Redo Logging
4. Undo/redo logging

Failure modes and solutions

Erroneous data entry

- Typos
  → Write constraints and triggers

Media failures

- Local disk failure, head crashes
  → Parity checks, RAID, archiving and copying

System crashes

- Transaction state lost due to power loss and software errors
  → Logging and recovery

Our focus today

Overview: Recovery Mechanism

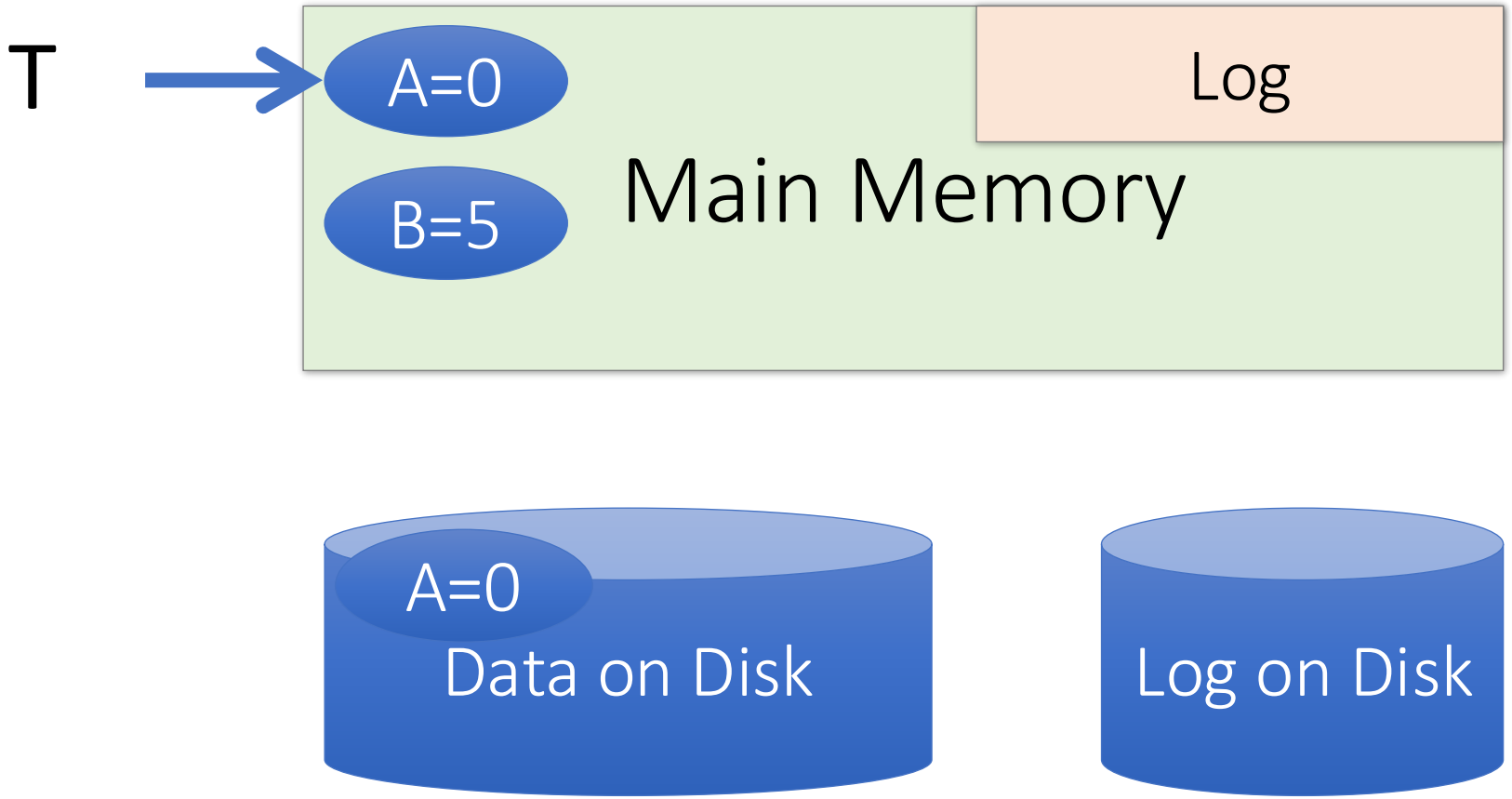
Crash scenario	Recovery Action	ACID property
Transaction has not committed, but some updates reached disk	UNDO incomplete transactions	Atomicity
Transaction committed, but some updates have not reached disk	REDO committed transactions	Durability

Recovery must ensure that incomplete transactions have no effect and committed transactions have all their effects.

