Failure Recovery

Hector Garcia-Molina and
Mahmoud SAKR

Integrity or consistency constraints

- Predicates data must satisfy
- Examples:
  - x is key of relation R
  - $x \rightarrow y$ holds in R
  - Domain(x) = {Red, Blue, Green}
  - α is valid index for attribute x of R
  - no employee should make more than twice the average salary

Definition:

- Consistent state: satisfies all constraints
- Consistent DB: DB in consistent state

Observation: DB cannot be consistent always!

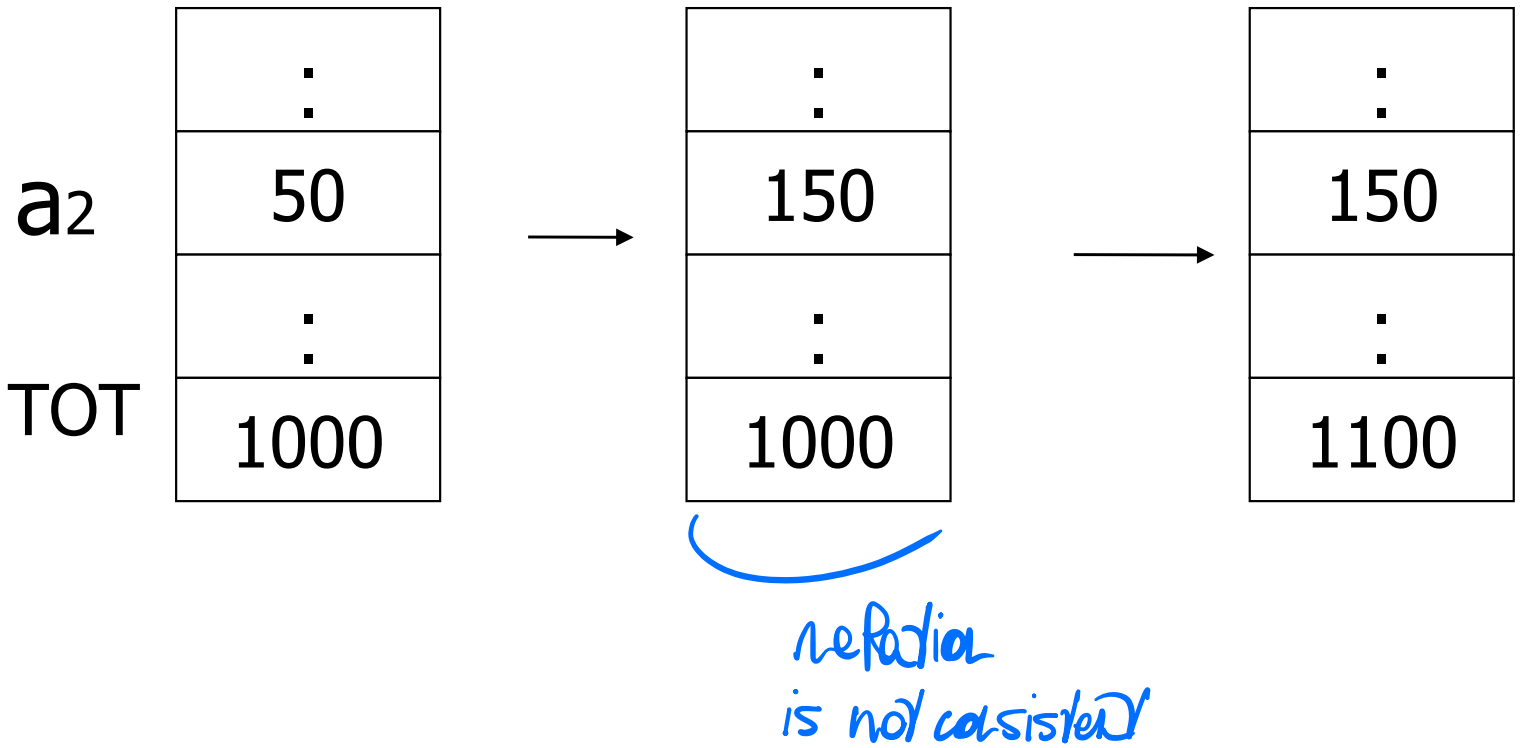
Example: $a_1 + a_2 + \ldots + a_n = TOT$

Deposit \$100 in a_2:
$$\begin{cases} a_2 \leftarrow a_2 + 100 \\ TOT \leftarrow TOT + 100 \end{cases}$$

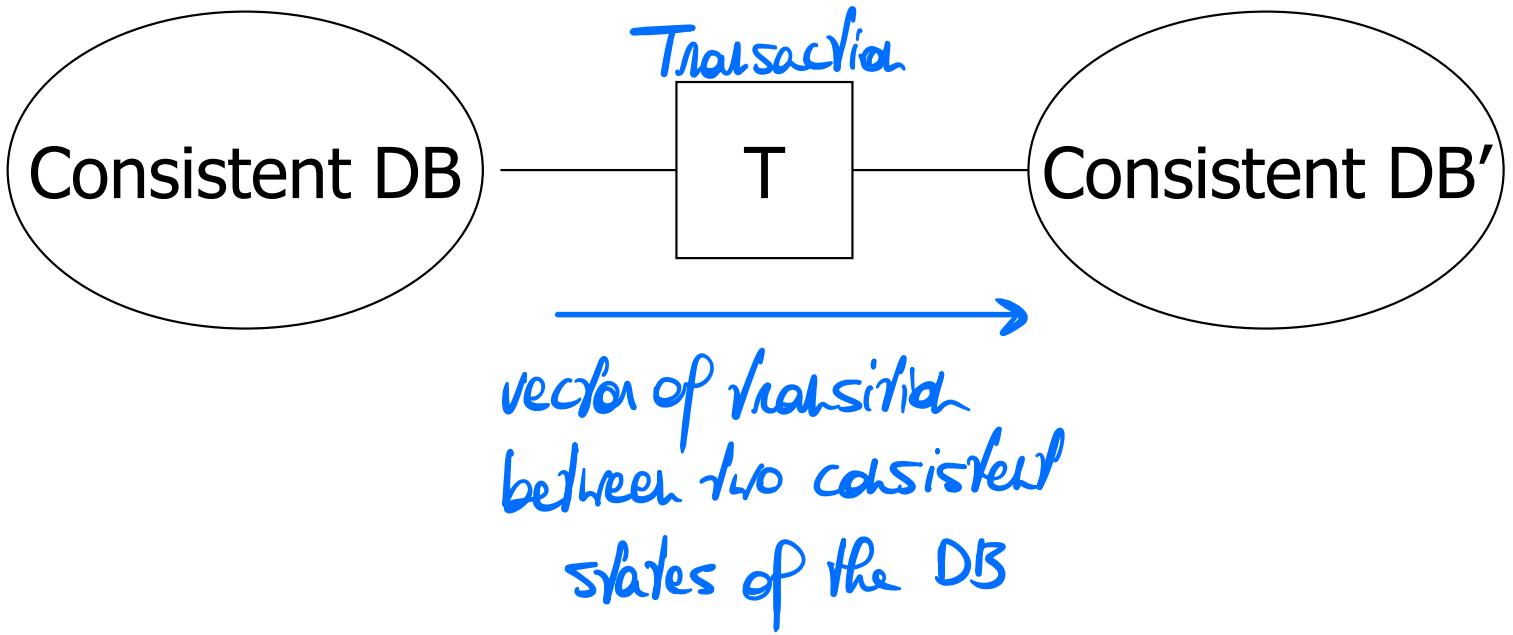
Example: $a_1 + a_2 + \ldots a_n = TOT$

Deposit \$100 in a_2: $a_2 \leftarrow a_2 + 100$

$TOT \leftarrow TOT + 100$



Transaction: collection of actions that preserve consistency



Big assumption:

If T starts with consistent state +

T executes in isolation ⟶ transaction, when only actions taken by the DB, leaves the DB in a consistent state if the DB was consistent before the transaction

⇒ T leaves consistent state

Correctness (informally)

- If we stop running transactions,
  DB left consistent
- Each transaction sees a consistent DB

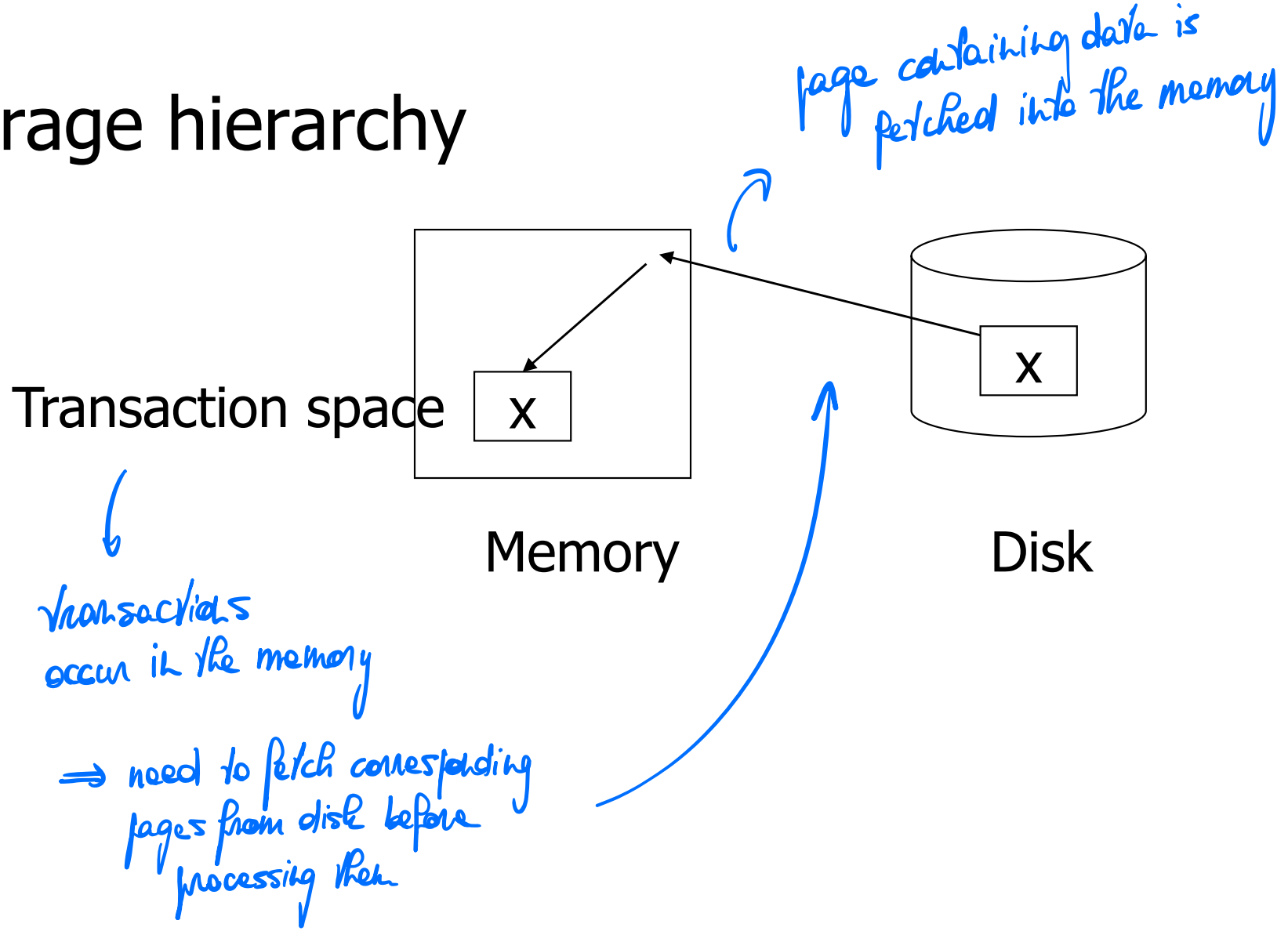
How can consistency be violated?

