Ullman et al. : Database System Principles

Notes 08: Failure Recovery

Integrity or correctness of data

- Would like data to be "accurate" or "correct" at all times

EMP

Name	Age
White	52
Green	3421
Gray	1

Integrity or consistency constraints

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

Definition:

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

Constraints (as we use here) may
not capture "full correctness"

Example 1 Transaction constraints

- When salary is updated,
  new salary > old salary
- When account record is deleted,
  balance = 0

Note: could be "emulated" by simple constraints, e.g.,

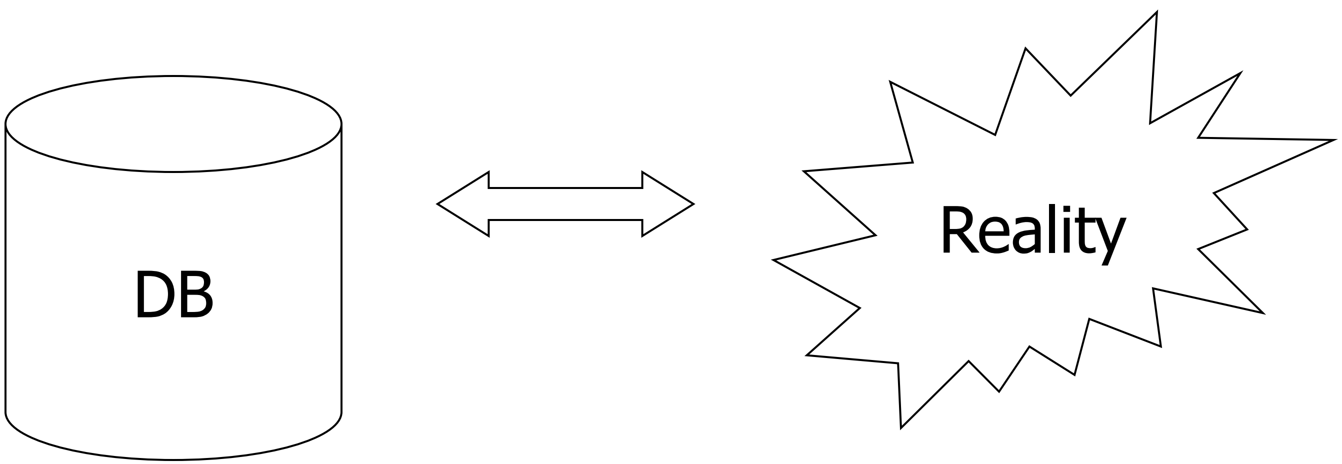
if deleted = true, then balance = 0

account

Acct #	….	balance	deleted?

Constraints (as we use here) may not capture "full correctness"

Example 2 Database should reflect real world



☞ in any case, continue with constraints...

Observation: DB cannot be consistent always!

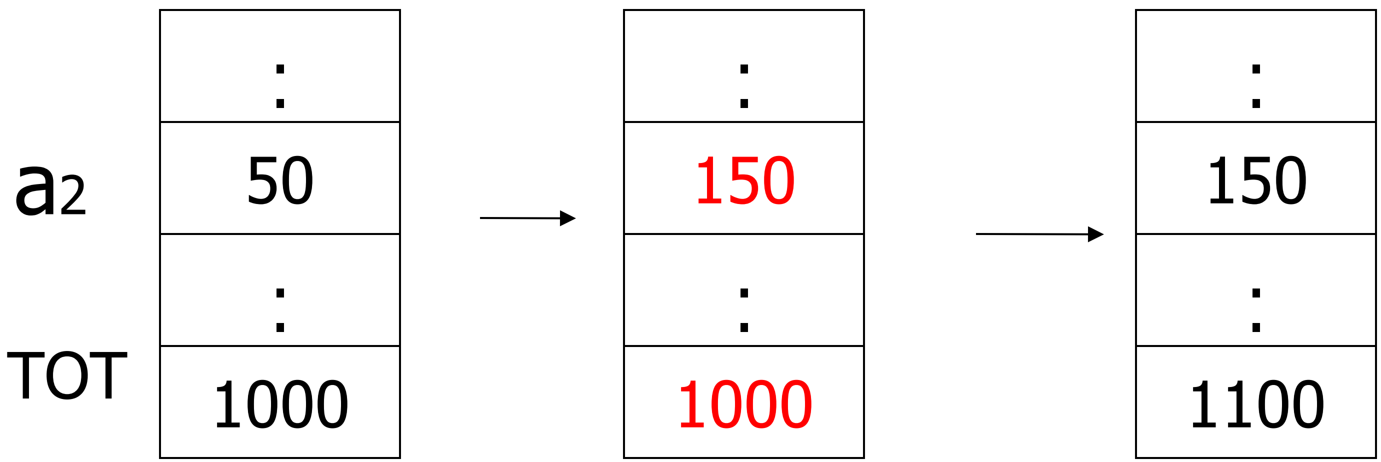
Example: $a_1 + a_2 + \ldots. a_n = TOT$ (constraint)

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

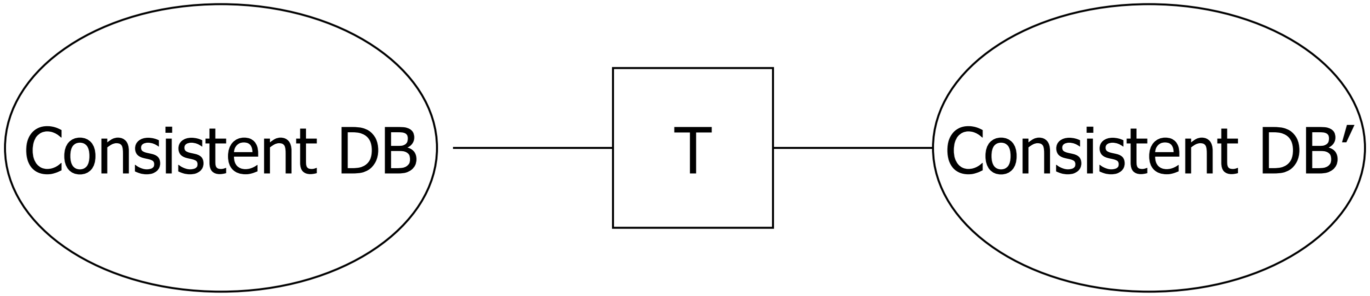
Example: $a_1 + a_2 + \ldots . a_n = TOT$ (constraint)

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

How can constraints be violated?