1. Write-Ahead Logging (WAL) and Commit Protocol

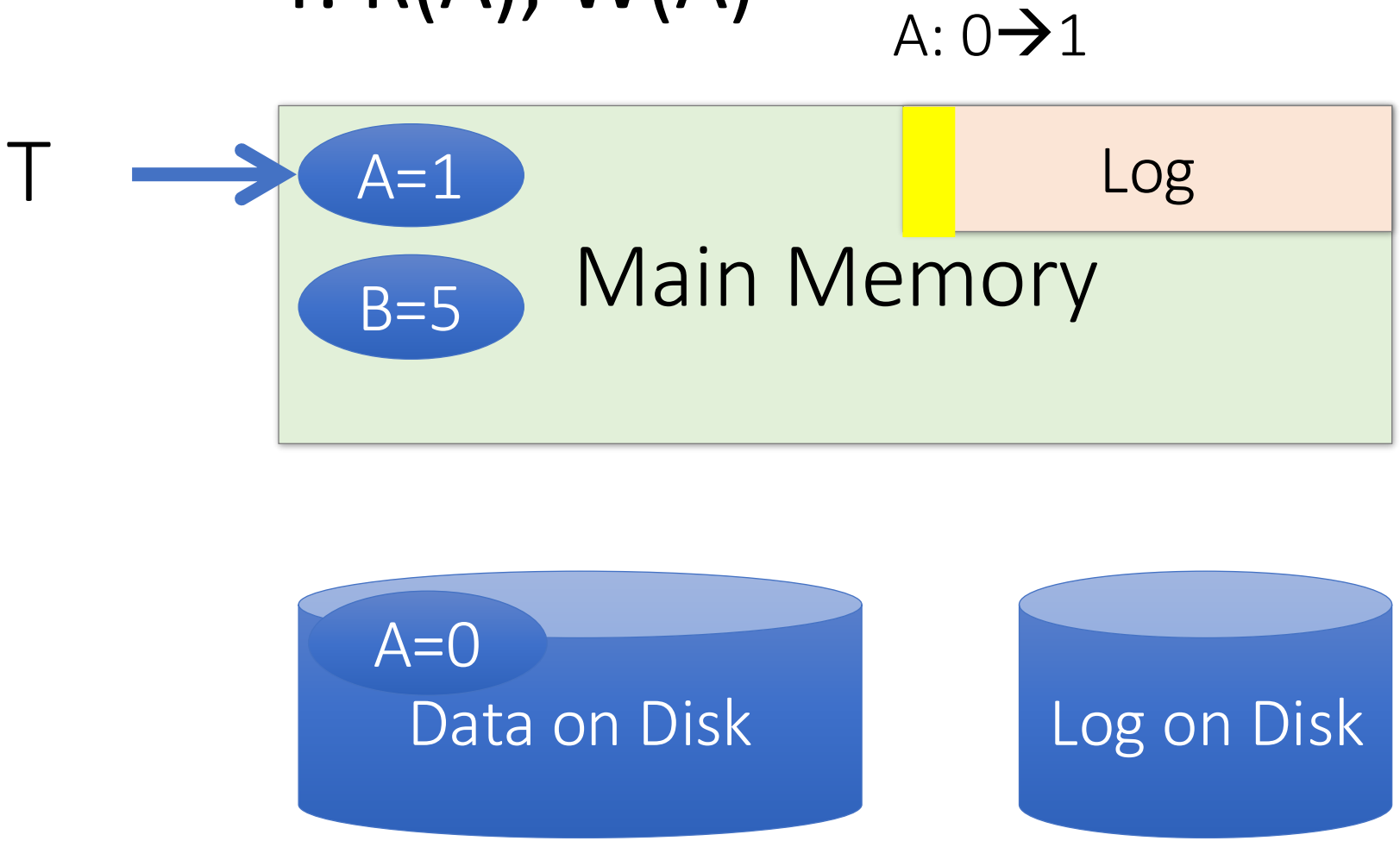
A picture of logging

T: R(A), W(A)



A picture of logging

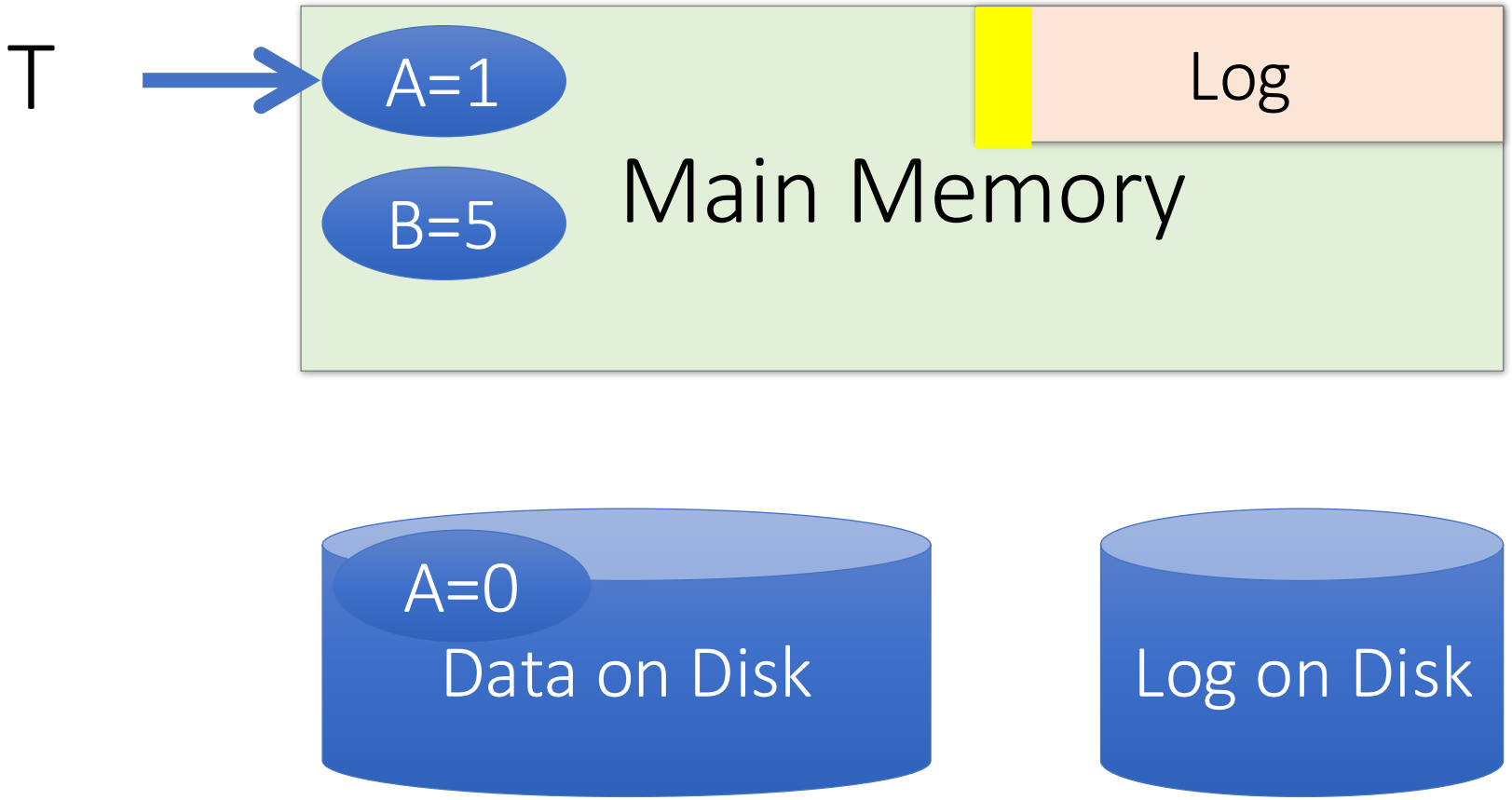
T: R(A), W(A)



A picture of logging

T: R(A), W(A)

A: 0→1



Operation recorded in log in main memory!