- System crash
  - memory lost
  - cpu halts, resets

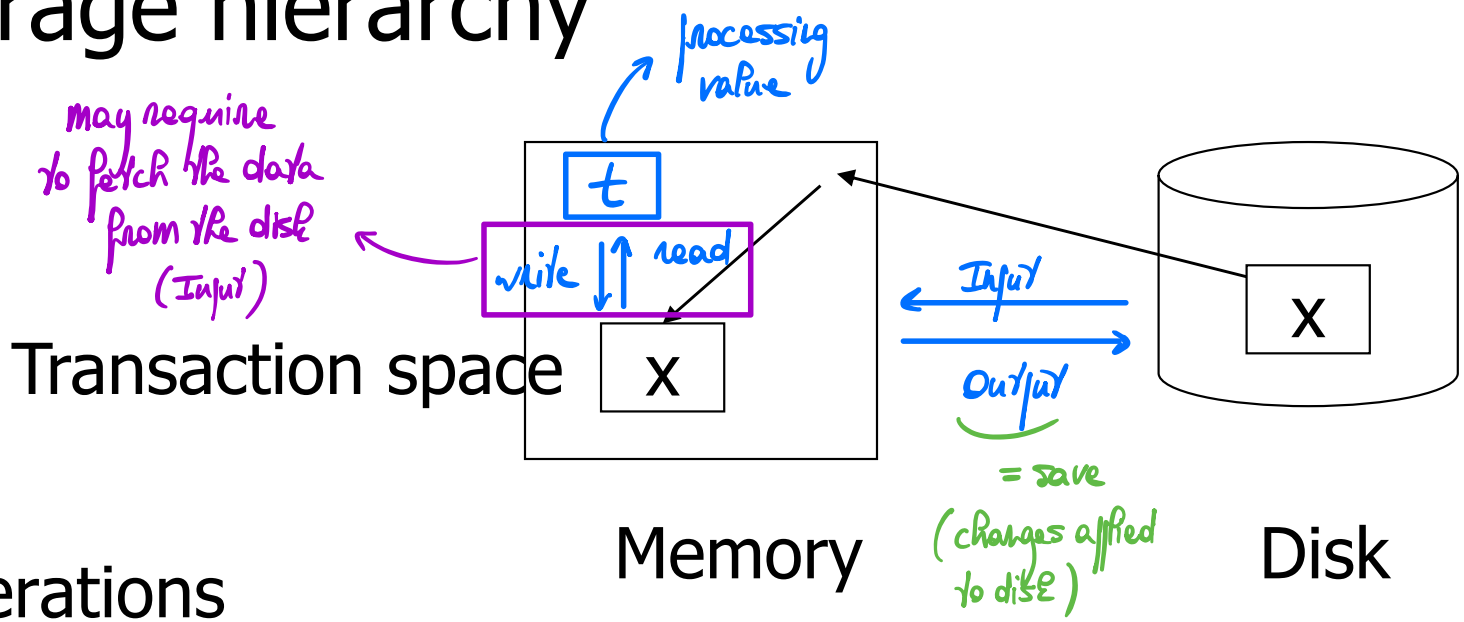
what we are dealing with in this chapter

- Media failure, catastrophes

Storage hierarchy



Storage hierarchy



Operations

- Input (x): block containing x → memory
- Output (x): block containing x → disk
- Read (x,t): do input(x) if necessary, t ← value of x in block
- Write (x,t): do input(x) if necessary, value of x in block ← t

if something has been written the data it contained is considered "save action"

may need to input (x) if not yet in memory

updating value of x

Key problem Unfinished transaction

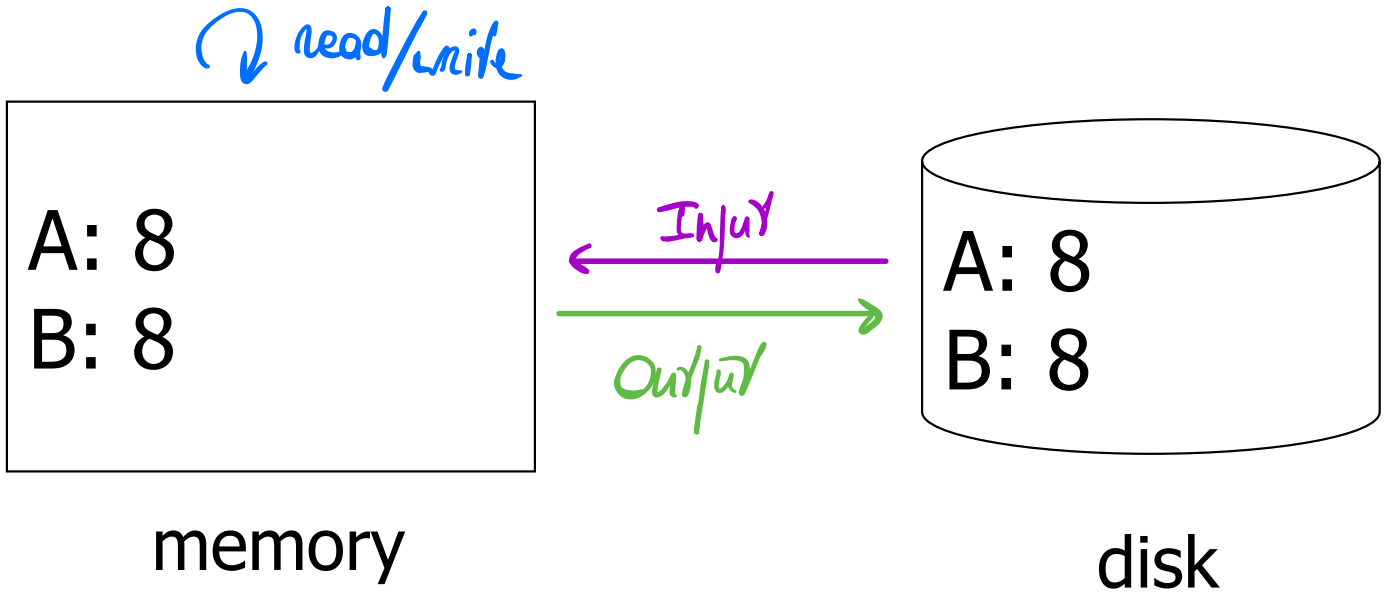
Example Constraint: A=B

T_1: $A \leftarrow A \times 2$

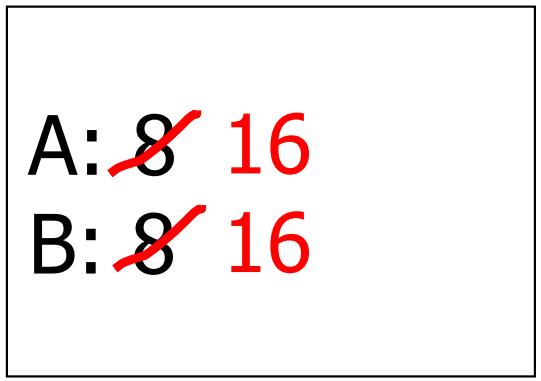
$B \leftarrow B \times 2$

T1: Read (A,t); t ← t×2
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);

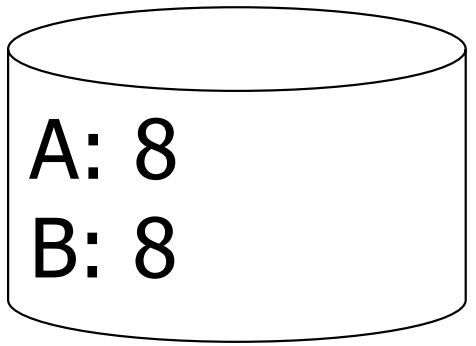
write occurs in transaction space
↳ then output to the disk



```
T1:  Read (A,t);  t ← t×2
     Write (A,t);
     Read (B,t);  t ← t×2
     Write (B,t);
     Output (A);
     Output (B);
```



memory

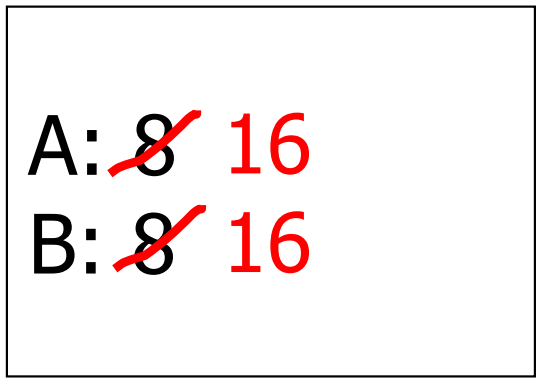


disk

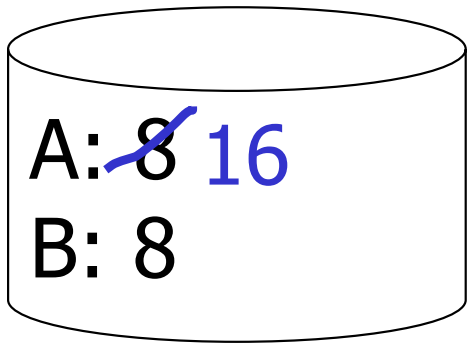
```
T1:  Read (A,t);  t ← t×2
     Write (A,t);
     Read (B,t);  t ← t×2
     Output (A);
     Output (B);
```

failure!

not consistent, data is corrupted



memory



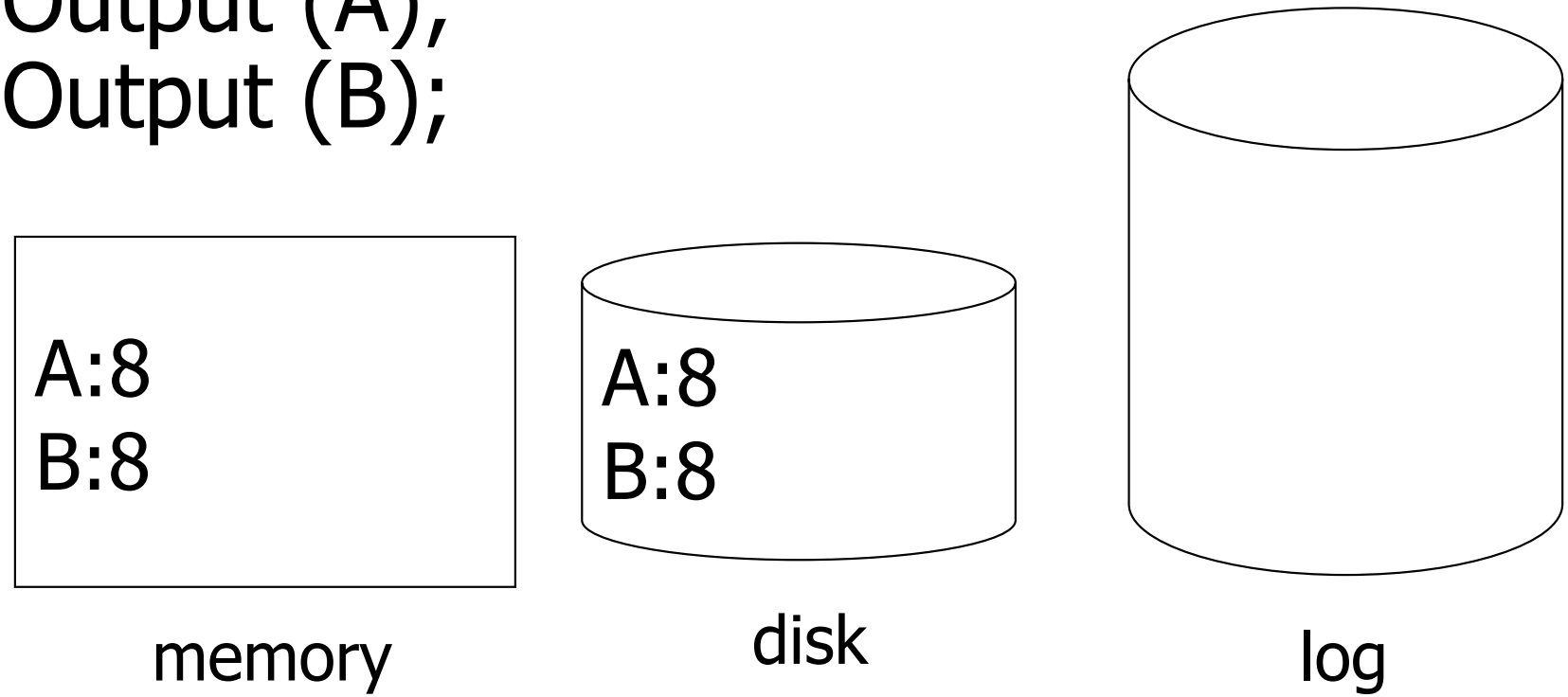
disk

- Need <u>atomicity:</u> execute all actions of a transaction or none at all

we have to log important events to keep track of the changes in case of system failure

