Lecture 05.02

Recovery from failures

By Marina Barsky
Winter 2017, University of Toronto

Definition:

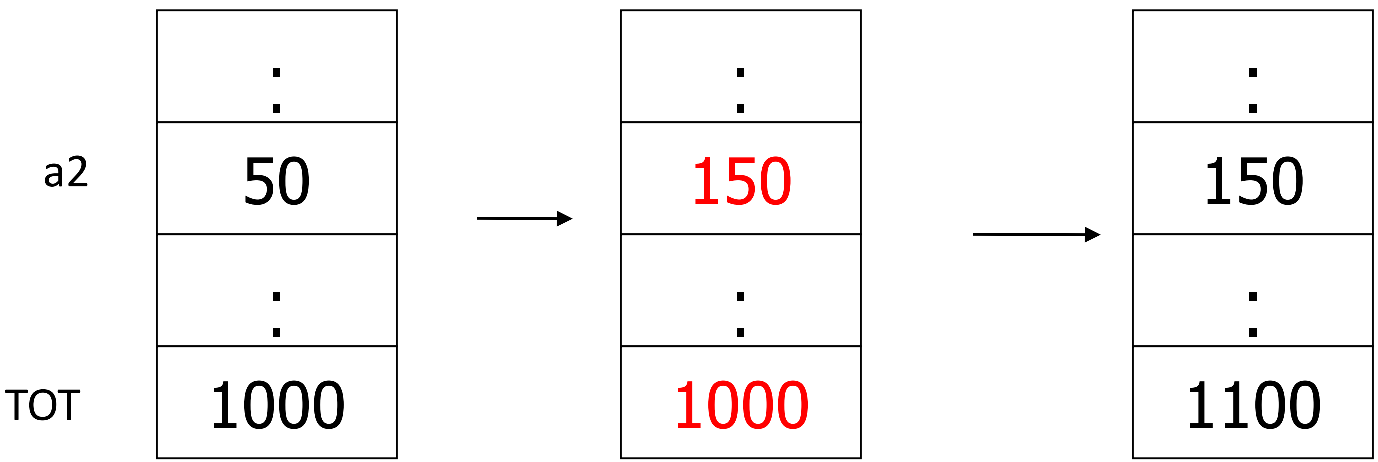
- *Consistent state*: all constraints are satisfied
- *Consistent DB*: DB in consistent state

Observation:
DB cannot be consistent at all times

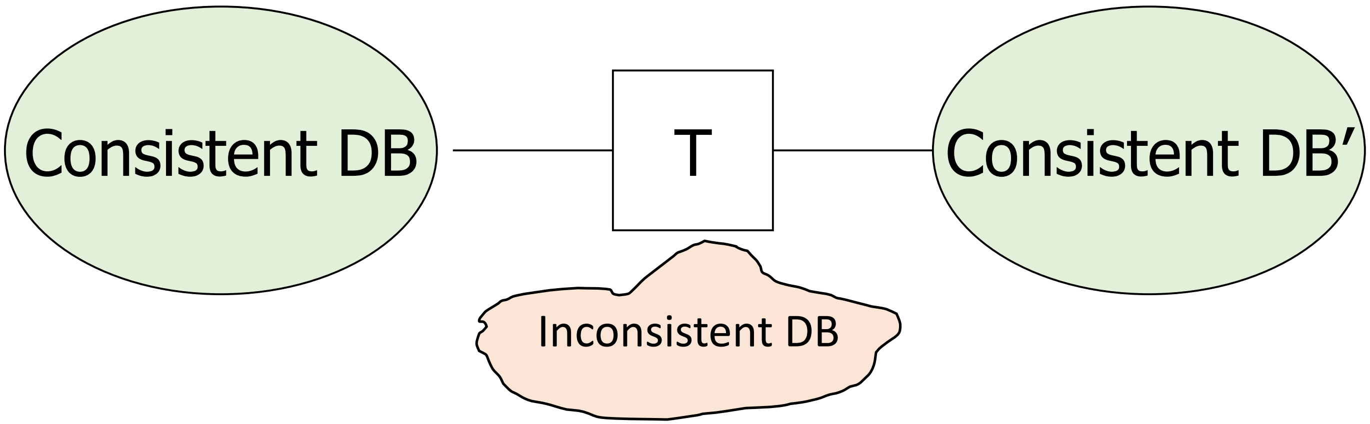
Example: a1 + a2 +…. an = TOT (constraint)

Deposit $100 in a2: $a2 \leftarrow a2 + 100$

$TOT \leftarrow TOT + 100$



Transaction: collection of actions that bring DB from one consistent state to another



If T starts with consistent state + T executes in isolation

⇒T leaves in a consistent state

We learned how to ensure that concurrent (interleaving) actions appear as if each transaction runs in isolation

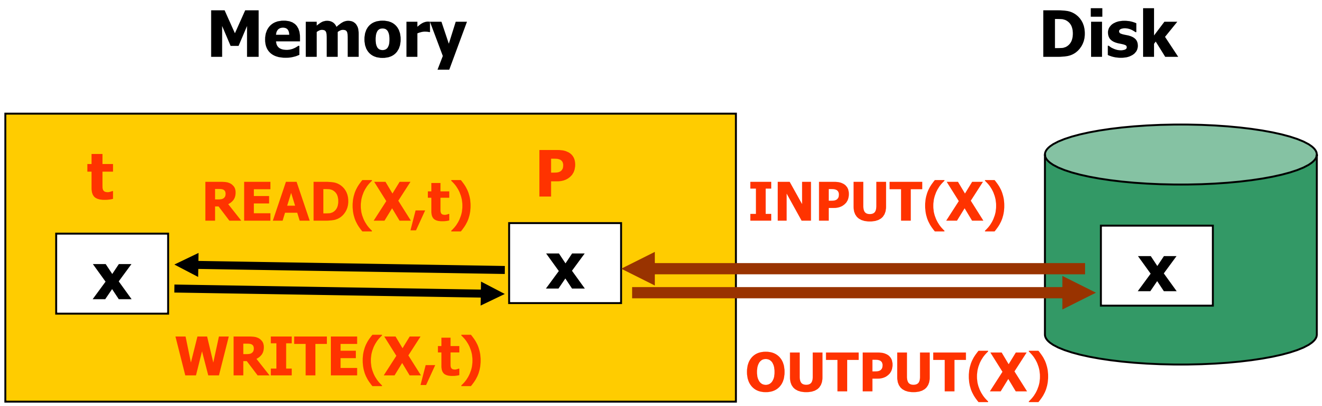
When we may end up with an inconsistent DB?

- Erroneous data entry
- Transaction bug (application programmer error)
- DBMS bug (DBMS programmer error)
- Other program bug (overrides memory page)
- System and media failures
  - power loss
  - memory failure
  - processor stop
  - disk crash
  - catastrophic failure: earthquake, flood, end of world

Coping with system failures

- Logging (undo, redo, undo-redo)
- Recovery using log
- Checkpointing
- Redundancy:
  - Replicate disk storage (RAID)
  - Memory parity
  - Archiving

Primitive operations of transactions



There are 3 address spaces involved in a transaction:

1. The disk blocks
2. The main memory (buffer) pages
3. The local variables of a Transaction

Operations:

- *Between buffer and disk:*
  - Input (x): block containing x $\rightarrow$ memory buffer
  - Output (x): block containing x $\rightarrow$ disk
- *Between transaction and buffer pages:*
  - Read (x,t): do input(x) if necessary
    t $\leftarrow$ value of x in page
  - Write (x,t): do input(x) if necessary
    value of x in page $\leftarrow$ t

Example: effect of transaction on state of memory and disk

A=8
B=8
Constraint: A=B (on disk)

T1: $A \leftarrow A \times 2$
$B \leftarrow B \times 2$

Action	*t*	*M-A*	*M-B*	*D-A*	*D-B*
1. READ(A,t)	8	8		8	8
2. t := t*2	16	8		8	8
3. WRITE(A,t)	16	16		8	8
4. READ(B,t)	8	16	8	8	8
5. t := t*2	16	16	8	8	8
6. WRITE(B,t)	16	16	16	8	8
7. OUTPUT(A)	16	16	16	16	8
8. OUTPUT(B)	16	16	16	16	16

Example: effect of transaction on state of memory and disk

A=8
B=6
Constraint: A=B

T1: A ← A × 2
B ← B × 2

Action	*t*	*M-A*	*M-B*	*D-A*	*D-B*
1. READ(A,t)	8	8		8	8
2. t := t*2	16	8		8	8
3. WRITE(A,t)	16	16		8	8
4. READ(B,t)	8	16	8	8	8
5. t := t*2 16	16	8	8	8	
6. WRITE(B,t)	16	16	16	8	8
7. OUTPUT(A)	16	16	16	16	8
8. OUTPUT(B)	16	16	16	16	16

CRASH

Example: effect of transaction on state of memory and disk

A=8
B=6
Constraint: A=B

T1: $A \leftarrow A \times 2$
$B \leftarrow B \times 2$

Action	***t***	***M-A***	***M-B***	***D-A***	***D-B***
1. READ(A,t)	**8**	**8**		**8**	**8**
2. t := t*2	**16**	**8**		**8**	**8**
3. WRITE(A,t)	**16**	**16**		**8**	**8**
4. READ(B,t)	**8**	**16**	**8**	**8**	**8**
5. t := t*2 16	**16**	**8**	**8**	**8**	
6. WRITE(B,t)	**16**	**16**	**16**	**8**	**8**
7. OUTPUT(A)	**16**	**16**	**16**	**16**	**8**

Inconsistent DB!

How to prevent an inconsistent state?

- We cannot prevent an inconsistent state, but we can arrange for the problem to be repaired

- Running the transaction again may not fix the problem
- Need ***atomicity***: execute
  - **all** actions of a transaction
  - **or none** at all

Solution 1:
undo logging (immediate modification on disk)

Log records

- A log is a file opened for append only
- It consists of log records, each telling something about what some transaction has done.

Log records:

<T , START>*:* This record indicates that transaction *T* has begun.

<T , COMMIT >*:* Transaction T has completed successfully and will make no more changes to database elements.

<T, ABORT >*:* Transaction T could not complete successfully.

