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!

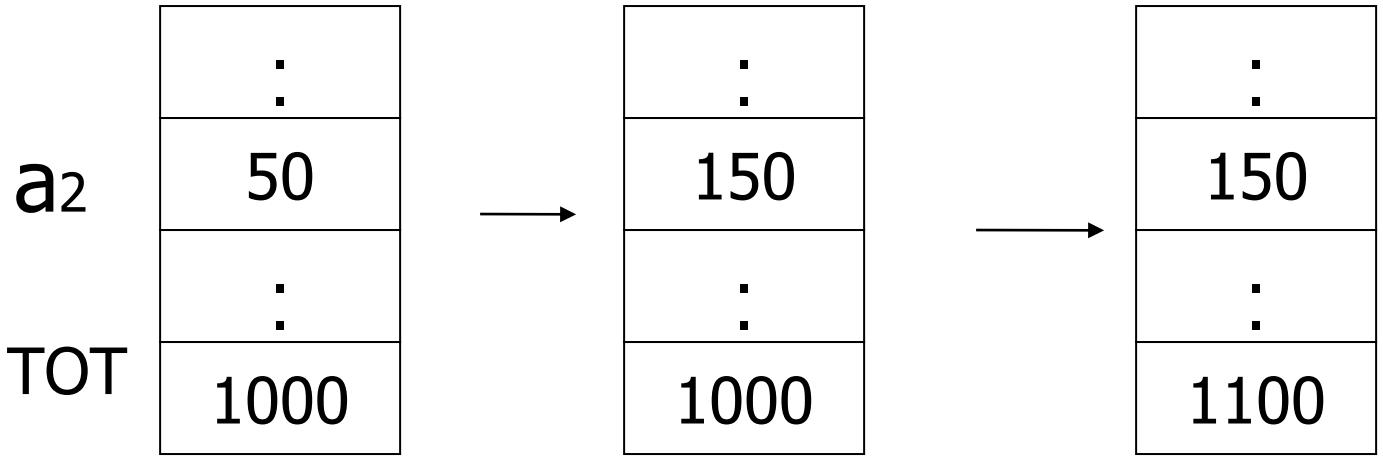
Example: $a_1 + a_2 + \ldots a_n = TOT$

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

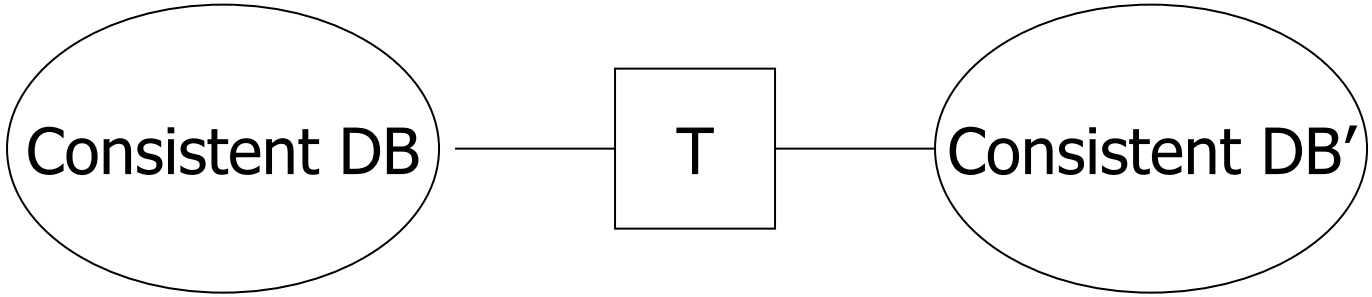
Example: $a_1 + a_2 + \ldots a_n = TOT$

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

$\Rightarrow$ T leaves consistent state

To execute in isolation means leaving the consistent state because some constraints won't be followed.

Correctness (informally)

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

If the database fails before, of after the transaction -> the DB is still consistent

But if it fails during a transaction, the DB looses its consistency

How can consistency be violated?

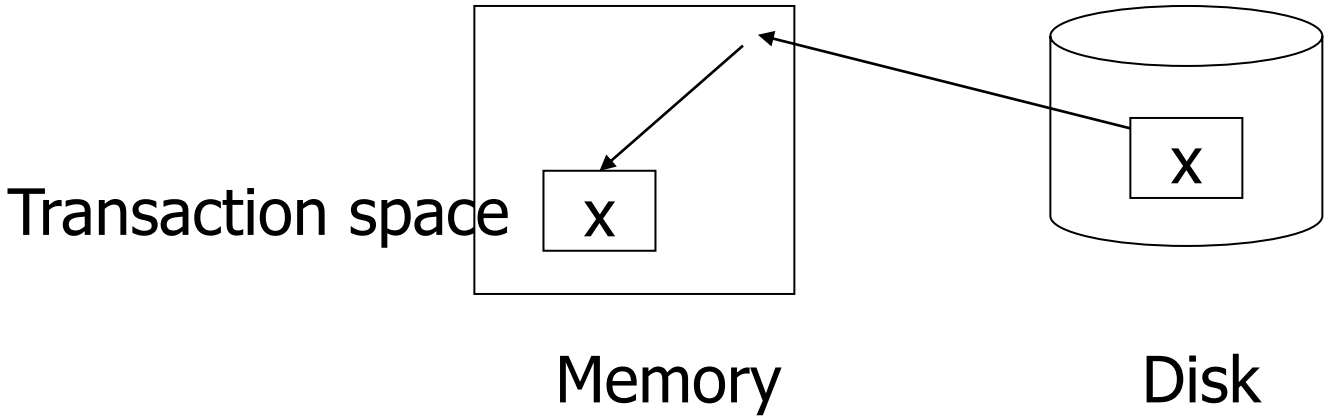
- System crash

  - memory lost
  - cpu halts, resets

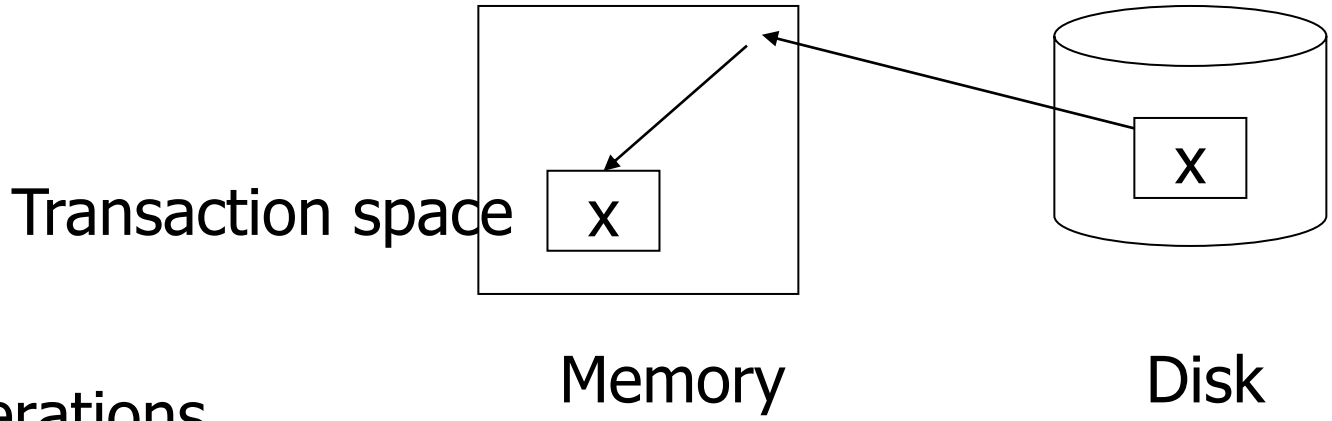
  A powerdown, a blue screen can violate the consistency of the DB

- Media failure, catastrophes

Storage hierarchy



Storage hierarchy



Operations

- Input (x): block containing x $\rightarrow$ memory
- Output (x): block containing x $\rightarrow$ disk
- Read (x,t): do input(x) if necessary, t $\leftarrow$ value of x in block
- Write (x,t): do input(x) if necessary, value of x in block $\leftarrow$ t