Undo logging (Immediate modification)

T_1: Read (A,t); $t \leftarrow t \times 2$ A=B
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);

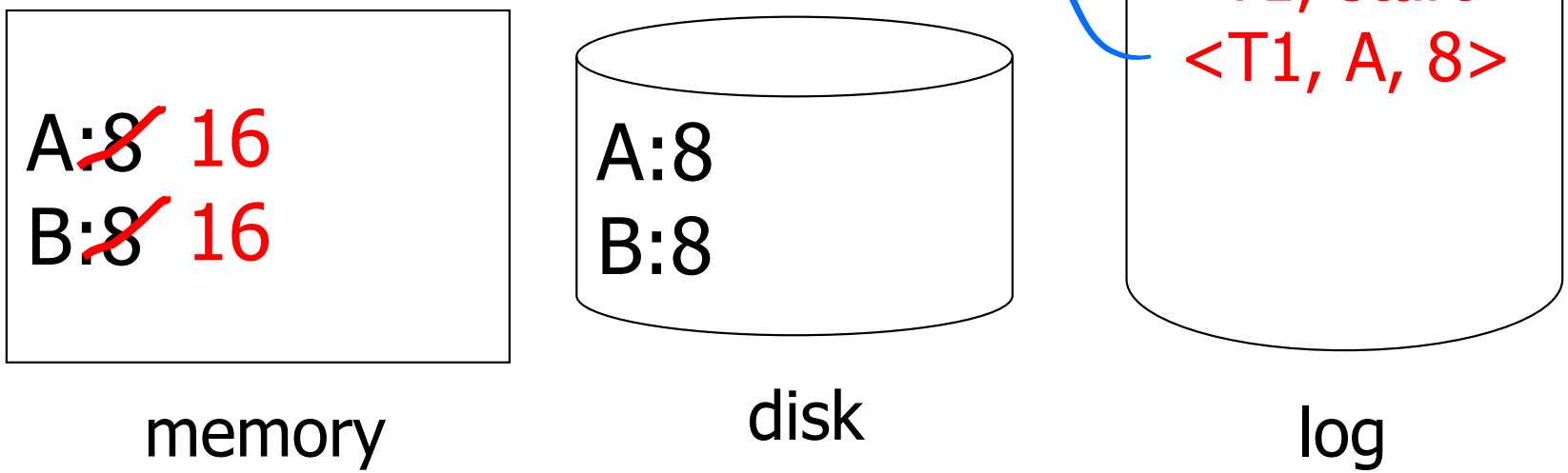


Undo logging (Immediate modification)

```
T1:  Read (A,t);  t ← t×2        A=B
     Write (A,t);
     Read (B,t);  t ← t×2
     Write (B,t);
     Output (A);
     Output (B);
```

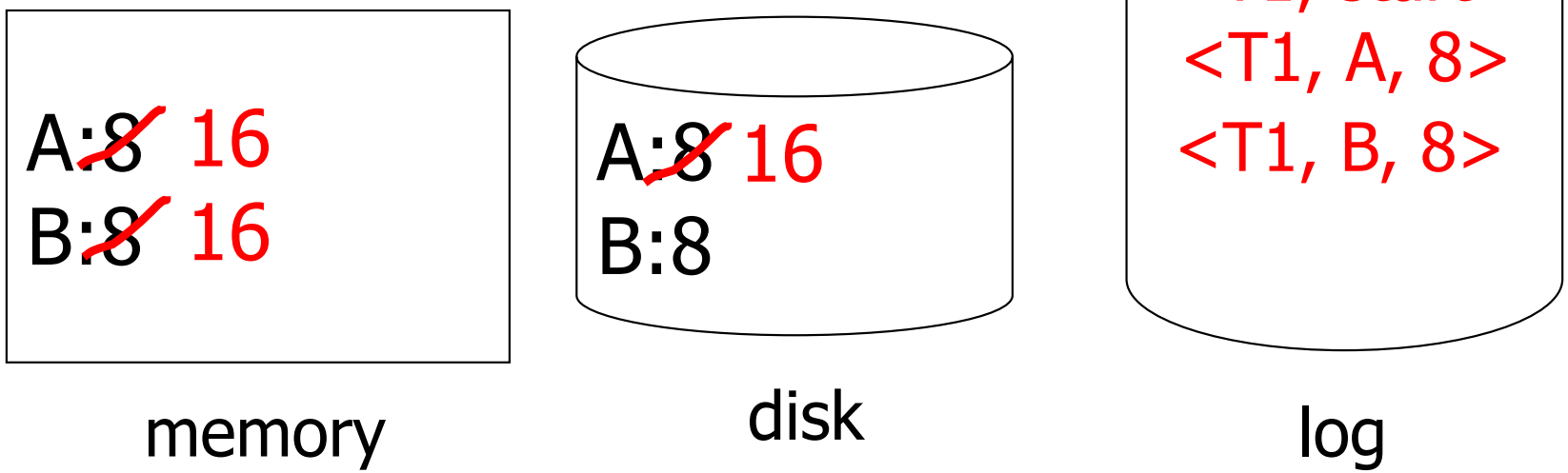
allows for recovering

we log old value before changing it in disk



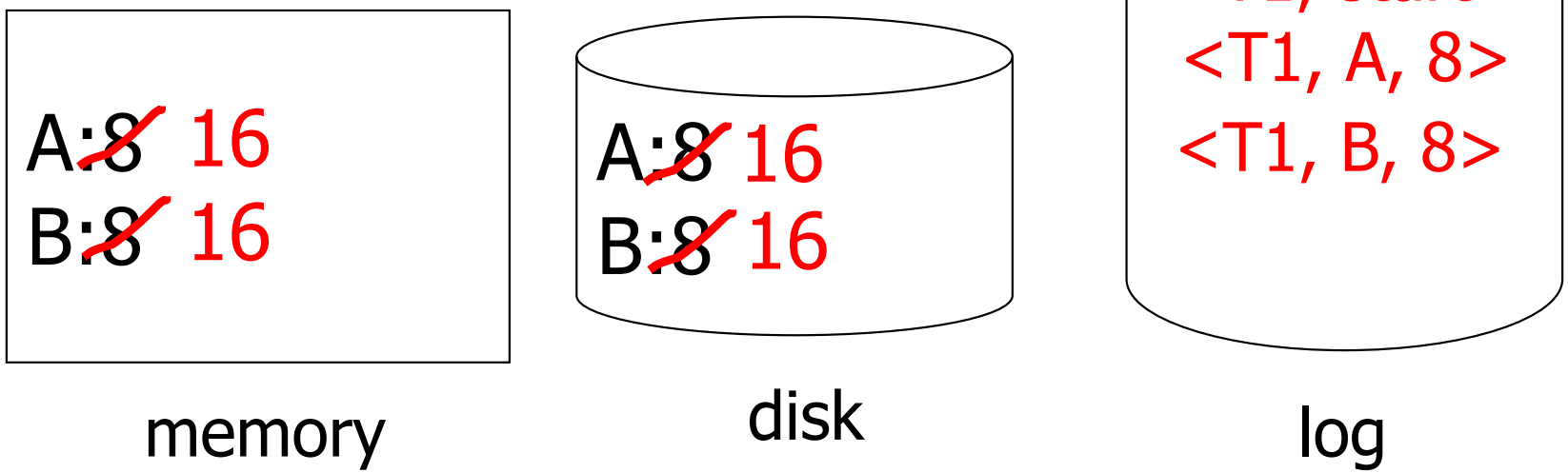
Undo logging (Immediate modification)

T1: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);



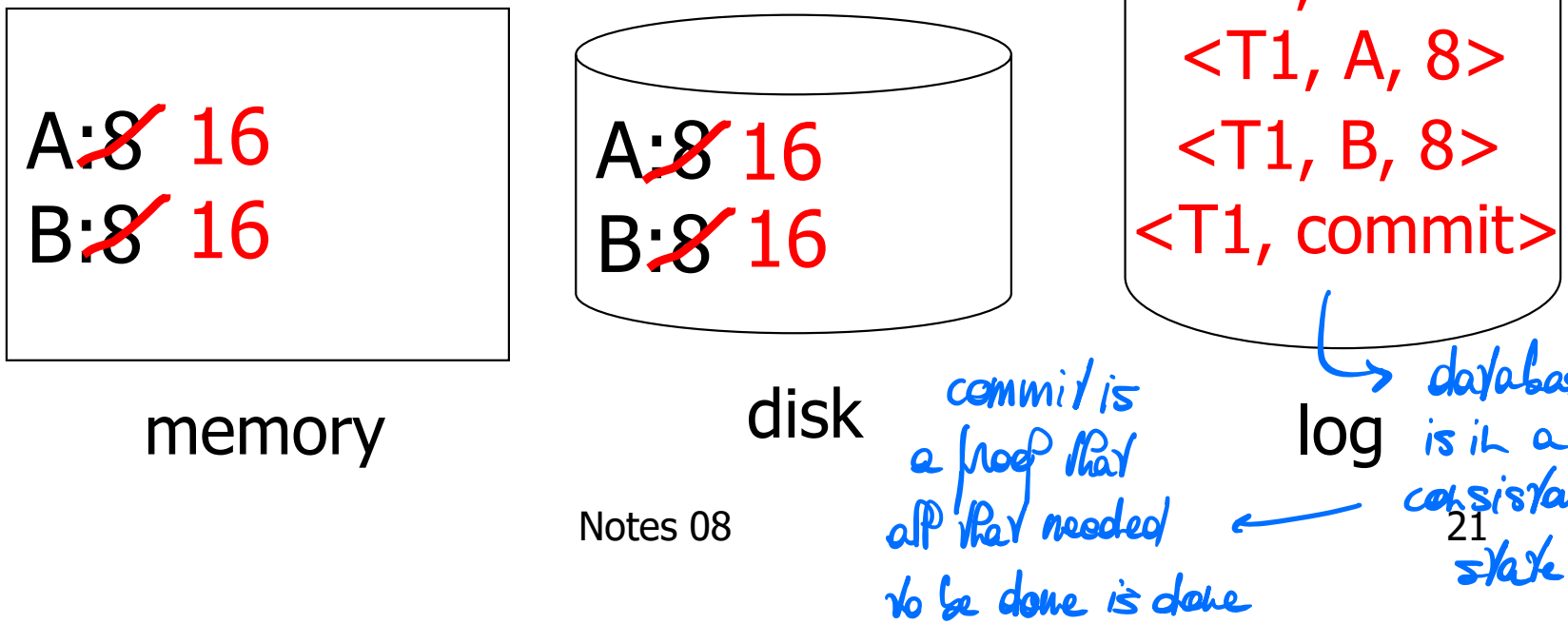
Undo logging (Immediate modification)

T1: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);