<T, X, v>*:* Transaction T has changed database element *X* , and its **old** value was *v.*

Undo logging rules

(1) For every action generate update 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 written to log, **all writes** of transaction must be reflected **on disk** (forced to disk)

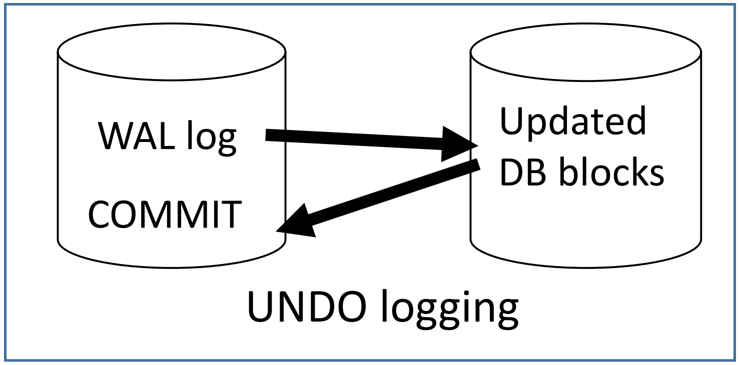
This is called force rule

Undo log - must write **to disk** in the following order:

1. The log records indicating that some db elements have changed.

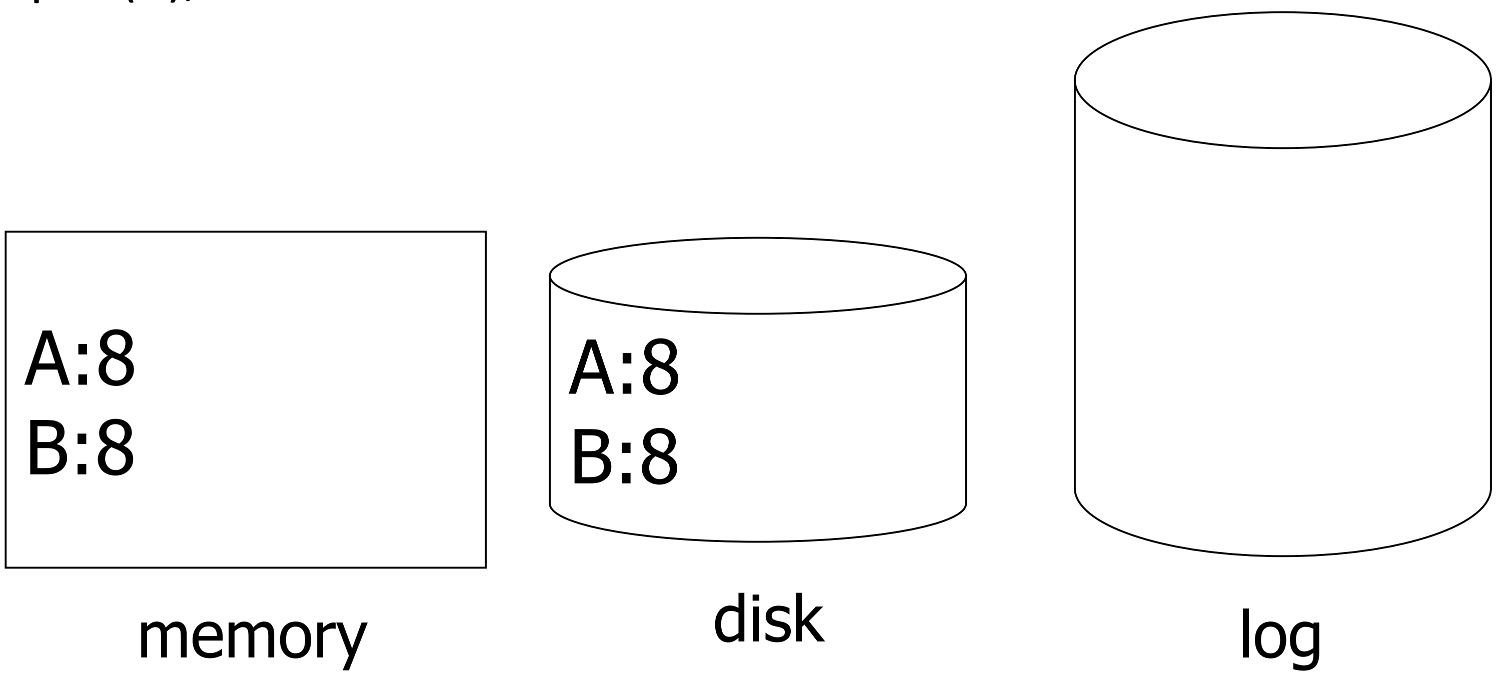
2. The changed database elements themselves.

3. The COMMIT log record.



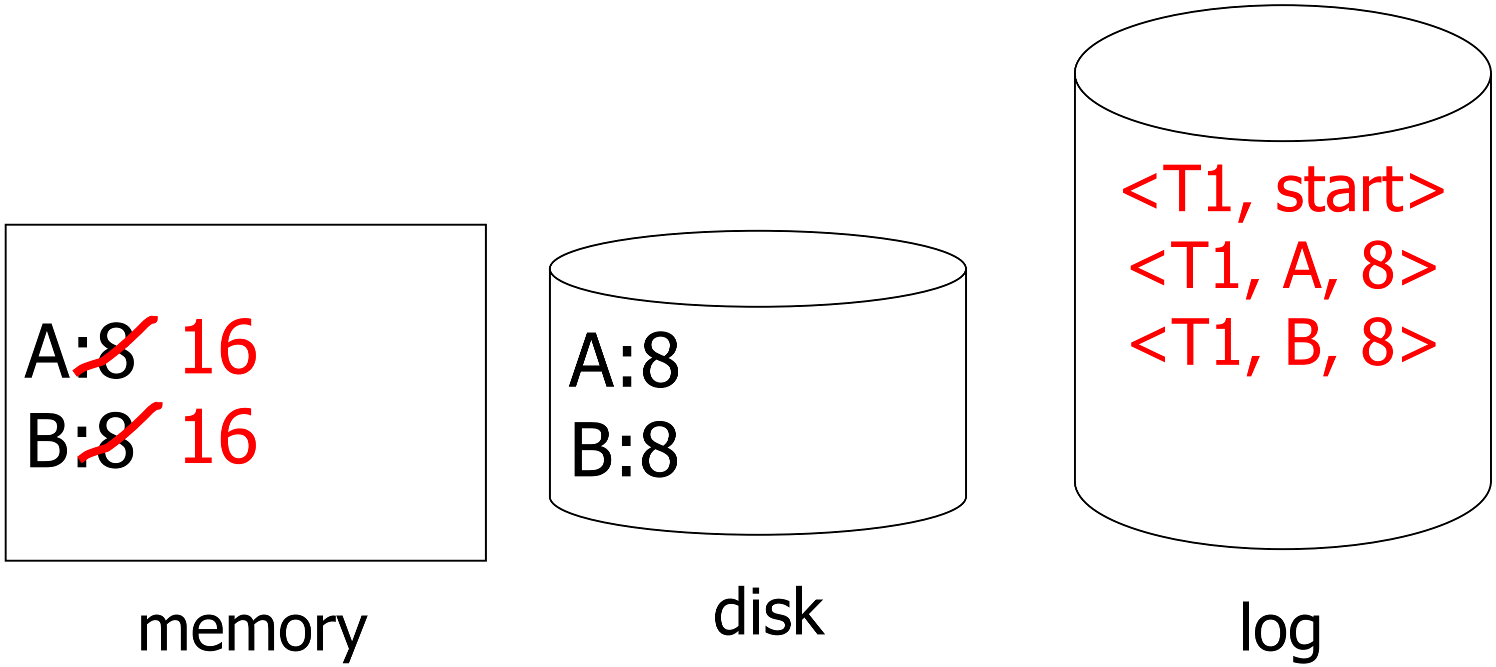
Example: Undo logging (Immediate modification)

T1: 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);



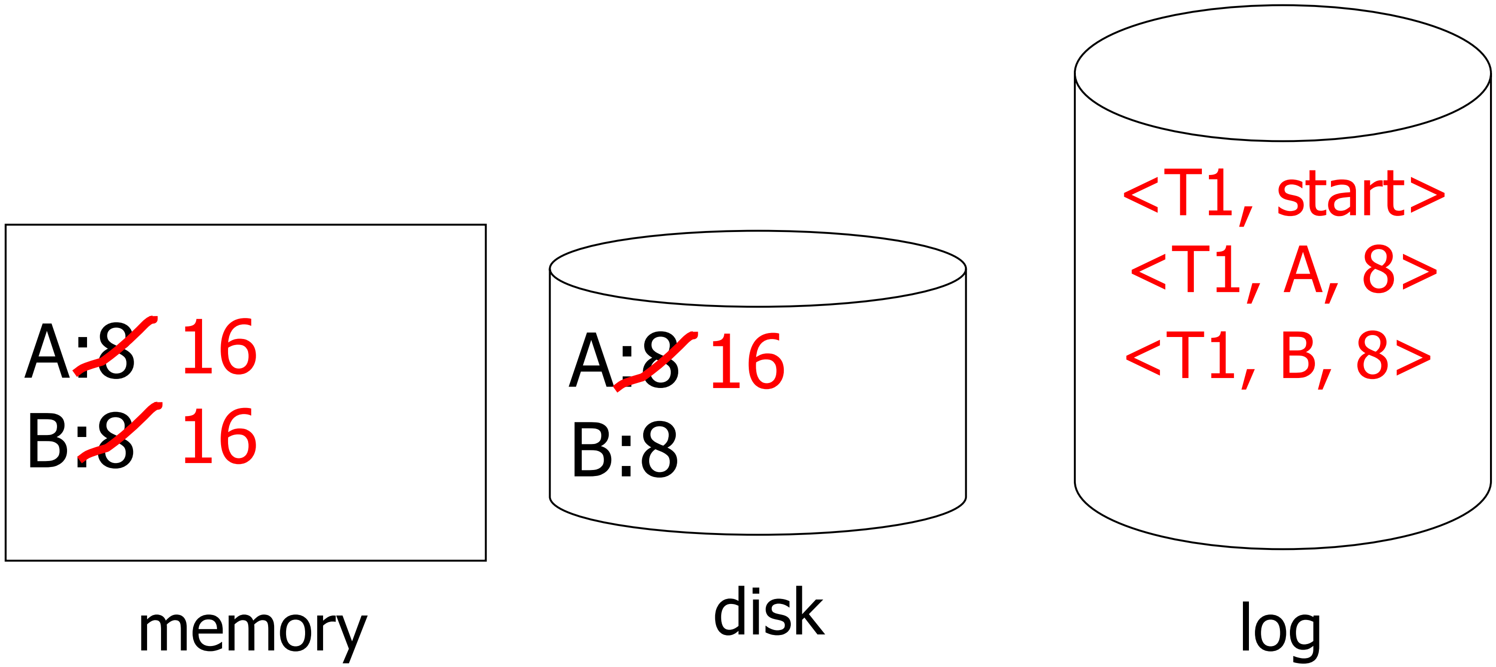
Example: Undo logging (Immediate modification)