Transaction Commit Protocol

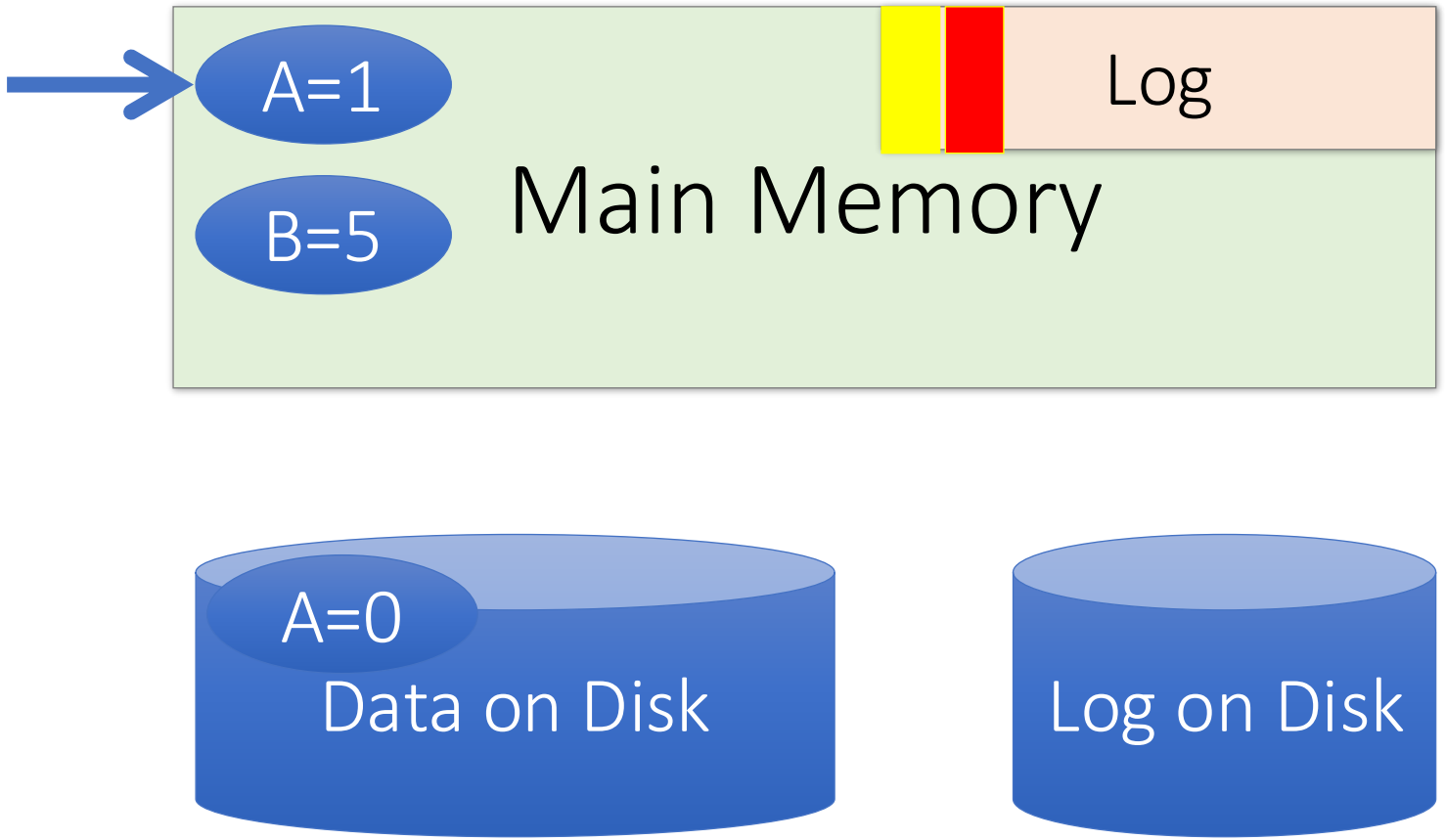
- COMMIT marks the transaction as successfully completed.
- Once COMMIT succeeds, its changes must survive a crash (durability).
- A transaction performs updates in memory, but updates in memory may be lost during a crash.

What needs to be written to disk, and in what order, to ensure durability?

Incorrect Commit Protocol #1

T: R(A), W(A)

A: 0→1



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!