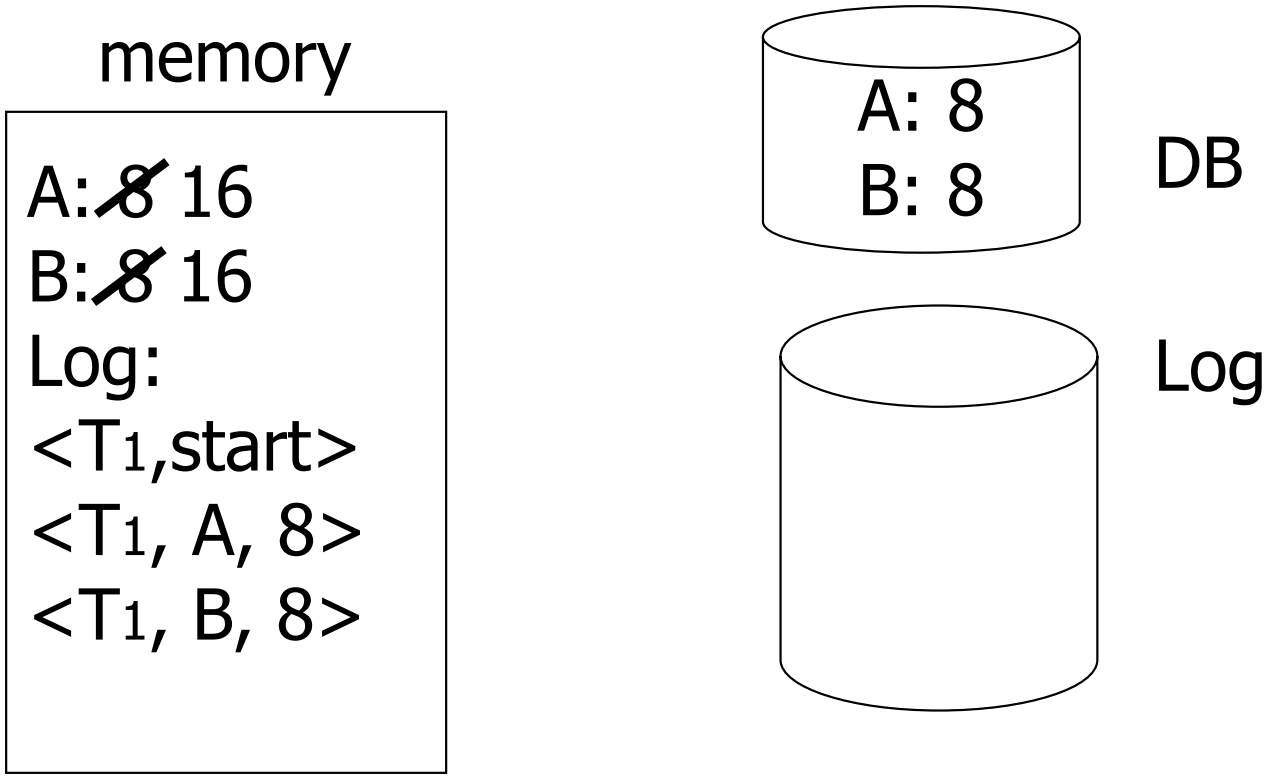
Undo logging (Immediate modification)

T1: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);



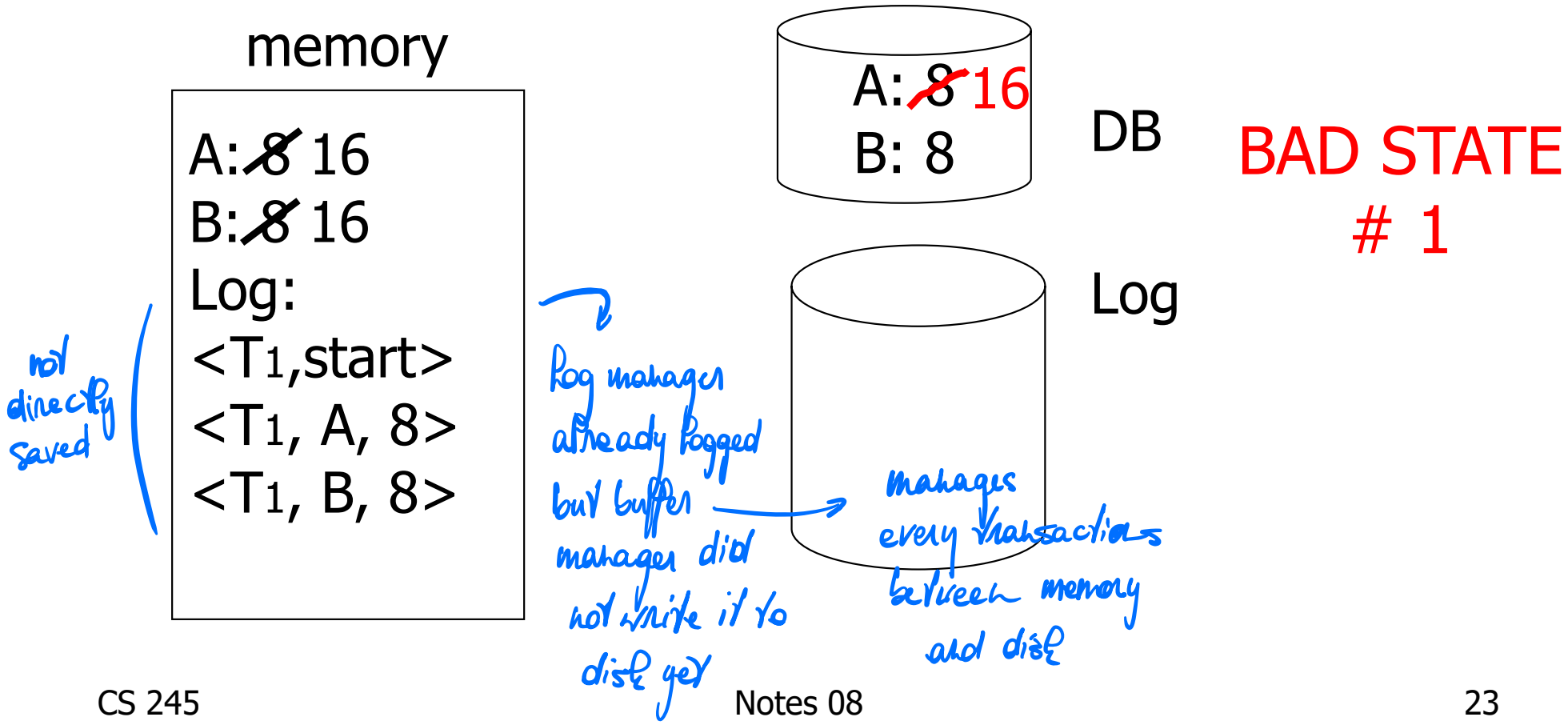
One "complication"

- Log is first written in memory
- Not written to disk on every action



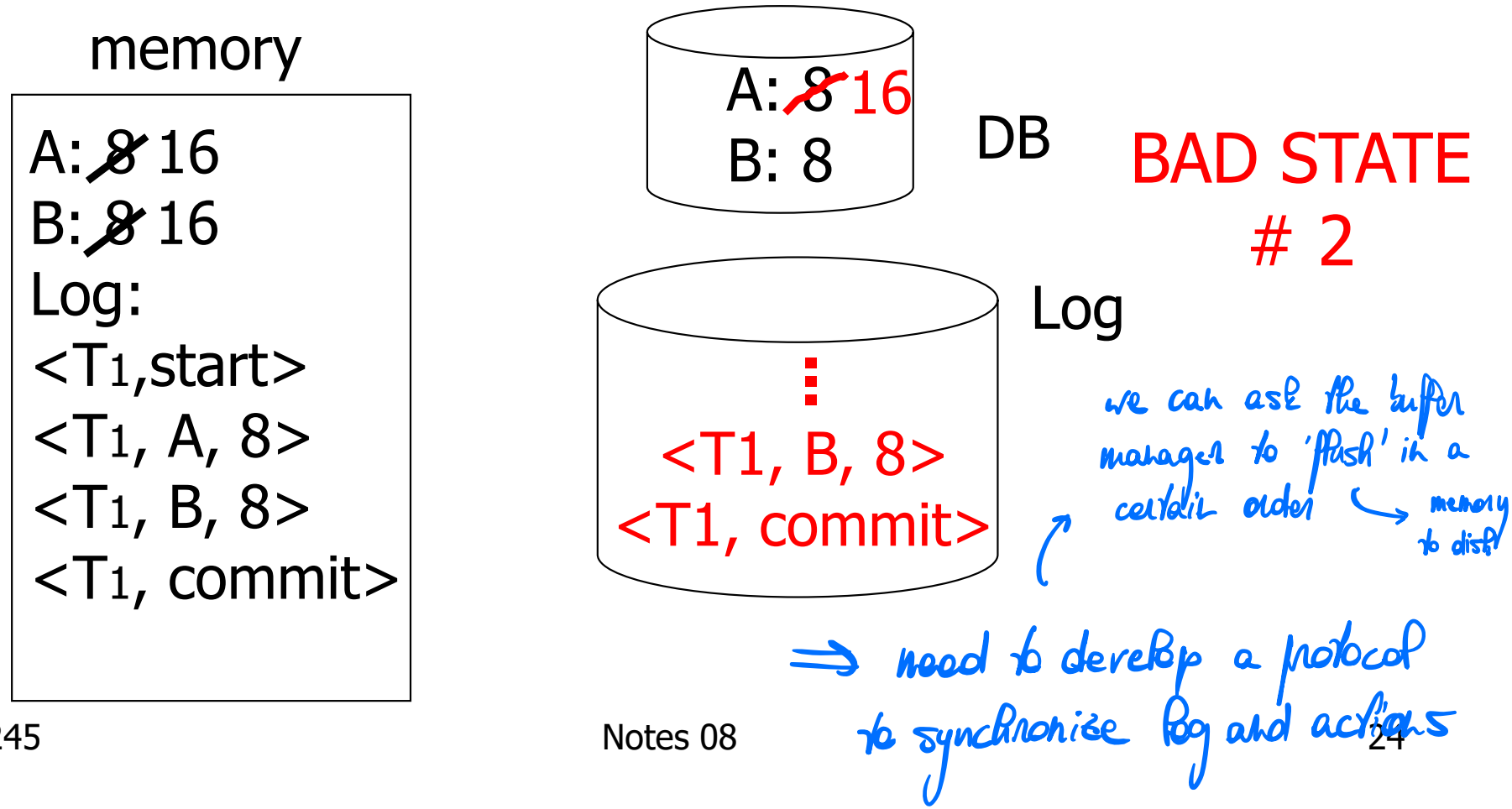
One "complication"

- Log is first written in memory
- Not written to disk on every action



One "complication"

- Log is first written in memory
- Not written to disk on every action



Undo logging rules

(1) For every action generate undo log record (containing old value)

(2) Before x is modified on disk, log records pertaining to x must be on disk (write ahead logging: WAL)

(3) Before commit is flushed to log, all writes of transaction must be reflected on disk

Undo logging example

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<T, A, 8>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<T, B, 8>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	
10)	OUTPUT(B)	16	16	16	16	16	
11)							<COMMIT T>
12)	FLUSH LOG						

memory

disk

→ flush A and B modif log before changing them

) change value in disk by outputting

↳ when done, we flush the termination of actia log

Figure 17.3: Actions and their log entries

Recovery rules: Undo logging

- For every Ti with <Ti, start> in log:
  - If <Ti,commit> or <Ti,abort> in log, do nothing
  - Else
    - For all <Ti, *X*, *v*> in log:
      - write (*X*, *v*)
      - output (*X*)
    - Write <Ti, abort> to log

it has already been dealt with
↳ either it is finished or we did nothing

we undo the changes

we save to disk

↳ log the fact that T_i was cancelled and we didn't change anything

Recovery rules: Undo logging

- For every Ti with <Ti, start> in log:
  - If <Ti,commit> or <Ti,abort> in log, do nothing
  - Else
    - For all <Ti, *X*, *v*> in log:
      - write *(X, v)*
      - output (*X*)
    - Write <Ti, abort> to log

☒IS THIS CORRECT??

Recovery rules: Undo logging

(1) Let S = set of transactions with
<Ti, start> in log, but no
<Ti, commit> (or <Ti, abort>) record in log

(2) For each <Ti, X, v> in log,
in reverse order (latest $\rightarrow$ earliest) do:
- if Ti $\in$ S then
  - write (X, v)
  - output (X)

(3) For each Ti $\in$ S do
- write <Ti, abort> to log

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<$T, A, 8$>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<$T, B, 8$>
8)	FLUSH LOG						
9)	OUTPUT(A)	16	16	16	16	8	failure
10)	OUTPUT(B)	16	16	16	16	16	
11)							<COMMIT T>
12)	FLUSH LOG						

9/10) Start and previous values are logged ⇒ undo { A = 16 → 8, B = 8 → 8 } but no commit

Then flush the abort transaction signal

↓

if failure here, abort was not flushed ⇒ undo exactly as before

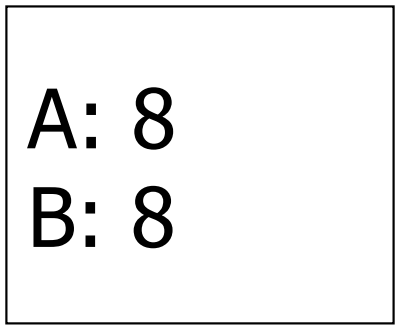
What if failure during recovery?