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);



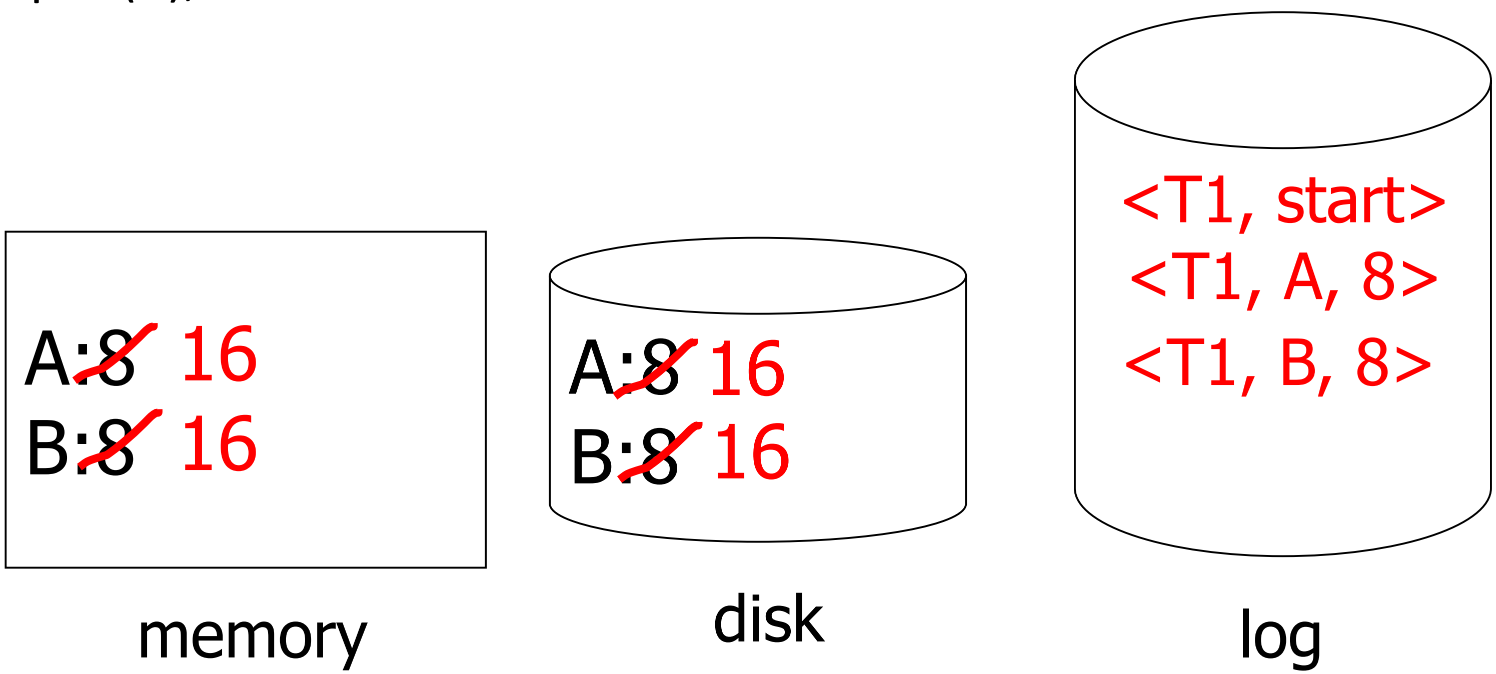
Example: Undo logging (Immediate modification)

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);



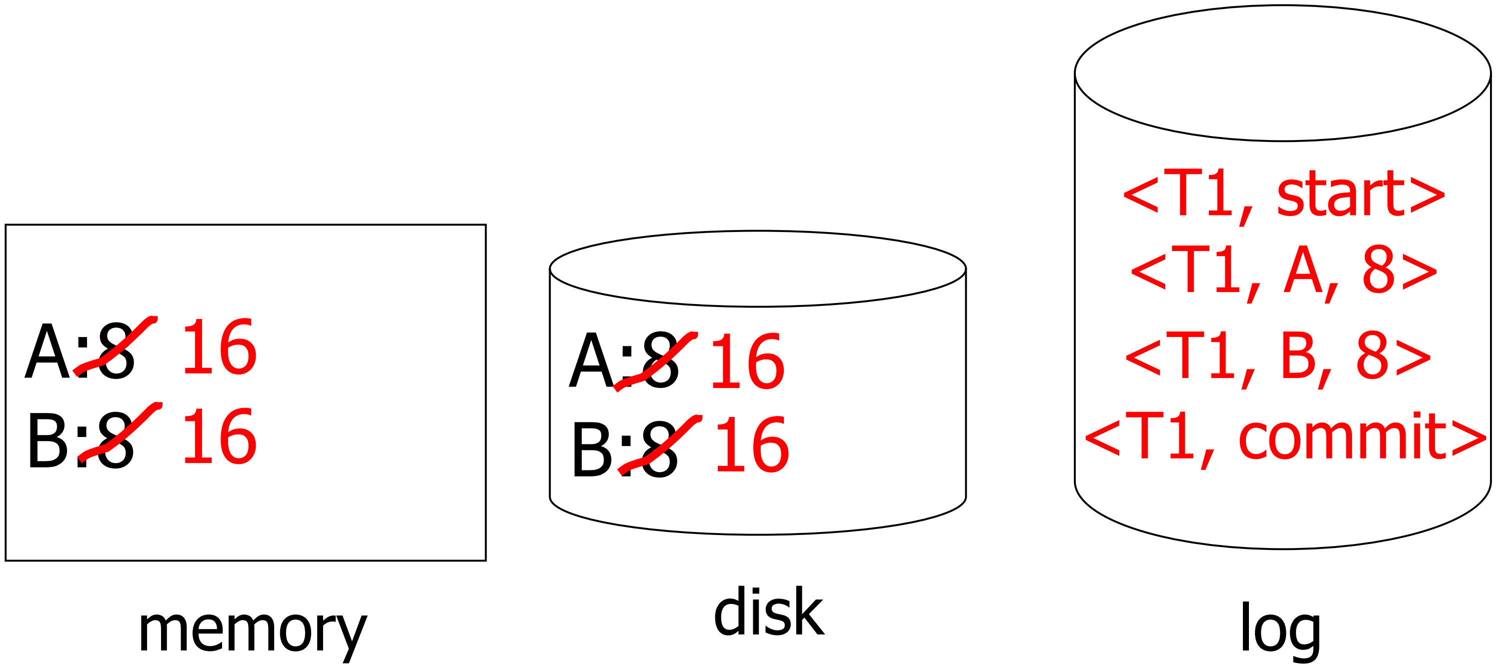
Example: Undo logging (Immediate modification)

T1: 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);



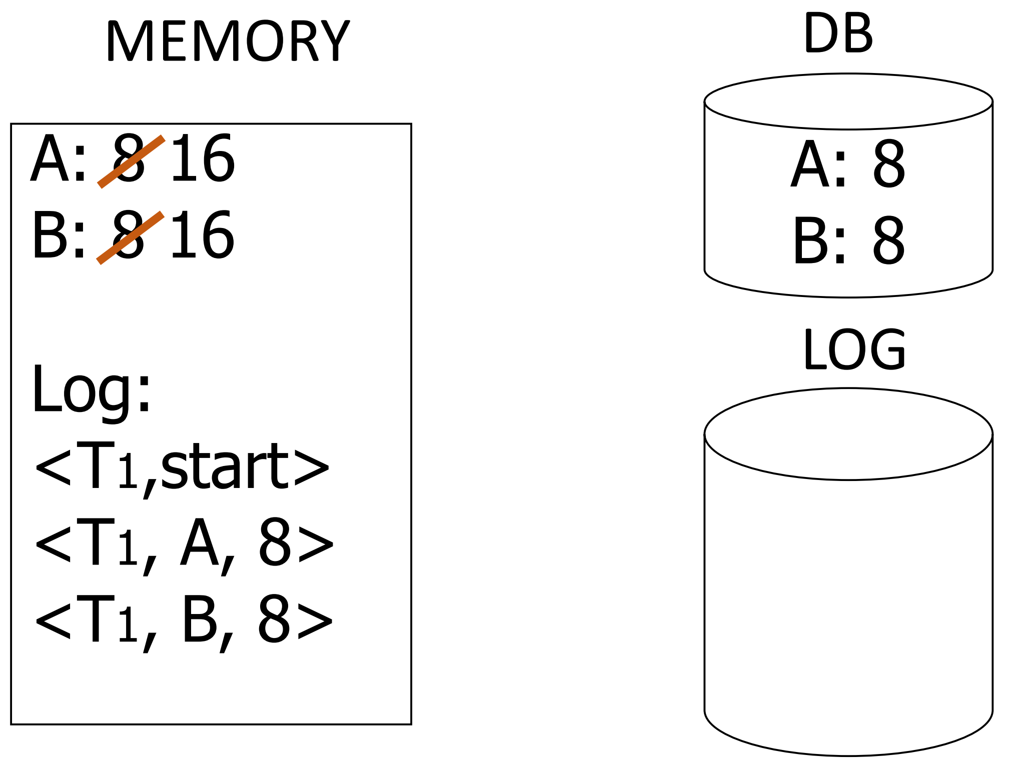
Example: Undo logging (Immediate modification)

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);



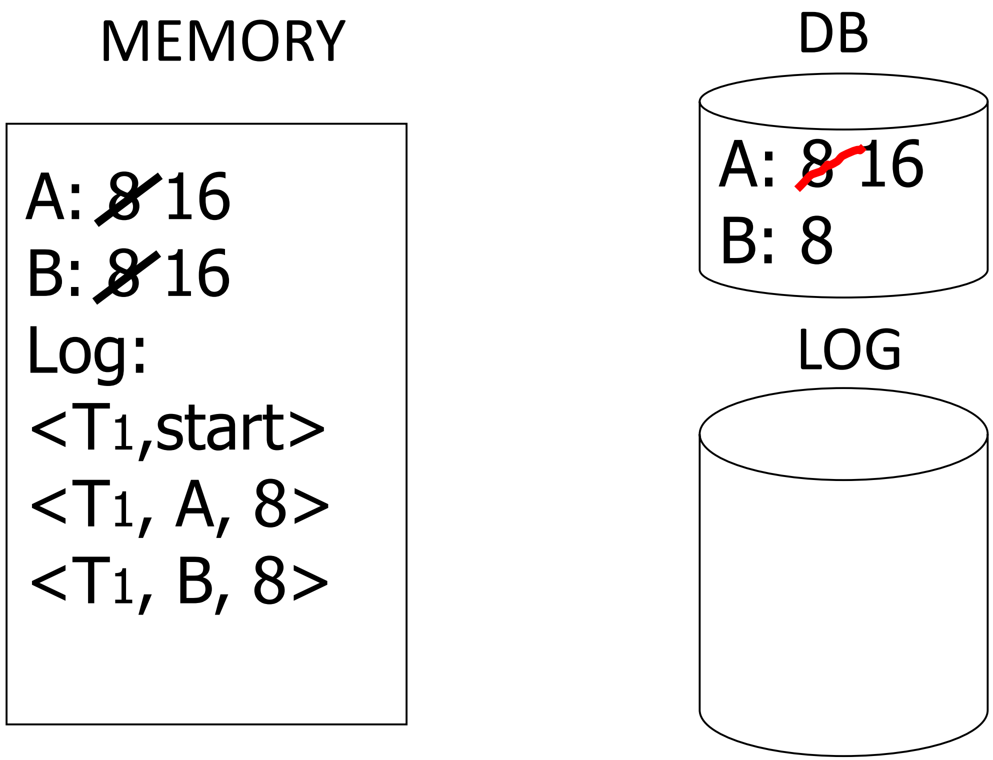
Flushing log to disk: explicitly

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



Flushing log to disk: explicitly

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



If database changed before log records reached disk

BAD STATE: cannot recover!

