Log-Based Recovery Schemes

If you are going to be in the logging business, one of the things that you have to do is to learn about heavy equipment.

Robert VanNatta,
Logging History of Columbia County

Integrity or consistency constraints

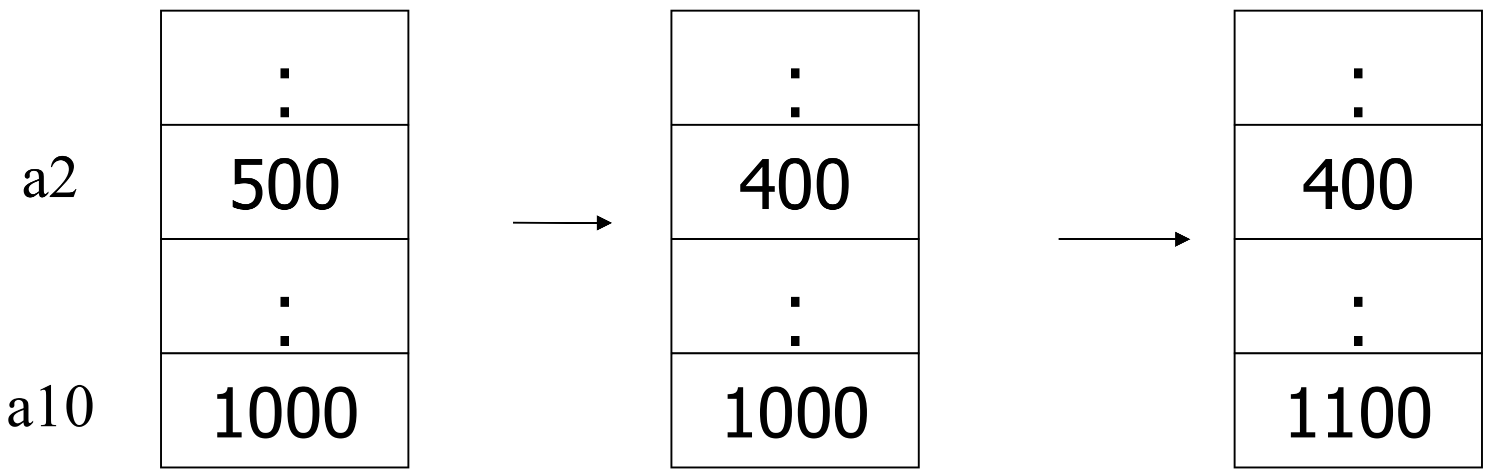
- Predicates/constraints data must satisfy, e.g.
  - x is key of relation R
  - $x \rightarrow y$ holds in R
  - Domain(x) = {Red, Blue, Green}
  - no employee should make more than twice the average salary
- Definitions
  - Consistent state: satisfies all constraints
  - Consistent DB: DB in consistent state

Observation:

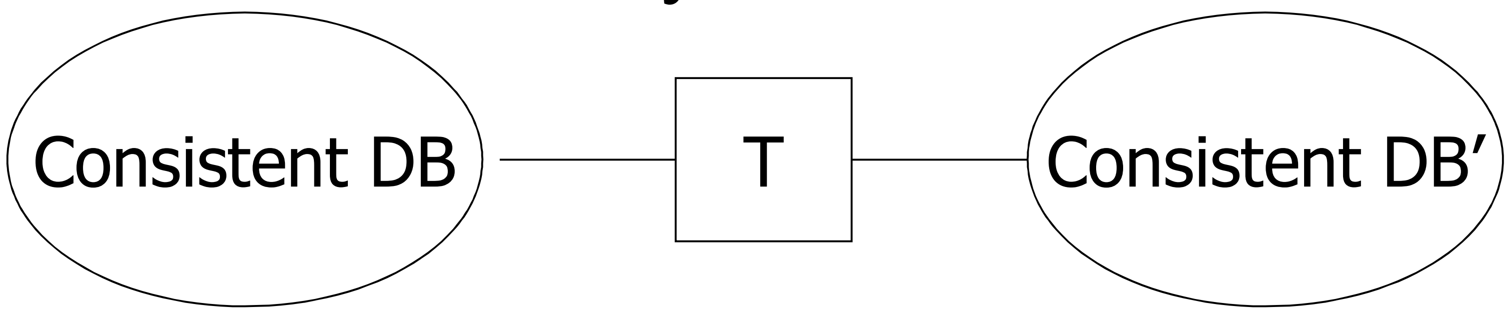
DB *cannot* always be consistent!

Example: Transfer 100 from a2 to a10

$$a2 \leftarrow a2 - 100$$
$$a10 \leftarrow a10 + 100$$



Transaction: collection of actions that preserve consistency



If T starts with a consistent state + T executes in isolation (and absence of errors)

$\Rightarrow$ T leaves a consistent state

Reasons for failures

- Transaction failures
  - Logical errors, deadlocks
- System crash
  - Power failures, operating system bugs etc
  - Memory data lost
- Disk failure
  - Disk Read-Write Head crashes

***STABLE STORAGE*: Data is *never* lost. Can approximate by using RAID and maintaining geographically distant copies of the data**

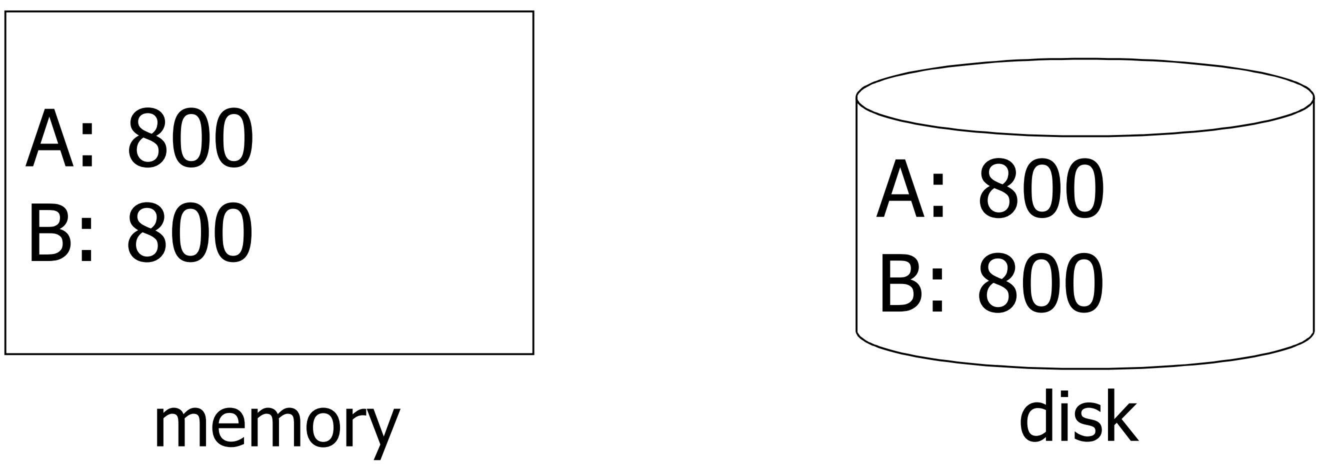
Key problem: Unfinished transaction

Example Transfer fund from A to B

T_1: $A \leftarrow A - 100$

$B \leftarrow B + 100$

T1: Read (A);
$A \leftarrow A-100$
Write (A);
Read (B);
$B \leftarrow B+100$
Write (B);



T1: Read (A);

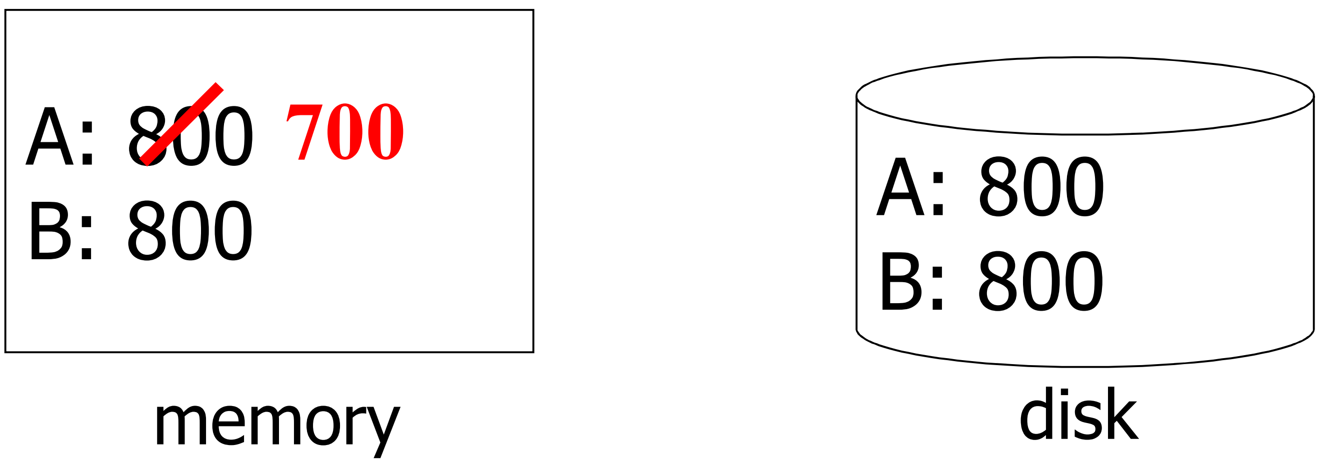
$A \leftarrow A\text{-}100$

Write (A);

Read (B);

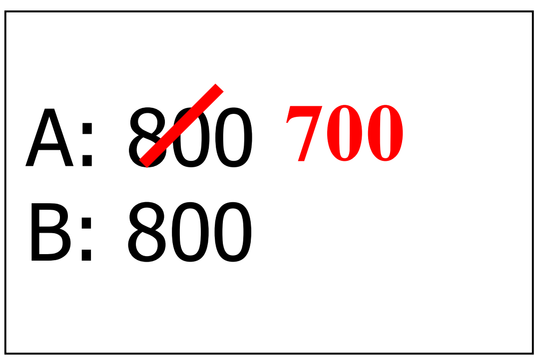
$B \leftarrow B+100$

Write (B);

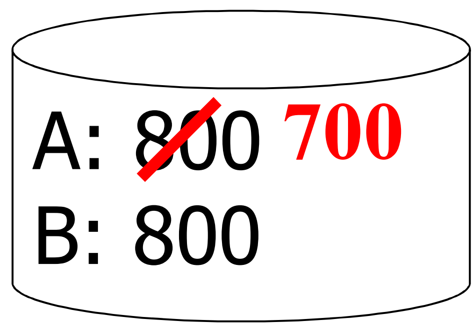


T1: Read (A);
$A \leftarrow A-100$
Write (A);
Read (B);
$B \leftarrow B+100$
Write (B);

Updated A value is written to disk.
This may be triggered ANYTIME
by explicit command or DBMS or OS