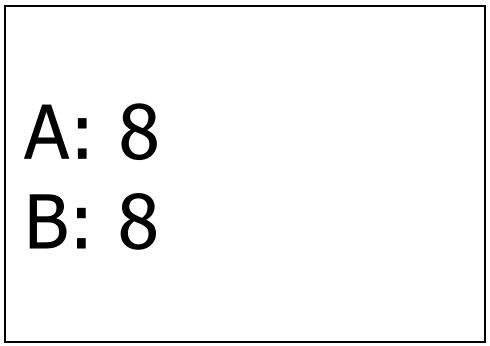
Key problem Unfinished transaction

Example Constraint: A=B

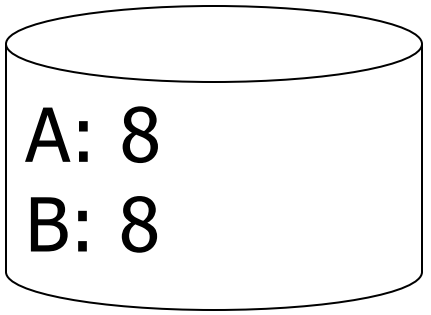
T_1: $A \leftarrow A \times 2$

$B \leftarrow B \times 2$

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

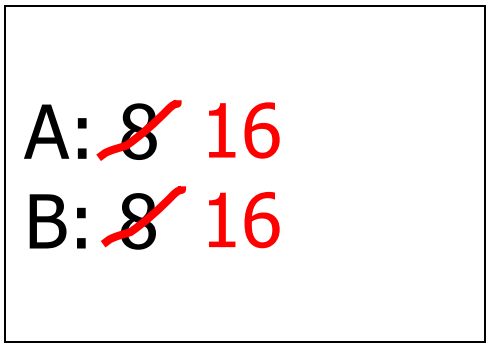


memory

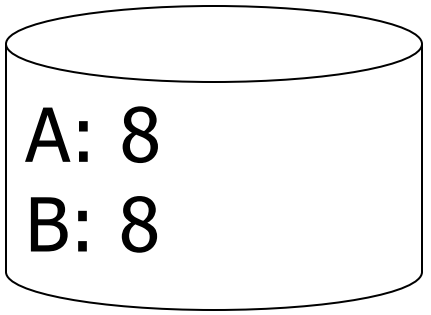


disk

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



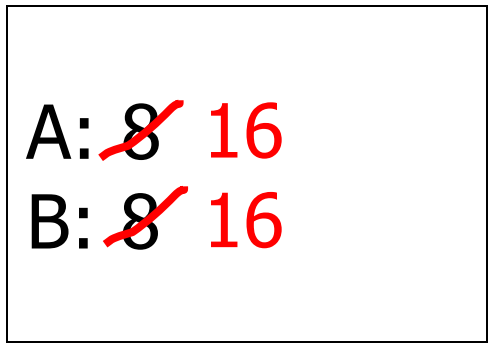
memory



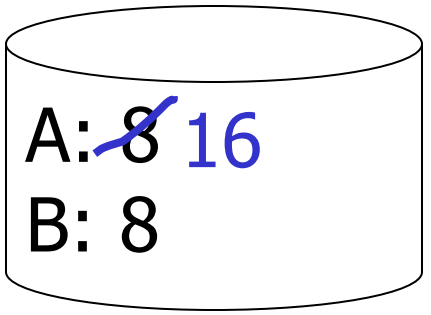
disk

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

failure!



memory

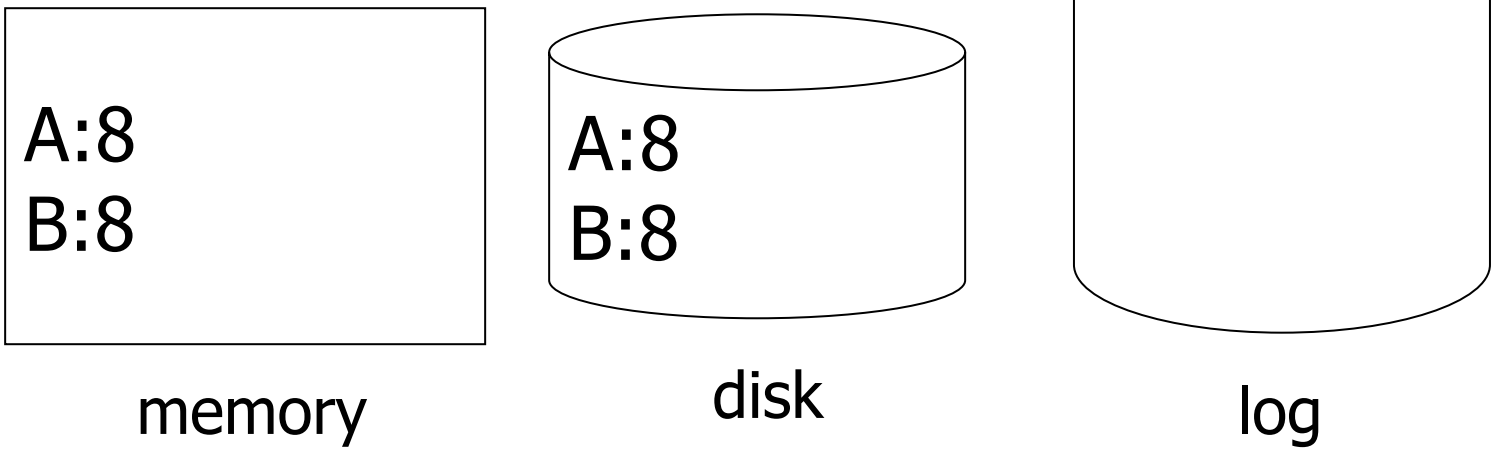


disk

- Need atomicity: execute all actions of a transaction or none at all

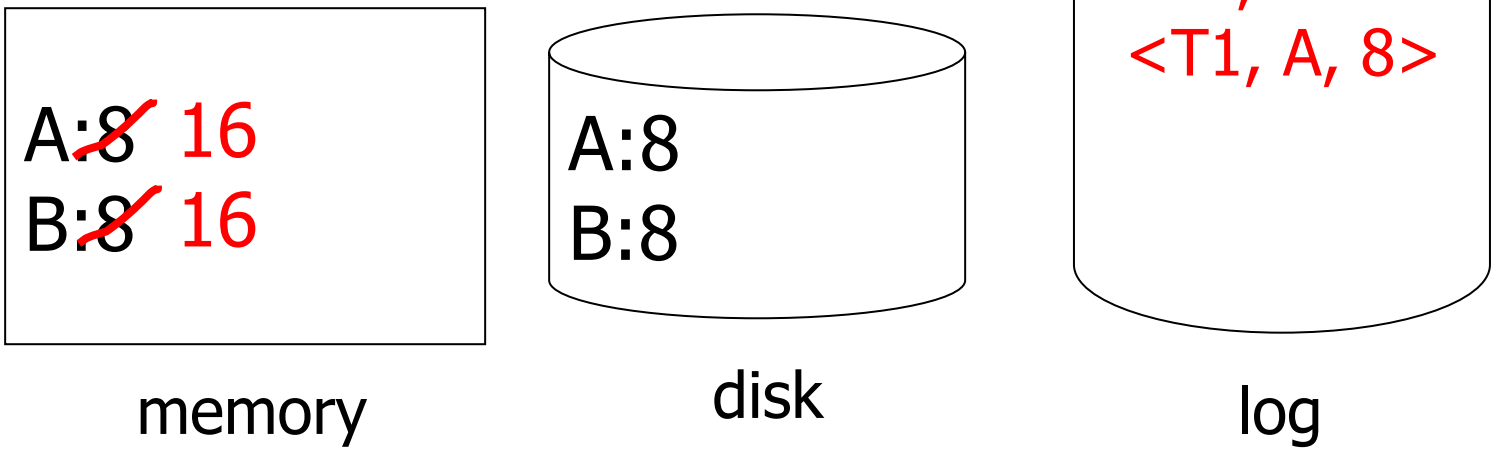
Undo logging (Immediate modification)

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);



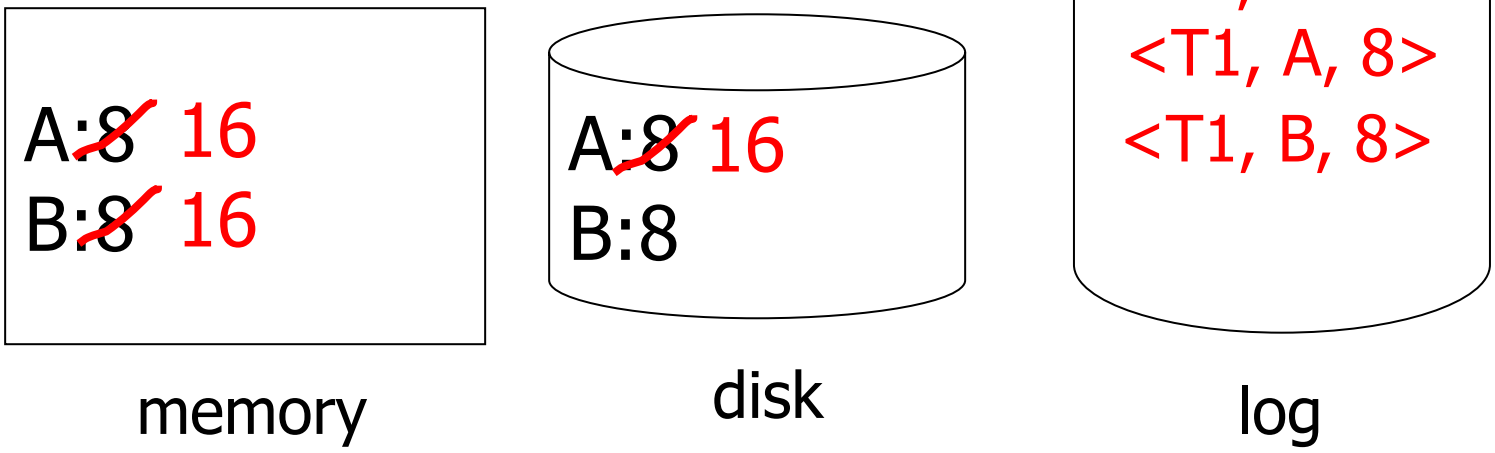
Undo logging (Immediate modification)

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);