Order of steps and disk writes in case of UNDO log

Step	*Activity*	*t*	*M-A*	*M-B*	*D-A*	*D-B*	*Log*
1)							<T,START>
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)							<T,COMMIT>
12)	FLUSH LOG						

Before writing to disk

After logging commit

Recovery using UNDO log

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

Because multiple uncommitted transactions could potentially modify the same element several times, the undo operations are in reverse order (latest $\rightarrow$ earliest)

What if failure during recovery?

No problem! Undo idempotent

Example: Recovery using Undo log

Step	*Activity*	*t*	*M-A*	*M-B*	*D-A*	*D-B*	*Log*
1)							<T,START>
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)							<T,COMMIT>
12)	FLUSH LOG						

The crash occurs after step (12). Then the <C0MMIT *T*> *record reached* disk before the crash. When we recover, we do not undo the results of *T*, and all log records concerning *T are ignored by the recovery manager.*

Example: Recovery using Undo log

Step	*Activity*	*t*	*M-A*	*M-B*	*D-A*	*D-B*	*Log*
1)							`<T,START>`
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)							`<T,COMMIT>`
12)	`FLUSH LOG`						

The crash occurs between steps (11) and (12). If <COMMIT *T> record reached* disk see previous case, if not, see next case.

Example: Recovery using Undo log

Step	*Activity*	*t*	*M-A*	*M-B*	*D-A*	*D-B*	*Log*
1)							<T,START>
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)							<T,COMMIT>
12)	FLUSH LOG						

The crash occurs between steps (10) and (11). Now, the COMMIT record surely was not written, so *T is incomplete and is undone.*

Example: Recovery using Undo log

Step	*Activity*	*t*	*M-A*	*M-B*	*D-A*	*D-B*	*Log*
1)							<T,START>
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)							<T,COMMIT>
12)	FLUSH LOG						

The crash occurs between steps (8) and (10). Again, *T is undone. In this* case the change to *A and/or B may not have reached disk. Nevertheless,* the proper value, 8, is restored for each of these database elements.

Example: Recovery using Undo log

Step	***Activity***	***t***	***M-A***	***M-B***	***D-A***	***D-B***	***Log***
1)							**<T,START>**
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)							**<T,COMMIT>**
12)	**FLUSH LOG**						

The crash occurs prior to step (8). Now, it is not certain whether any of the log records concerning *T have reached disk.* If the change to *A and/or B reached disk, then the corresponding* log record reached disk. Therefore if there were changes to *A and/or B made on disk by T, then the corresponding log record will cause the* recovery manager to undo those changes.

Problems with UNDO logging

- The buffer pages forced to disk before writing <COMMIT T>, at the time that could be not the best from the disk performance perspective
- Too many disk I/Os

- How can we save disk I/Os allowing changed data reside in memory buffers for a while?

Solution 2:
Redo logging

Redo logging rules

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

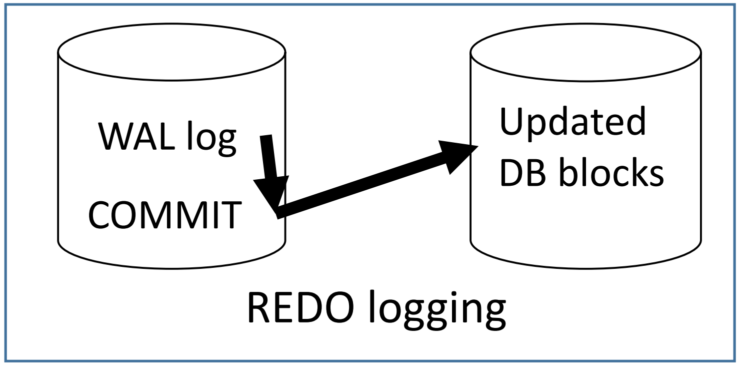
(2) Before X is modified on disk (DB), all log records for transaction that modified X (including commit) must be on disk

(3) Flush log at commit

Redo log - must write to disk in the following order:

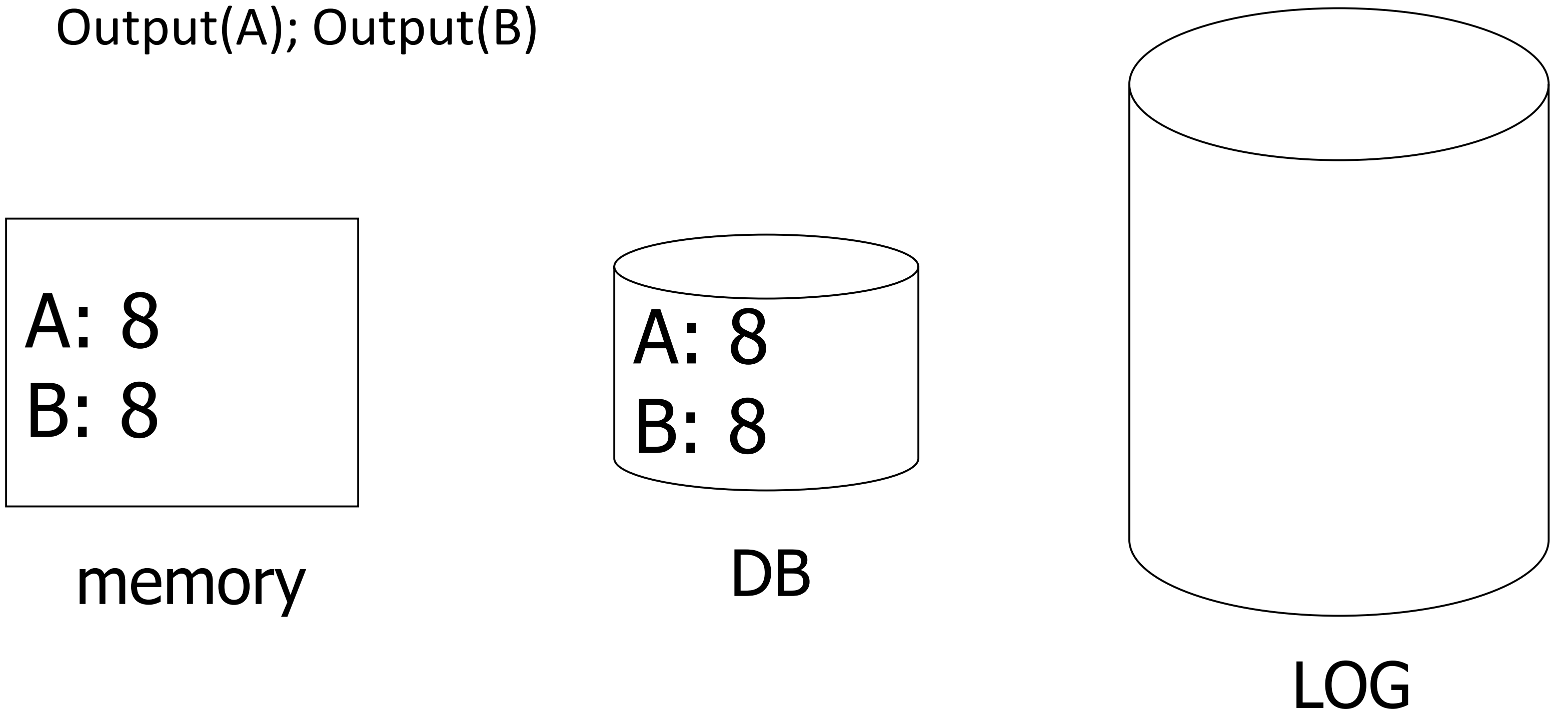
1. The log records indicating changed database elements.
2. The COMMIT log record.
3. The changed database elements themselves.