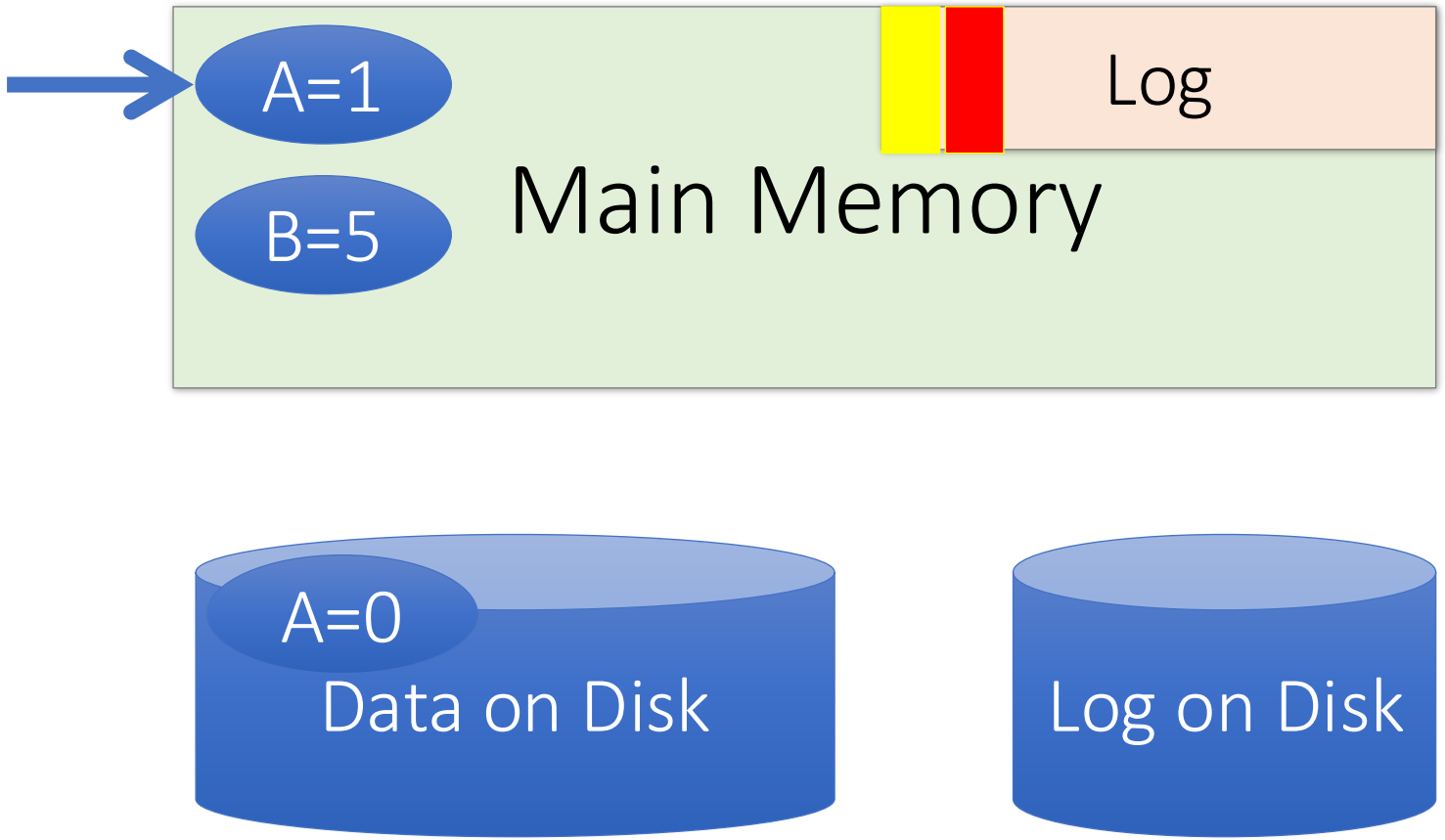
If we crash now, is T durable? Yes! Except…

How do we know whether T was committed??

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)

A: 0→1



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?

Write-ahead Logging (WAL) Commit Protocol

T: R(A), W(A)

T 

A: 0→1





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!

Write-Ahead Logging (WAL)

WAL only log writes!

Rule 1: Log before data

- Before writing a modified data page to disk, first flush the corresponding log record.
- Ensures recovery information exists even if uncommitted changes reach disk (supports atomicity).

Rule 2: Log at commit

- Before reporting a successful commit, the transaction's log records, including COMMIT, must be flushed.
- Ensures the log contains a durable record of committed transactions (supports durability).

Different Logging Schemes

WAL determines *when* log records must reach disk.

Different logging schemes determine *what* information is recorded and *how* recovery uses it.

Logging Scheme	When can data pages be flushed?	Recovery
Undo	Before commit; all must be flushed by commit	Undo incomplete
Redo	After commit	Redo committed
Undo/Redo	Before or after commit	Both

Transaction primitives

- INPUT(X): copy block X from disk to memory
- READ(X, t): copy X to transaction's local variable t
  (run INPUT(X) if X is not in memory)
- WRITE(X, t): copy value of t to X (run INPUT(X) if X is not in memory)
- OUTPUT(X): copy X from memory to disk

Transaction primitives

- Example transaction
  - Consistent state: $A = B$

Logical steps

$A := A * 2$
$B := B * 2$

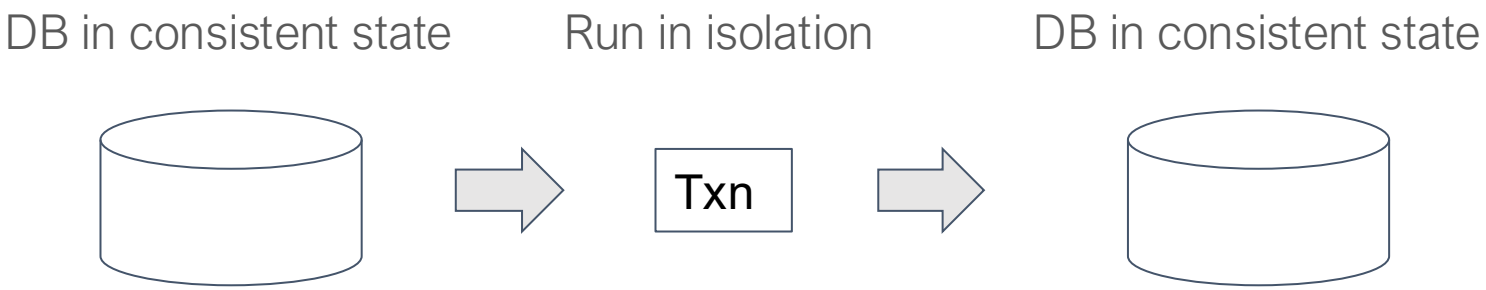
Execution

		Memory		Disk	
Action	t	A	B	A	B
READ(A, t)	8	8		8	8
$t := t * 2$	16	8		8	8
WRITE(A, t)	16	16		8	8
READ(B, t)	8	16	8	8	8
$t := t * 2$	16	16	8	8	8
WRITE(B, t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Recall: The Correctness Principle

A fundamental assumption about transaction is:

If a transaction executes in the absence of any other transactions or system errors, and it starts with the database in a consistent state, then the database is also in a consistent state when the transactions ends.