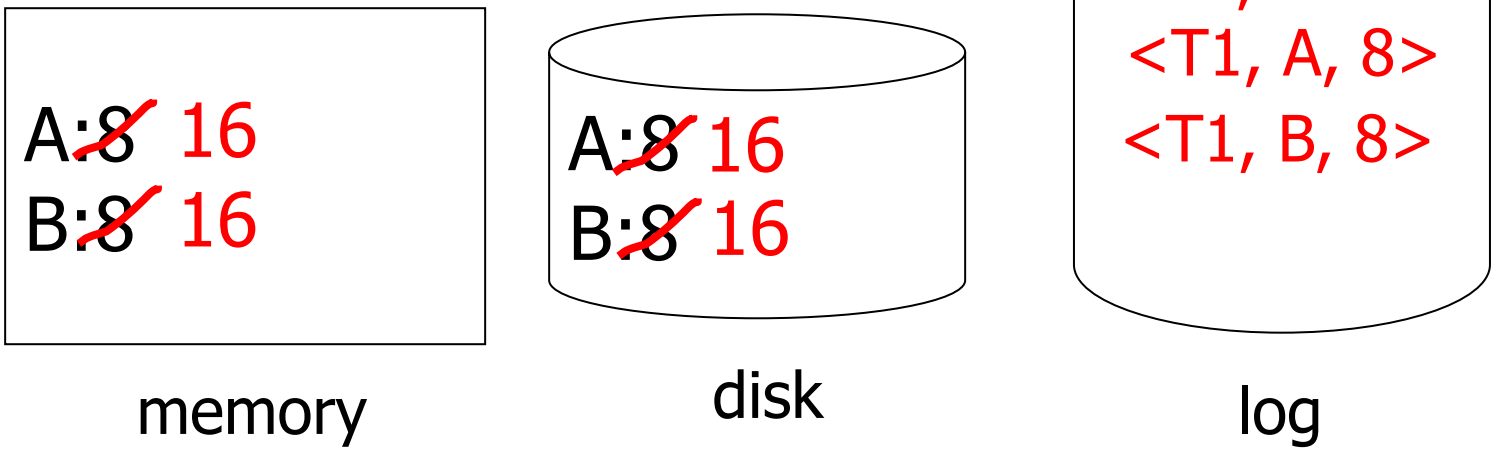
Undo logging (Immediate modification)

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);



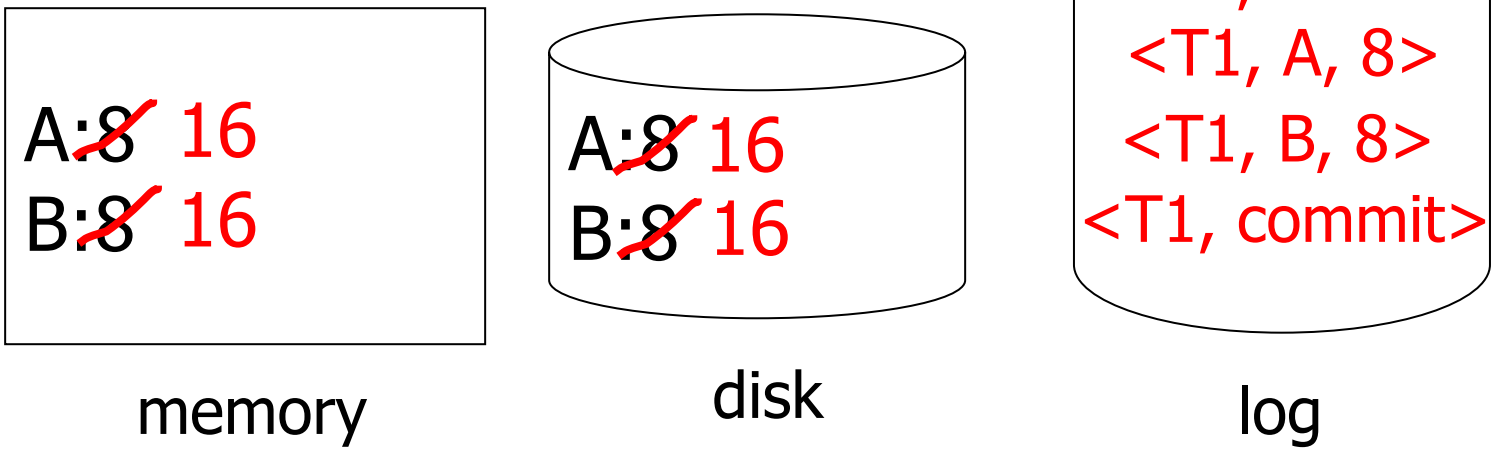
Undo logging (Immediate modification)

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);



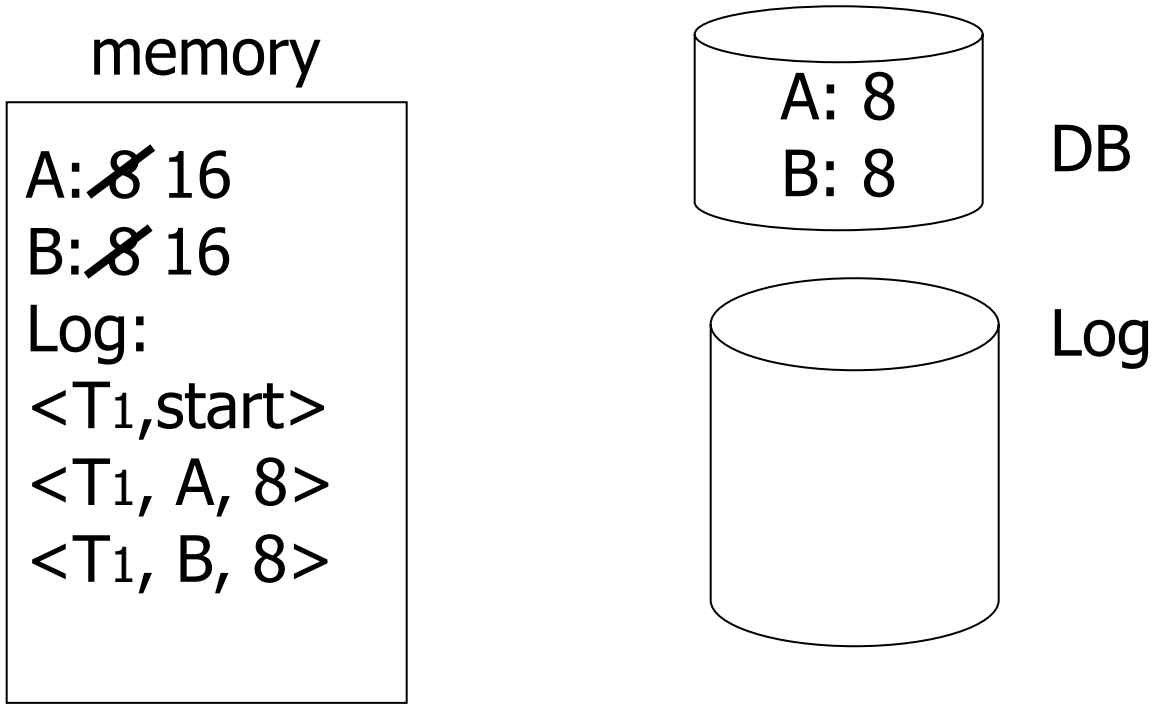
Undo logging (Immediate modification)

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);



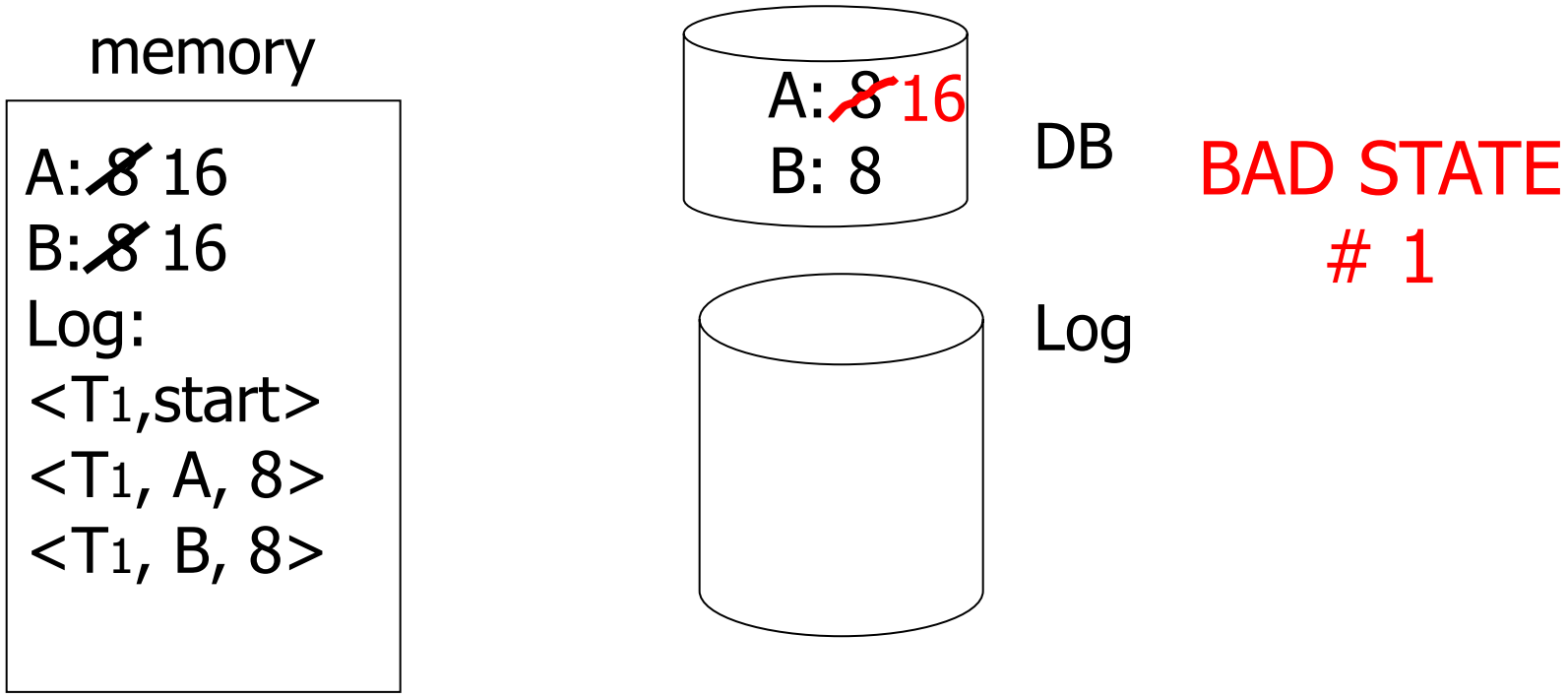
One "complication"