- Transaction bug (incomplete transaction)
- DBMS bug (some process)
- Hardware failure

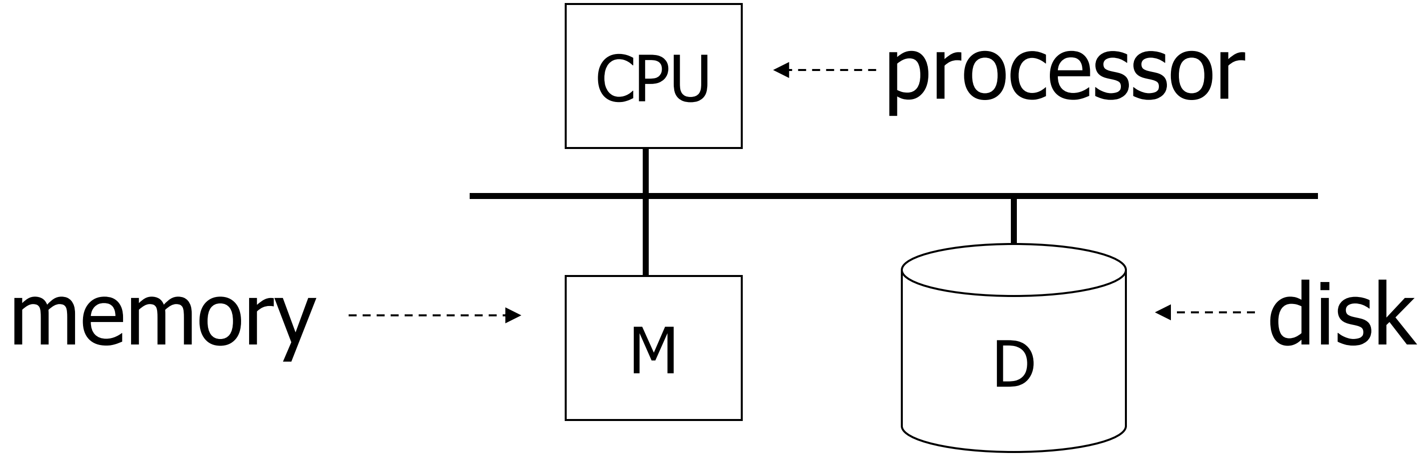
    e.g., disk crash alters balance of account

- Data sharing

e.g.: T1: give 10% raise to programmers

T2: change programmers $\Rightarrow$ systems analysts

Our failure model



- Events
  - Desired
  - Undesired
    - Expected
    - Unexpected

Desired events: see product manuals….

Undesired expected events:

System crash

- memory lost
- cpu halts, resets

that's it!!

Undesired Unexpected: Everything else!

<u>Undesired Unexpected:</u> Everything else!

Examples:

- Disk data is lost
- Memory lost without CPU halt
- CPU implodes wiping out universe….

Is this model reasonable?

Approach: Add low level checks + redundancy to increase probability model holds

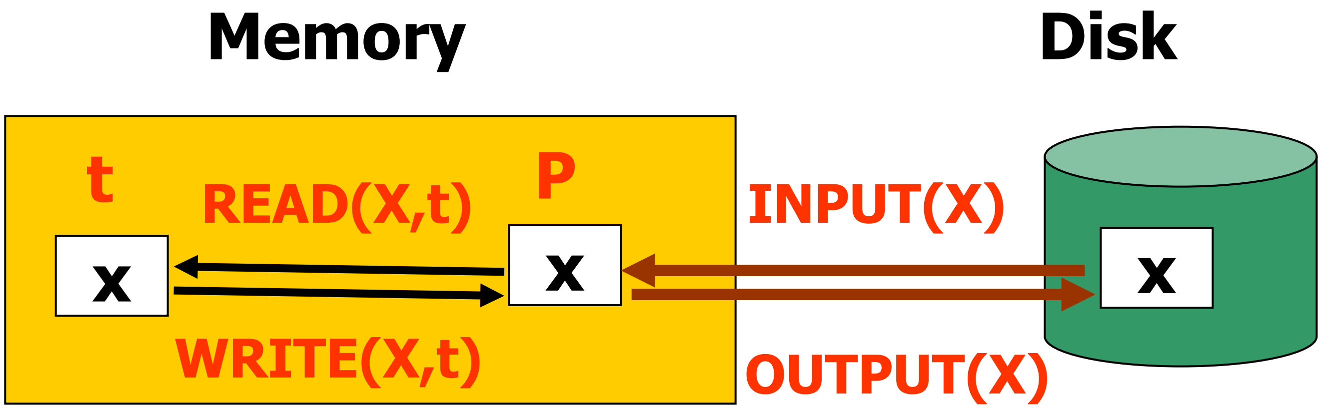
E.g.,
- Replicate disk storage (RAID)
- Memory parity
- CPU checks

The primitive operations of Transactions

There are 3 important address spaces:

1. The disk blocks
2. The shared main memory
3. The local address space of a Transaction

Basic operations



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

Steps of a transaction and its effect on memory and disk

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

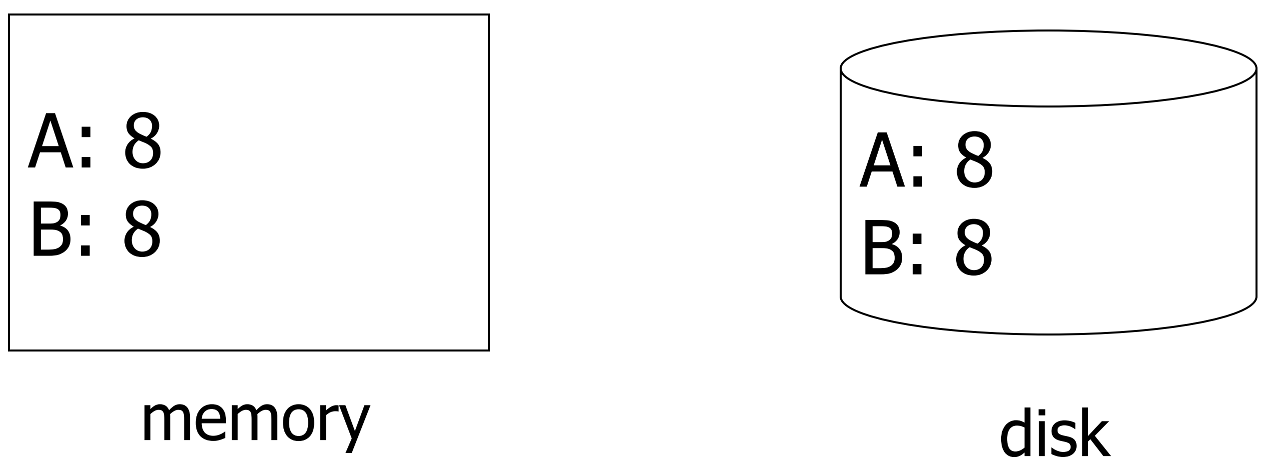
Key problem   Unfinished transaction

Example   Constraint: A=B

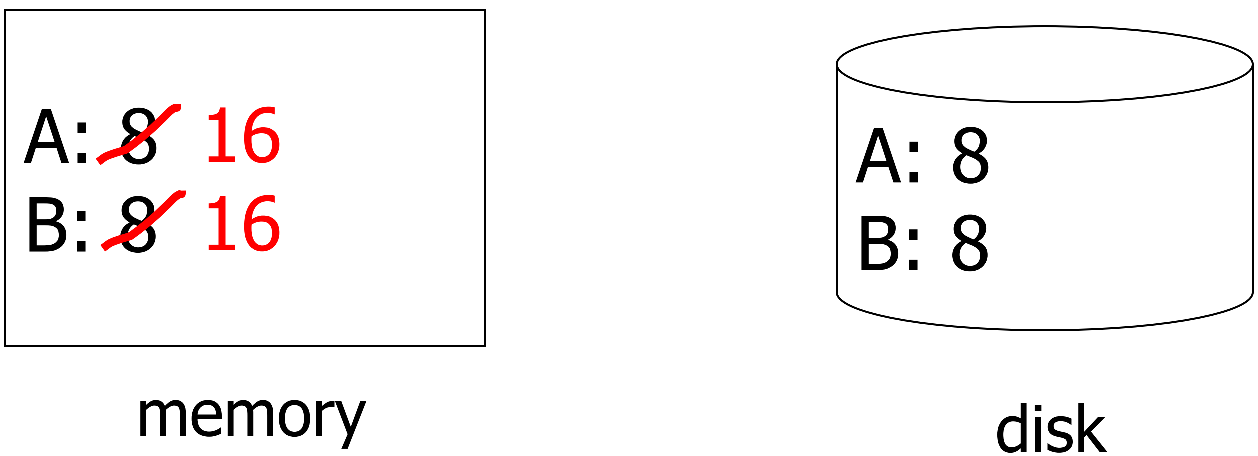
T_1: $A \leftarrow A \times 2$

$B \leftarrow B \times 2$

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);

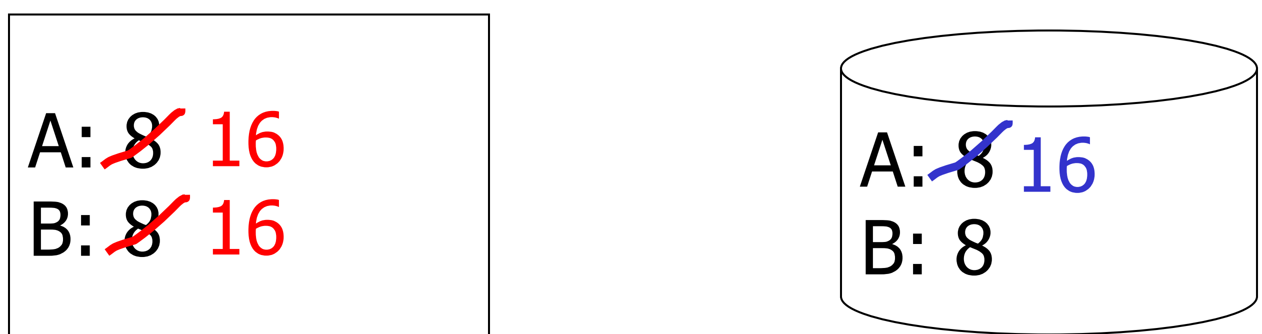


T_1: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



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

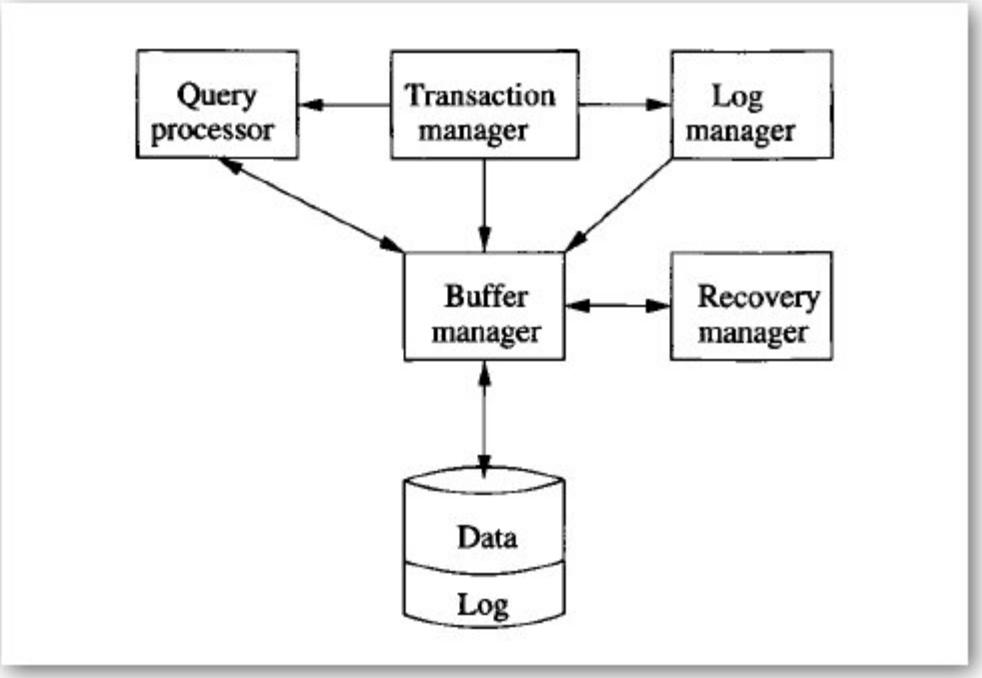
disk

Steps of a transaction and its effect on memory and disk

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

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

One solution: undo logging
(immediate modification on disk)



The transaction manager will send messages about actions of transactions to the log manager, to the buffer manager about when it is possible or necessary to copy the buffer back to disk, and to the query processor to execute the queries and other database operations that comprise the transaction.