Transaction primitives

- Example transaction
  - Consistent state: $A = B$

Logical steps

$A := A * 2$
$B := B * 2$

Execution

		Memory		Disk	
Action	t	A	B	A	B
READ(A, t)	8	8		8	8
$t := t * 2$	16	8		8	8
WRITE(A, t)	16	16		8	8
READ(B, t)	8	16	8	8	8
$t := t * 2$	16	16	8	8	8
WRITE(B, t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Consistent

Transaction primitives

- Example transaction
  - Consistent state: $A = B$

Logical steps

$A := A * 2$
$B := B * 2$

Execution

		Memory		Disk	
Action	t	A	B	A	B
READ(A, t)	8	8		8	8
$t := t * 2$	16	8		8	8
WRITE(A, t)	16	16		8	8
READ(B, t)	8	16	8	8	8
$t := t * 2$	16	16	8	8	8
WRITE(B, t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Consistent

Transaction primitives

- Example transaction
  - Consistent state: $A = B$

Logical steps

$A := A * 2$
$B := B * 2$

Execution

		Memory		Disk	
Action	t	A	B	A	B
READ(A, t)	8	8		8	8
$t := t * 2$	16	8		8	8
WRITE(A, t)	16	16		8	8
READ(B, t)	8	16	8	8	8
$t := t * 2$	16	16	8	8	8
WRITE(B, t)	16	16	16	8	8
OUTPUT(A)	16	16	16	16	8
OUTPUT(B)	16	16	16	16	16

Not consistent!
Either reset $A = 8$
or advance $B = 16$

2. Undo logging

Undo logging

- Idea: Undo incomplete transactions, and ignore committed ones

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Undo log format:

<*T*, *X*, *v*>: T updated database element X whose old value is v

Undo logging

- Idea: Undo incomplete transactions, and ignore committed ones

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

T started

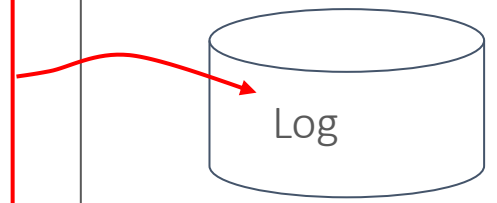
T changed *A*, and its former value is 8

T completed successfully

Undo logging

- Idea: Undo incomplete transactions, and ignore committed ones

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						



Rule 1 (WAL rule):
Flush the undo log record before writing the modified data to disk

Undo logging

- Idea: Undo incomplete transactions, and ignore committed ones

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						



Rule 1:
<*T, A,* 8> must be flushed to disk before new *A* is written to disk (same for *B*)

Rule 2 (Undo-logging commit rule):
Flush all modified data to disk before flushing the COMMIT record.

Undo logging

- Idea: Undo incomplete transactions, and ignore committed ones

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Flush order:

a) Log record

b) Data pages

c) COMMIT record

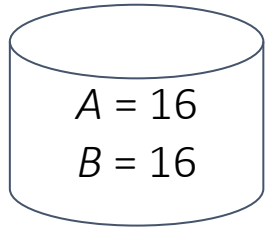
Recovery using undo logging

* Simplifying assumption: use entire log

	Memory			Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

Recovery

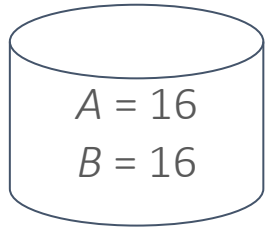


Recovery using undo logging

* Simplifying assumption: use entire log

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Recovery



Observe <COMMIT *T*> record

Crash

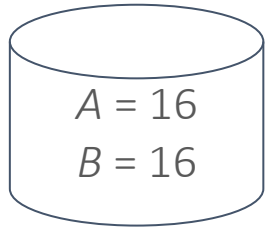
Recovery using undo logging

* Simplifying assumption: use entire log

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

Recovery



Ignore (*T* was committed)



Observe <COMMIT *T*> record

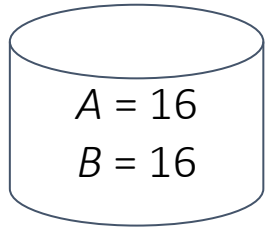
Recovery using undo logging

* Simplifying assumption: use entire log

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

Recovery



Ignore (*T* was committed)

Ignore (*T* was committed)

Observe <COMMIT *T*> record

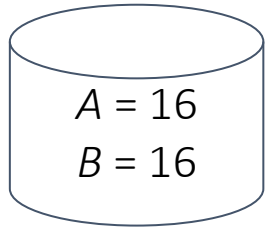
Recovery using undo logging

* Simplifying assumption: use entire log

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

Recovery



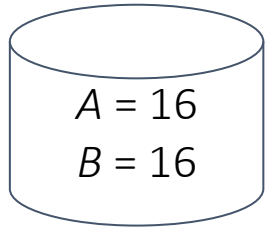
Recovery using undo logging

* Simplifying assumption: use entire log

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

Recovery



<COMMIT *T*> may or may not have been flushed to disk. If so, same as previous scenario. If not, *T* is considered incomplete

Recovery using undo logging

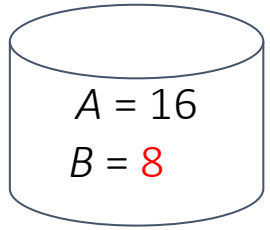
* Simplifying assumption: use entire log

	Memory			Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

If *T* was incomplete, set *B* to previous value 8 on disk

Recovery



Recovery using undo logging

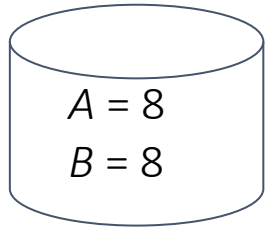
* Simplifying assumption: use entire log

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

If *T* was incomplete, set *A* to previous value 8 on disk

Recovery



Recovery using undo logging

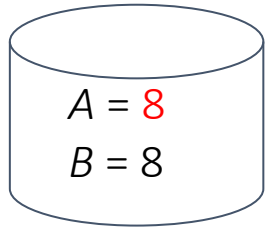
* Simplifying assumption: use entire log

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

Recovery

Write <ABORT *T*> to log and flush to disk



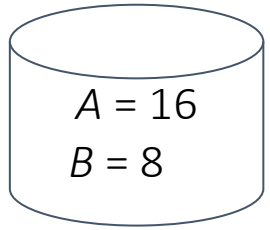
Recovery using undo logging

* Simplifying assumption: use entire log

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

Recovery



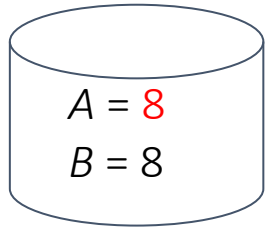
Recovery using undo logging

* Simplifying assumption: use entire log

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	
						<COMMIT *T*>
FLUSH LOG						

Crash

Recovery



Same recovery as before, but only *A* is set to previous value

What happens if the system crashes during the recovery?

Recovery is idempotent: if the system crashes during recovery, we can safely repeat it.