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



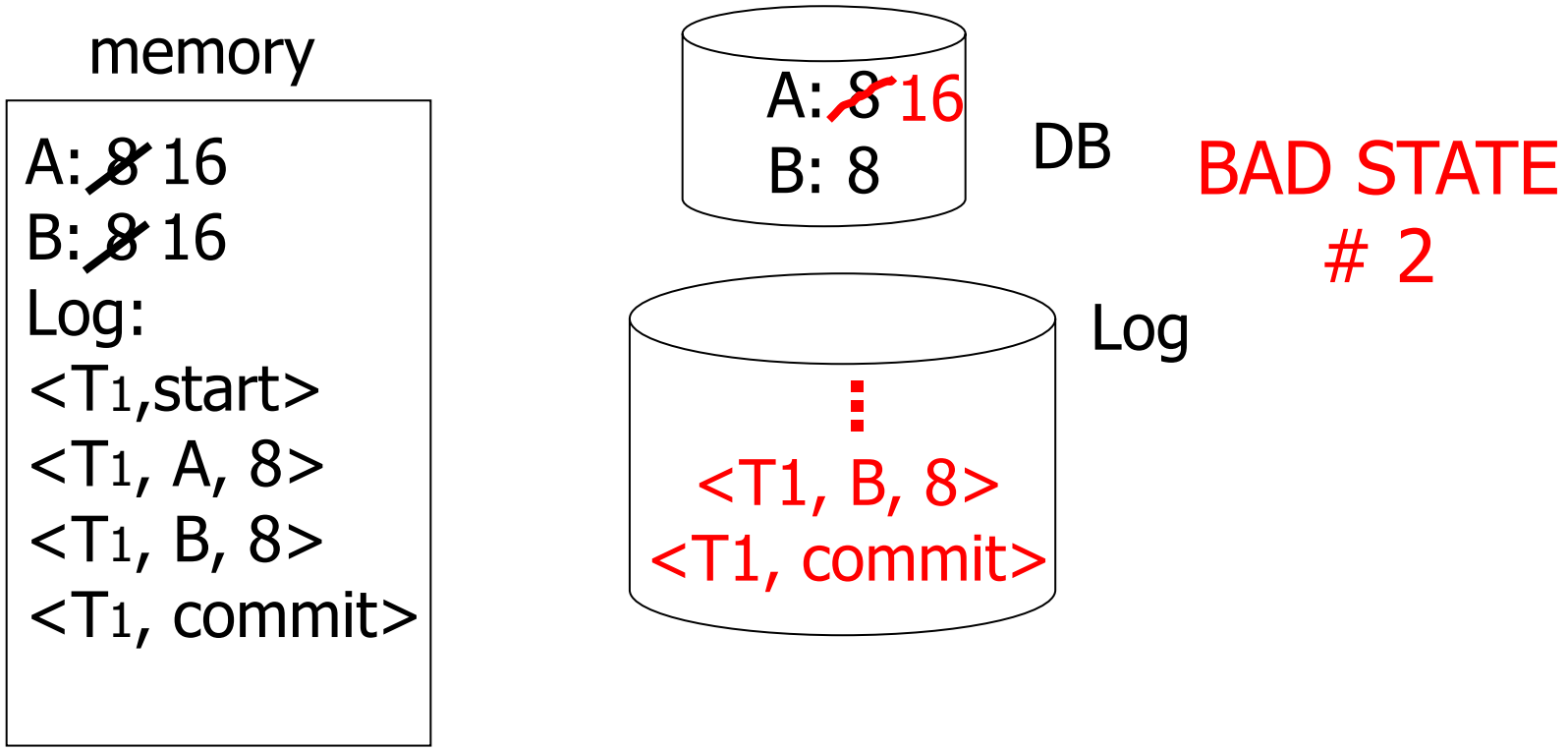
One "complication"

- 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						

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

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

Not correct because the order of the x is not good
Need to be bottom up and not top down!

☒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)

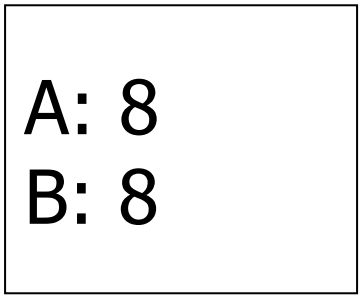
(3) For each Ti $\in$ S do
- write <Ti, abort> to log

What if failure during recovery?

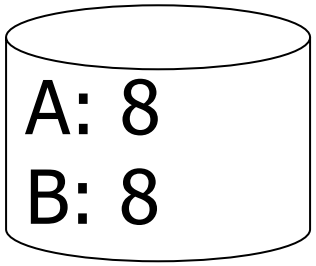
No problem! ⇨ Undo idempotent

Redo logging (deferred modification)

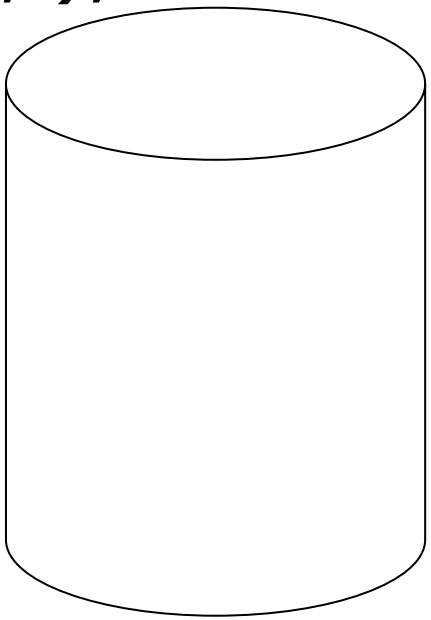
T_1: Read(A,t); t← t×2; write (A,t);
Read(B,t); t←t×2; write (B,t);
Output(A); Output(B)



memory



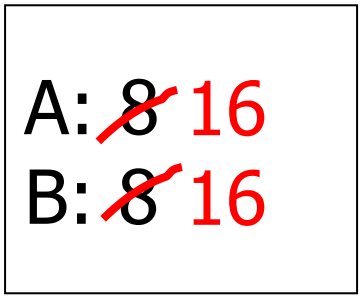
DB



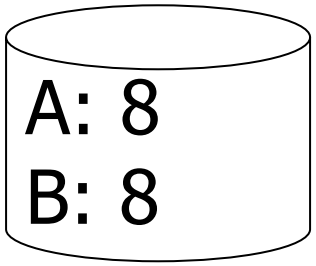
LOG

Redo logging (deferred modification)