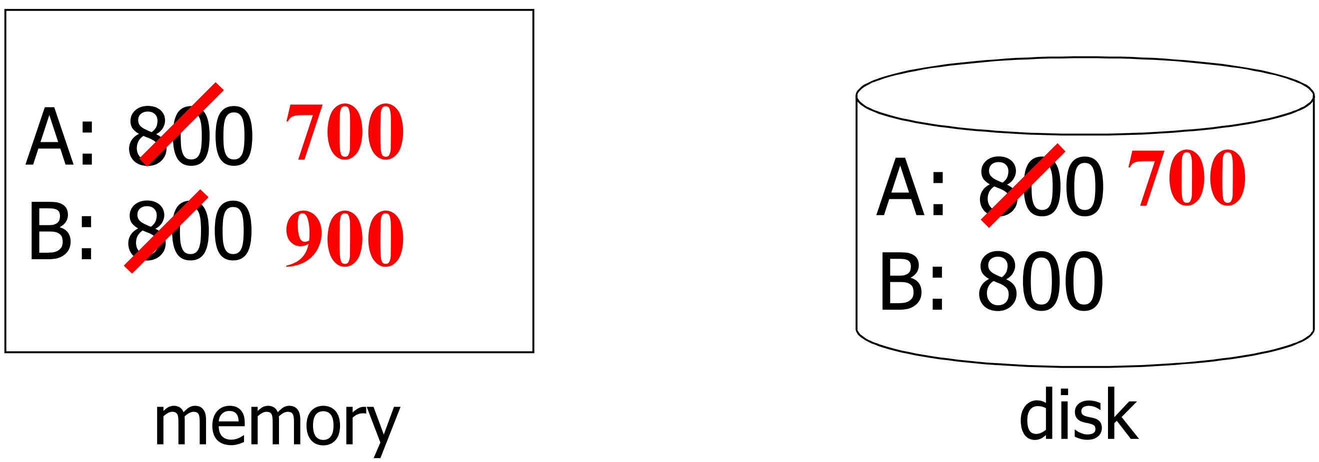


memory



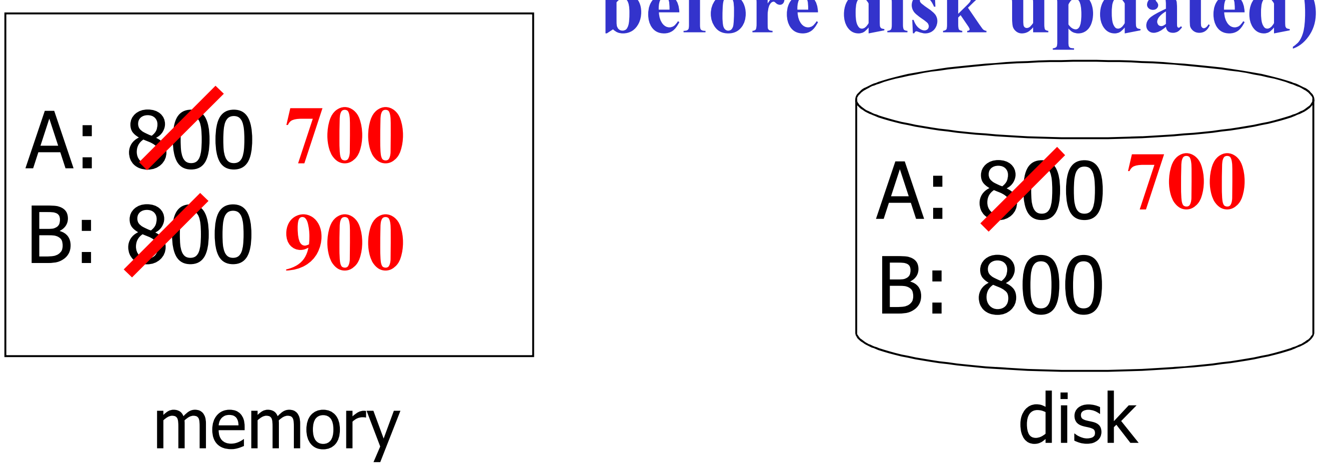
disk

T1: Read (A);
A $\leftarrow$ A-100
Write (A);
Read (B);
B $\leftarrow$ B+100
Write (B);

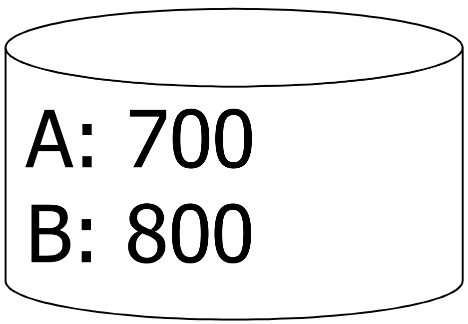


T1: Read (A);
$A \leftarrow A-100$
Write (A);
Read (B);
$B \leftarrow B+100$
Write (B);

Failure before commit (memory content lost before disk updated)!



What is the disk content before the crash?



disk

Disk not updated yet

A = 700; B = 800?

A = 800; B = 700?

Disk fully updated

A = 800; B = 800?

Disk partially updated

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

Recovery Manager

- Recovery Manager guarantees atomicity and durability properties of Xacts
  - Undo: remove effects of aborted Xact to preserve atomicity
  - Redo: re-installing effects of committed Xact for durability
- Processes three operations:
  - Commit(T) - install T's updated "pages" into disk
  - Abort(T) - restore all data that T updated to their prior values
  - Restart - recover database to a consistent state from system failure
    - abort all active Xacts at the time of system failure
    - installs updates of all committed Xacts
- Desirable properties:
  - Add little overhead to the *normal processing* of Xacts
  - Recover quickly from a failure

Interaction Between Recovery and Buffer Managers: Dirty pages in buffer pool

- Can a dirty page updated by Xact T be written to disk before T commits?

Yes -> steal policy -> need to remember the old value (good is that frees up buffer page from doing this again)
No -> no steal policy

- Must all dirty pages that are updated by Xact T be written to disk when T commits?

Yes -> Force policy (continues the contract between user and DBMS that committed transaction is guaranteed
No -> No Force policy -> need to remember the new value

Recovery schemes: Design options

- Four possible design options

	Force	No-force
Steal	Undo & no redo	Undo & redo
No steal	No undo & no redo	No undo & redo

No-steal policy $\Rightarrow$ No undo
Force policy $\Rightarrow$ No redo

Which is the best solution??

Log-based Recovery

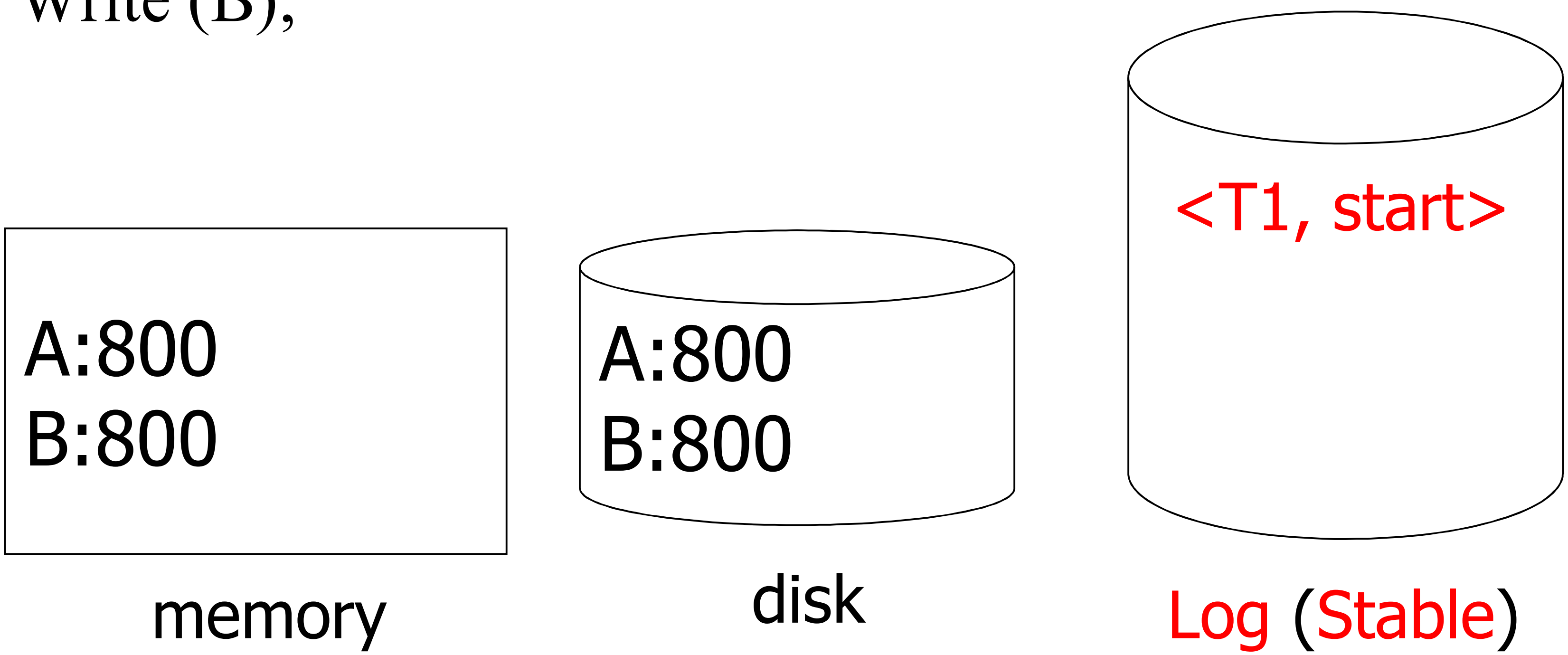
- Log (aka trail/journal): history of actions executed by DBMS
    - Contains a log record for ***each write***, commit, & abort
- Each log record has a unique identifier called **Log Sequence Number (LSN)**
- Log is stored as a sequential file of records in stable storage
    - LSN of log record = address of log record

UNDO & NO REDO

One Solution: Undo logging (Immediate modification/Steal-Force)

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);

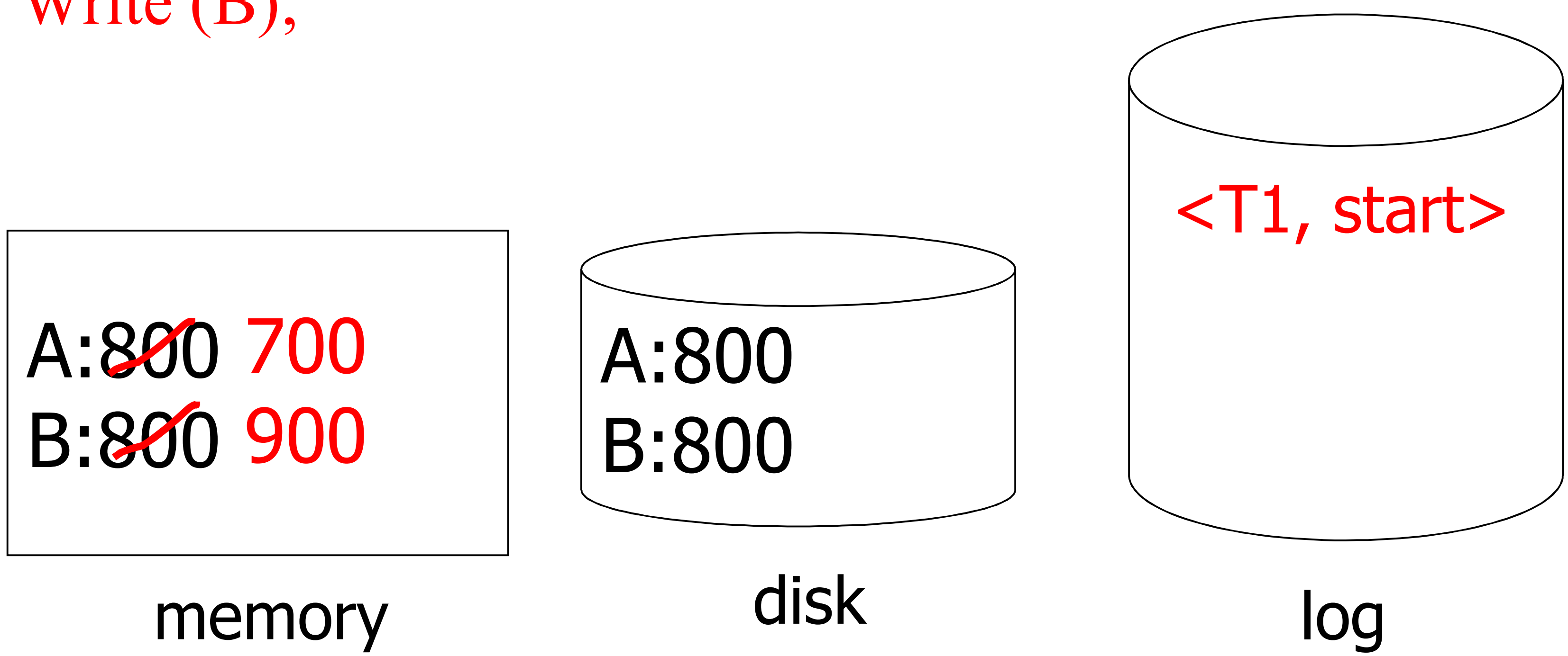
Undo log: <TID, Object, oldValue>
(not showing LSN)



One Solution: Undo logging (Immediate modification)