The log manager maintains the log. It must deal with the buffer manager, since space for the log initially appears in main-memory buffers, and at certain times these buffers must be copied to disk.

A log is a file of log records, each telling something about what some transaction has done.

Log records:

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

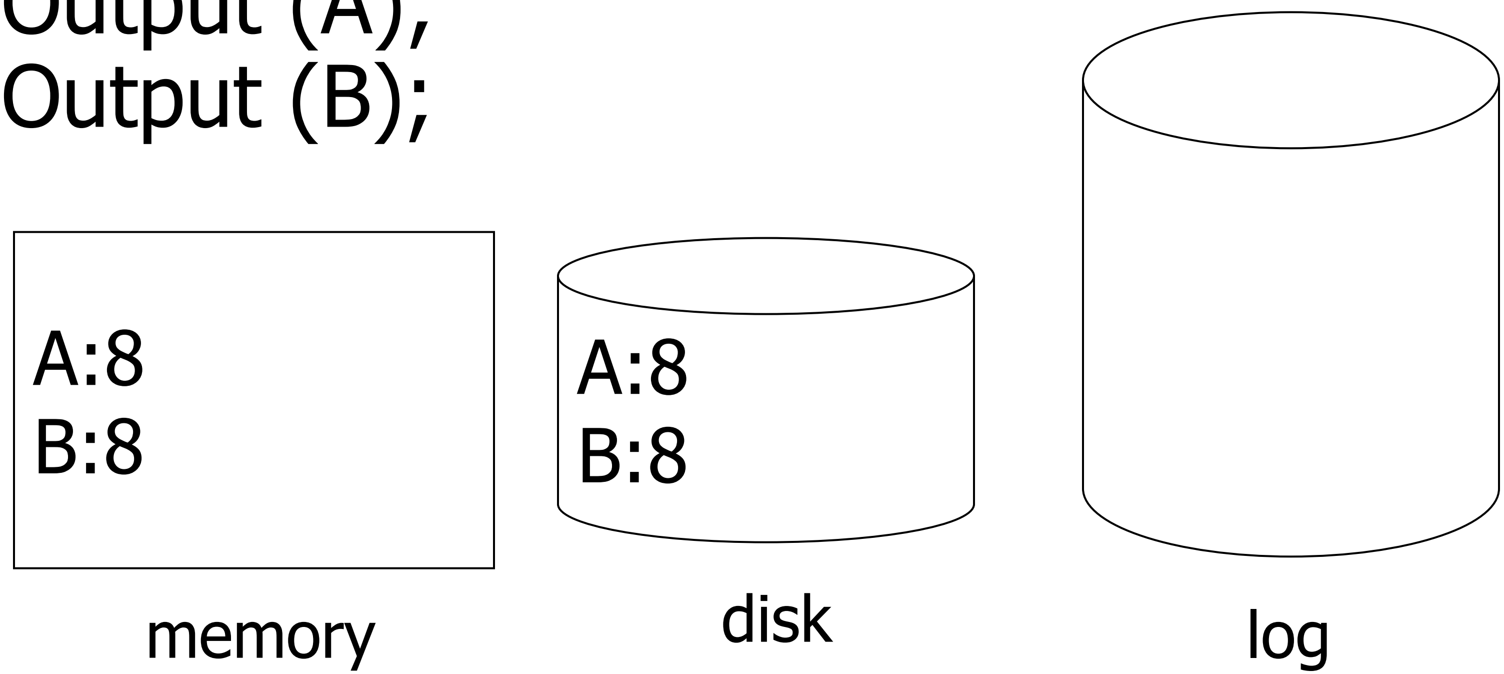
<T , COMMIT >: Transaction T has completed successfully and will make no more changes to database elements.

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

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

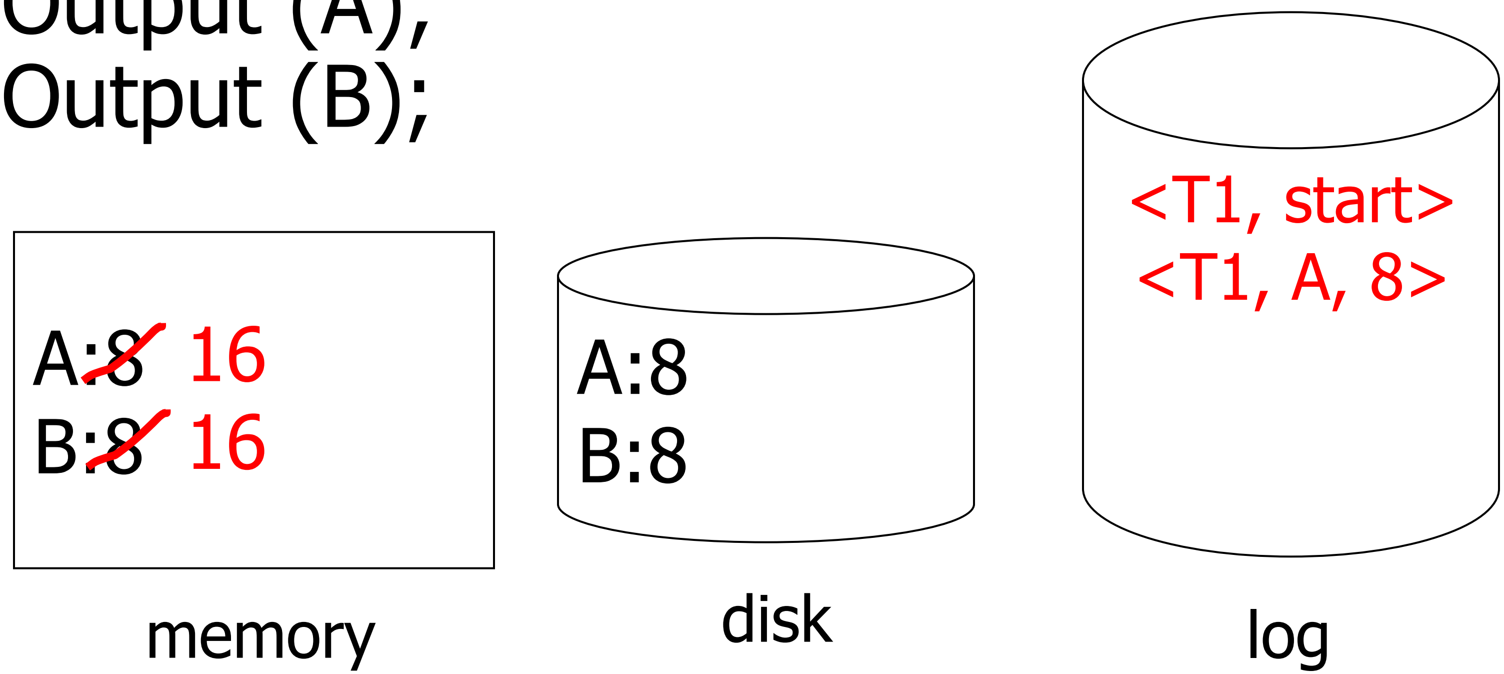
Undo logging (Immediate modification)