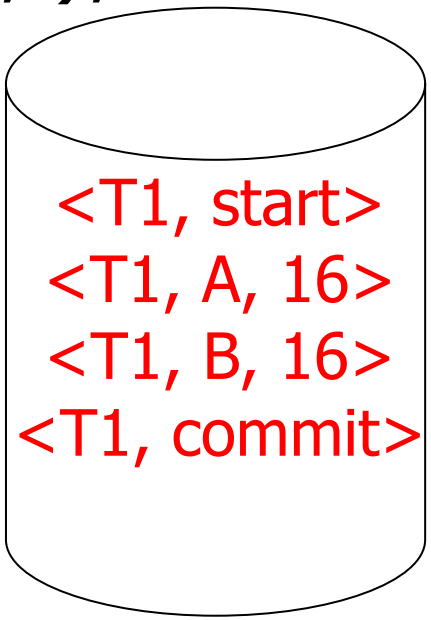
T_1: Read(A,t); t← t×2; write (A,t);
Read(B,t); t←t×2; write (B,t);
Output(A); Output(B)



memory



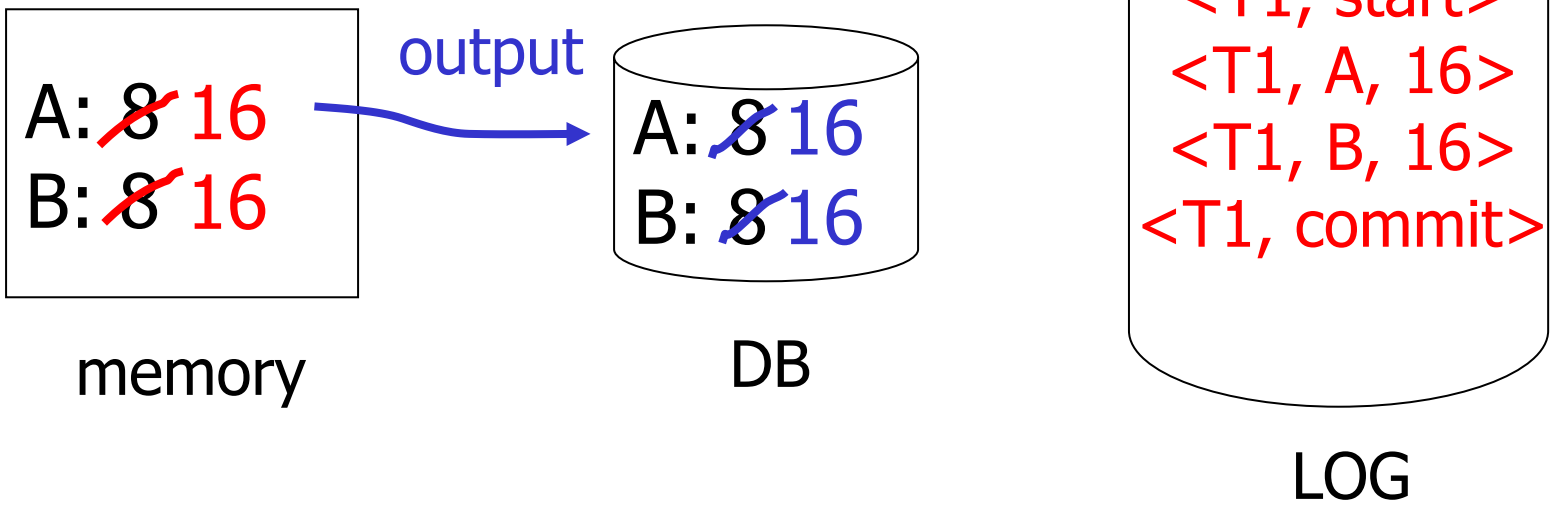
DB



LOG

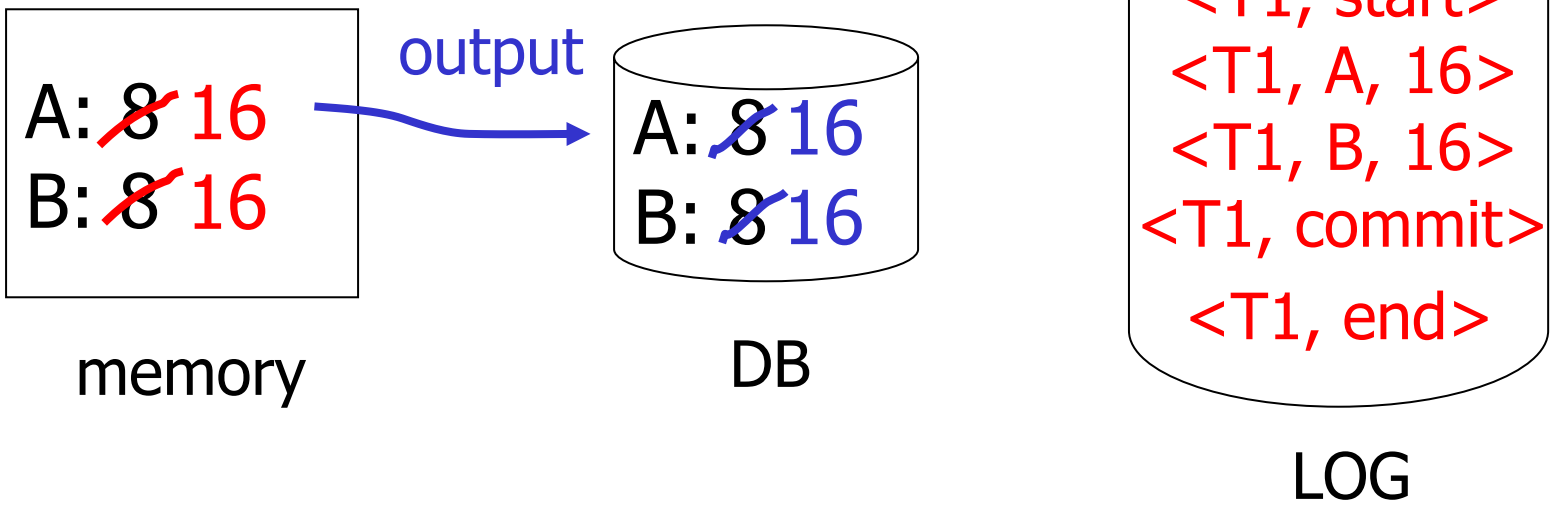
Redo logging (deferred modification)

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)

(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 $\rightarrow$ latest) do:

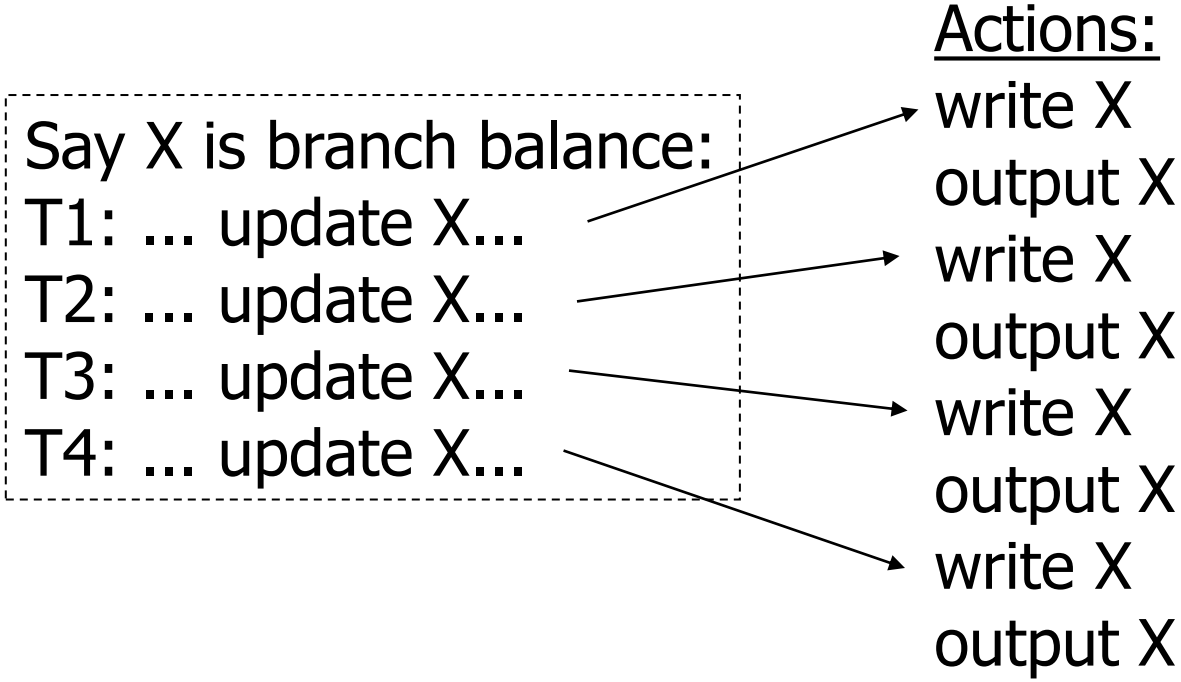
- if Ti $\in$ S then { Write(X, v); Output(X) }