The changes remain in buffer until COMMIT log record reaches disk. That means that we cannot free dirty pages, until transaction is complete, we cannot *steal* them – this is called no steal rule



Example: 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)

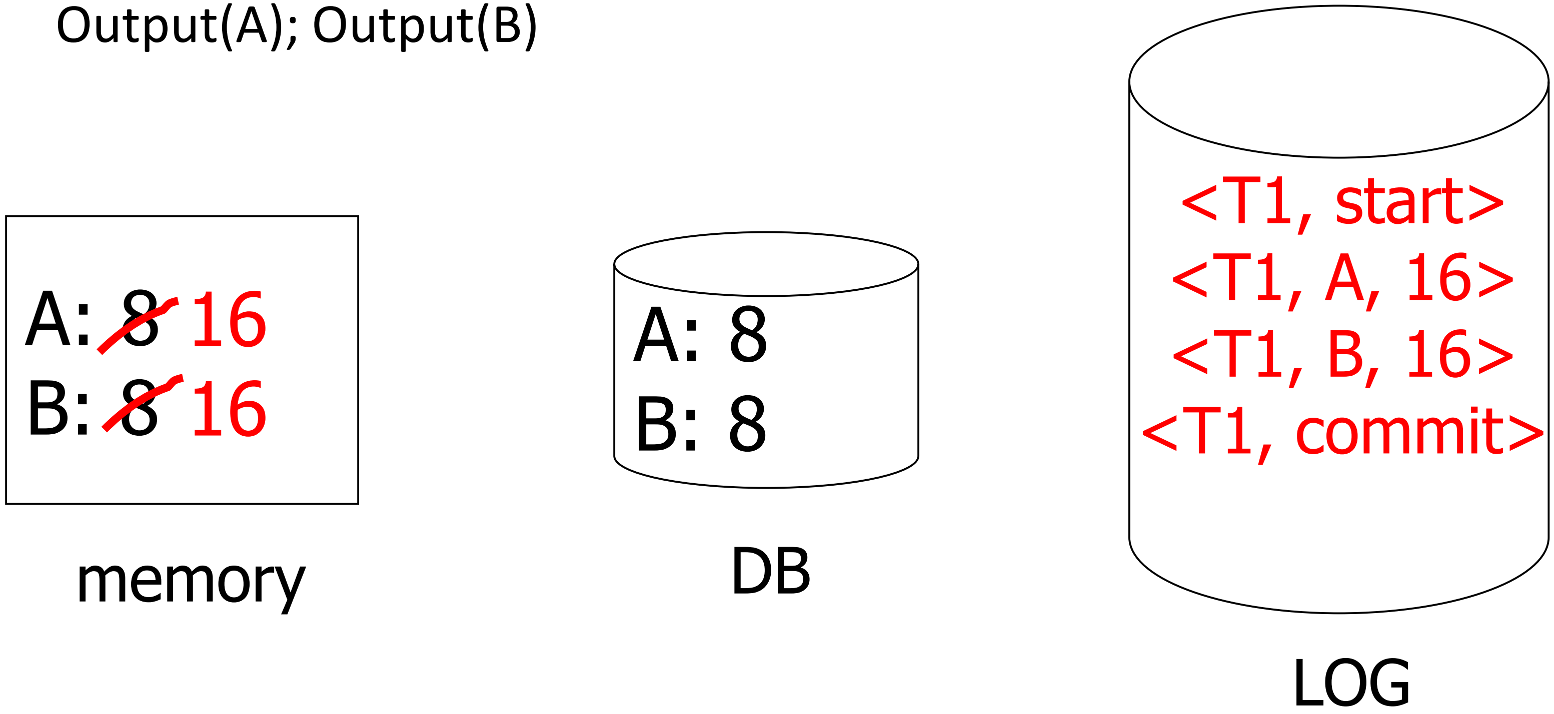


Example: Redo logging (deferred modification)

T1: Read(A,t); t t×2; write (A,t);

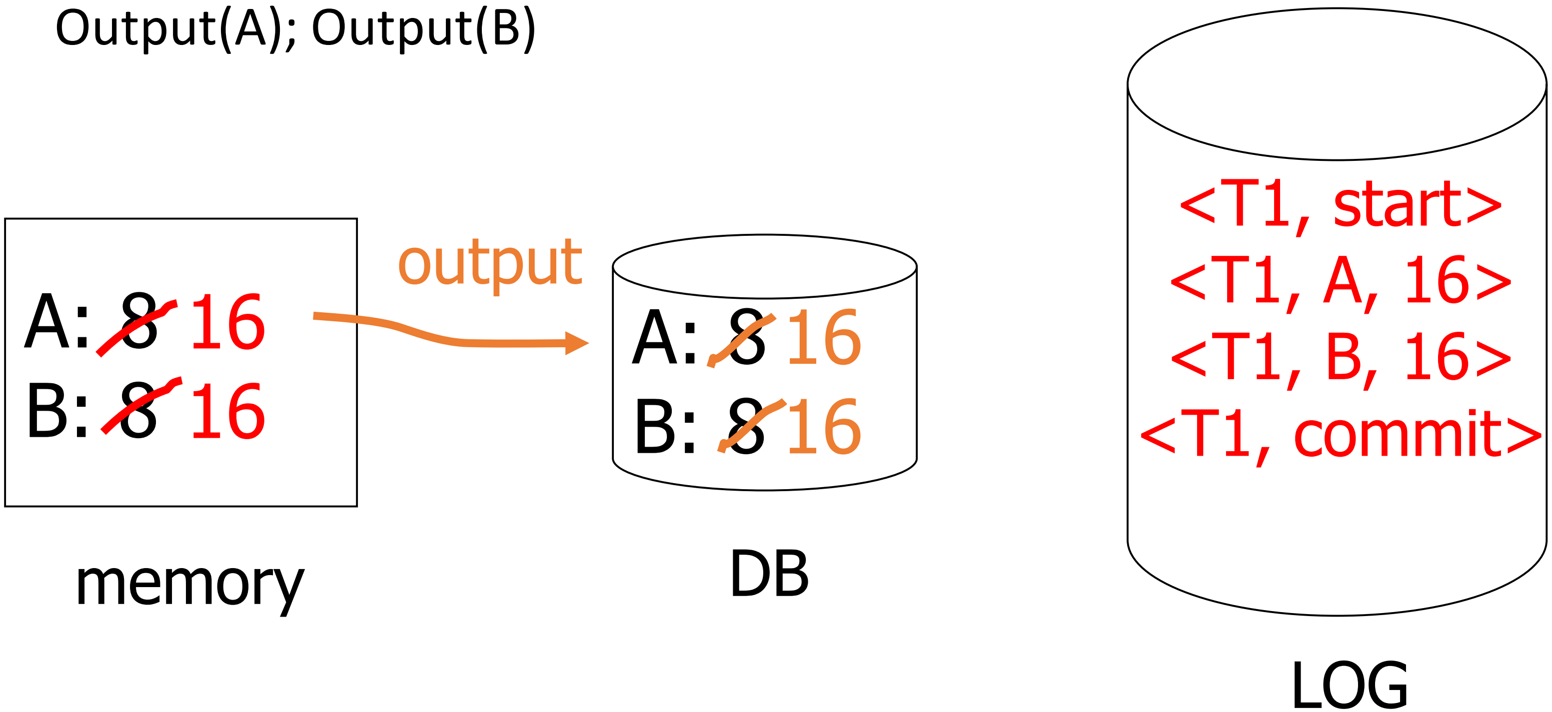
Read(B,t); t t×2; write (B,t);

Output(A); Output(B)



Example: 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)



Order of steps and disk writes in case of REDO log

Step	*Activity*	*t*	*M-A*	*M-B*	*D-A*	*D-B*	*Log*
1)							<T,START>
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)							<T,COMMIT>
9)	FLUSH LOG						
10)	OUTPUT(A)	16	16	16	16	8	
11)	OUTPUT(B)	16	16	16	16	16	

After logging commit

Recovery using REDO log

For each Ti with <Ti, commit> in log:

  For all <Ti, X, v> in log:

    Write(X, v)

    Output(X)

For each Ti without commit, write <Ti, abort>

Because we need to replay committed transactions in the order they were executed, the redo operations are in forward order (earliest → latest)

Key drawbacks

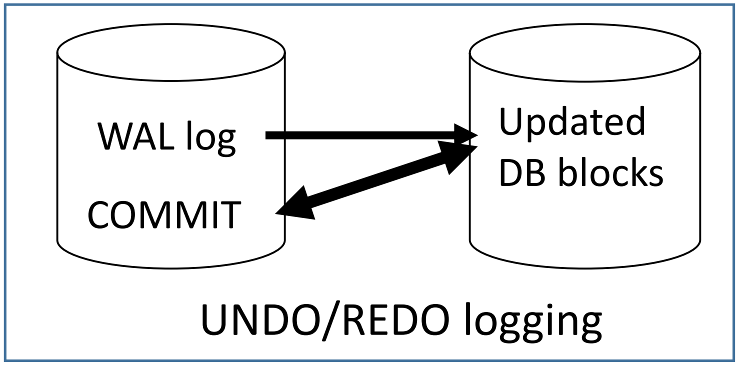
- *Undo logging:* need frequent disk writes
- *Redo logging:* need to keep all modified blocks in memory until commit

Solution: undo/redo logging – increased flexibility at the expense of larger log

Undo/redo logging

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

- Page with X can be flushed before or after <COMMIT T> is written
- Log record has to be flushed before corresponding updated page (WAL)
- Flush log after <COMMIT T> is written (solves problem of delayed commitment)



Undo/redo logging rules

UR1 Before modifying any database element X on disk because of changes made by some transaction T, it is necessary that the update log record <T, X , v, w> appear on disk.

UR2 A <C0MMIT T> record must be flushed to disk **as soon as it appears in the log**

Undo/redo recovery policy

1. Redo all the committed transactions in the order earliest-first, and
2. Undo all the uncommited transactions in the order latest-first.

Log needs to be truncated

- Log can become larger than DB itself
- It takes too much time to check all the log records when recovery is needed

- We want to truncate some old log records, which are no longer needed

- Can we delete everything prior to **`<T,COMMIT>`?**

No, because the actions of some uncommitted transactions are interleaving

Solution: checkpointing

Quiescent checkpointing

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

Every transaction executed before checkpoint has finished and the log can be truncated

Problem: while waiting for all active transactions to complete, DB appears stalled to its users

Solution:
non-quiescent checkpointing

Non-quiescent checkpointing

1. Write log record <START CKPT(T1, …Tk)>

(T1 … Tk are active transactions)

and flush log.

2. Wait until all T1 … Tk commit or abort, but don't prohibit other transactions from starting.

3. When all T1 … Tk have completed, write a log record <END CKPT> and flush the log.

Recovery using UNDO log with checkpointing – in words

Scanning log backwards:

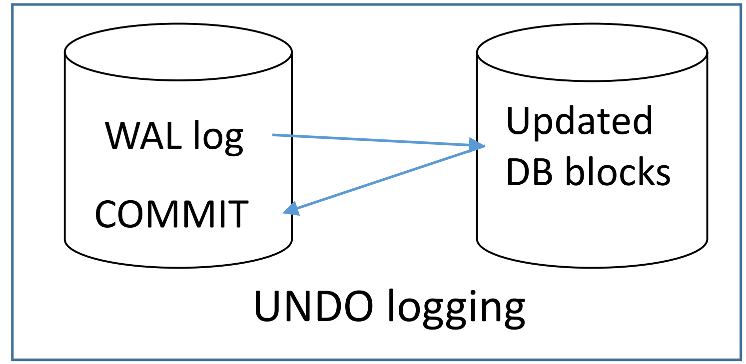
- If we first meet an <END CKPT> record, then we know that all incomplete transactions began after the <START CKPT (T1, ... ,Tk)> record.