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);



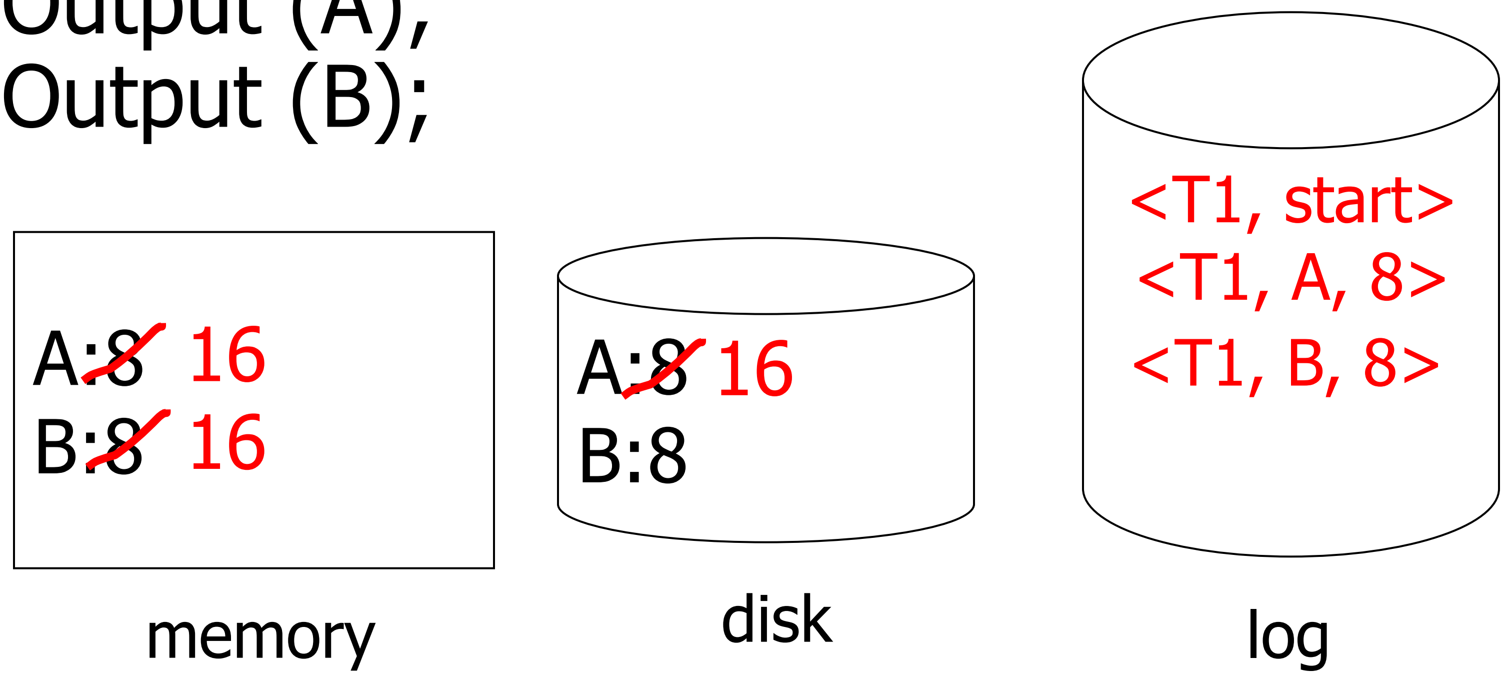
Undo logging (Immediate modification)

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);



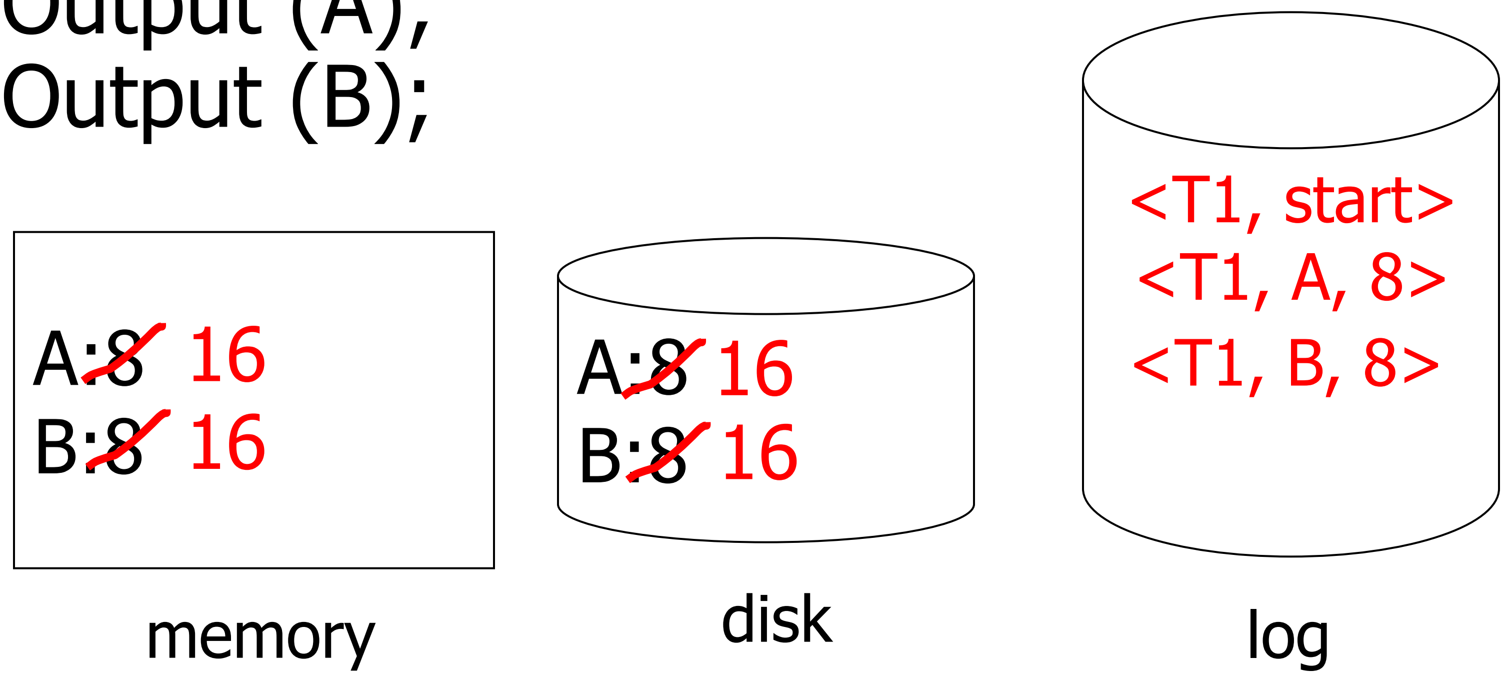
Undo logging (Immediate modification)