Top down

(3) For each Ti $\in$ S, write <Ti, end>

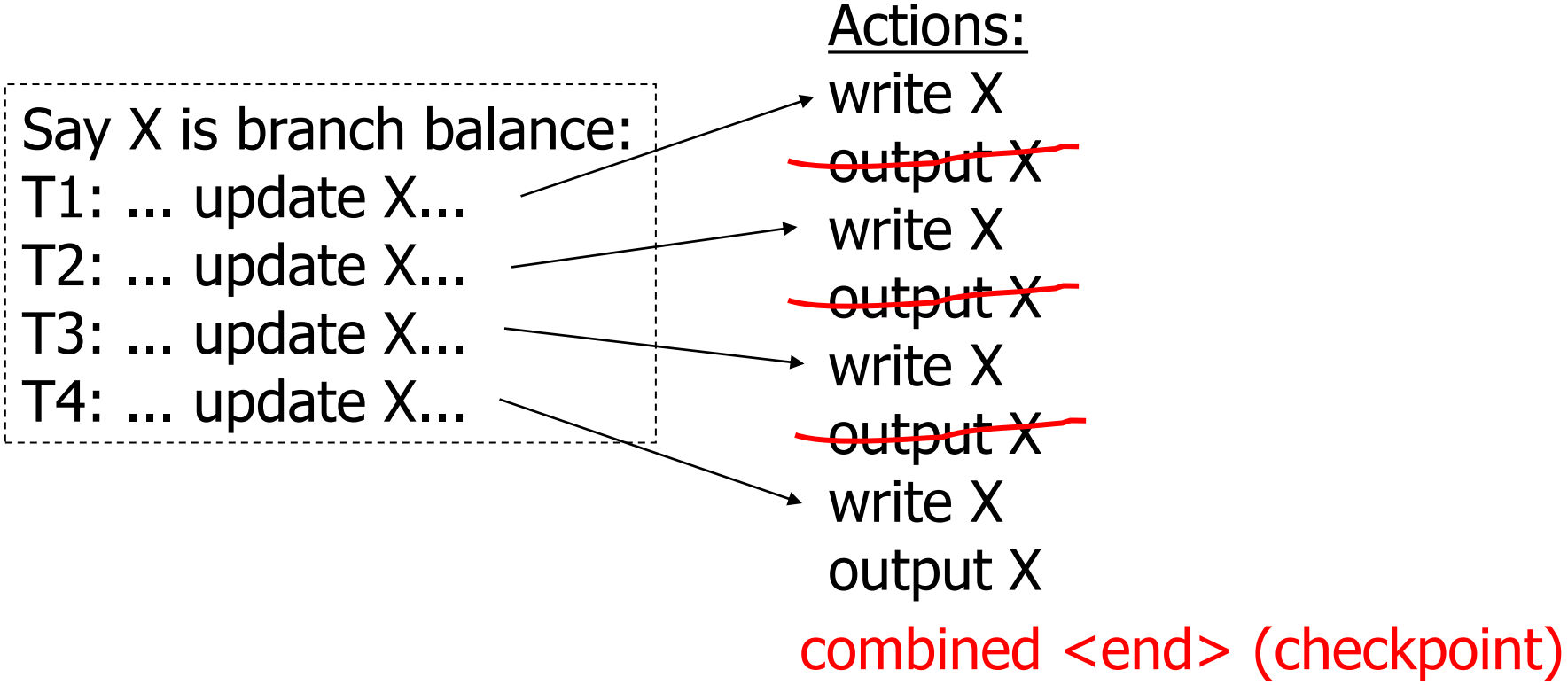
Combining <Ti, end> Records

- Want to delay DB flushes for hot objects



Combining <Ti, end> Records

- Want to delay DB flushes for hot objects



Solution: Checkpoint

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

Periodically:

(1) Do not accept 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)

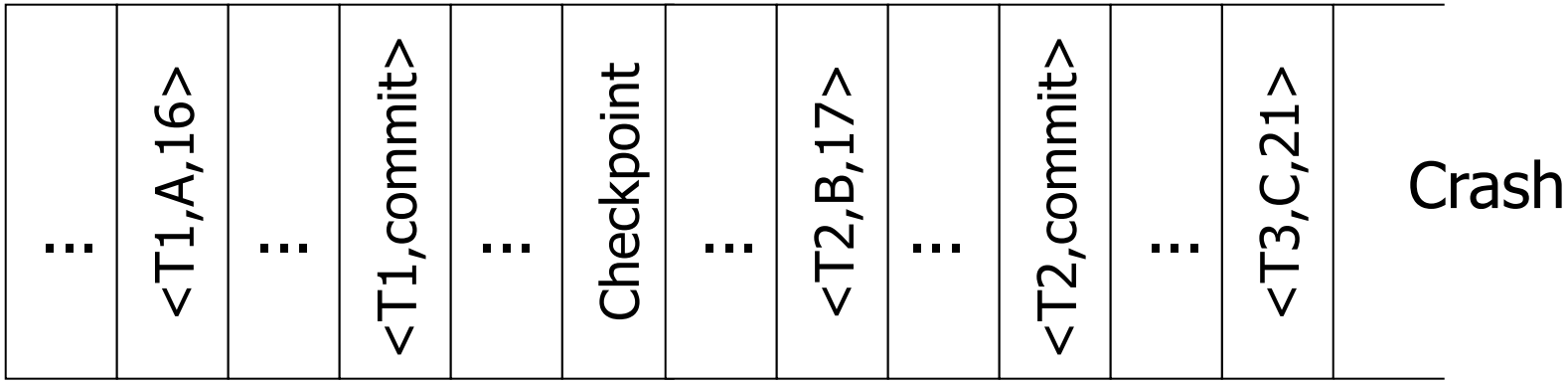
(5) Write "checkpoint" record on disk (log)

(6) Resume transaction processing

A checkpoint allows to know where to restart not to redo everything (the log file can be VERY long)

Example: what to do at recovery?

Redo log (disk):



Key drawbacks:

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

Solution: undo/redo logging!

Update $\Rightarrow$ <Ti, Xid, New X val, Old X val>

page X

Rules

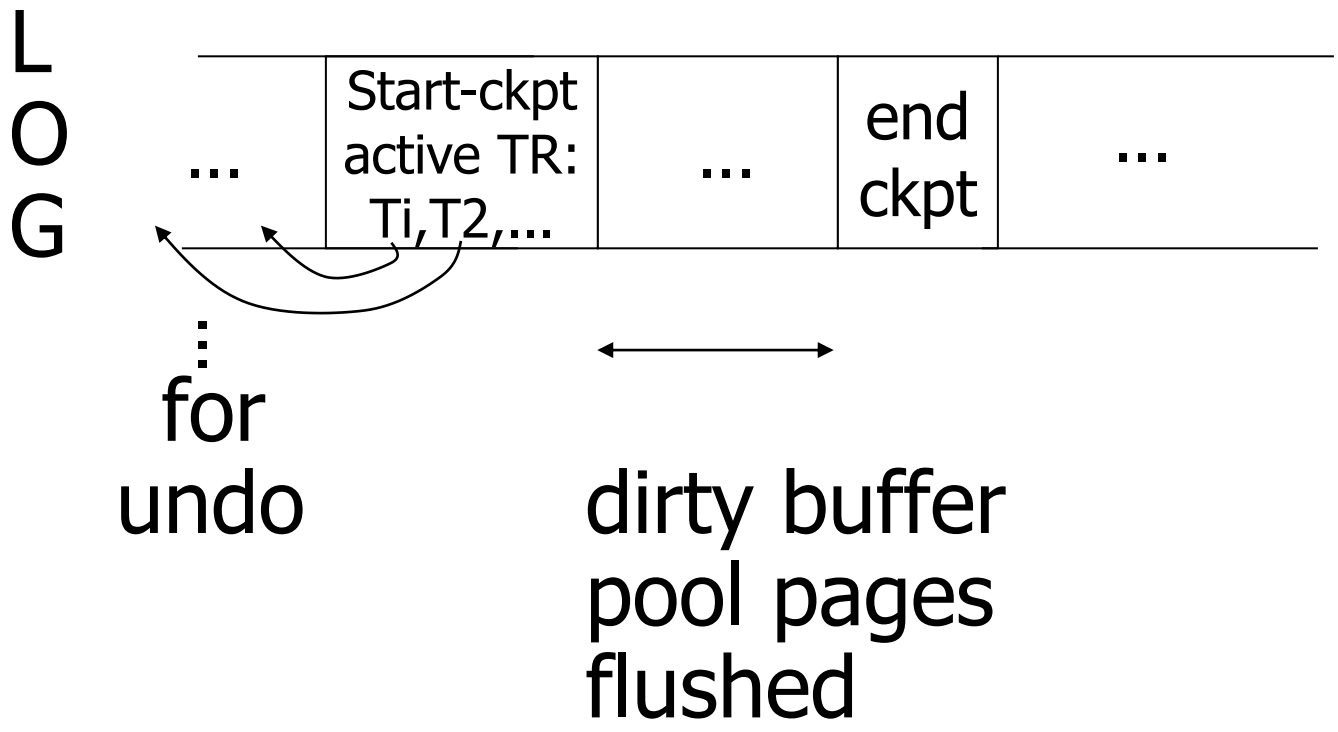
- Page X can be flushed before or after Ti commit
- Log record flushed before corresponding updated page (WAL)
- Flush at commit (log only)

Example: Undo/Redo logging

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						
9)	OUTPUT(A)	16	16	16	16	8	
10)							<COMMIT T>
11)	OUTPUT(B)	16	16	16	16	16	

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

Non-quiesce checkpoint



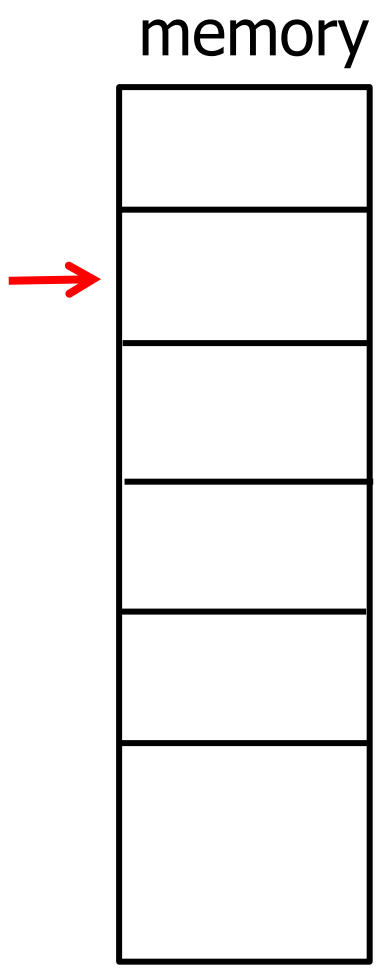
Any comment that appear before, we don't need to redo/repeat them