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);

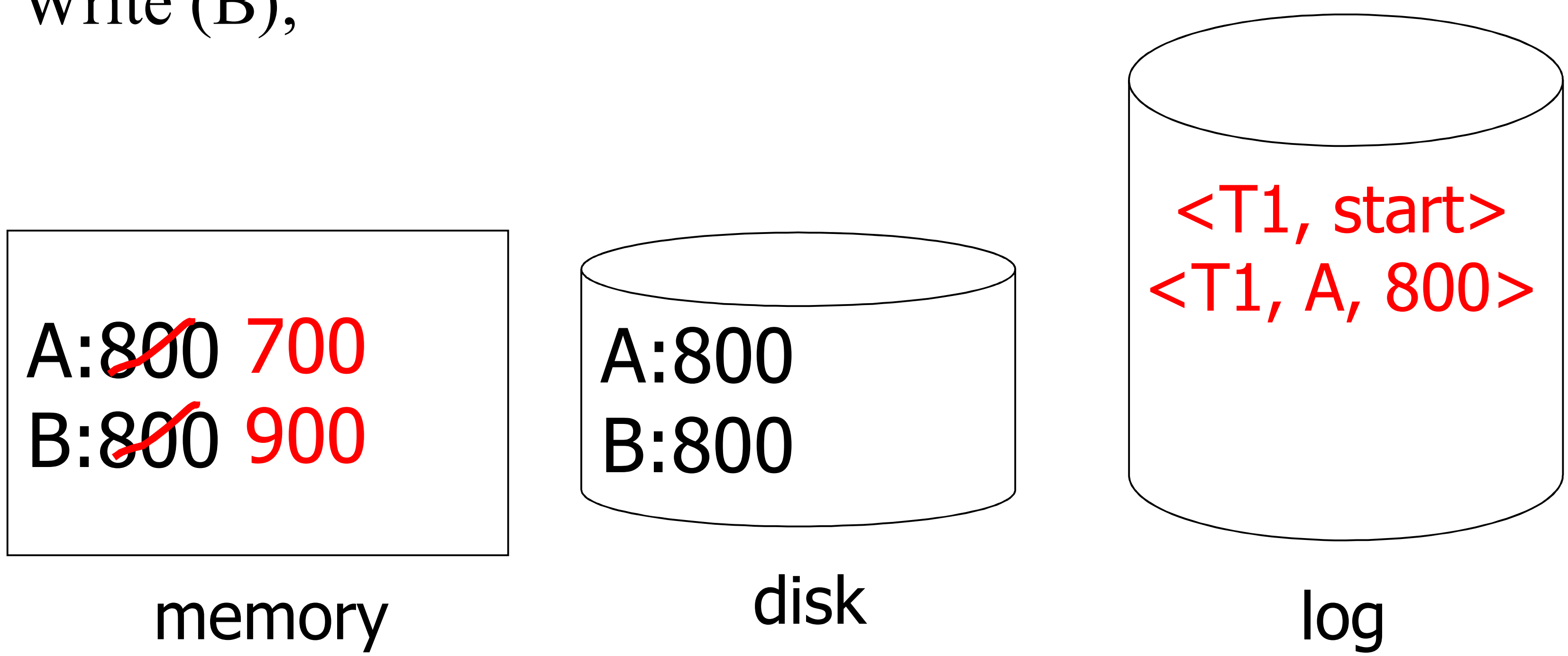
Undo log: <TID, Object, oldValue>



One Solution: Undo logging (Immediate modification)

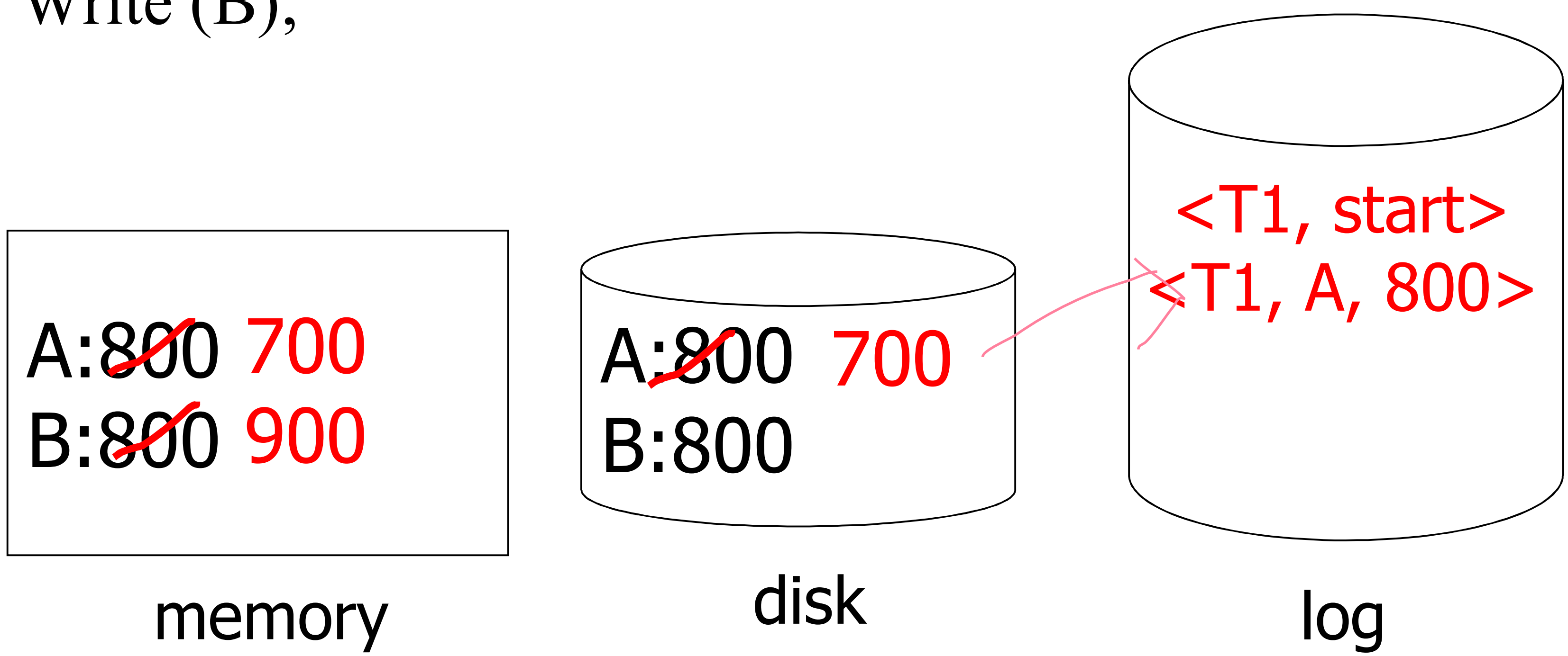
T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);

Undo log: <TID, Object, oldValue>



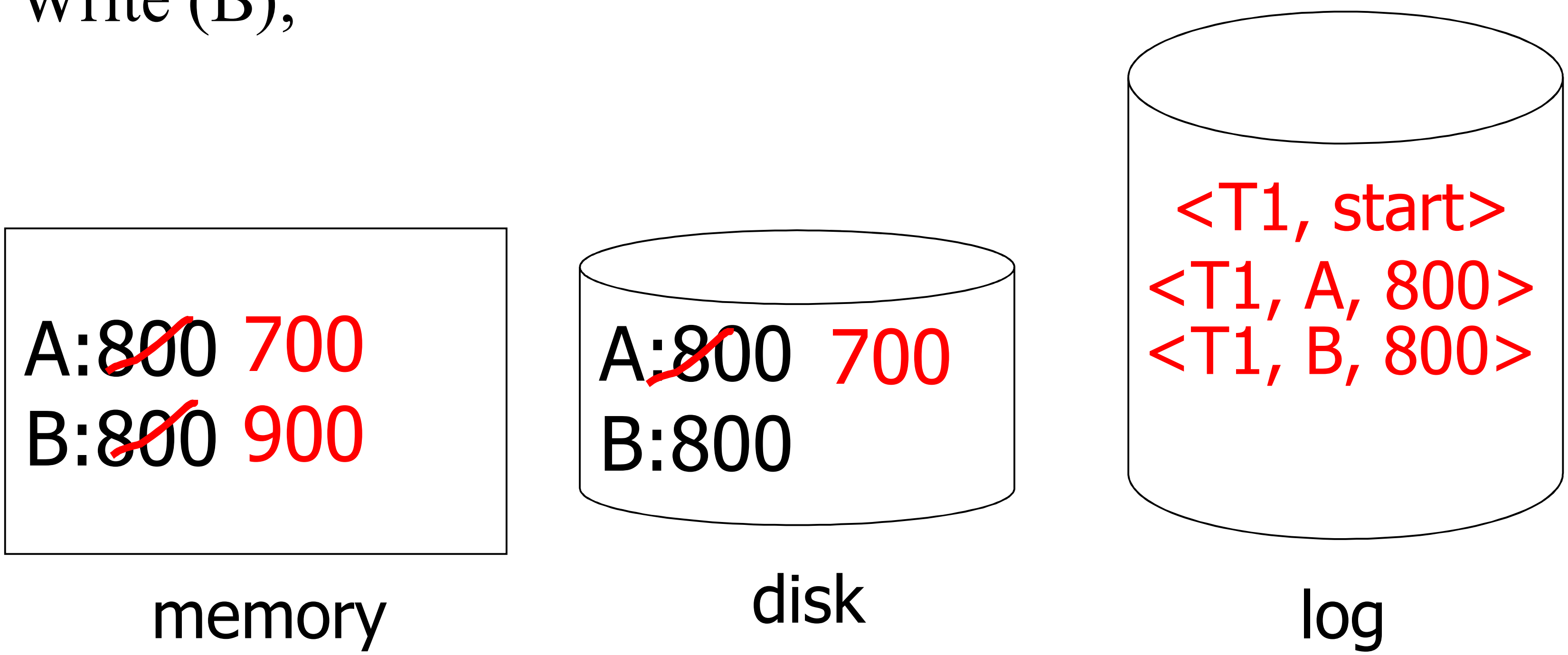
One Solution: Undo logging (Immediate modification)

T1: Read (A); $A \leftarrow A-100$
Write (A);
Read (B); $B \leftarrow B+100$
Write (B);



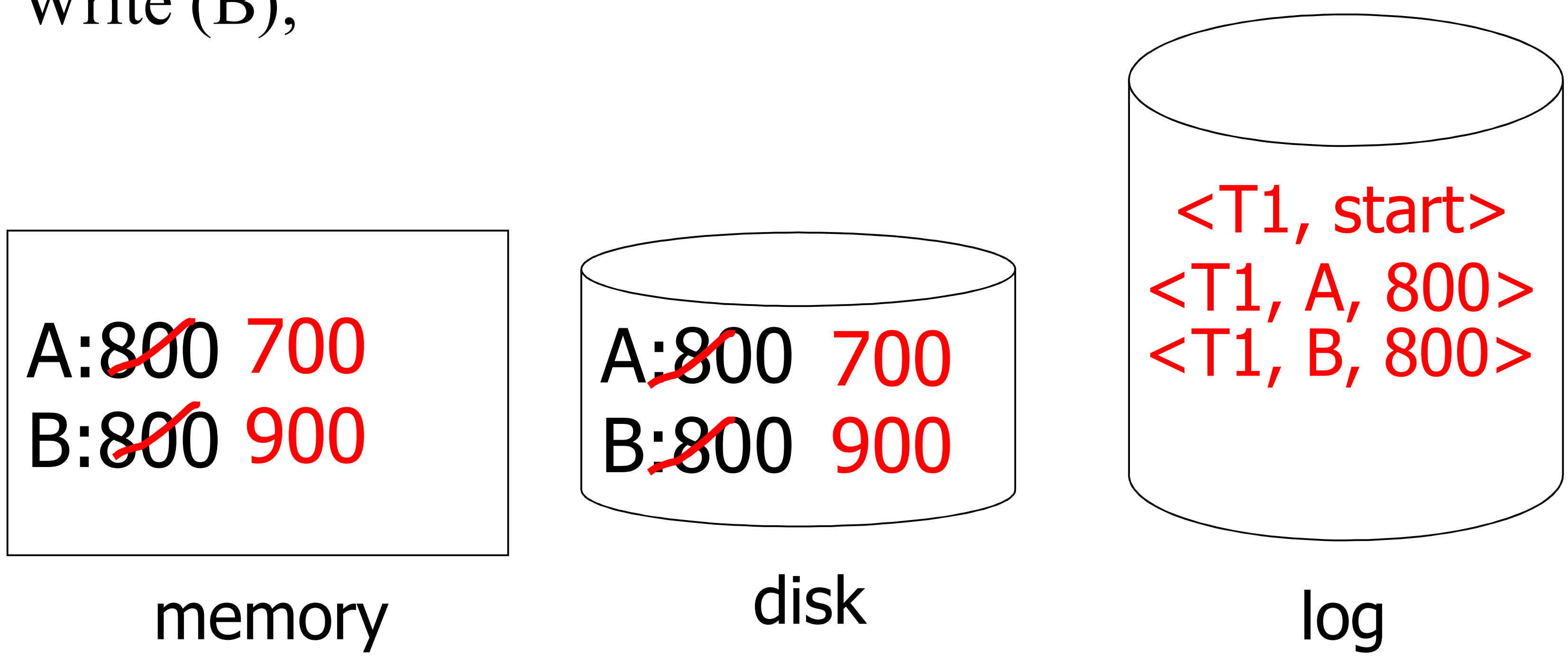
One Solution: Undo logging (Immediate modification)