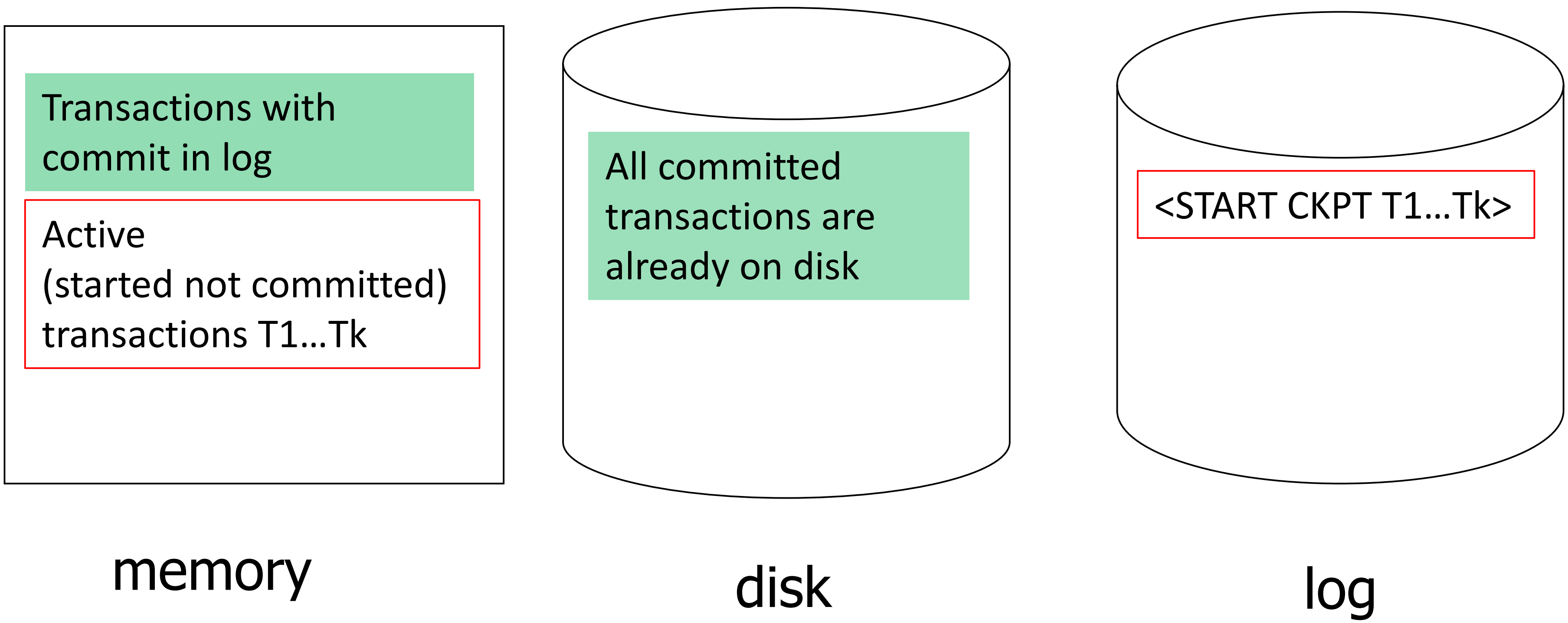
We may thus scan backwards as far as this <START CKPT>, and then stop; previous log is useless and may be discarded after the recovery.

- If we first meet a record < START CKPT (T1, ... , Tk)>, then the crash occurred during the checkpoint. We need scan no further back than the start of the earliest of these incomplete transactions.

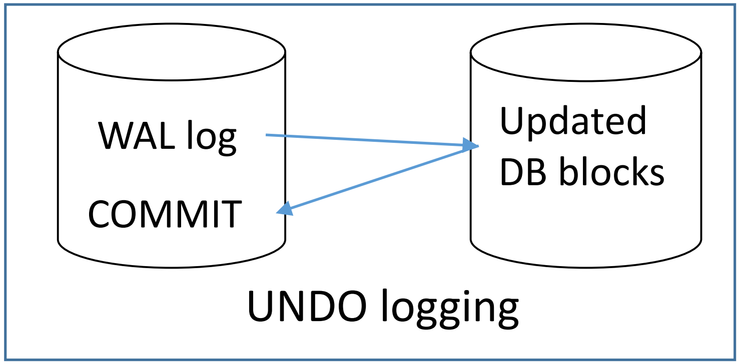
- General rule: once <END CKPT> is written, we can discard the log prior to the preceding <START CKPT> record