Image source: https://insightrsblog.com/2010/09/17/do-you-want-to-recreate-your-entire-database/

In-class Exercise

Given the undo log, describe the action of the recovery manager (Assume the log shown is the log on disk at the time of the crash)

<START T>
<T, *A*, 10>
<START U>
<U, *B*, 20>
<T, *C*, 30>
<U, *D*, 40>
<COMMIT U>

Crash

Checkpointing: Why do we need it?

- Recovery may require scanning a long transaction log.
- We cannot simply discard log records after a transaction commits: other transactions may still be active.
- Checkpointing reduces recovery time by limiting how much of the log needs to be examined after a crash.

Quiescent checkpointing

- Pauses transaction processing, waits for all active transactions to finish, and saves a checkpoint before resuming.

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>

Stop accepting new transactions

Quiescent checkpointing

- Pauses transaction processing, waits for all active transactions to finish, and saves a checkpoint before resuming.

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<T2, *C*, 15>
<T1, *D*, 20>
<COMMIT T1>
<COMMIT T2>

Stop accepting new transactions

Wait until all transactions commit or abort

Quiescent checkpointing

- Pauses transaction processing, waits for all active transactions to finish, and saves a checkpoint before resuming.

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<T2, *C*, 15>
<T1, *D*, 20>
<COMMIT T1>
<COMMIT T2>
<CKPT>

Stop accepting new transactions

Wait until all transactions commit or abort

Flush log + data

Write <CKPT> and flush

Quiescent checkpointing

- Pauses transaction processing, waits for all active transactions to finish, and saves a checkpoint before resuming.

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<T2, *C*, 15>
<T1, *D*, 20>
<COMMIT T1>
<COMMIT T2>
<CKPT>
<START T3>
<T3, E, 25>
<T3, F, 30>

Stop accepting new transactions

Wait until all transactions commit or abort

Flush log + data

Write <CKPT> and flush

Resume transactions

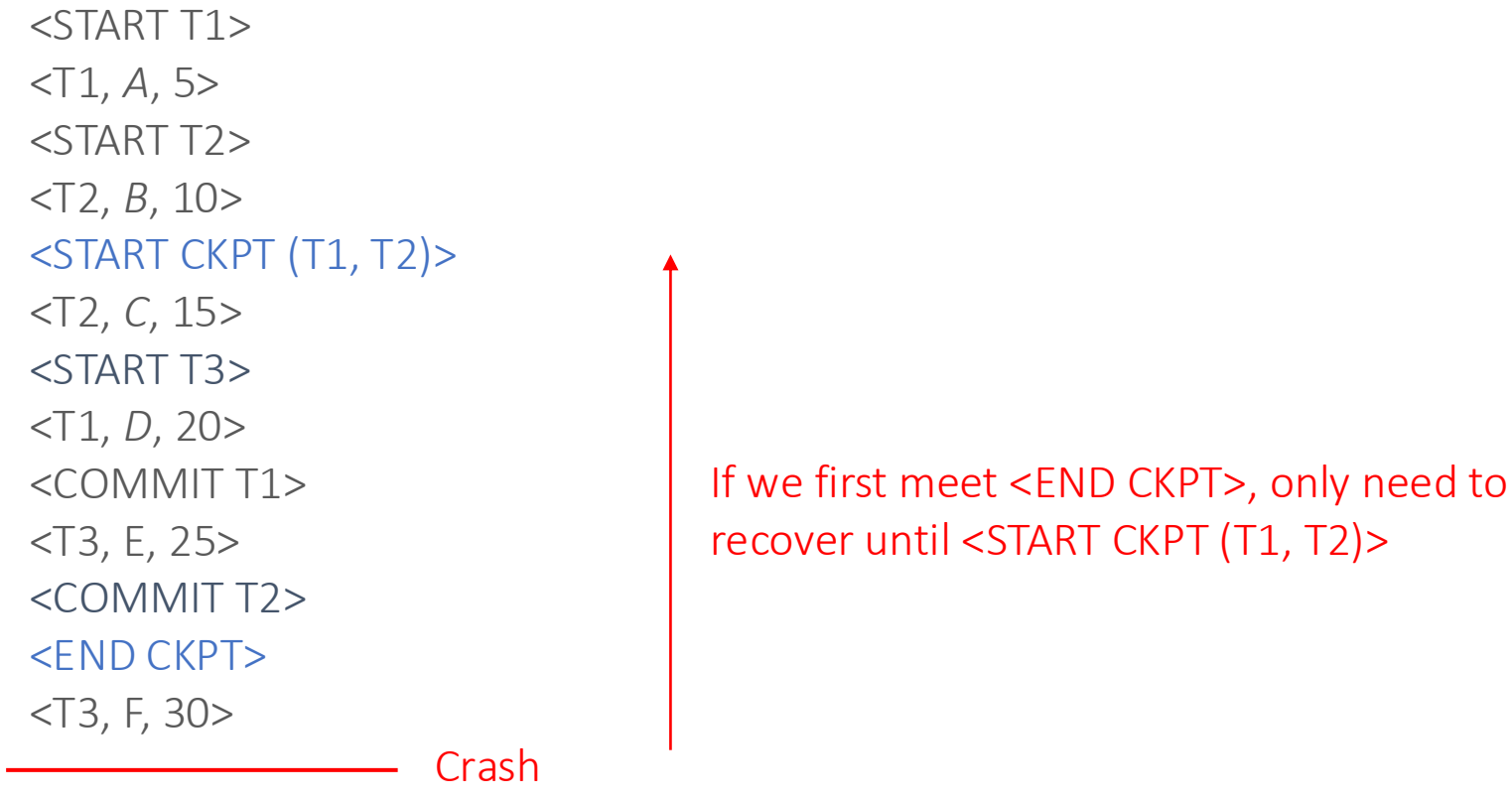
Nonquiescent checkpointing

- Motivation: avoid shutting down system while checkpointing
- Checkpoint all active transactions, but allow new transactions to enter system

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<START CKPT (T1, T2)> Write log record and flush log
<T2, *C*, 15>
<START T3>
<T1, *D*, 20>
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT> Write log record and flush log
<T3, F, 30>

Nonquiescent checkpointing

- Motivation: avoid shutting down system while checkpointing
- Checkpoint all active transactions, but allow new transactions to enter system