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



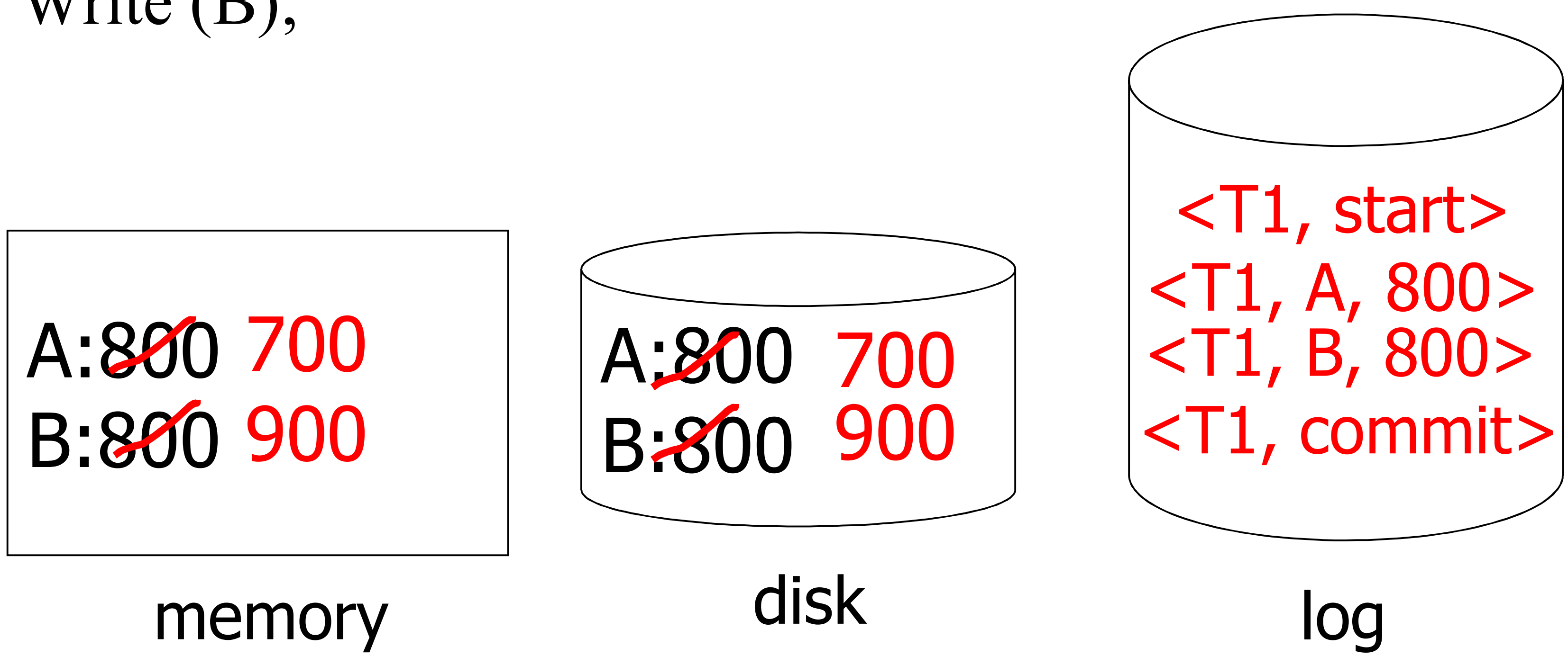
One Solution: Undo logging (Immediate modification)

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



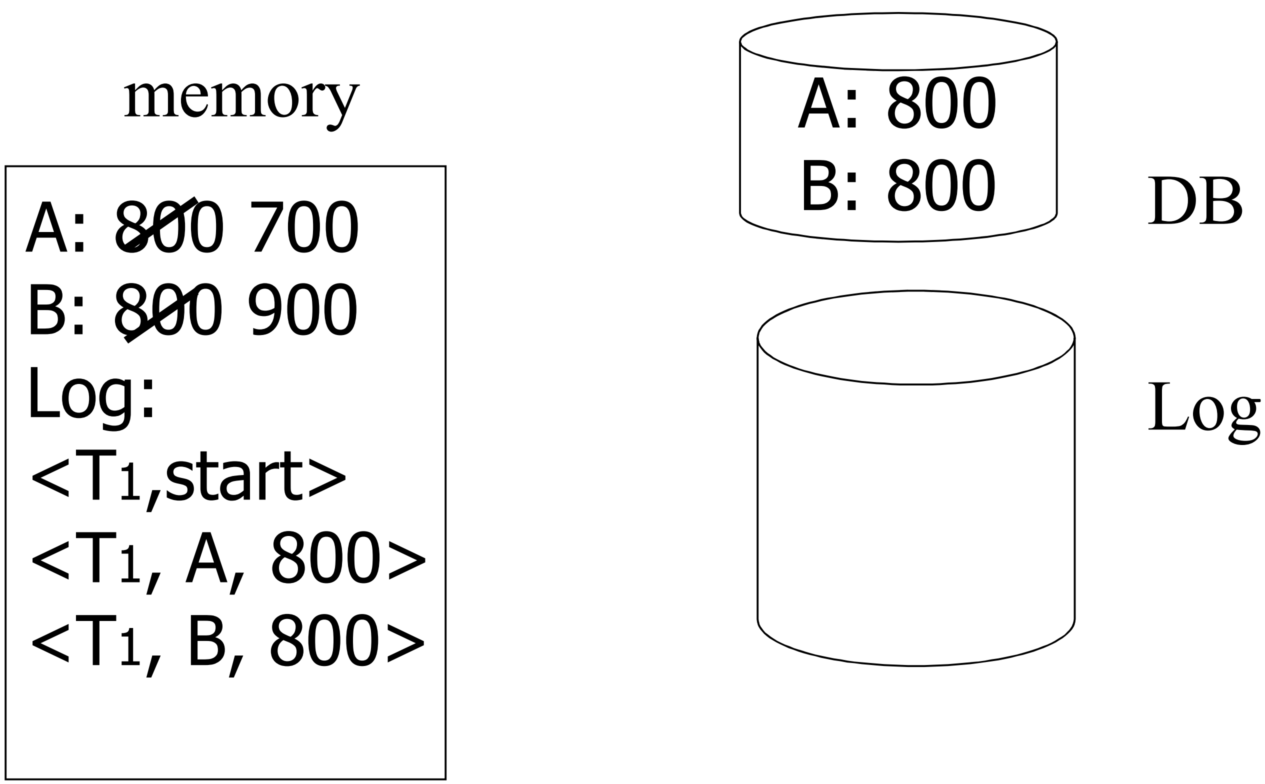
One Solution: Undo logging (Immediate modification)