No problem! ➡ Undo idempotent

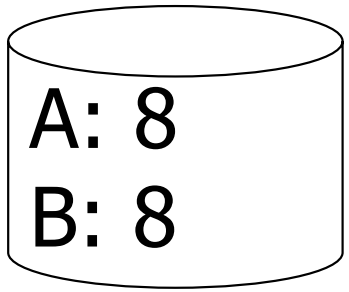
opposite of before
↳ we want to put Bg in disk before the modificat

Redo logging (deferred modification)

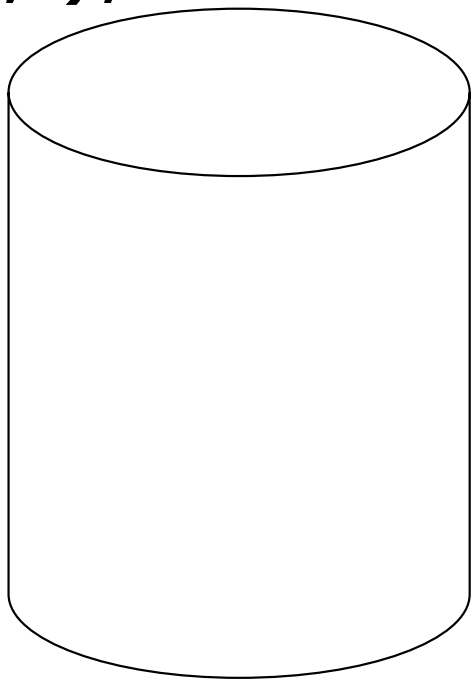
T_1: Read(A,t); t← t×2; write (A,t);
Read(B,t); t←t×2; write (B,t);
Output(A); Output(B)



memory



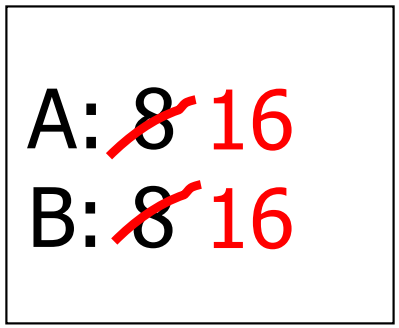
DB



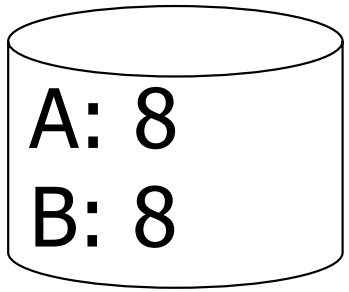
LOG

Redo logging (deferred modification)

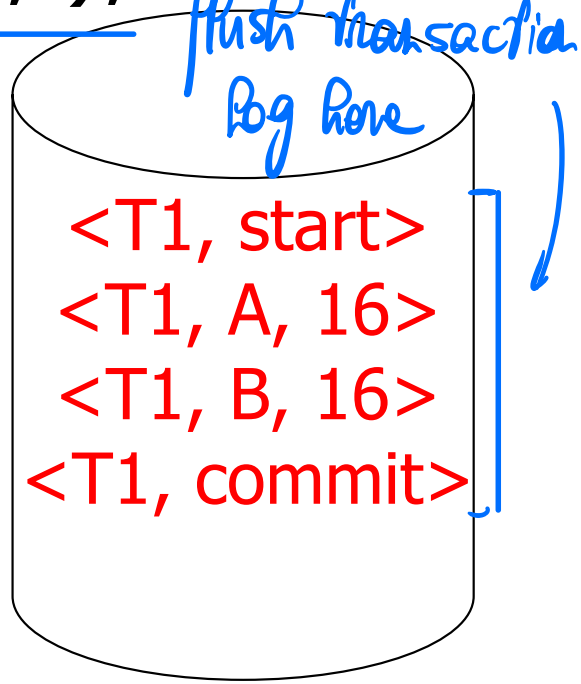
T1: Read(A,t); t← t×2; write (A,t);
Read(B,t); t←t×2; write (B,t);
Output(A); Output(B)



memory



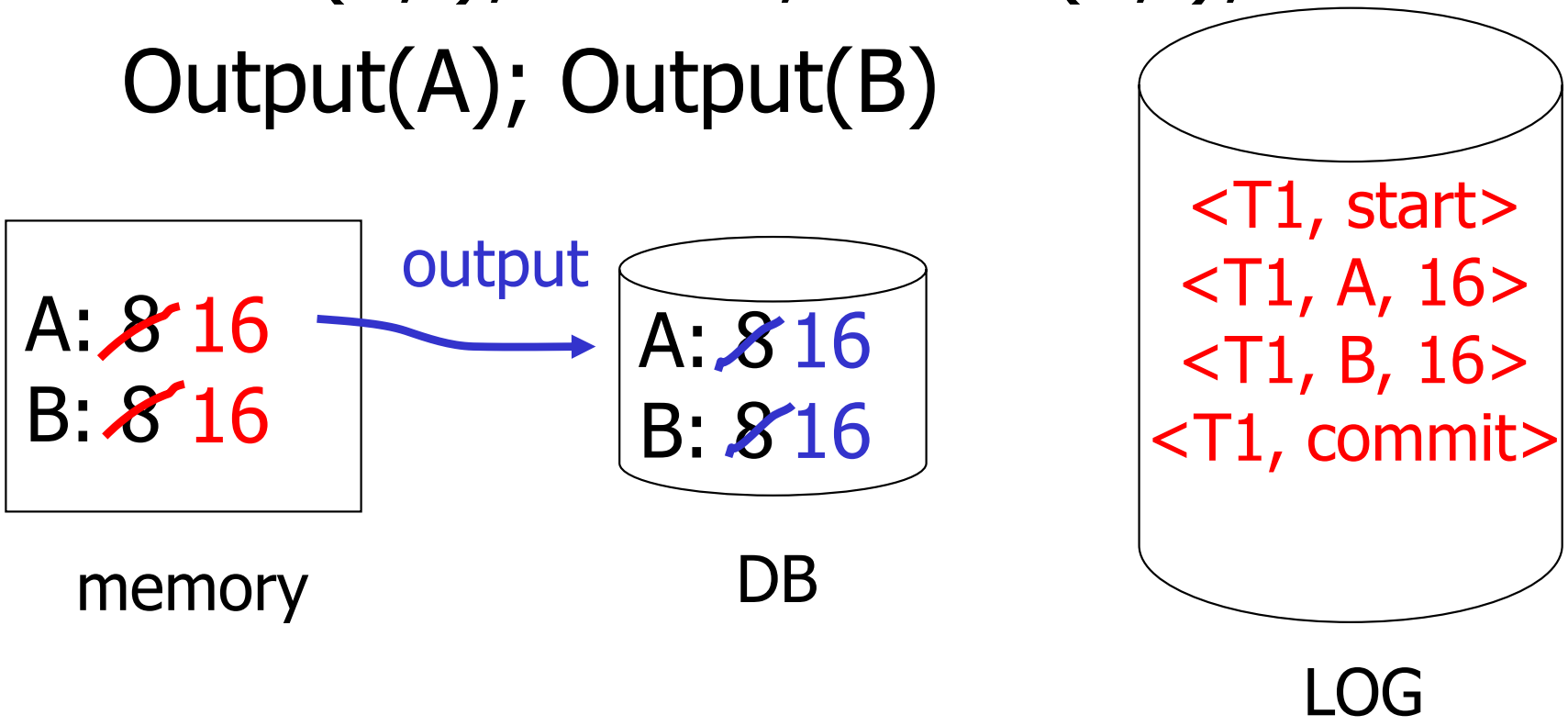
DB



LOG

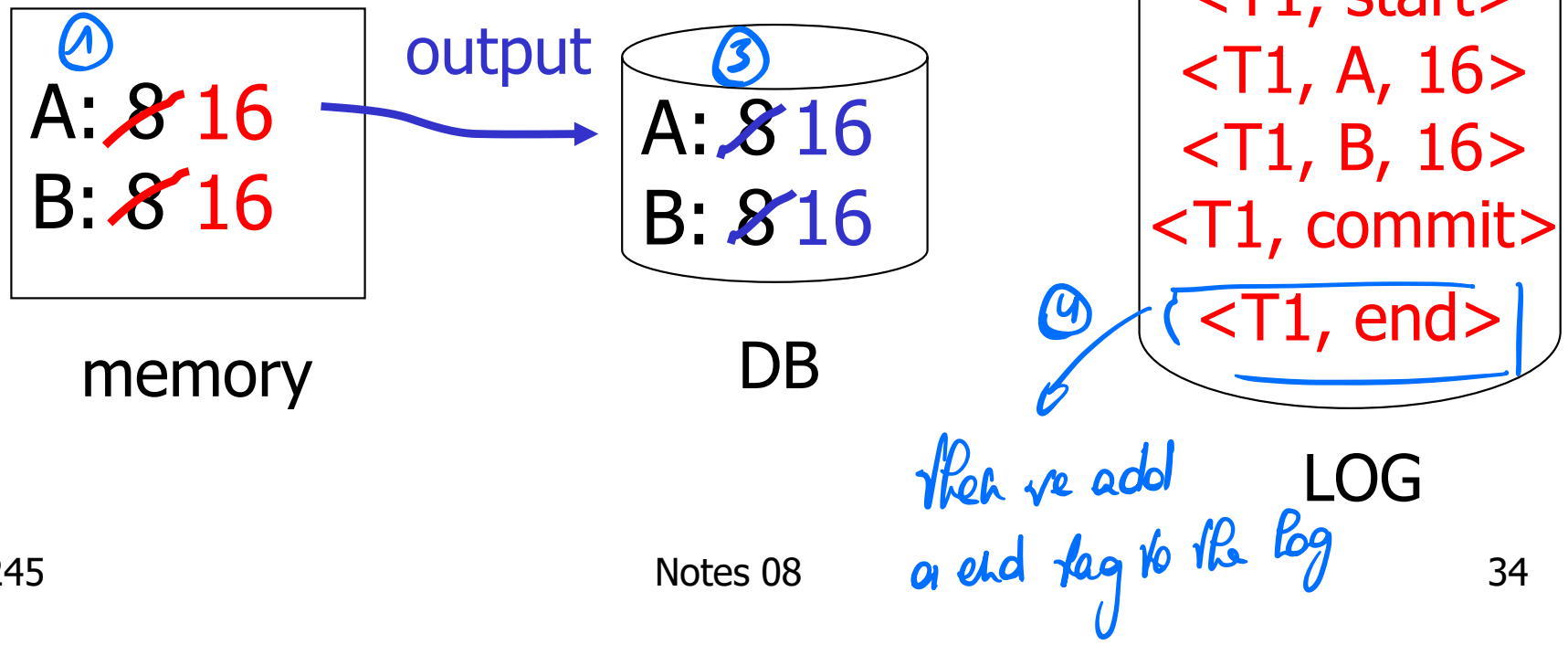
Redo logging (deferred modification)

T_1: Read(A,t); t← t×2; write (A,t);
Read(B,t); t←t×2; write (B,t);
Output(A); Output(B)



Redo logging (deferred modification)

T1: Read(A,t); t← t×2; write (A,t);
Read(B,t); t←t×2; write (B,t);
Output(A); Output(B)



Redo logging rules

(1) For every action, generate redo log record (containing new value)

we want the transaction to happen

(2) Before X is modified on disk (DB), all log records for transaction that modified X (including commit) must be on disk

(3) Flush log at commit

(4) Write END record after DB updates flushed to disk

Redo logging example

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<$T, A, 16$>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<$T, B, 16$>
8)							<COMMIT T>
9)	FLUSH LOG						
10)	OUTPUT(A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

Figure 17.7: Actions and their log entries using redo logging

Recovery rules: Redo logging

- For every Ti with <Ti, commit> in log:
  - For all <Ti, X, v> in log:
    - Write(X, v)
    - Output(X)

Recovery rules: Redo logging

- For every Ti with <Ti, commit> in log:
  - For all <Ti, X, v> in log:
    - Write(X, v)
    - Output(X)

☒IS THIS CORRECT??