After the start of the checkpoint, redo. Before, don't redo

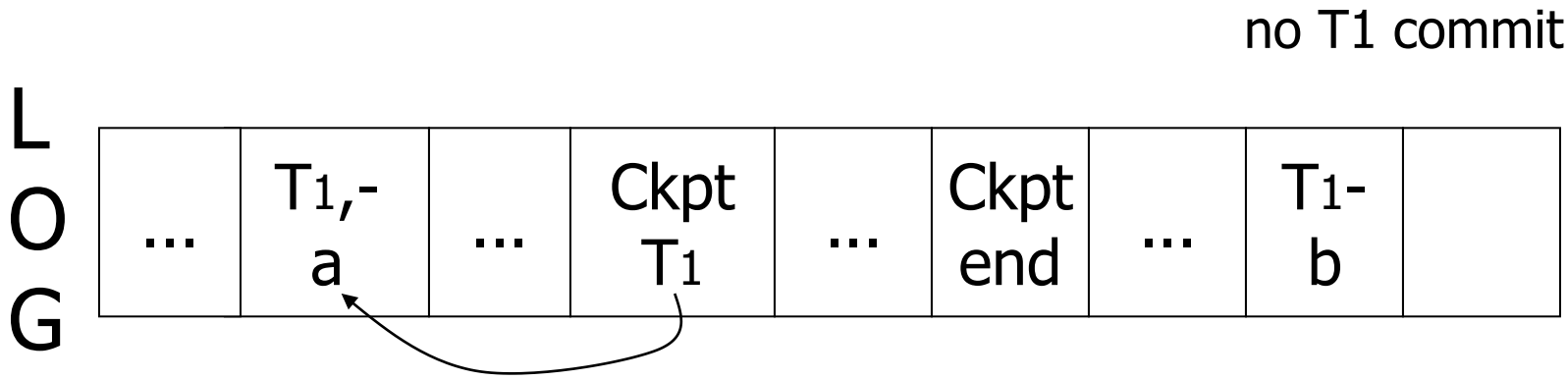
Non-quiesce checkpoint

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

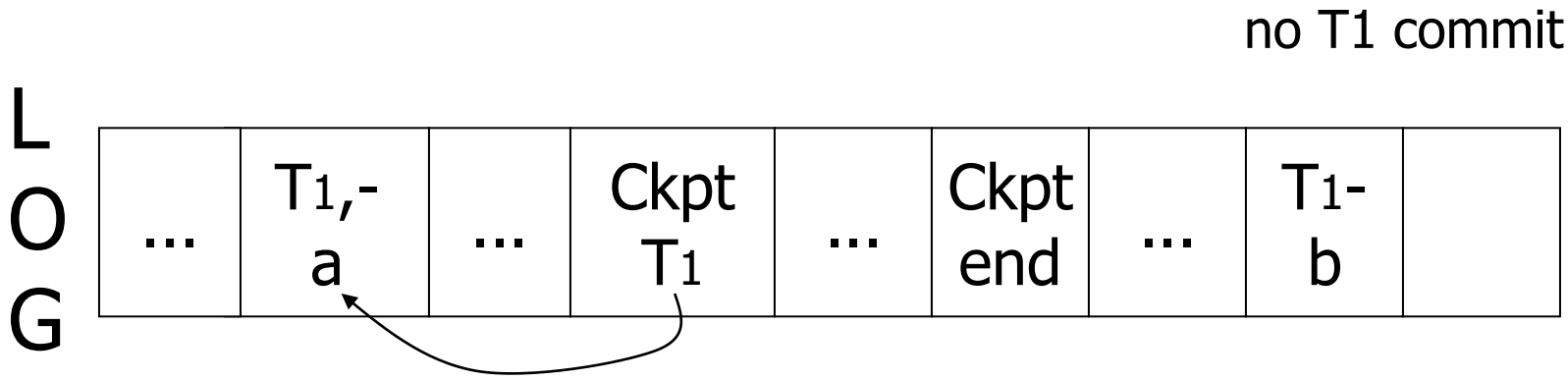
[transactions run concurrently]



Examples what to do at recovery time?



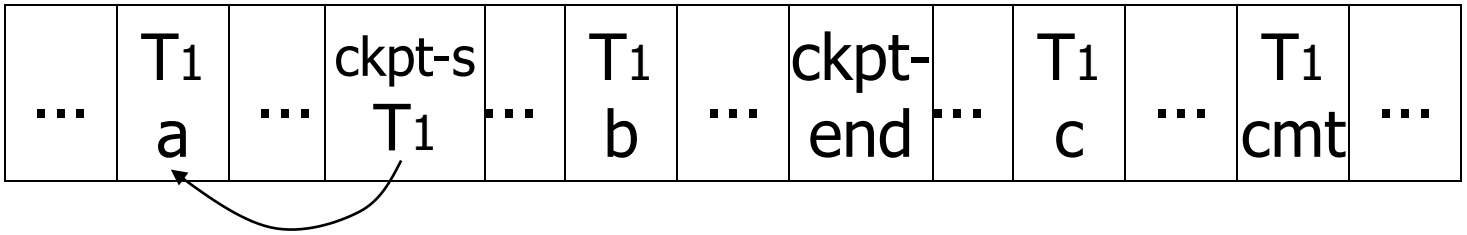
Examples what to do at recovery time?



☒ Undo T_1 (undo a,b)