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



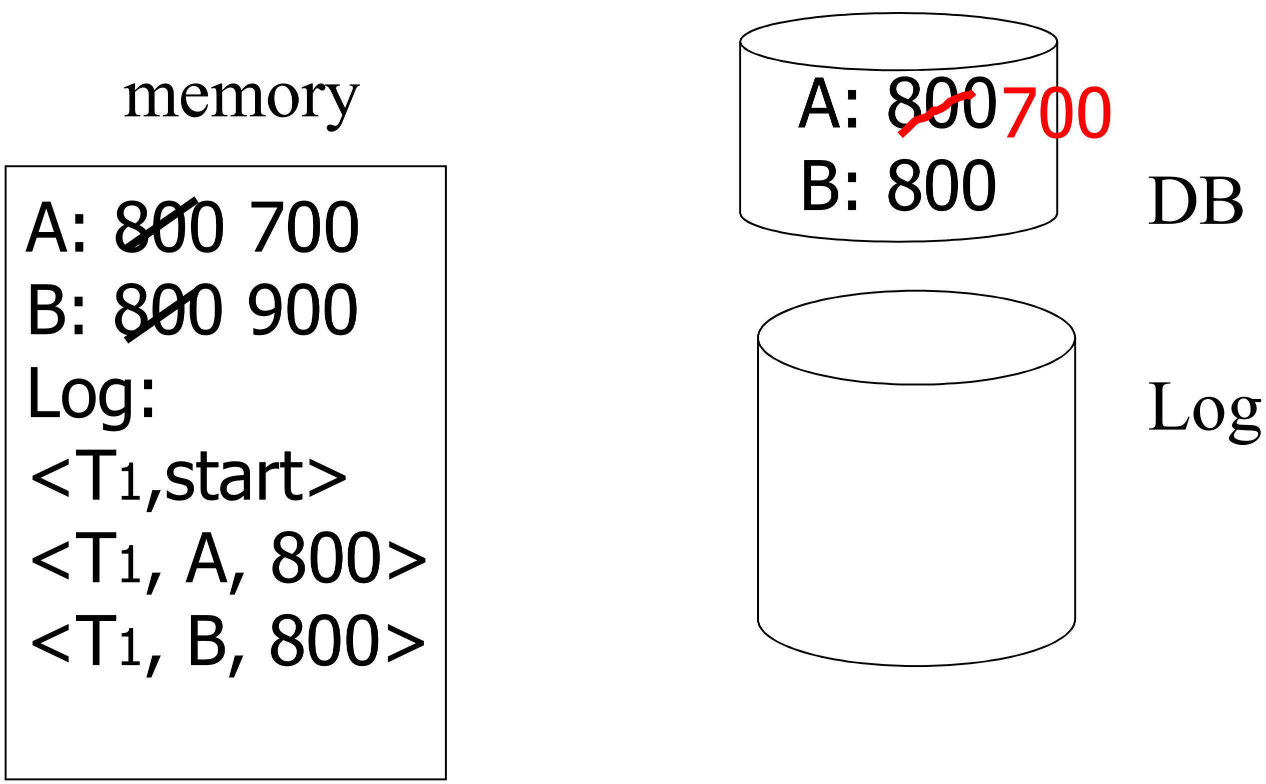
Complications

- Log is first written in memory



Complications

- Log is first written in memory

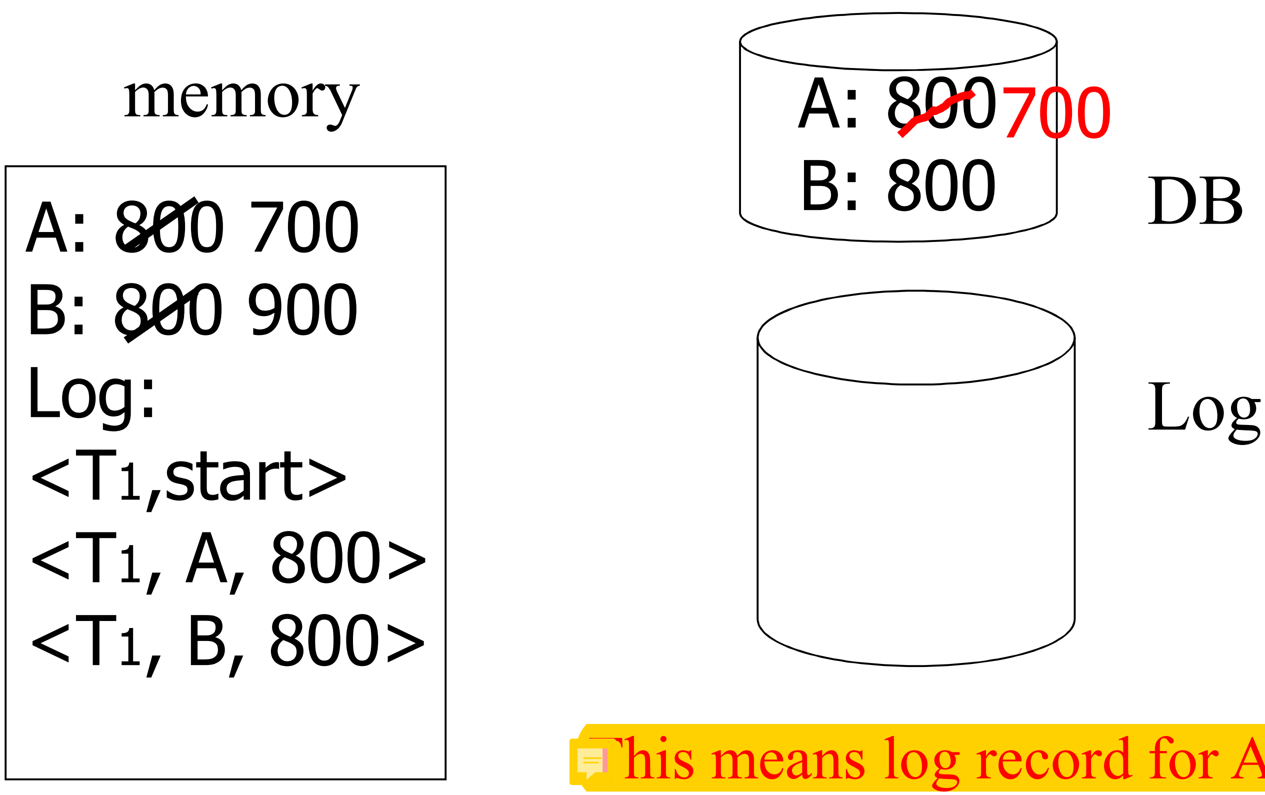


BAD STATE
1

Failure occurs after partial updates on disk but before log is written to disk

Complications

- Log is first written in memory



BAD STATE
1

Failure occurs after partial updates on disk but before log is written to disk

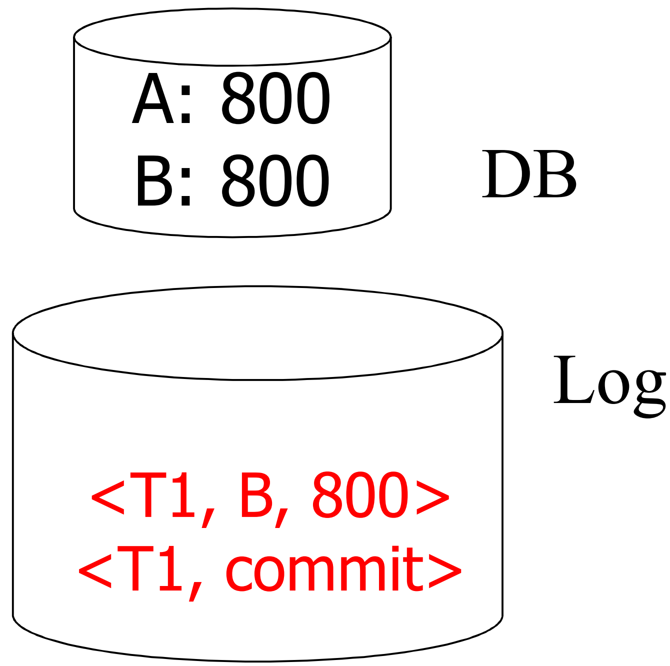
This means log record for A ***must be on log disk*** before A can be updated on data disk (DB)

Complications

- Log is first written in memory
- Updates are not written to disk on every action

memory

A: ~~800~~ 700
B: ~~800~~ 900
Log:
<T1,start>
<T1, A, 800>
<T1, B, 800>
<T1, commit>



Complications

- Log is first written in memory
- Updates are not written to disk on every action

memory

A: ~~800~~ 700
B: ~~800~~ 900
Log:
<T1,start>
<T1, A, 800>
<T1, B, 800>

A: ~~800~~ 700
B: 800

DB

BAD STATE
2

<T1, B, 800>
<T1, commit>

Log

All logs are on disk (***including commit log***) but only partial updates on disk.

Complications

- Log is first written in memory
- Updates are not written to disk on every action

memory

A: ~~800~~ 700
B: ~~800~~ 900
Log:
<T1,start>
<T1, A, 800>
<T1, B, 800>

A: ~~800~~ 700
B: 800

DB

BAD STATE # 2

<T1, B, 800>
<T1, commit>

Log

All logs are on disk (including commit log) but only partial updates on disk.

Before COMMIT log is written to Log, ***all updates must be on disk (DB)***

Undo logging rules

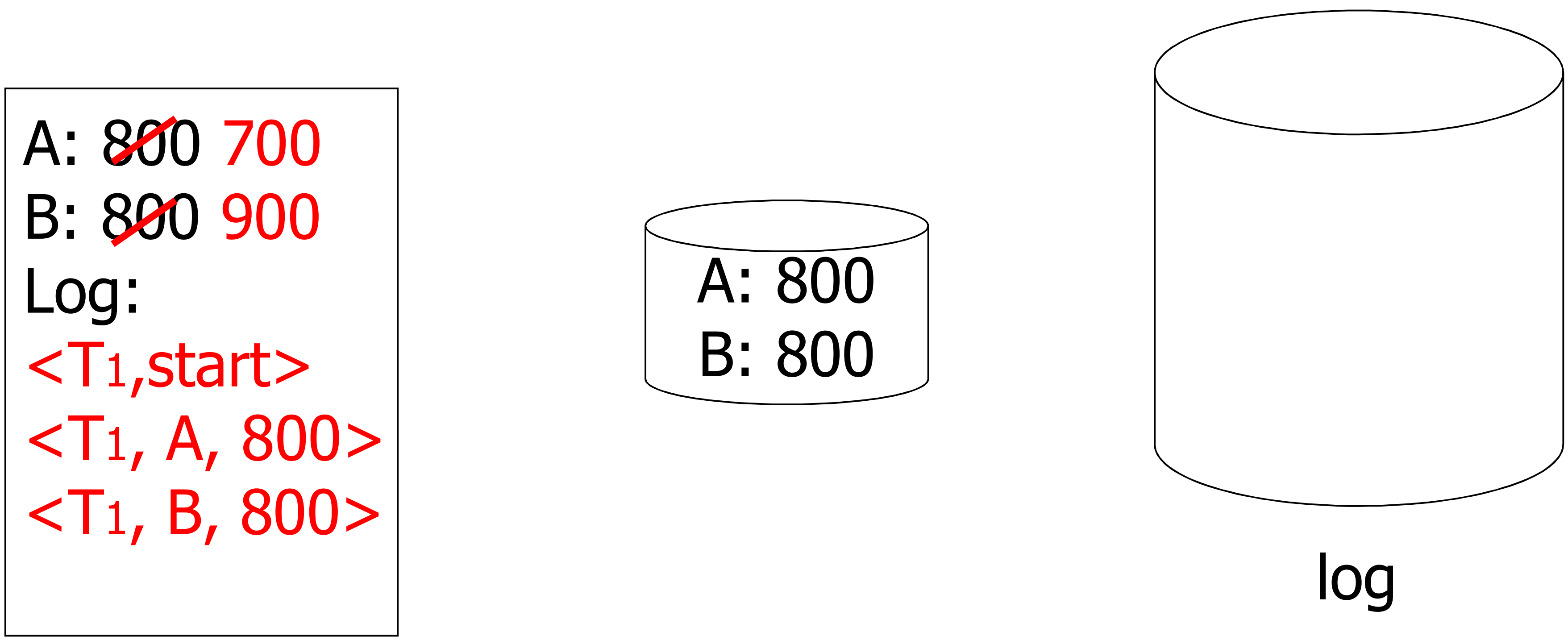
(1) For every action generate undo log record (containing *old* value)

(2) Before *x* is modified on disk, log record 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

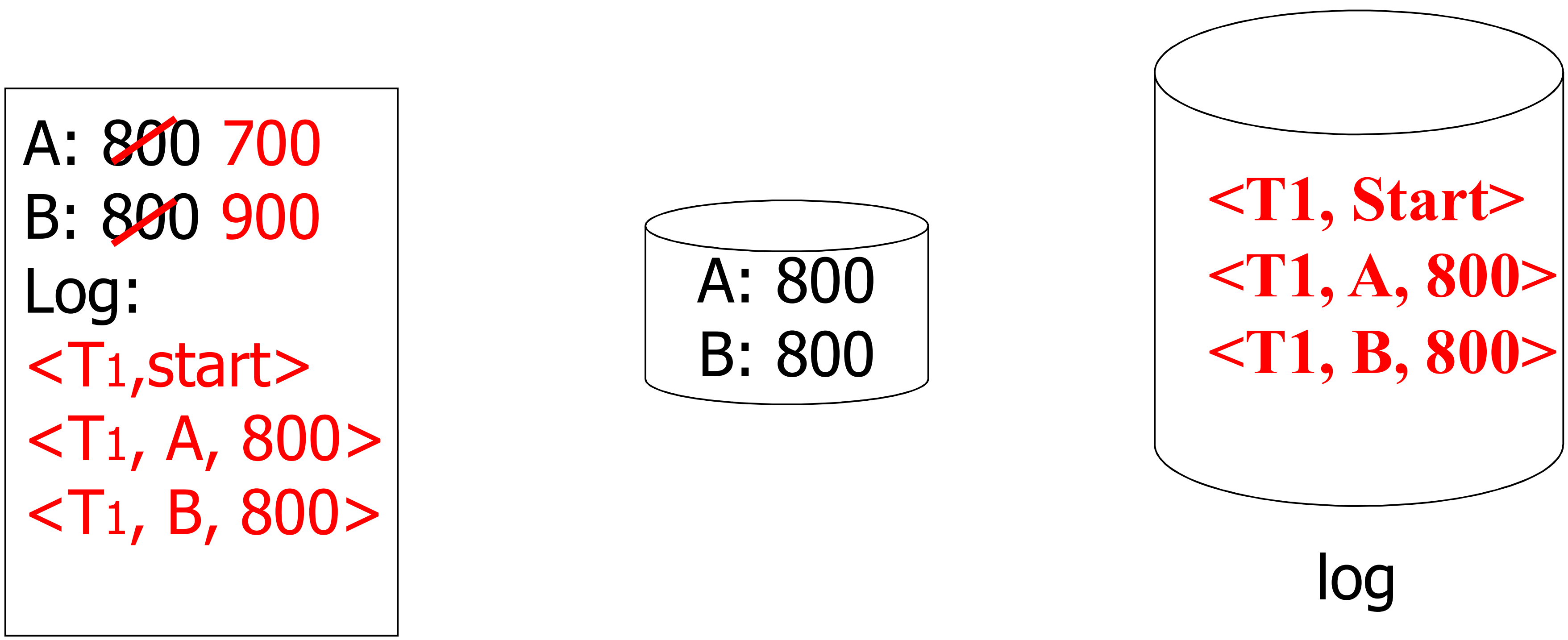
Undo Logging

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



Undo Logging

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



Undo Logging

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);