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);



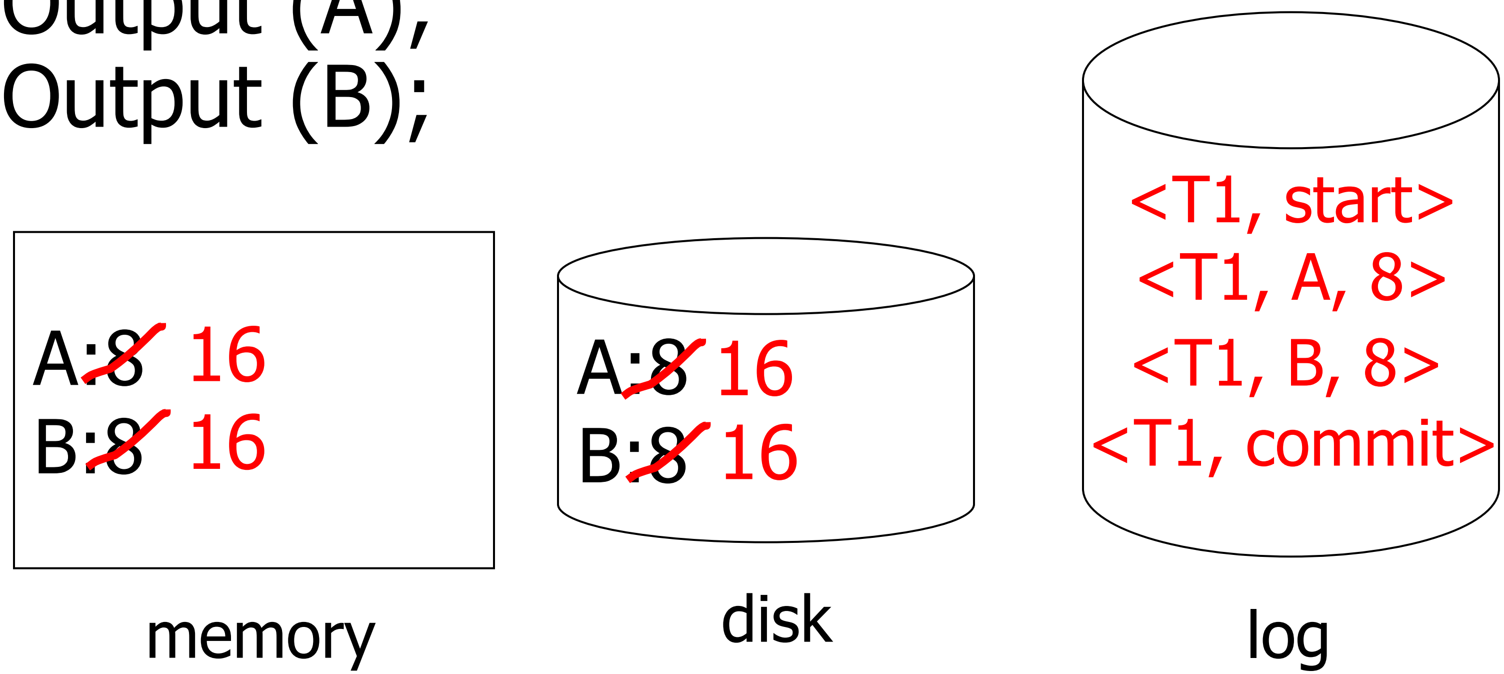
Undo logging (Immediate modification)

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);



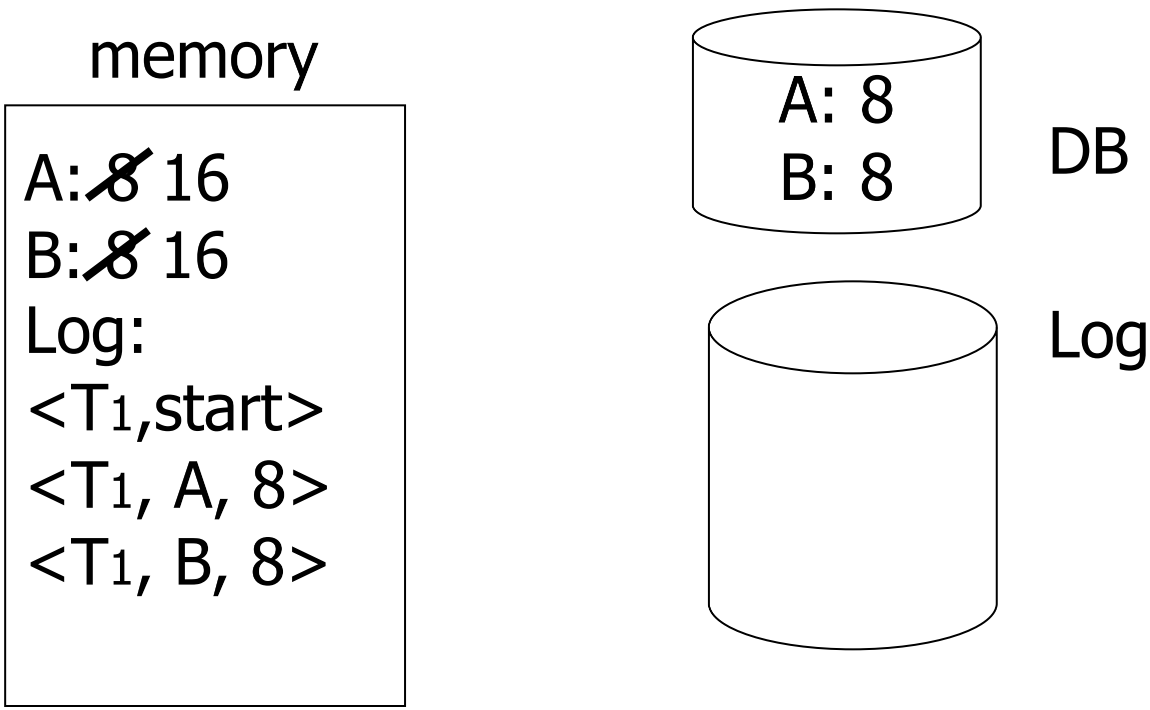
Undo logging (Immediate modification)