Recovery rules: Redo logging

(1) Let S = set of transactions with <Ti, commit> (and no <Ti, end>) in log

(2) For each <Ti, X, v> in log, in forward order (earliest → latest) do:

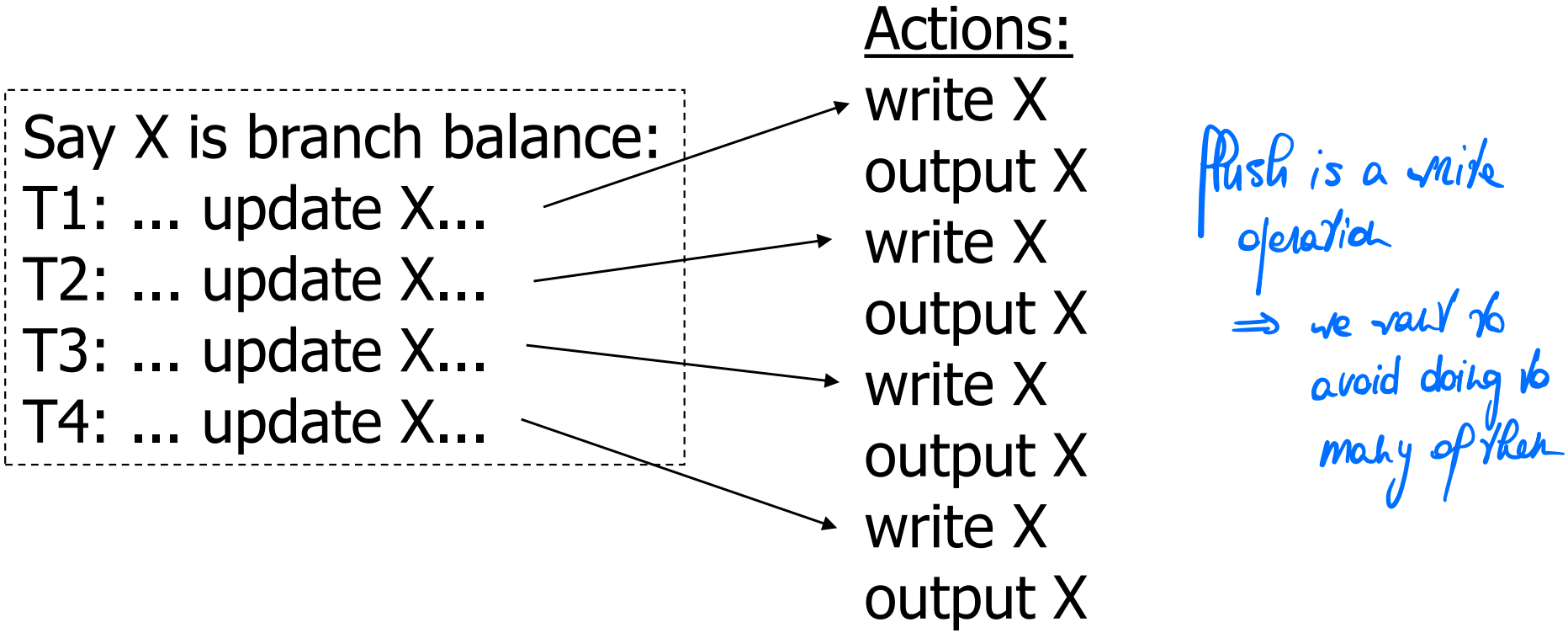
- if Ti ∈ S then
  - Write(X, v)
  - Output(X)

(3) For each Ti ∈ S, write <Ti, end>

↳ we are not returning to previous state, we end the transactions that did not finish ⇒ transactions happen in the order in which they were requested

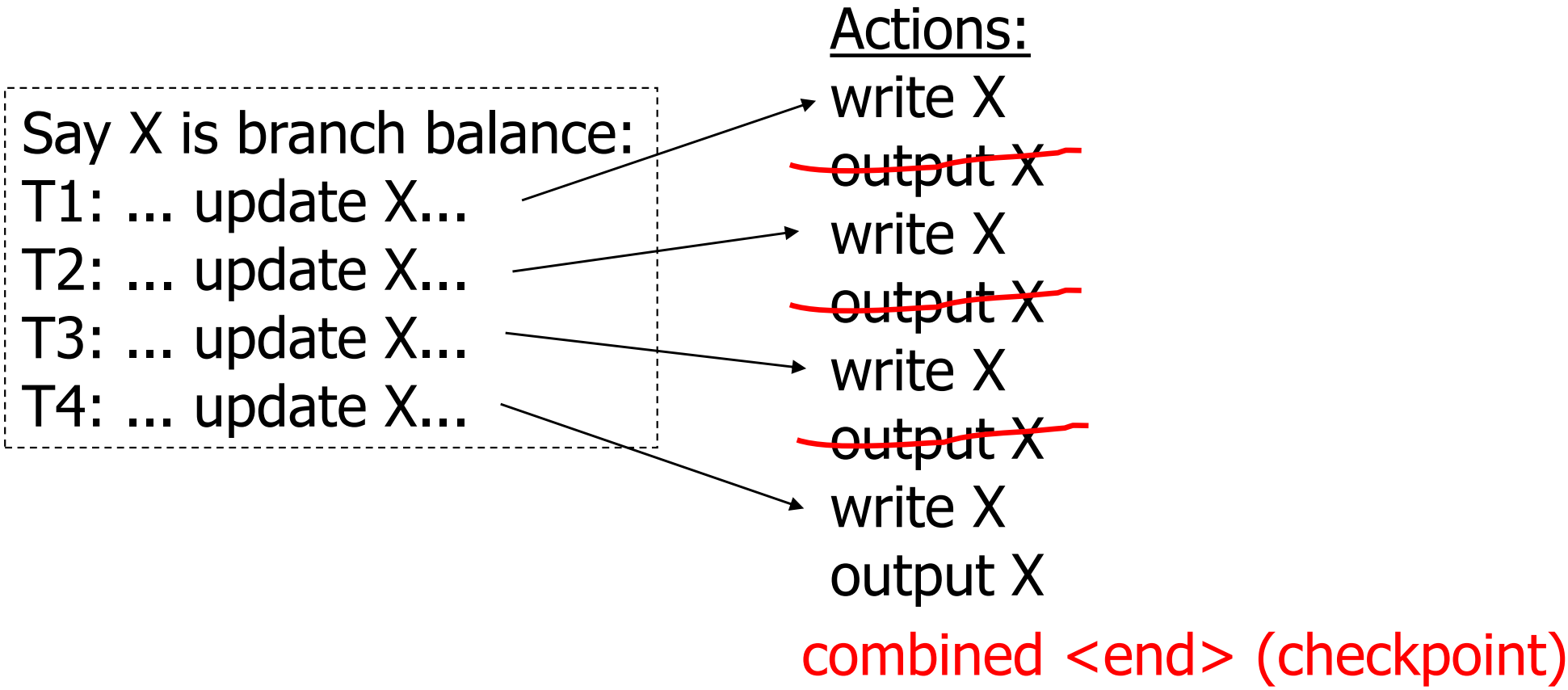
Combining <Ti, end> Records

- Want to delay DB flushes for hot objects



Combining <Ti, end> Records

- Want to delay DB flushes for hot objects



Solution: Checkpoint

good for not having an ever increasing log that we need to scan entirely at each crash

- no <ti, end> actions>
- simple checkpoint

make the log such that all that came before does not need to be processed in case of crash recovery

only need to process what comes after for undo or redo

Periodically:

(1) Do not accept new transactions

freeze the DB periodically, blocking new transactions

(2) Wait until all transactions finish

(3) Flush all log records to disk (log)

(4) Flush all buffers to disk (DB) (do not discard buffers)

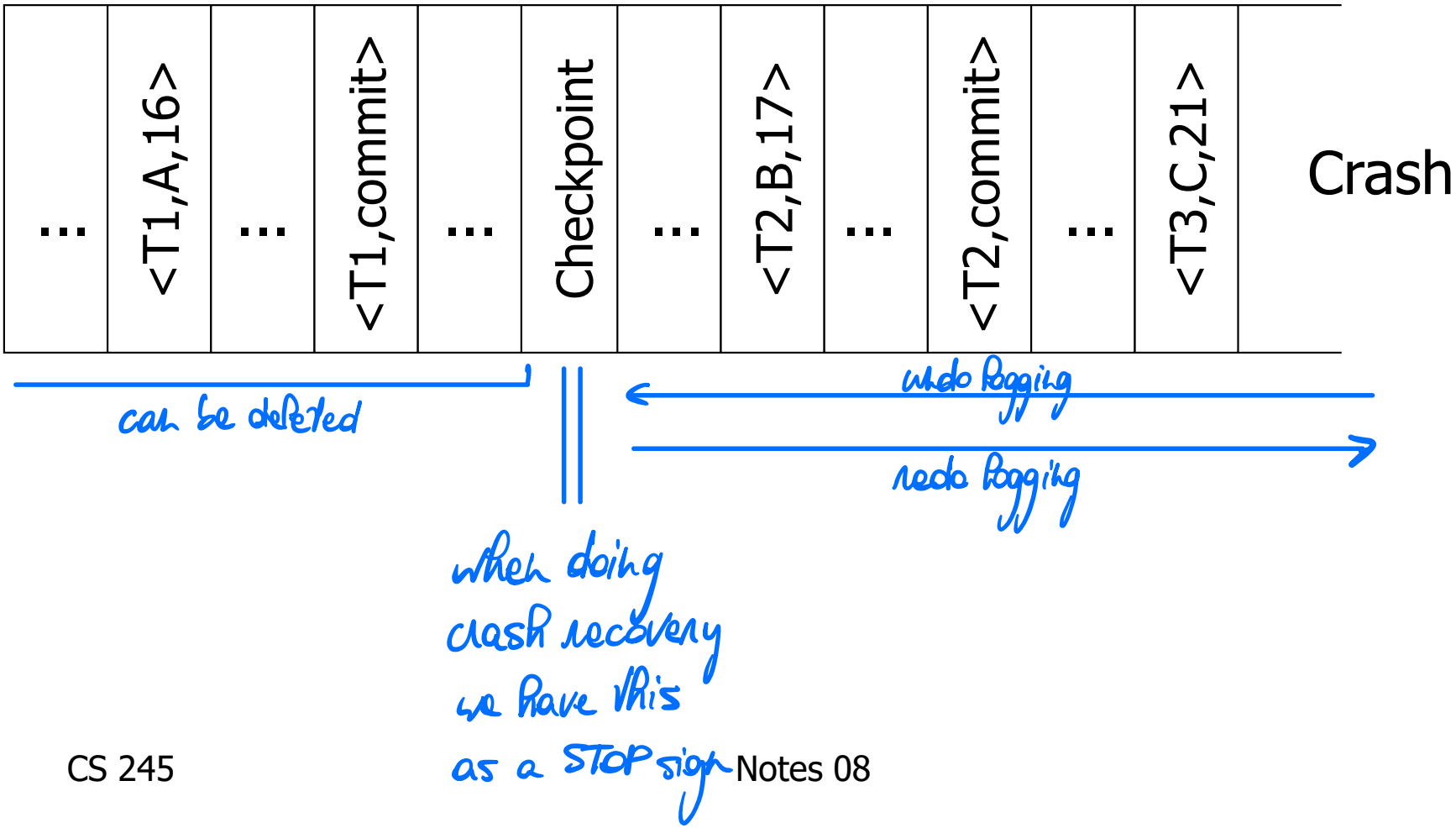
(5) Write "checkpoint" record on disk (log)

(6) Resume transaction processing

"Simple Checkpoints" ensure that everything before each checkpoint is completely processed and won't require any crash recovery actions

Example: what to do at recovery?

Redo log (disk):



Key drawbacks:

put data to disk early and free memory
but we flush more often
⇒ I/O cost

- *Undo logging:* increased I/O cost
- *Redo logging:* need to keep all modified blocks in memory until commit

stack things in memory for a longer period
↳ less I/O

Solution: undo/redo logging!

Update ⇒ <Ti, Xid, New X val, Old X val>

page X

reference to the page

Rules

- Page X can be flushed before or after Ti commit
- Log record flushed before corresponding updated page (WAL)
- Flush at commit (log only)

always log before output

write ahead logging

↳ and flush log when we commit

without commit, the consistency might not be complete

can't be sure that we are not missing an action

undo

else, no idea

if no commit : undo

commit = no more action to do

Example: Undo/Redo logging

business is correct !!