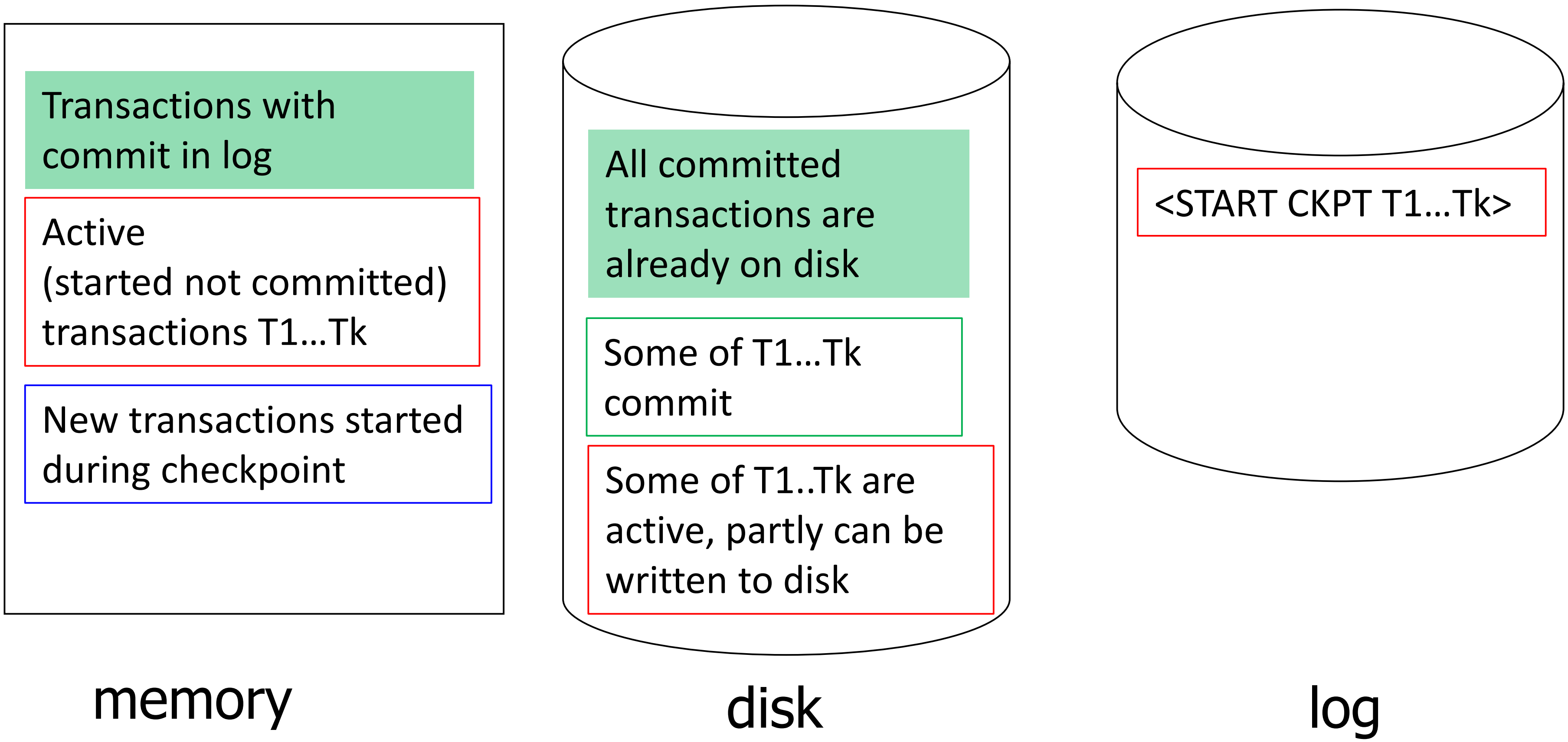
Recovery using undo – start checkpoint



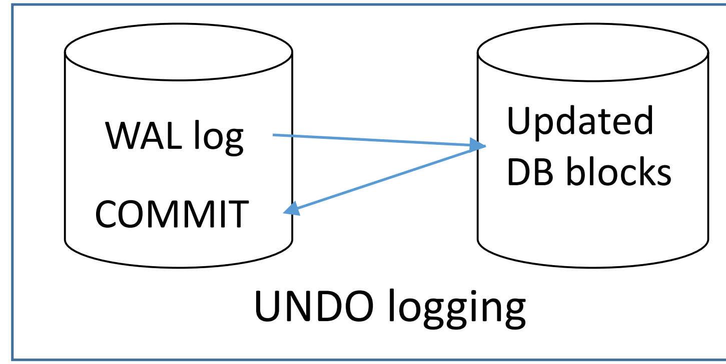


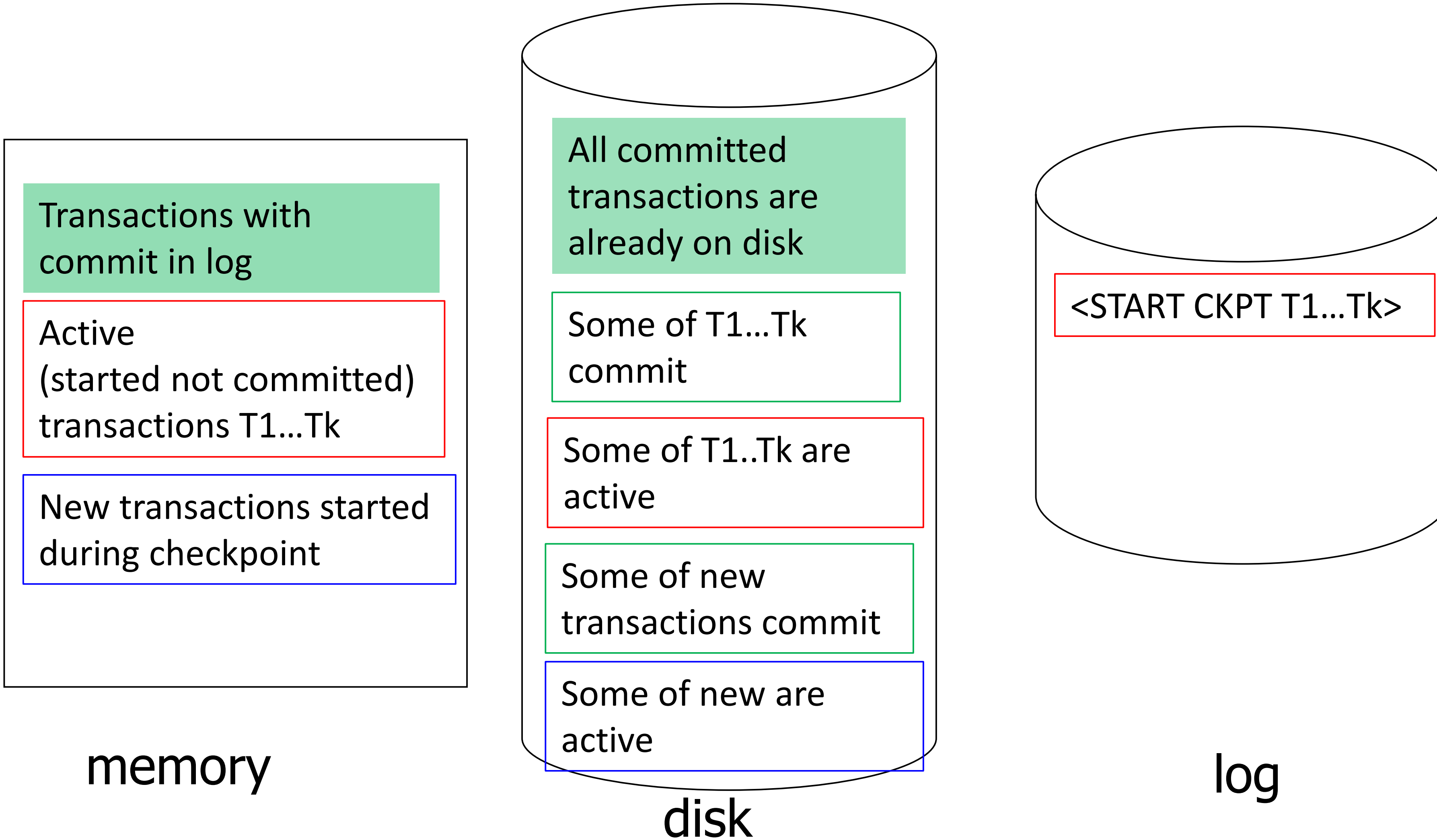
Recovery using undo – during checkpoint



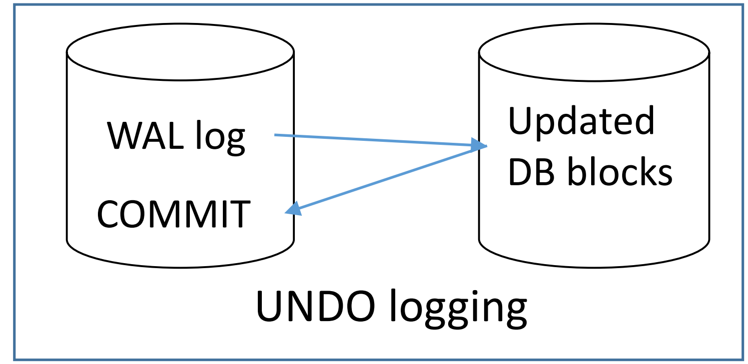


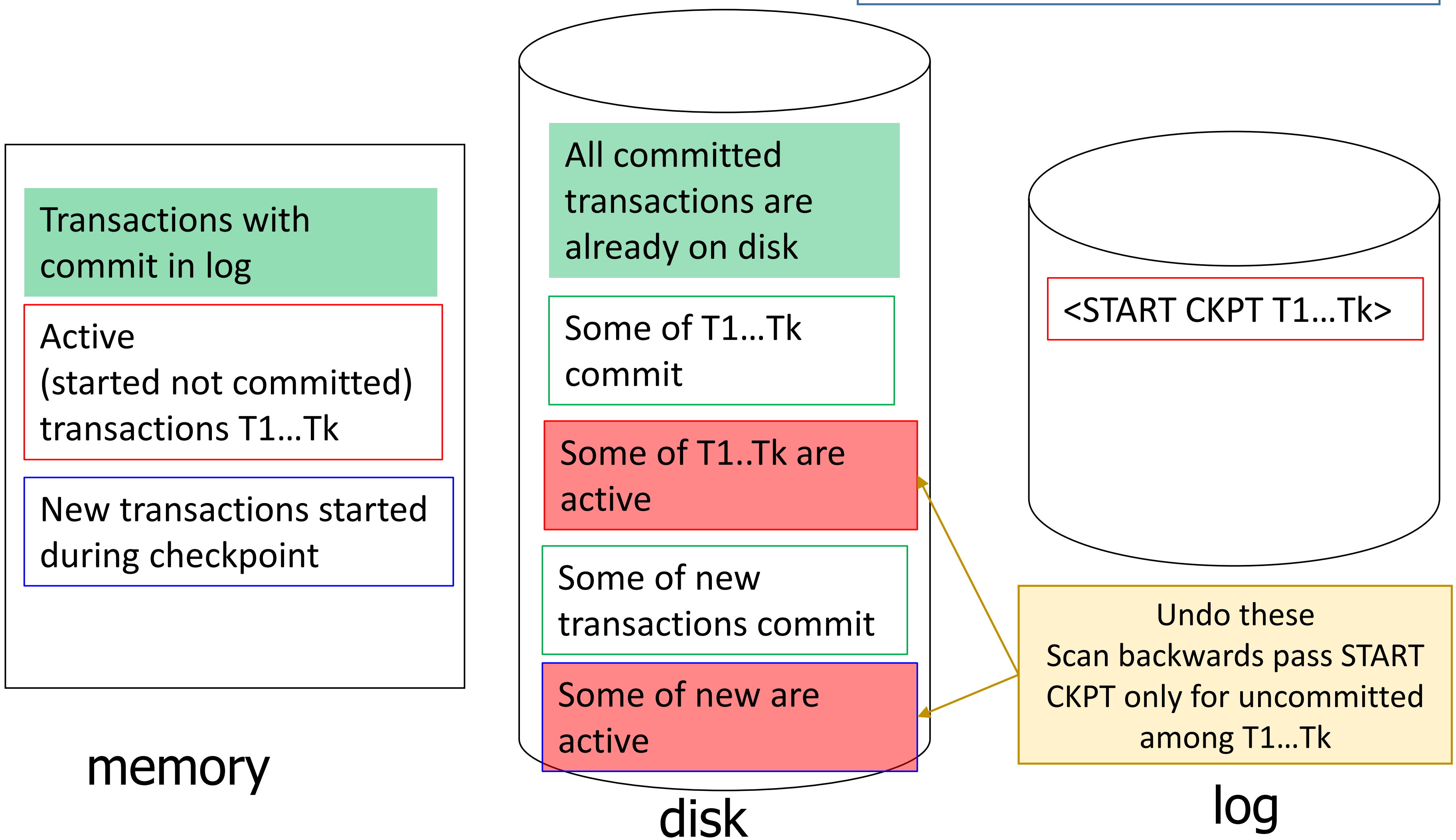
Recovery using undo – during checkpoint



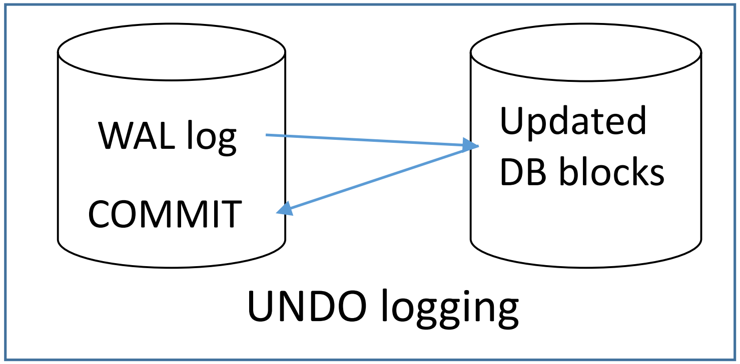


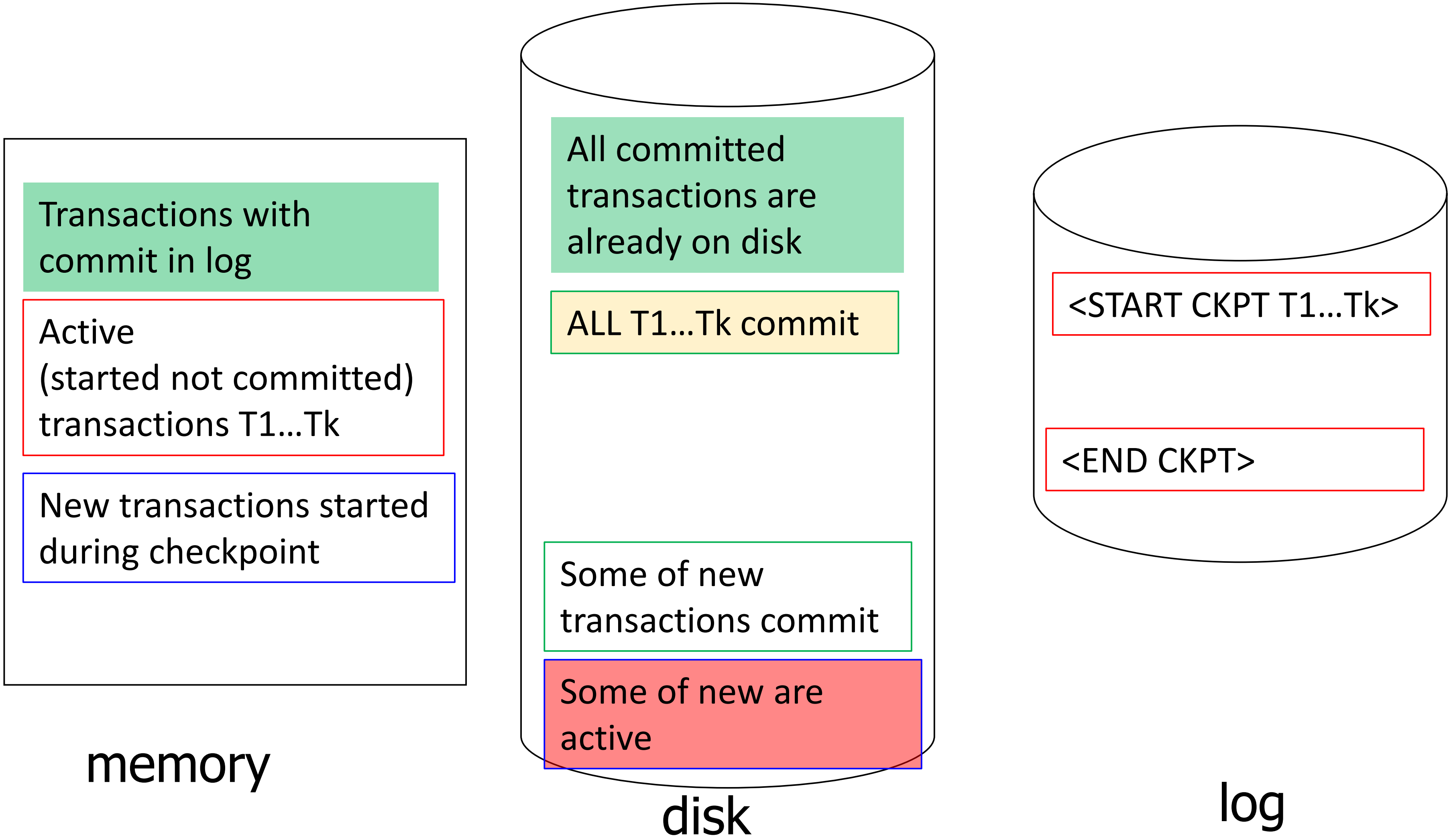
Recovery using undo – failure during checkpoint