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);



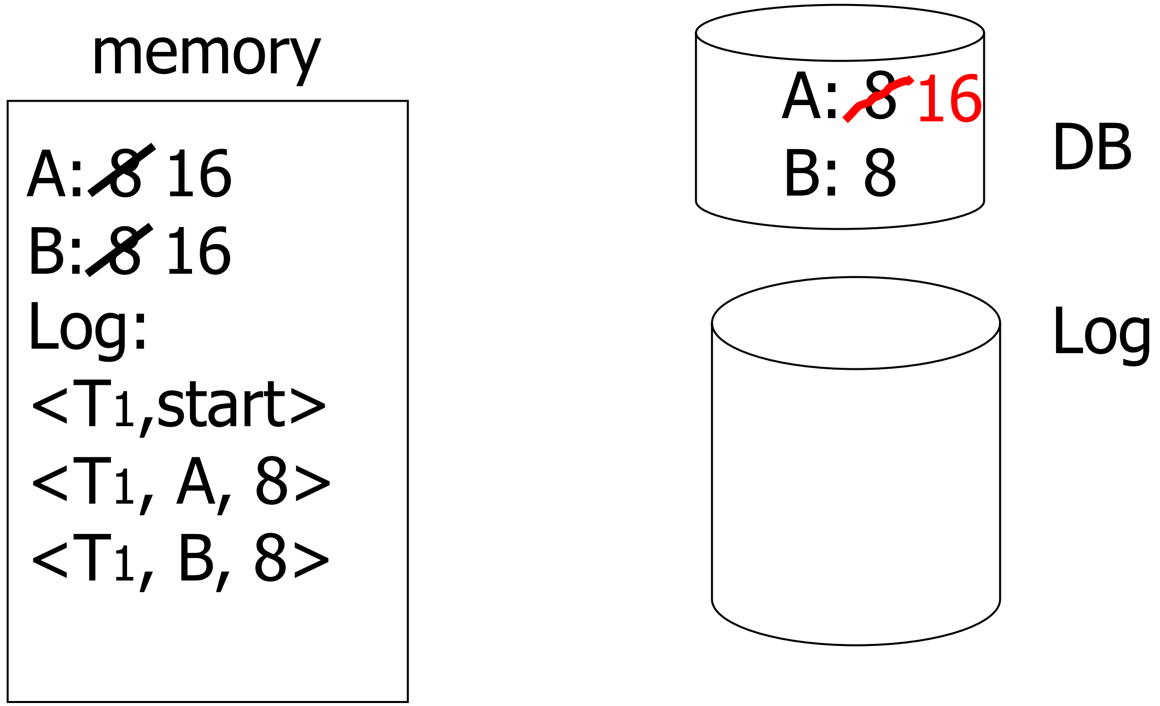
One “complication”

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



One "complication"

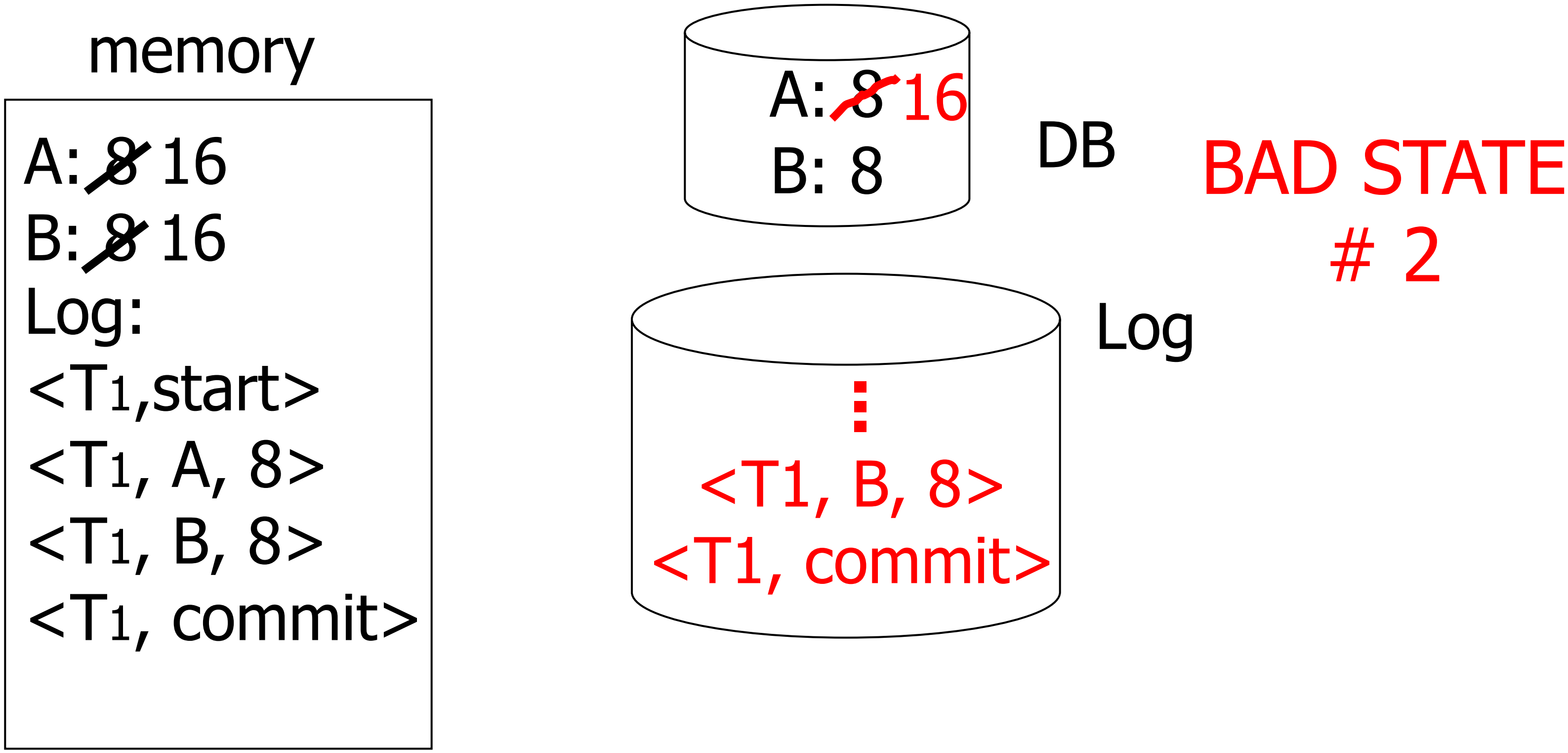
- Log is first written in memory
- Not written to disk on every action