↳ if commit : redo

↳ if is a committed transaction

Step	Action	t	M-A	M-B	D-A	D-B	Log
1)							<START T>
2)	READ(A,t)	8	8		8	8	
3)	t := t*2	16	8		8	8	
4)	WRITE(A,t)	16	16		8	8	<$T, A, 8, 16$>
5)	READ(B,t)	8	16	8	8	8	
6)	t := t*2	16	16	8	8	8	
7)	WRITE(B,t)	16	16	16	8	8	<$T, B, 8, 16$>
8)	FLUSH LOG → WAL						
9)	OUTPUT(A)	16	16	16	16	8	
10)							<COMMIT T>
11)	OUTPUT(B)	16	16	16	16	16	↳ no need to flush the commit

no need to log manually, we can let the buffer manager decide when

any order

Figure 17.9: A possible sequence of actions and their log entries using undo/redo logging

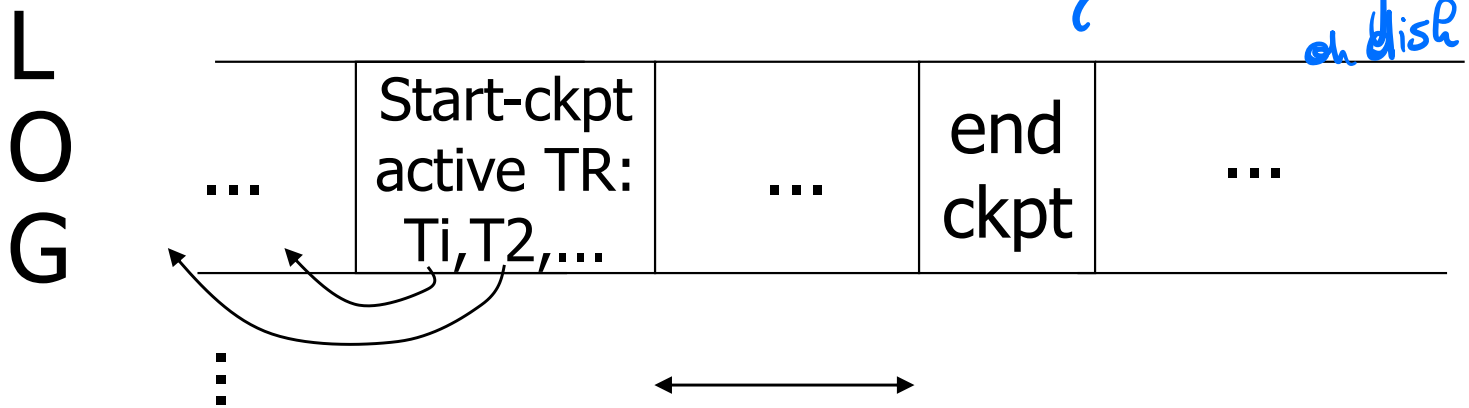
Simple Checkpoints are not affordable anymore, logging size becomes critical, we have to scan the whole log

free for buffer manag. to chose

NOT PART OF THE EXAM TOPICS

Non-quiesce checkpoint

for those who commited before starting checkpoint everything is on disk



for undo

dirty buffer pool pages flushed

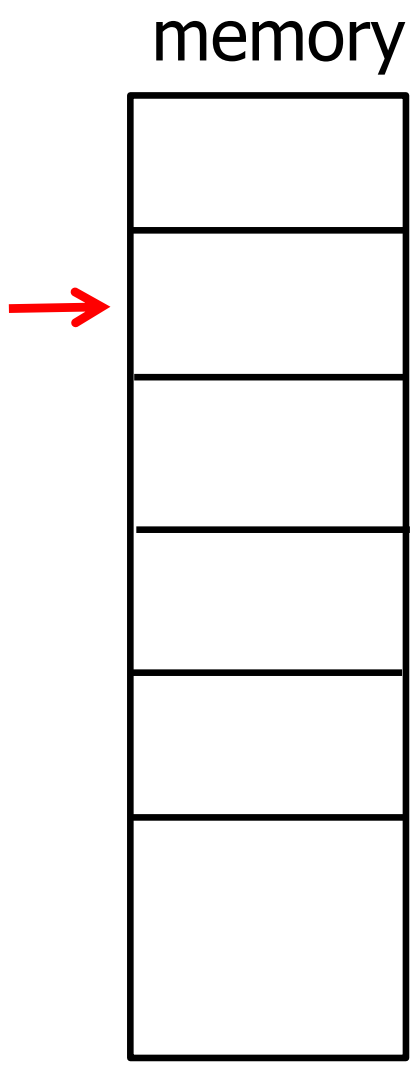
even if we see a commit, it does not mean that the changes occured in the disk

we still have uncertainty about this part

Non-quiesce checkpoint

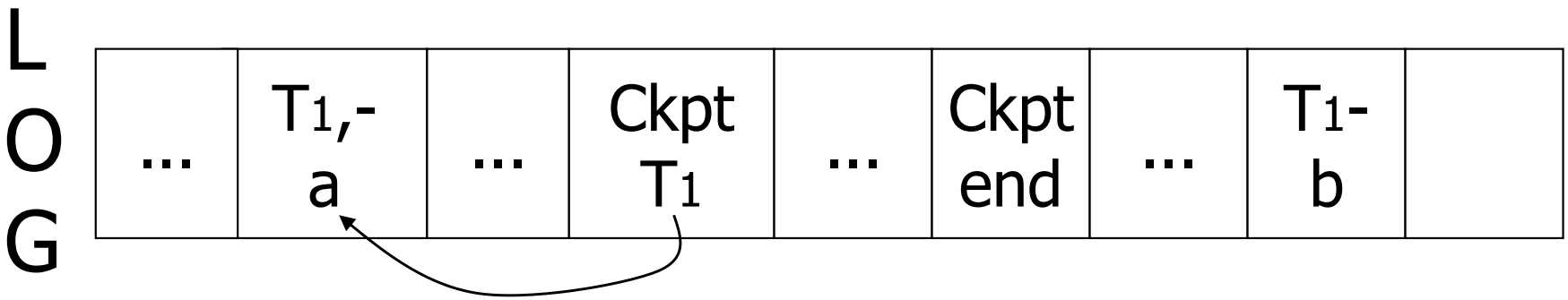
checkpoint process:
for i := 1 to M do
    output(buffer i)

[transactions run concurrently]

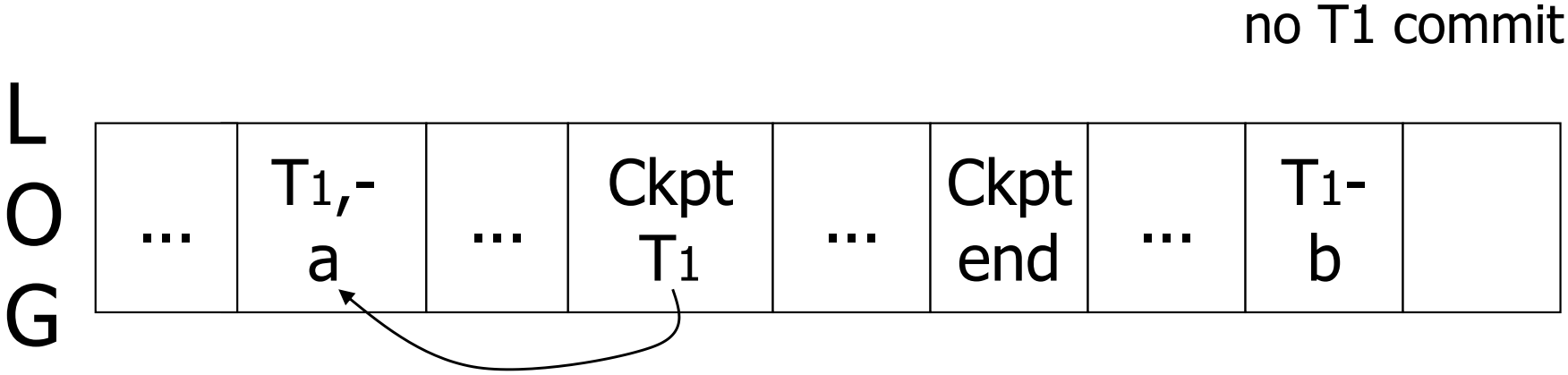


Examples what to do at recovery time?

no T1 commit



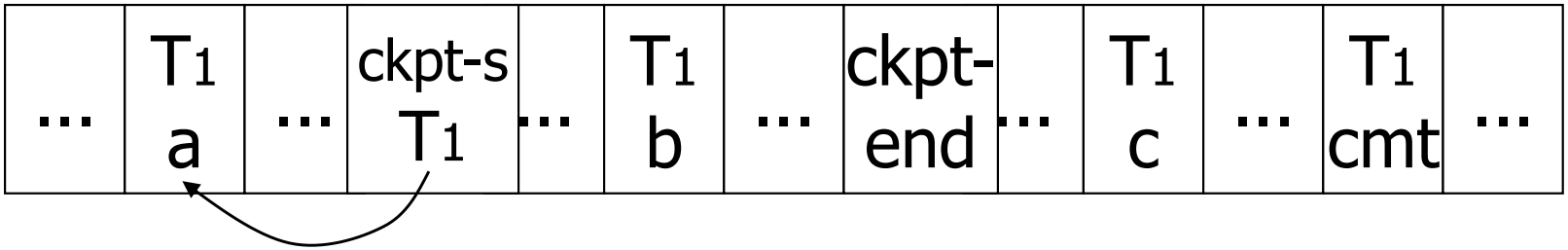
Examples what to do at recovery time?



☒ Undo T_1 (undo a,b)