A: ~~800~~ 700
B: ~~800~~ 900
Log:
<T1,start>
<T1, A, 800>
<T1, B, 800>

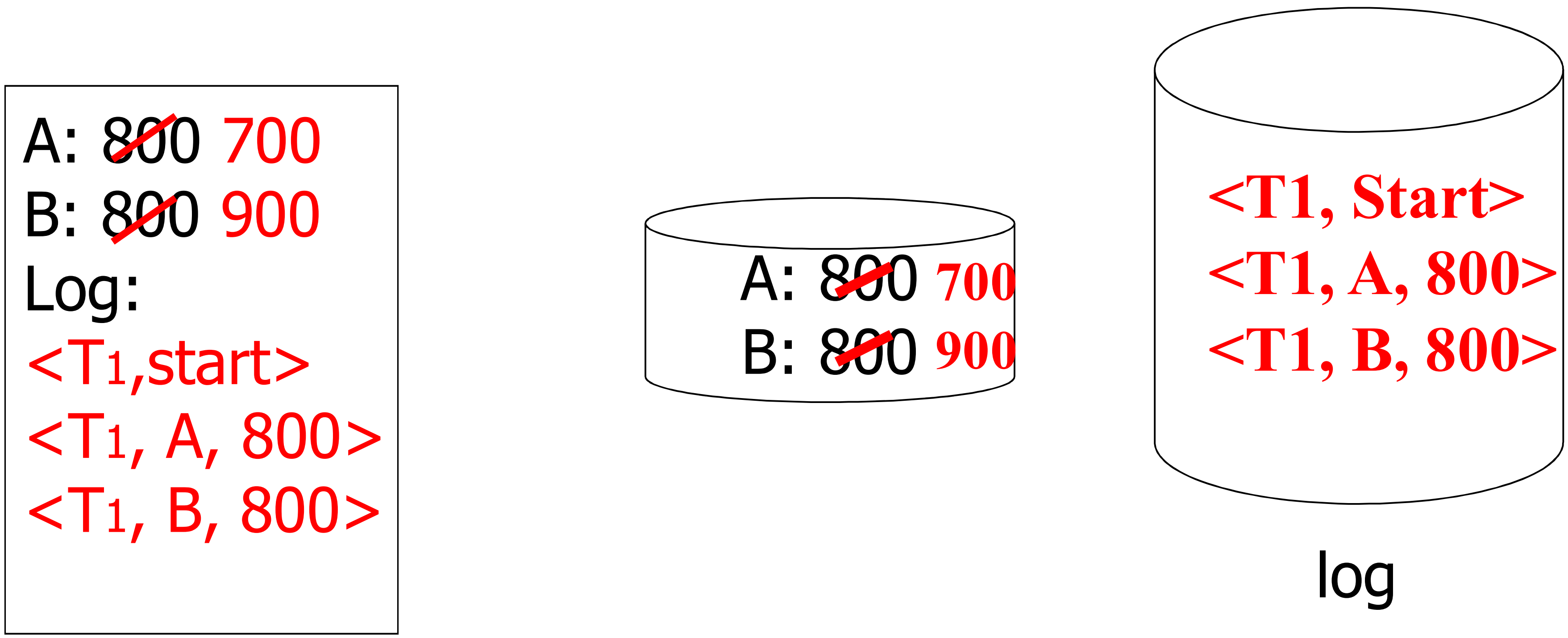
A: ~~800~~ 700
B: 800

<T1, Start>
<T1, A, 800>
<T1, B, 800>

log

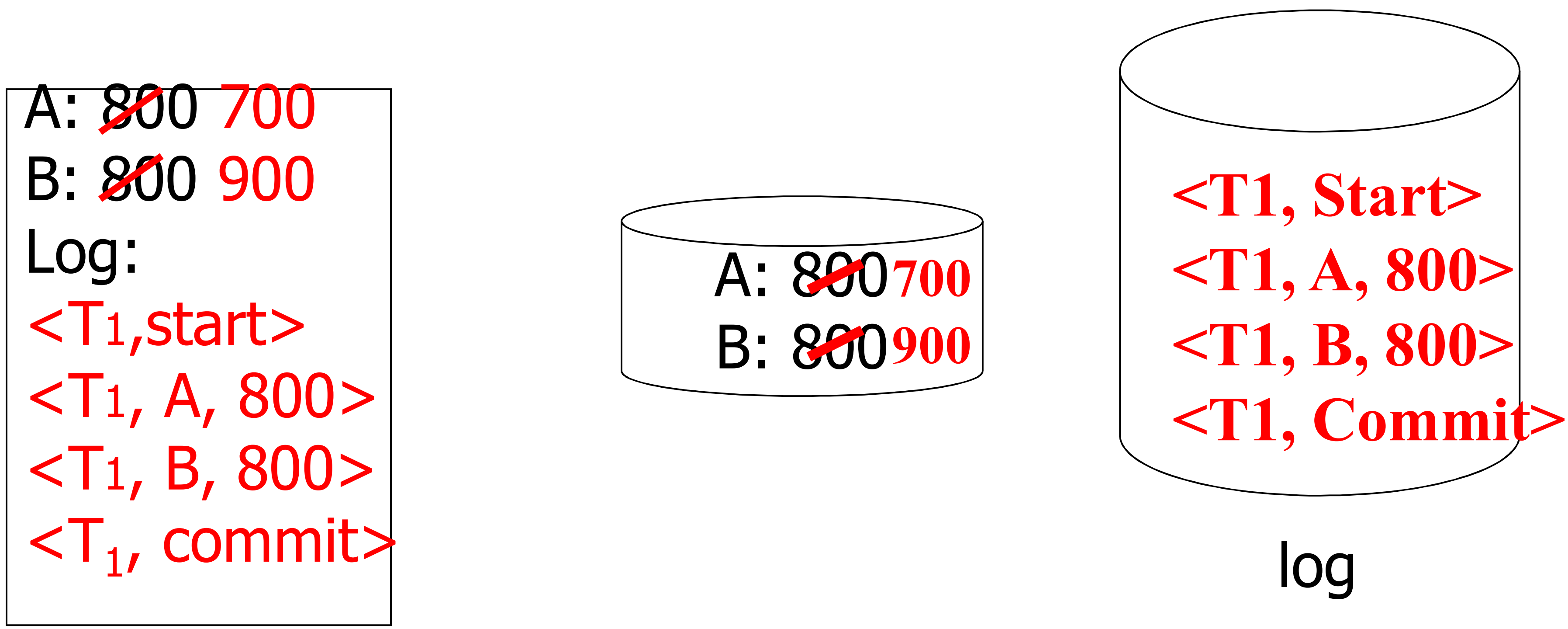
Undo Logging

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



Undo Logging

T1: Read (A); $A \leftarrow A-100$
Write (A);
Read (B); $B \leftarrow B+100$
Write (B);



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

- What about those with Commit/Abort?

(2) For each <Ti, X, v> in log,

in ***reverse order*** (latest $\rightarrow$ earliest) do:

- if Ti $\in$ S then
  - $X \leftarrow v$
  - Update disk

(3) For each Ti $\in$ S do

- write <Ti, abort> to log

What if failure during recovery?

No problem! Undo is idempotent

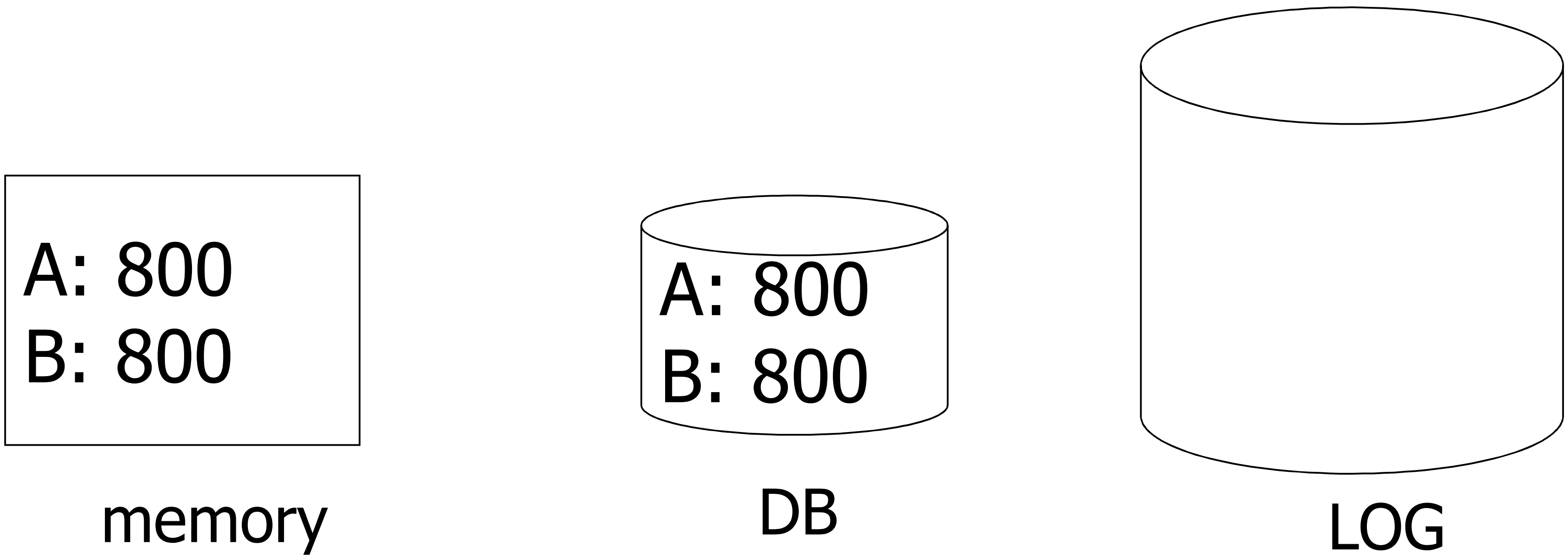
Redo logging (deferred modification/no-steal-no-force)

- In UNDO logging, we remember only the "old" value
- How about remembering only the "new" (updated) values instead?
  - Log record of the form <TID, object, newValue>
- What does this mean?
  - NO old values, so NO updates must be written to disk until a transaction commits!
  - All updates have to be buffered in memory!
    - Can also store dirty pages in temporary disk storage but very inefficient

Redo logging (deferred modification)

T1: Read(A); A ← A-100; write (A);

Read(B); B ← B+100; write (B);

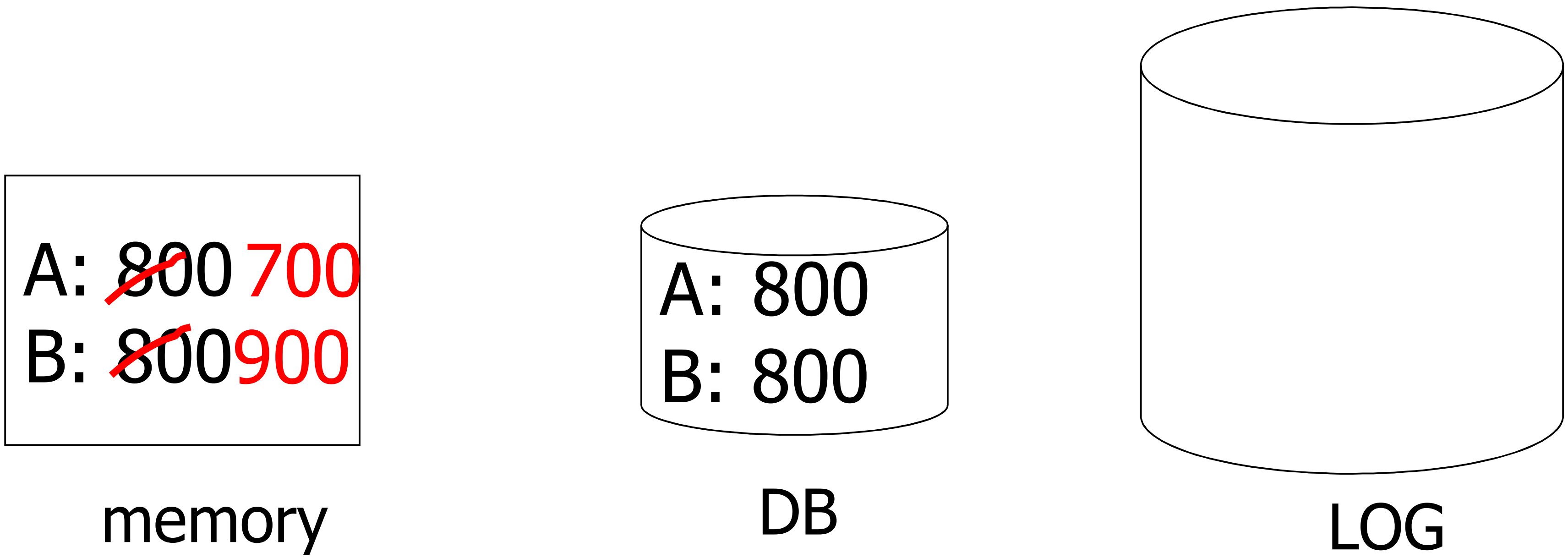


Redo log: <TID, Object, newValue>

Redo logging (deferred modification)