Nonquiescent checkpointing

- Motivation: avoid shutting down system while checkpointing
- Checkpoint all active transactions, but allow new transactions to enter system

<START T1>
<T1, *A*, 5>
<START T2>
<T2, *B*, 10>
<START CKPT (T1, T2)>
<T2, *C*, 15>
<START T3>
<T1, *D*, 20>
Crash
<COMMIT T1>
<T3, E, 25>
<COMMIT T2>
<END CKPT>
<T3, F, 30>



If we first meet <START CKPT (T1, T2)>, only need to recover until <START T1>

3. Redo logging

Redo logging

Redo logging ignores incomplete transactions and repeats committed ones

- Undo logging cancels incomplete transactions and ignores committed ones

<*T*, *X*, *v*> now means *T* wrote new value *v* for database element *X*

One rule: all log records (e.g., <*T*, *X*, *v*> and <COMMIT *T*>) must appear on disk before modifying any database element *X* on disk

- => Uncommitted pages can not reach disk

Redo logging

- Example

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 16>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 16>
						<COMMIT *T*>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	

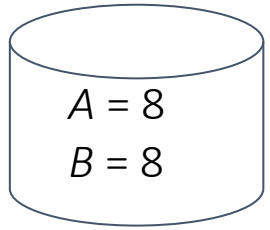
Recovery with redo logging

- Scan log forward and redo committed transactions

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 16>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 16>
						<COMMIT *T*>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	

Crash

Recovery



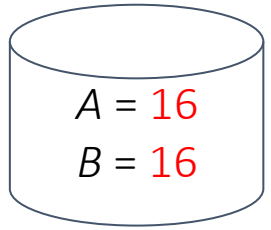
Recovery with redo logging

- Scan log forward and redo committed transactions

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 16>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 16>
						<COMMIT *T*>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	

Crash

Recovery



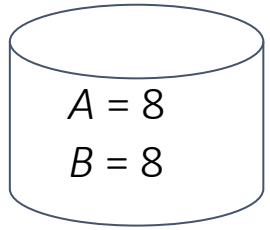
Recovery with redo logging

- Scan log forward and redo committed transactions

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 16>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 16>
						<COMMIT *T*>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	

Crash

Recovery



Recovery with redo logging

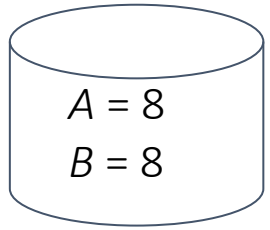
- Scan log forward and redo committed transactions

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 16>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 16>
						<COMMIT *T*>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
OUTPUT(*B*)	16	16	16	16	16	

Crash

Recovery

Do nothing



Nonquiescent checkpointing for redo log

- Write to disk all DB elements modified by committed transactions

<START T1>
<T1, *A*, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 10>
<START CKPT (T2)>

Nonquiescent checkpointing for redo log

- Before writing <END CKPT>, flush all updates from transactions that committed before <START CKPT>.

<START T1>
<T1, *A*, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 10>
<START CKPT (T2)>
<T2, *C*, 15>
<START T3>
<T3, *D*, 20>
<END CKPT>

Flush updates from transactions committed before checkpoint start. (i.e., T1)

Nonquiescent checkpointing for redo log

- Before writing <END CKPT>, flush all updates from transactions that committed before <START CKPT>.

<START T1>
<T1, *A*, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 10>
<START CKPT (T2)>
<T2, *C*, 15>
<START T3>
<T3, *D*, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

Flush updates from transactions committed before checkpoint start. (i.e., T1)

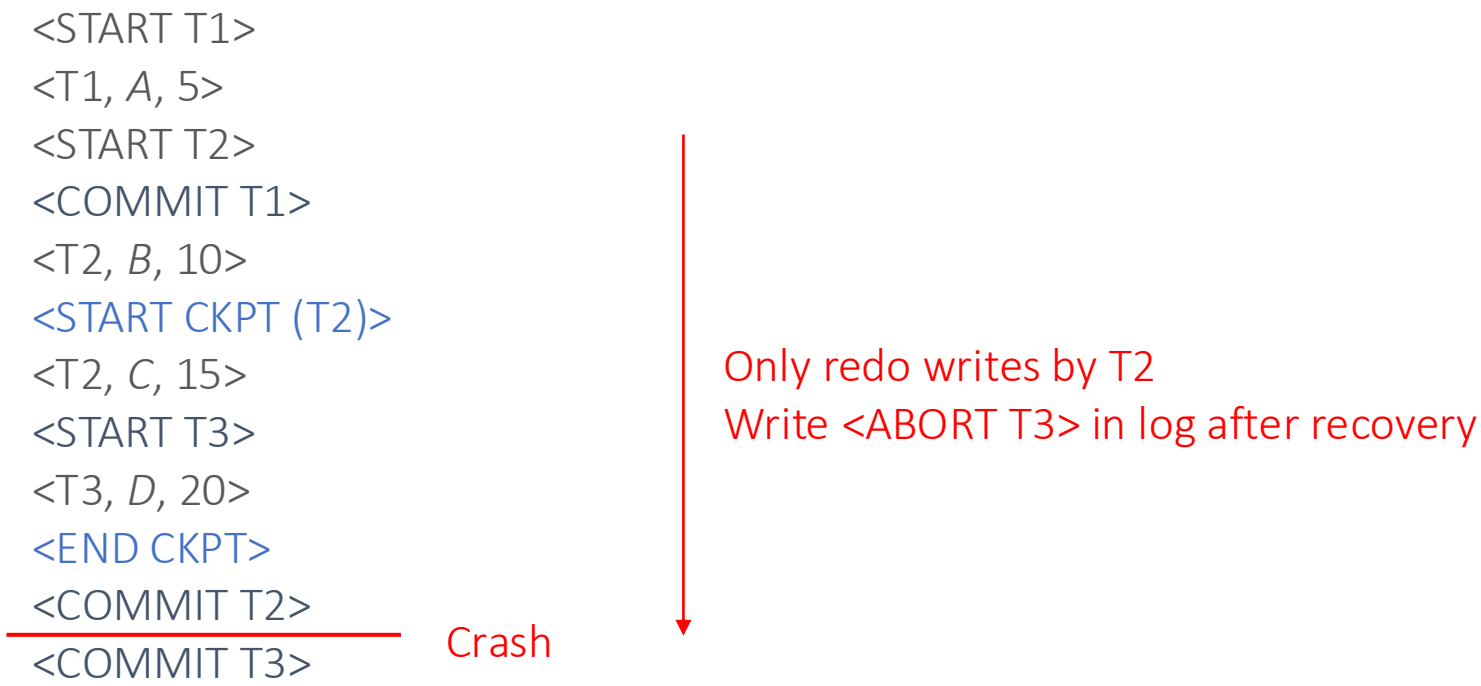
Nonquiescent checkpointing for redo log

- After crash, redo committed transactions that either started after START CKPT or were active during START CKPT