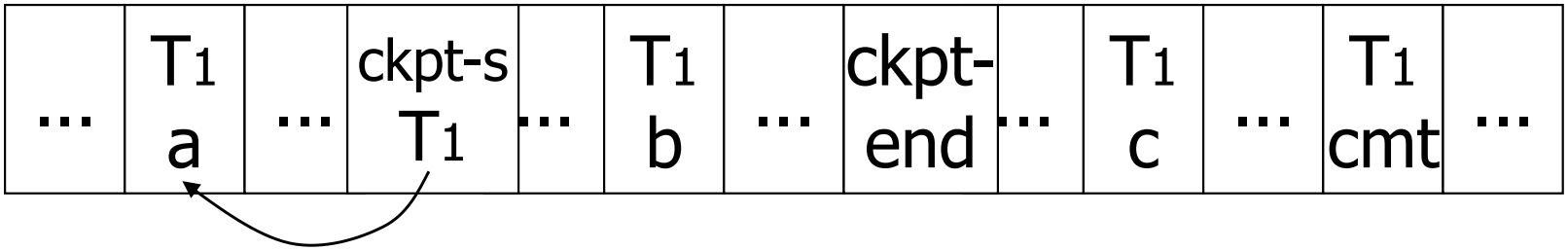
Example

LOG



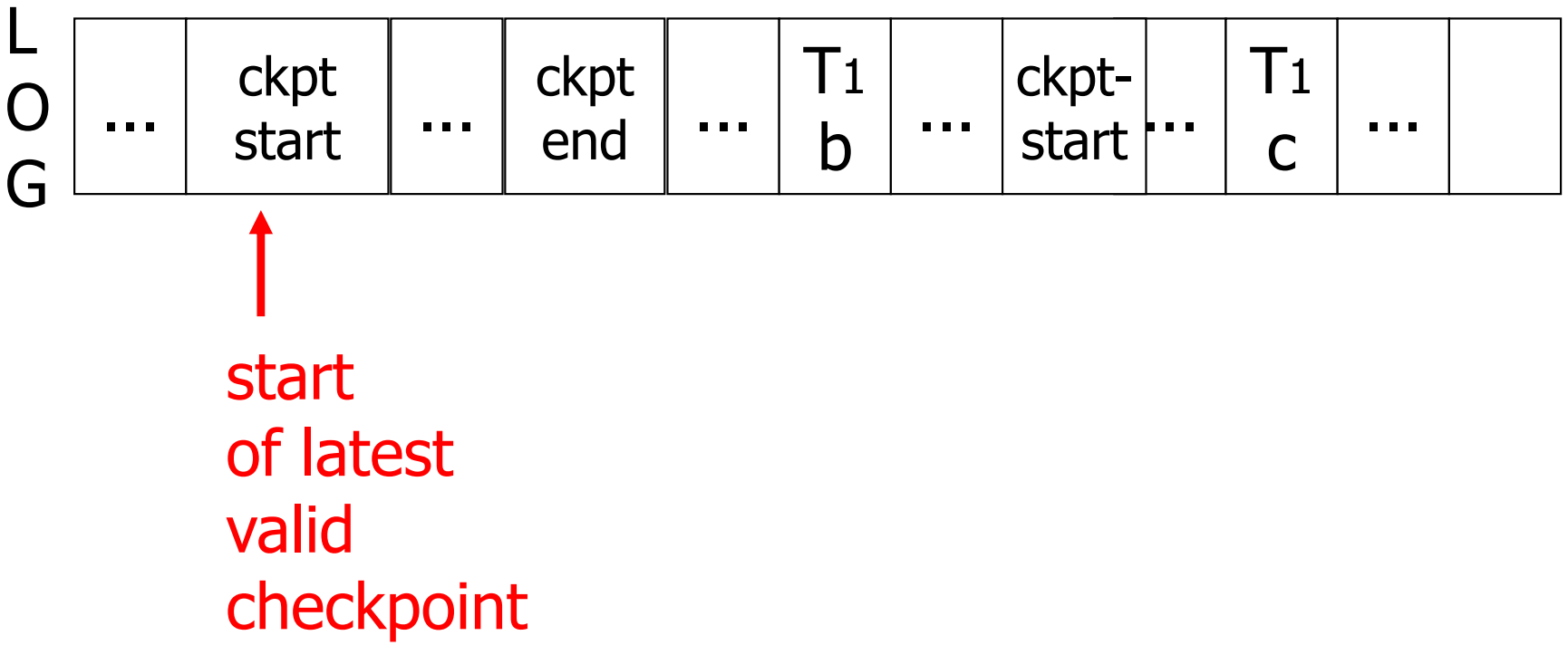
Example

LOG



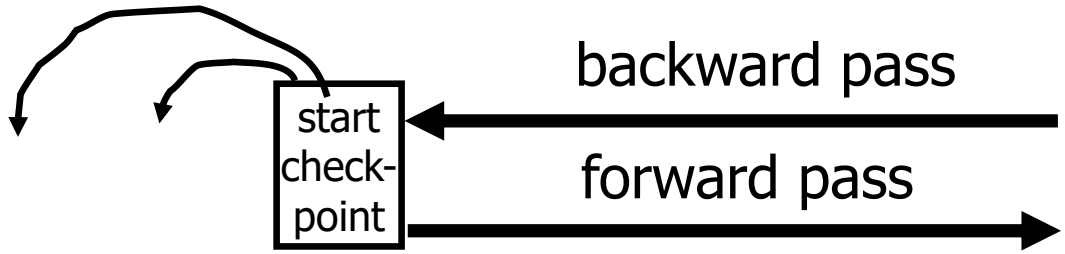
☒ Redo T1: (redo b,c)

Recover From Valid Checkpoint:



Recovery process:

- Backwards pass (end of log ➲ latest valid checkpoint start)
  - construct set S of committed transactions
  - undo actions of transactions not in S
- Undo pending transactions
  - follow undo chains for transactions in (checkpoint active list) - S
- Forward pass (latest checkpoint start ➲ end of log)
  - redo actions of S transactions



Real world actions

E.g., dispense cash at ATM

$T_i = a_1\ a_2\ \ldots\ a_j\ \ldots\ a_n$

$

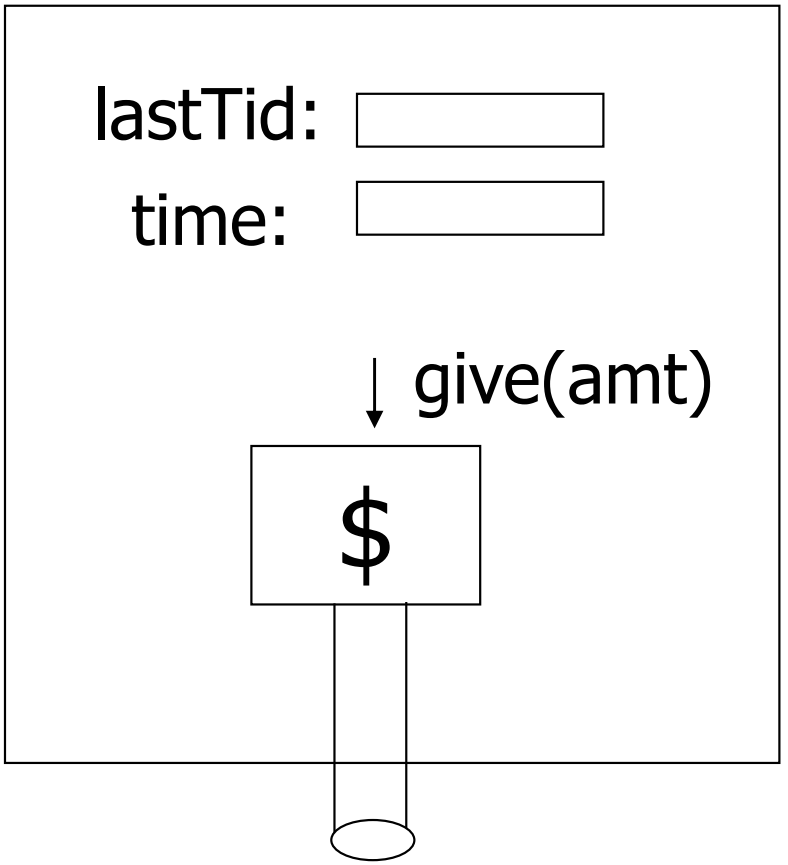
Solution

(1) execute real-world actions after commit

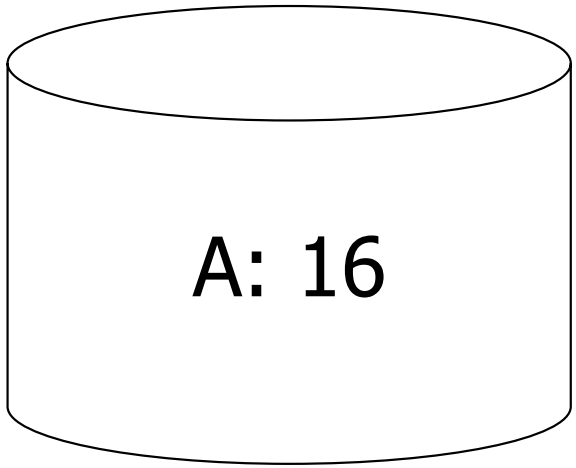
(2) try to make idempotent

Give$$

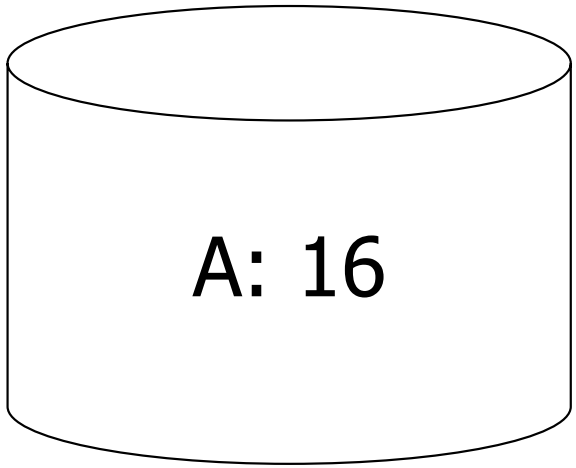
(amt, Tid, time)



Media failure (loss of non-volatile storage)



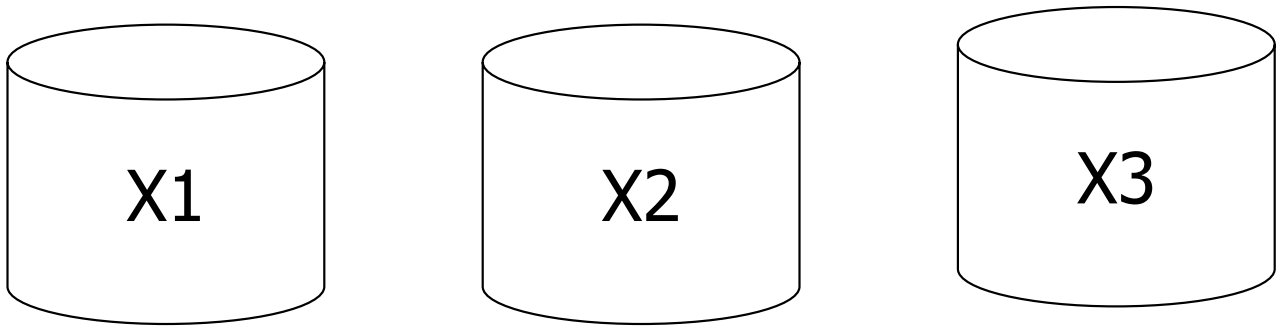
Media failure (loss of non-volatile storage)



Solution: Make copies of data!

Example 1 Triple modular redundancy

- Keep 3 copies on separate disks
- Output(X) --> three outputs
- Input(X) --> three inputs + vote

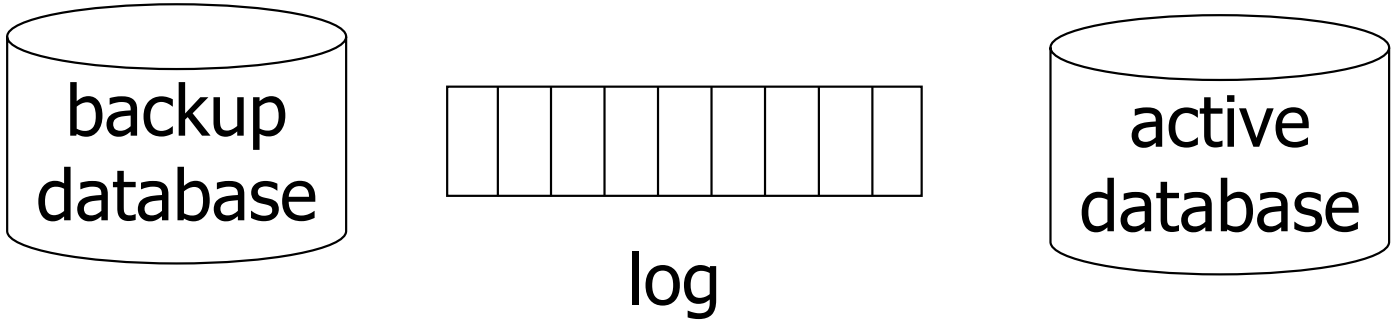


Example #2 Redundant writes, Single reads

- Keep N copies on separate disks
- Output(X) --> N outputs
- Input(X) --> Input one copy
  - if ok, done
  - else try another one

⇔ Assumes bad data can be detected

Example #3: DB Dump + Log



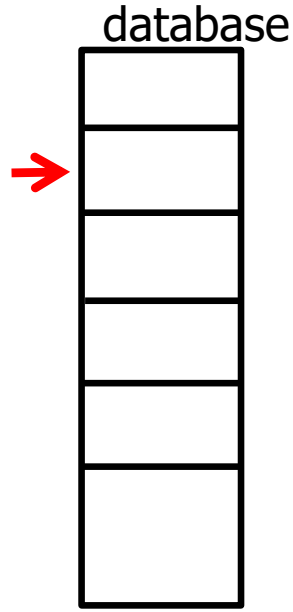
- If active database is lost,
  - restore active database from backup
  - bring up-to-date using redo entries in log

Backup Database

- Just like checkpoint,
  except that we write full database

```
create backup database:
for i := 1 to DB_Size do
   [read DB block i; write to backup]

[transactions run concurrently]
```

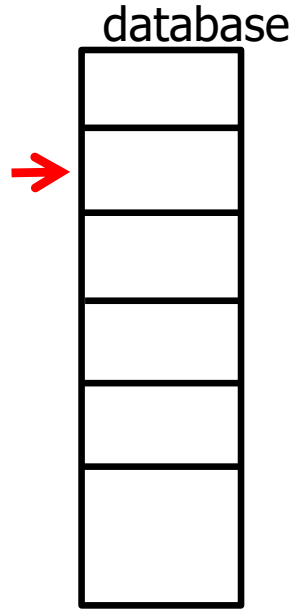


Backup Database

- Just like checkpoint,
  except that we write full database

```
create backup database:
for i := 1 to DB_Size do
   [read DB block i; write to backup]

[transactions run concurrently]
```



- Restore from backup DB and log:
  Similar to recovery from checkpoint and log

Summary

- Consistency of data
- One source of problems: failures
  - Logging
  - Redundancy
- Another source of problems:
  Data Sharing (not in this lecture)