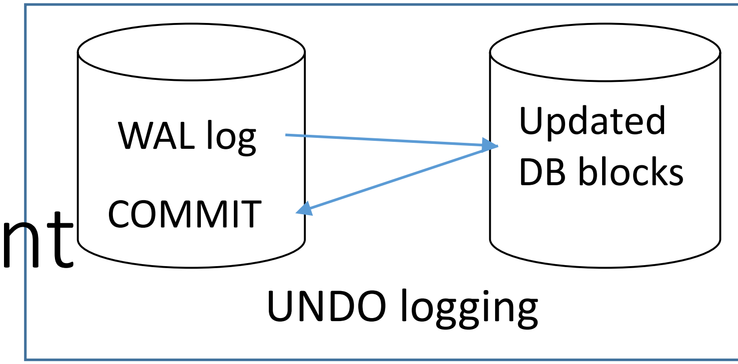


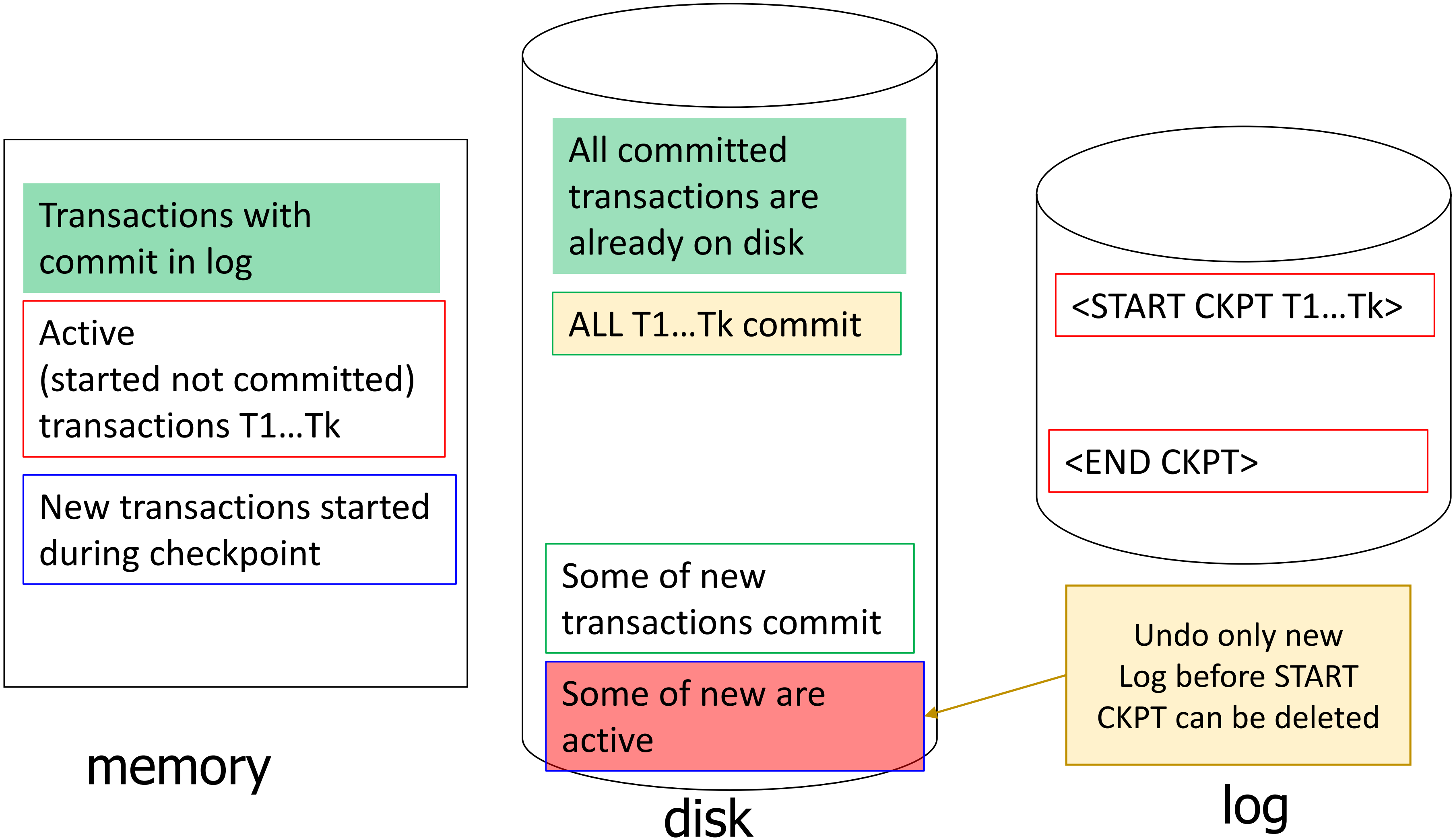
Recovery using undo – end checkpoint



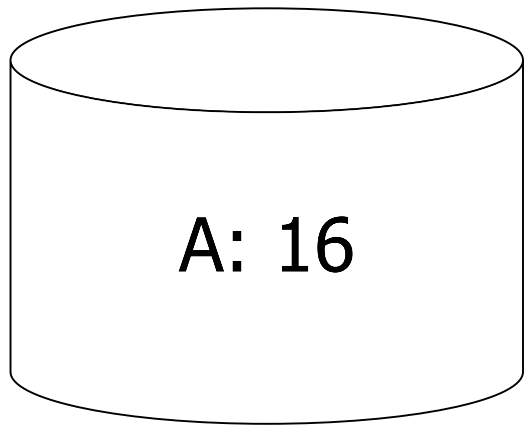


Recovery using undo – failure after end checkpoint





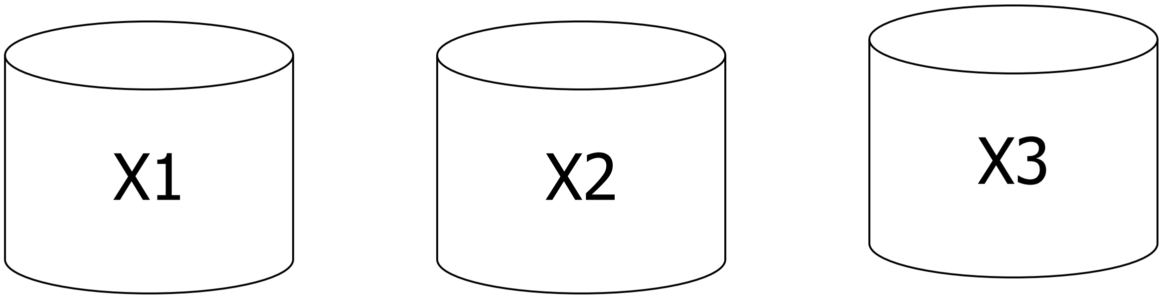
Media failure
(loss of non-volatile storage)



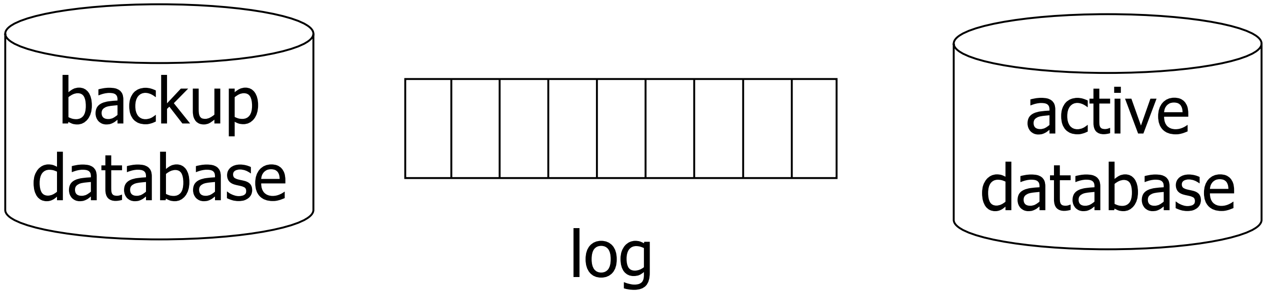
Solution: Make copies of data!

Triple modular redundancy

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



DB Dump + Log



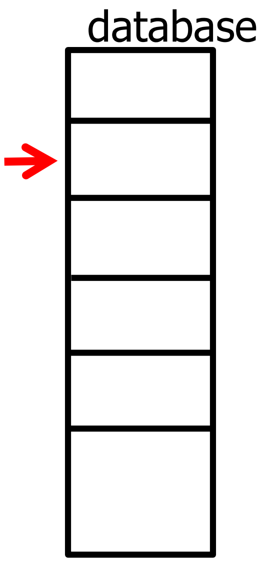
- If active database is lost,
  - restore active database from backup
  - bring up-to-date using redo entries in log

Non-quiescent archiving

- 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]
```



- To restore – we need the dump and the log created during the backup

Summary

- To preserve DB consistency: need mechanisms to get out of an inconsistent state created due to failure
- Two main recovery techniques: logging and redundant copies
- The most flexible logging protocol: undo/redo
- Checkpoints prevent log from indefinite growth

Mechanisms that guarantee ACID transactions

- ***A**tomicity*: recovery with undo/redo logging
- ***C**onsistency*: serializable schedules, logging for the event of crash
- ***I**solation*: serializable schedules, locking
- ***D**urability*: write-ahead logging, redundant copies