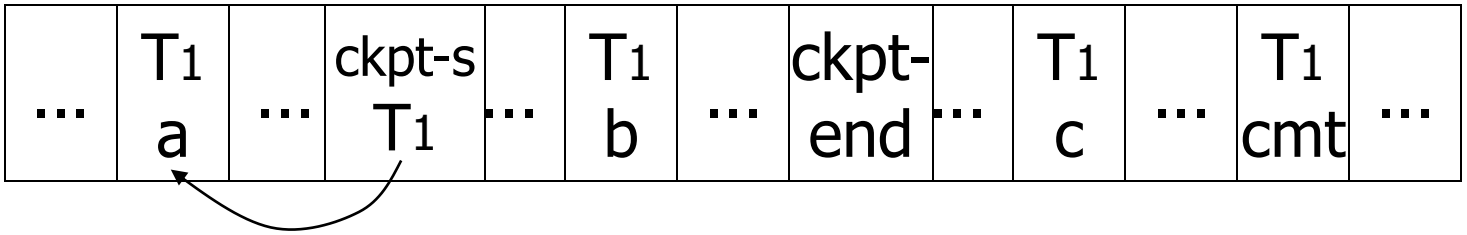
Example

LOG



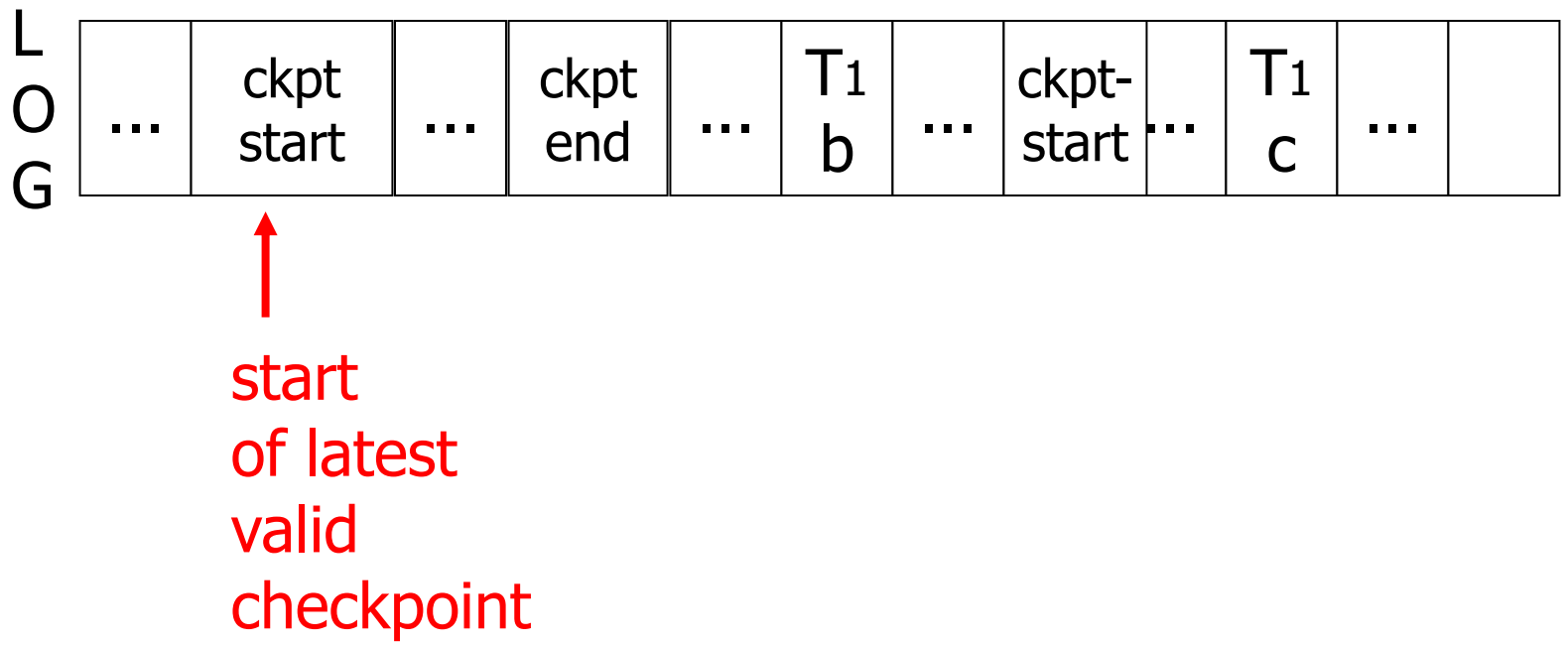
Example

LOG



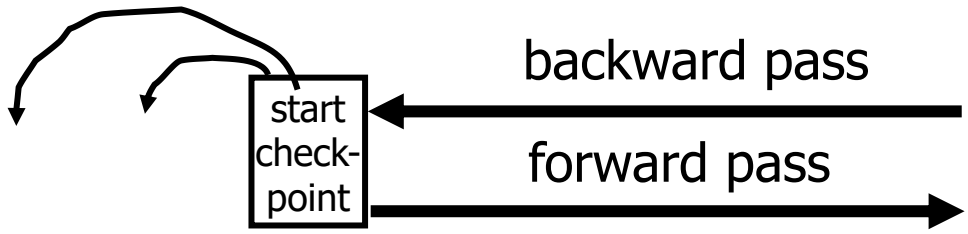
☒ Redo T1: (redo b,c)

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

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

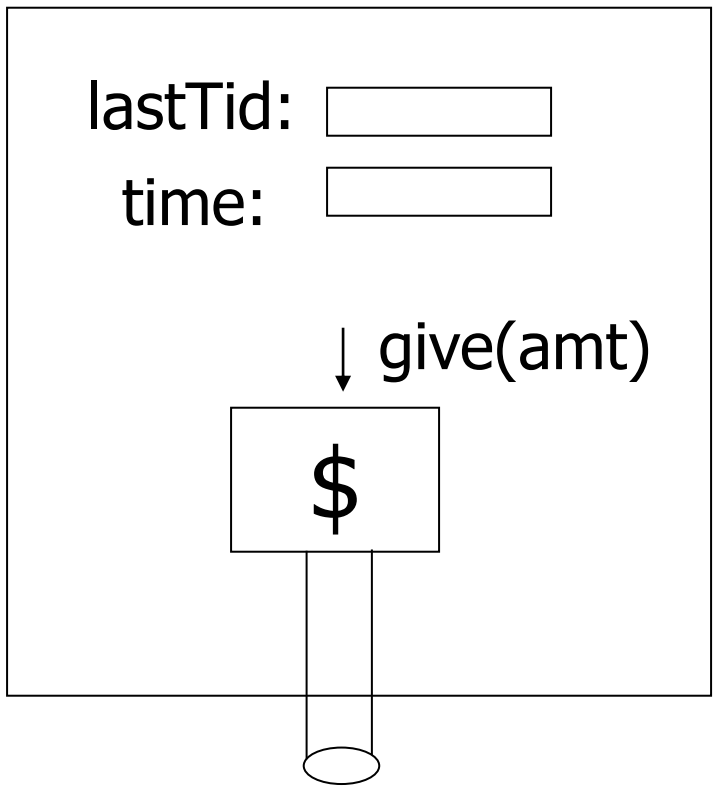
$

Solution

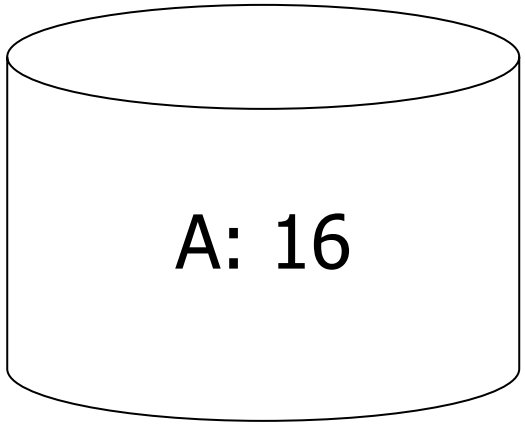
(1) execute real-world actions after commit

(2) try to make idempotent

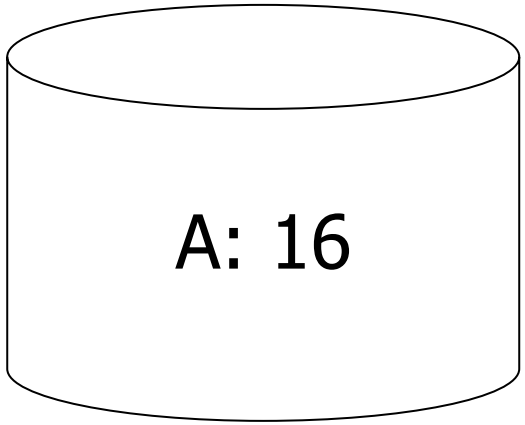
Give$$
(amt, Tid, time)



Media failure (loss of non-volatile storage)



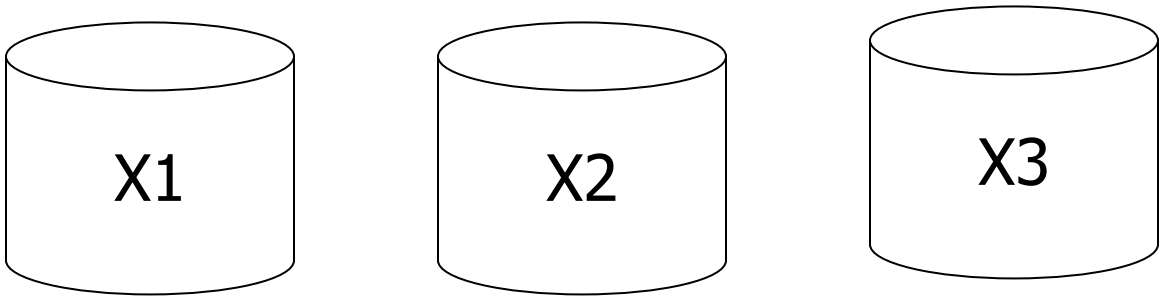
Media failure (loss of non-volatile storage)



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

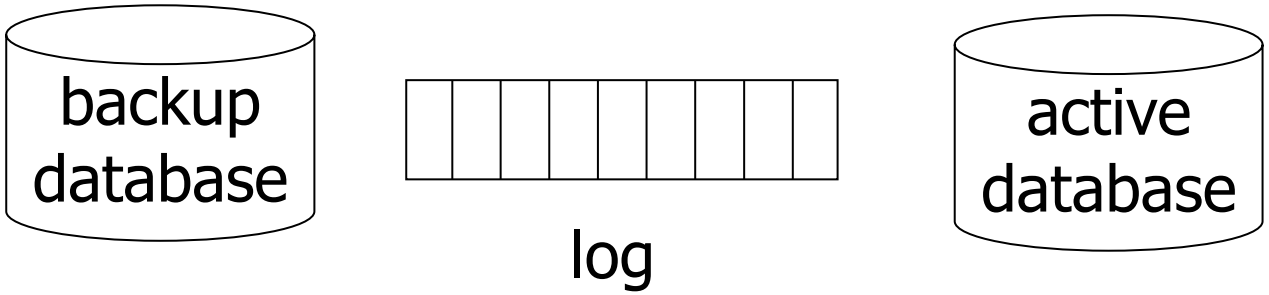


Example #2 Redundant writes, Single reads

- 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



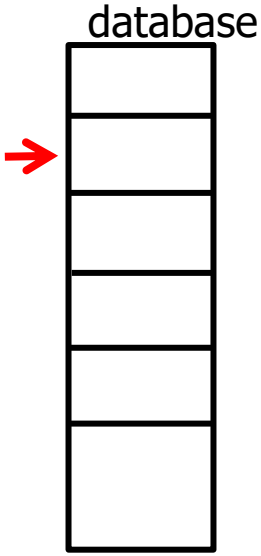
- If active database is lost,
  - restore active database from backup
  - bring up-to-date using redo entries in log

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]

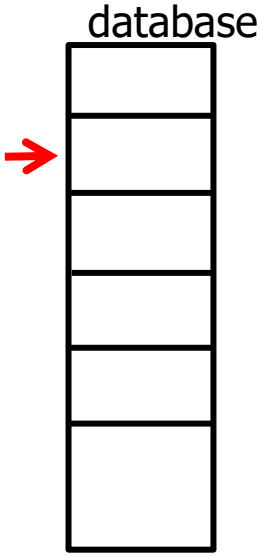


Backup Database

- 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)