T1: Read(A); A ← A-100; write (A);

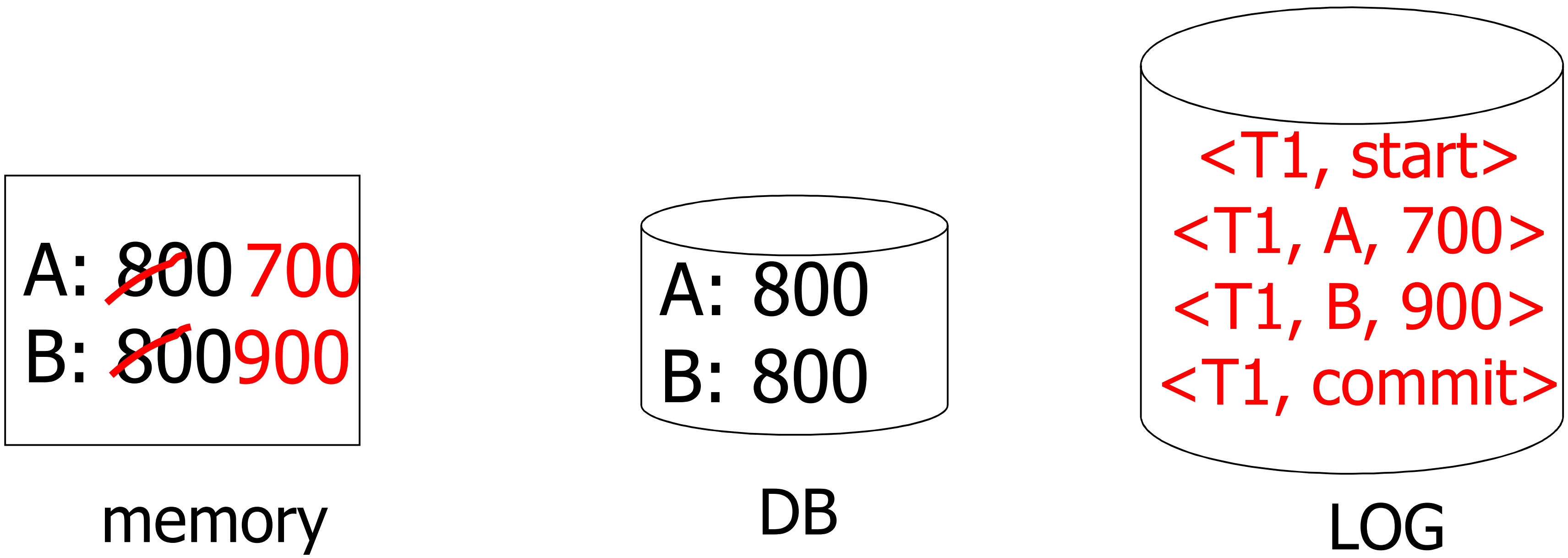
Read(B); B ← B+100; write (B);



Redo logging (deferred modification)

T1: Read(A); A ← A-100; write (A);

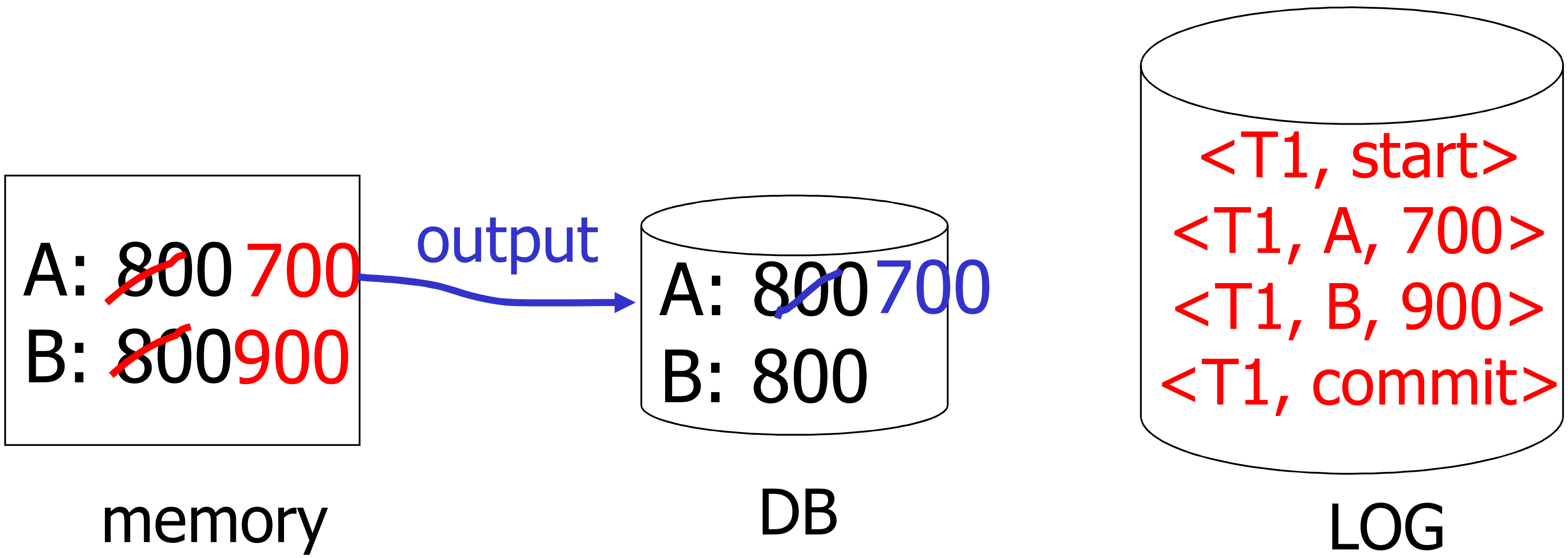
Read(B); B ← B+100; write (B);



Redo logging (deferred modification)

T1: Read(A); A ← A-100; write (A);

Read(B); B ← B+100; write (B);



writing out from memory is optional - since the log already has the information of the committed transaction, it can update from there

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

Recovery rules: Redo logging

(1) Let S = set of transactions with <Ti, commit> in log

(2) For each <Ti, X, v> in log, in **forward order** (earliest $\rightarrow$ latest) do:

- if Ti $\in$ S then $\begin{cases} X \leftarrow v \\ \text{Update X on disk} \end{cases}$

Redo is also idempotent

Key drawbacks:

- *Undo logging (steal/force)*
  - increase the number of disk I/Os
- *Redo logging (no-steal/no-force)*
  - need to keep all modified blocks in memory until commit

Another Solution: undo/redo logging!

Update $\Rightarrow$ <TID, object, newValue, oldValue>
page X

Rules:

1) Page X can be flushed before or after Ti commit
2) Log record flushed before corresponding updated page (WAL)
3) All log records flushed at commit

This is adopted in IBM DB2 – known as the Aries Recovery Manager

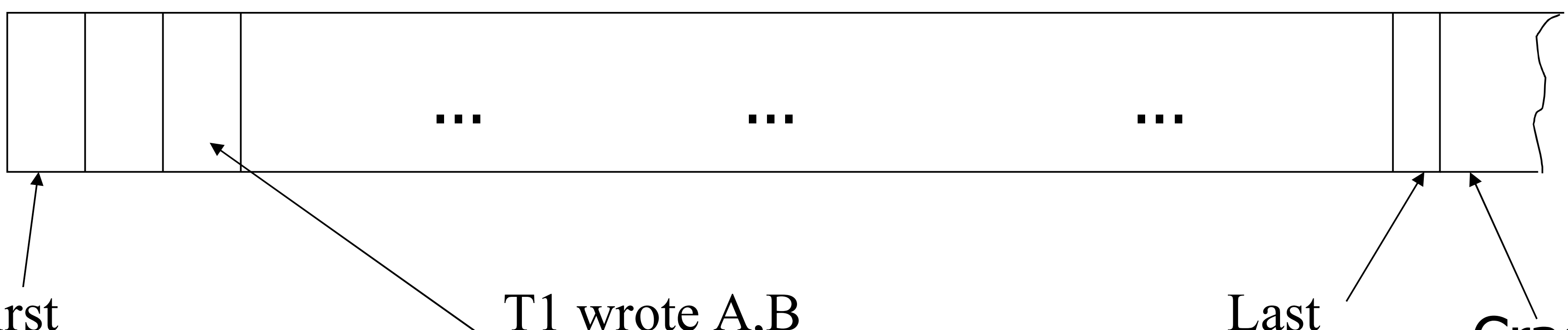
Recovery process:

- Backwards pass
  - construct set S of committed transactions
  - undo actions of transactions not in S
- Forward pass
  - redo actions of S transactions

Checkpointing

Recovery can be very, very SLOW !

Redo log:



First Record (1 year ago)

T1 wrote A,B Committed a year ago

Last Record

Crash

--> STILL, Need to redo after crash!!

Solution: Checkpoint (simple version)

Periodically:

(1) Do not accept new transactions

(2) Wait until all (active) transactions finish

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

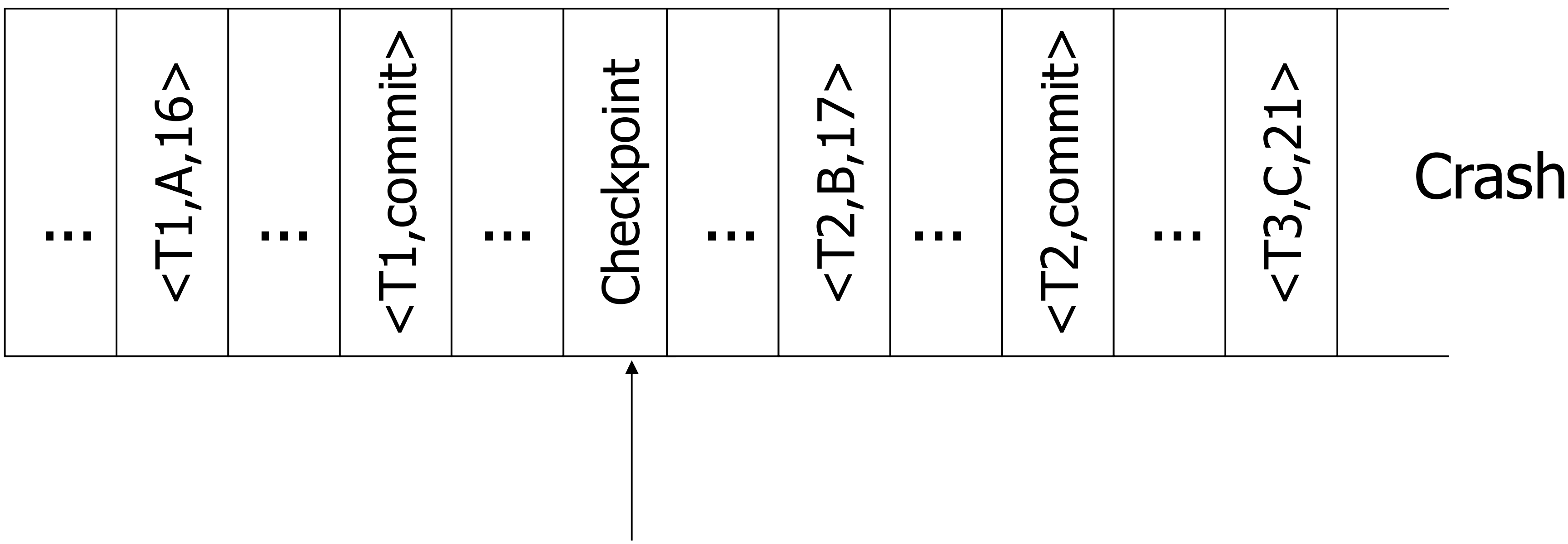
(4) Flush all buffers to disk (DB)

(5) Write “checkpoint” record on disk (log)

(6) Resume transaction processing

Example: what to do at recovery?