<START T1>
<T1, *A*, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 10>
<START CKPT (T2)>
<T2, *C*, 15>
<START T3>
<T3, *D*, 20>
<END CKPT>
<COMMIT T2>
Crash
<COMMIT T3>

Nonquiescent checkpointing for redo log

- After crash, redo committed transactions that either started after START CKPT or were active during START CKPT



4. Undo/redo logging

Undo/redo logging

Modified data pages can be flushed before or after a transaction commits.

<*T*, *X*, *v*, *w*> : *T* changed value of *X* from *v* to *w*

One rule:

- <*T*, *X*, *v*, *w*> must appear on disk before modifying *X* on disk
- Modified data pages may be flushed before or after COMMIT.

Undo/redo logging

- Example

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8, 16>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8, 16>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
						<COMMIT *T*>
OUTPUT(*B*)	16	16	16	16	16	

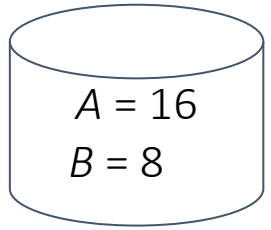
Recovery with undo/redo logging

- Redo all committed transactions and undo all incomplete transactions

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8, 16>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8, 16>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
						<COMMIT *T*>
OUTPUT(*B*)	16	16	16	16	16	

Crash

Recovery



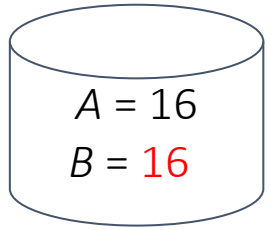
Recovery with undo/redo logging

- Redo all committed transactions and undo all incomplete transactions

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8, 16>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8, 16>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
						<COMMIT *T*>
OUTPUT(*B*)	16	16	16	16	16	

Crash

Recovery



T is commited
Redo by writing the value 16
for both *A* and *B* to the disk.

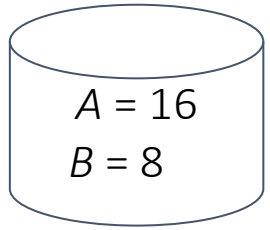
Recovery with undo/redo logging

- Redo all committed transactions and undo all incomplete transactions

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8, 16>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8, 16>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
						<COMMIT *T*>
OUTPUT(*B*)	16	16	16	16	16	

Crash

Recovery



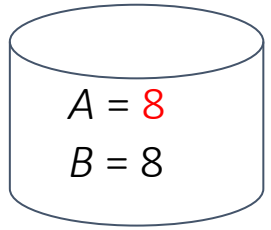
Recovery with undo/redo logging

- Redo all committed transactions and undo all incomplete transactions

		Memory		Disk		
Action	*t*	*A*	*B*	*A*	*B*	Log
						<START *T*>
READ(*A*, t)	8	8		8	8	
t := *t* * 2	16	8		8	8	
WRITE(*A*, *t*)	16	16		8	8	<*T*, *A*, 8, 16>
READ(*B*, *t*)	8	16	8	8	8	
t := *t* * 2	16	16	8	8	8	
WRITE(*B*, *t*)	16	16	16	8	8	<*T*, *B*, 8, 16>
FLUSH LOG						
OUTPUT(*A*)	16	16	16	16	8	
						<COMMIT *T*>
OUTPUT(*B*)	16	16	16	16	16	

Crash

Recovery



T is incomplete
Undo by resetting *A* and *B* to the previous value of 8

Nonquiescent checkpointing for undo/redo logging

- Flush all dirty pages (including uncommitted updates) before <END CKPT>.

<START T1>
<T1, *A*, 4, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 9, 10>
<START CKPT (T2)>

Nonquiescent checkpointing for undo/redo logging

- Flush all dirty pages (including uncommitted updates) before <END CKPT>.

<START T1>
<T1, *A*, 4, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 9, 10>
<START CKPT (T2)>
<T2, *C*, 14, 15>
<START T3>
<T3, *D*, 19, 20>
<END CKPT>

Flush all dirty data pages before writing <END CKPT>.

Nonquiescent checkpointing for undo/redo logging

- Simpler than other logging methods

<START T1>
<T1, *A*, 4, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 9, 10>
<START CKPT (T2)>
<T2, *C*, 14, 15>
<START T3>
<T3, *D*, 19, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

Flush all dirty data pages before writing <END CKPT>.

WAL still applies: flush the corresponding log records before flushing data pages.

Nonquiescent checkpointing for undo/redo logging

- After a crash, redo committed transactions, and undo uncommitted ones

<START T1>
<T1, *A*, 4, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 9, 10>
<START CKPT (T2)>
<T2, *C*, 14, 15>
<START T3>
<T3, *D*, 19, 20>
<END CKPT>
<COMMIT T2>
Crash
<COMMIT T3>

Nonquiescent checkpointing for undo/redo logging

- After a crash, redo committed transactions, and undo uncommitted ones

<START T1>
<T1, *A*, 4, 5>
<START T2>
<COMMIT T1>
<T2, *B*, 9, 10>
<START CKPT (T2)>
<T2, *C*, 14, 15>
<START T3>
<T3, *D*, 19, 20>
<END CKPT>
<COMMIT T2>
Crash
<COMMIT T3>

Redo T2 by setting C to 15 on disk
(No need to set B to 10 thanks to CKPT)
Undo T3 by setting D to 19 on disk

Summary

Write-ahead logging protocol

- Log before data
- Log at commit

Coping with System Failures

- Undo logging
- Redo logging
- Undo/redo logging
- Checkpointing