Redo log (disk):



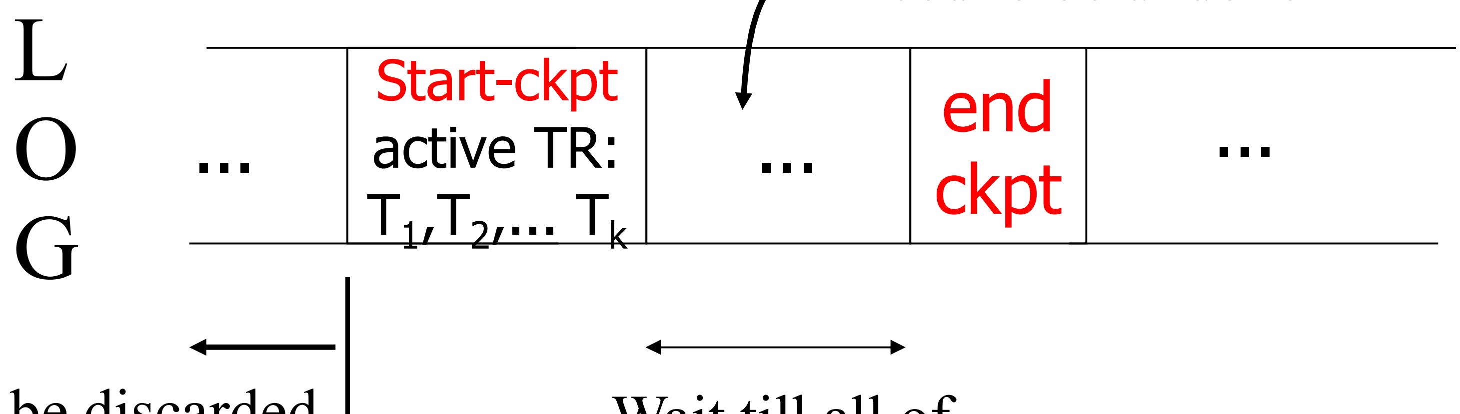
No need to examine log records before the most recent Checkpoint

Non-quiescent Checkpoint

- Processing continues in the midst of checkpointing
- No blocking of newly arrived transactions

Non-quiescent Checkpoint: Undo Log

Only uncommitted transactions started from this point onwards need to be undone



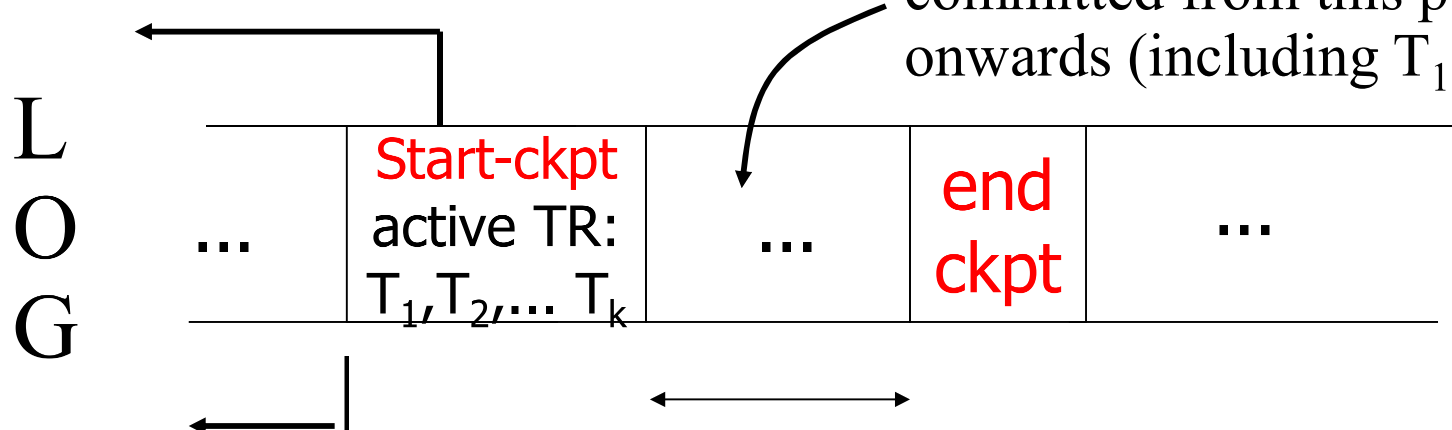
Can be discarded when checkpoint completes successfully

Wait till all of $T_1, \ldots, T_k$ commit or abort, but do not prohibit other transactions from starting

Non-quiescent Checkpoint: Redo Log

We do not need to look further back than the earliest of the Start of the active transactions $T_1 \ldots T_k$

Need to redo transactions that committed from this point onwards (including $T_1 \ldots T_k$)



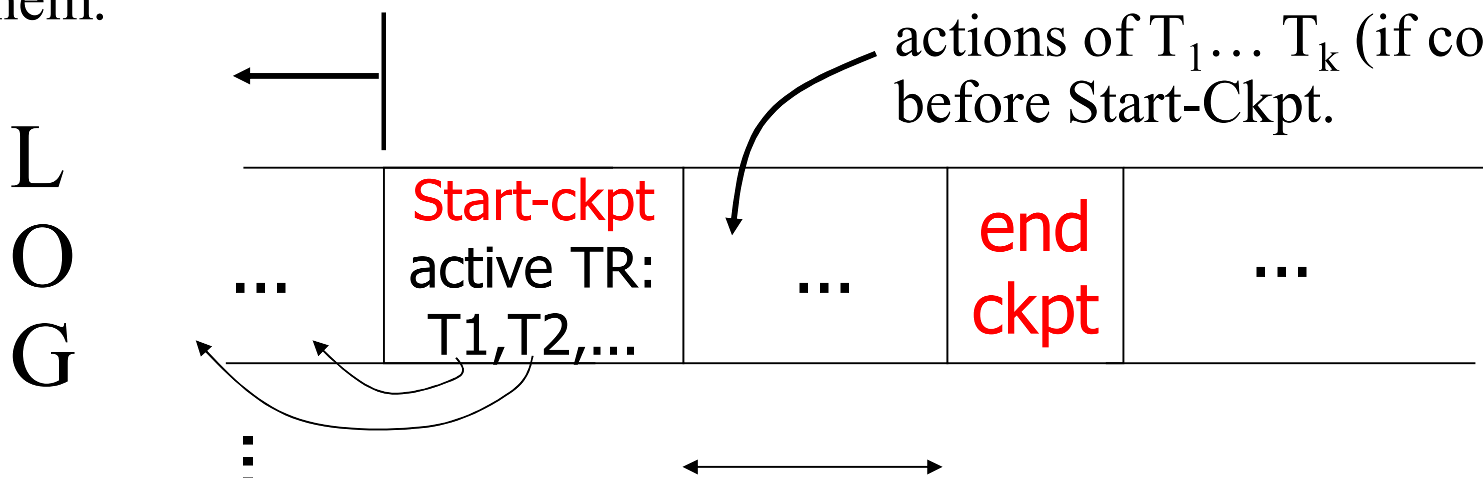
Committed transactions' updates all reflected on disk. So, no need to redo them.

Write to disk all database elements that were written to buffers but not yet to disk by transactions that had already committed when the START CKPT record was written to the log

Non-quiesce checkpoint (Undo/Redo logging)

Committed transactions' updates all reflected on disk. So, no need to redo them.

Need to redo actions of transactions committed from this point onwards. No need to redo actions of $T_1 \ldots T_k$ (if committed) before Start-Ckpt.



for Undo (because dirty pages are flushed)

All dirty buffer pages prior to Start-Ckpt are flushed without disrupting runtime operations

Examples: what to do at recovery time?

no T1 commit

L
O
G

…	T_1,- a	…	Ckpt T_1	…	Ckpt end	…	T_1- b	

- Undo T1 (undo a,b)

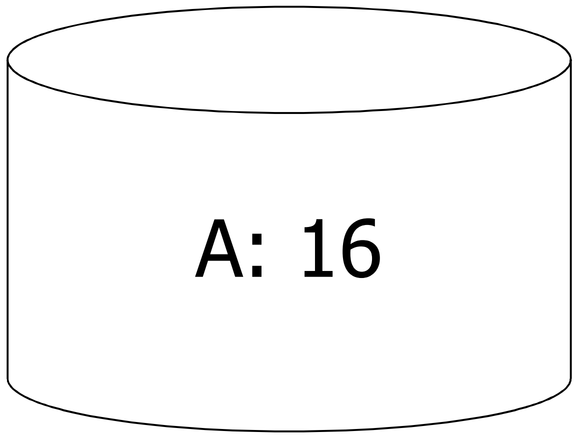
Example

L
O
G

…	T_1 a	…	ckpt-s T_1	…	T_1 b	…	ckpt-end	…	T_1 c	…	T_1 cmt	…

- Redo T1: (redo b,c)

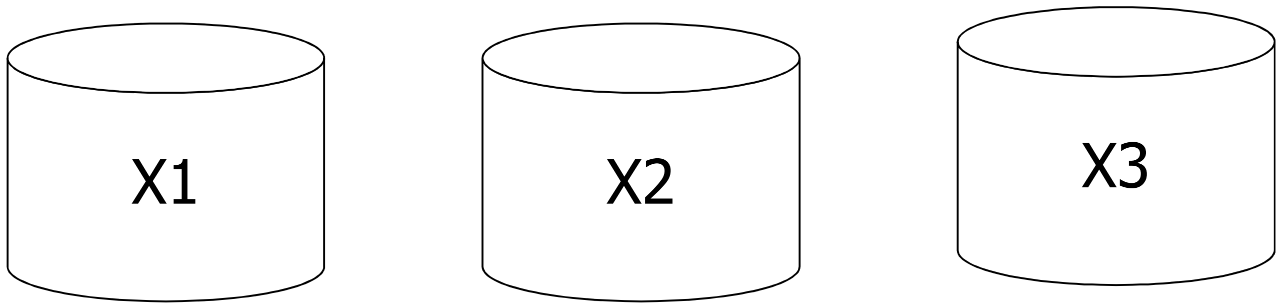
Media failure
(Loss of non-volatile storage)



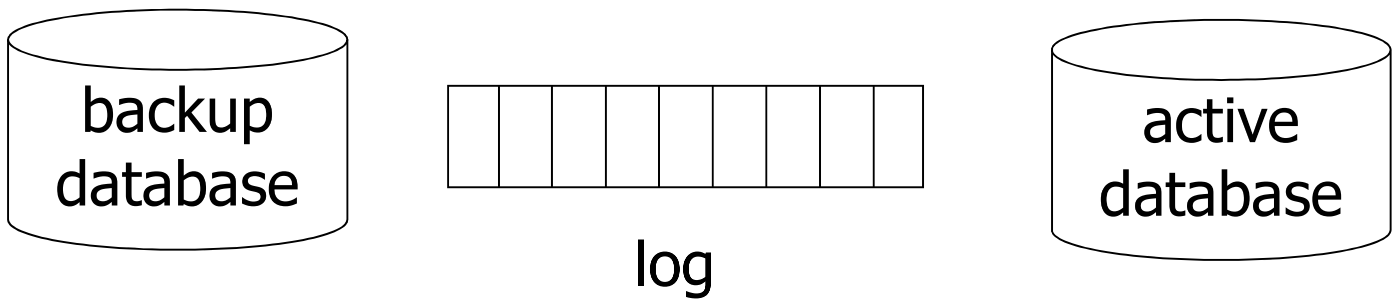
<u>Solution:</u> Make copies of data!

Triple Modular Redundancy

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



DB Dump + Log



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

Summary

- Consistency of data
- Two sources of problems:
  - Failures
    - Logging
    - Redundancy
  - Data sharing
    - Concurrency Control
- Log-based recovery mechanisms
  - Undo, Redo, Undo/Redo
  - What about No Undo/No Redo???