BAD STATE
1

One “complication”

- Log is first written in memory
- Written to disk before 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

Must write to disk in the following order (UNDO LOG)

1. The log records indicating changed database elements.

2. The changed database elements themselves.

3. The COMMIT log record.

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						

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

<u>Recovery rules:</u> 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??

<u>Recovery rules:</u> 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 $\begin{cases} \text{- write (X, v)} \\ \text{- output (X)} \end{cases}$

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

- write <Ti, abort> to log (plus FLUSH LOG)

What if failure during recovery?

No problem! Undo idempotent

Recovery from 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 <COMMIT *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.*

Recovery from 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.

Recovery from 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 as in the previous case.*

Recovery from 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.

Recovery from 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.

Checkpoint

- 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

Checkpoint

- non-quiescent checkpoint

1. Write log record <START CKPT(T1, …Tk)>
   T1 … Tk are active transactions, and flush log.

2. Wait until all Ti-s commit or abort, but don't prohibit other transactions from starting.

3. When all Ti-s have completed, write a log record <END CKPT> and flush the log.

Example: what to do at recovery?

Undo log (disk):

…	<T1,A,16>	…	<T1,commit>	…	Checkpoint	…	<T2,B,17>	…	<T2,commit>	…	<T3,C,21>	

Crash

When we scan the log backwards

If we first meet an <END CKPT> record, then we know that all incomplete transactions began after the previous

<START CKPT (T1, ... ,Tk)> record.

We may thus scan backwards as far as the next <START CKPT>, and then stop; previous log is useless and may as well have been discarded.

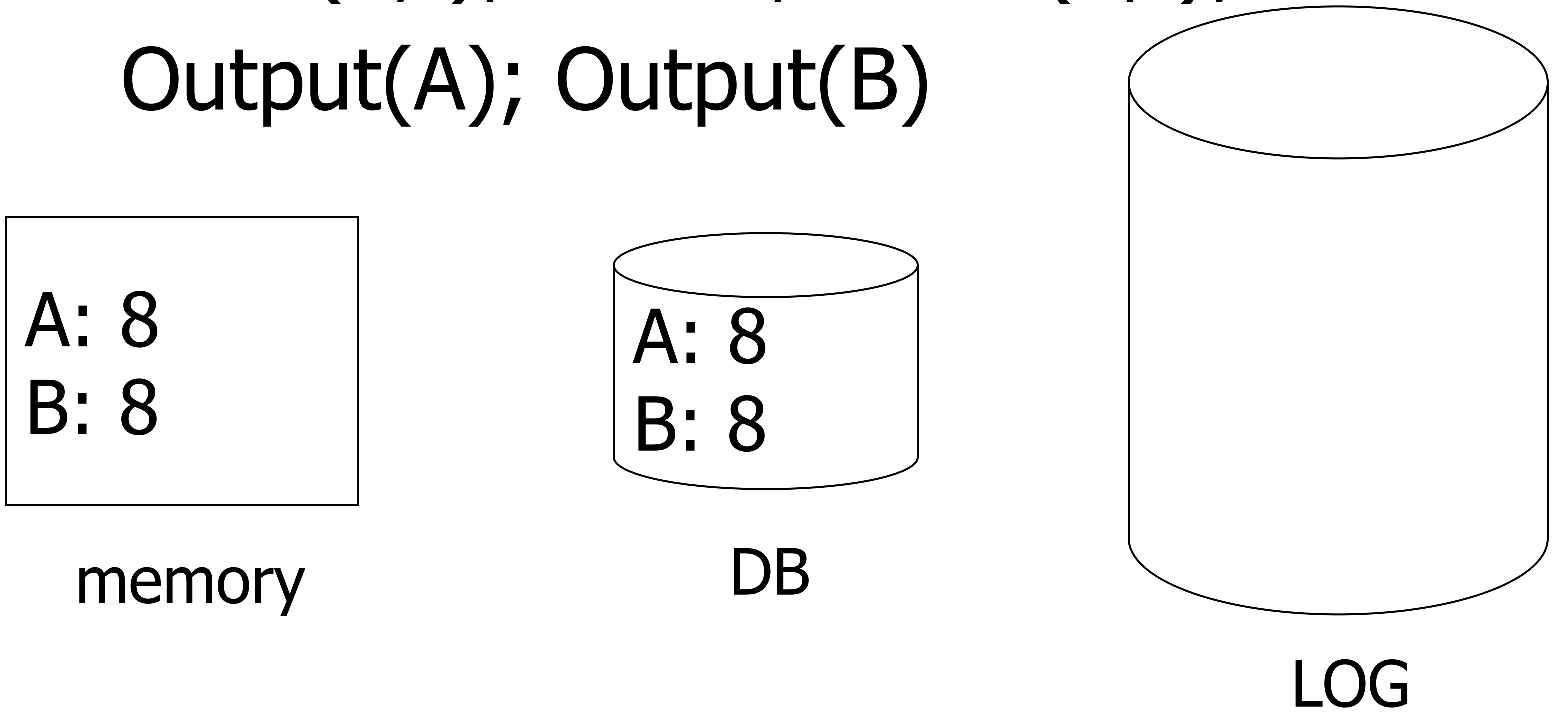
If we first meet a record < START CKPT (T1, ... , *Tk)>,* then the crash occurred during the checkpoint. We need scan no further back than the start of the earliest of these incomplete transactions.

To discuss:

- Redo logging
- Undo/redo logging, why both?
- Real world actions
- Checkpoints
- Media failures

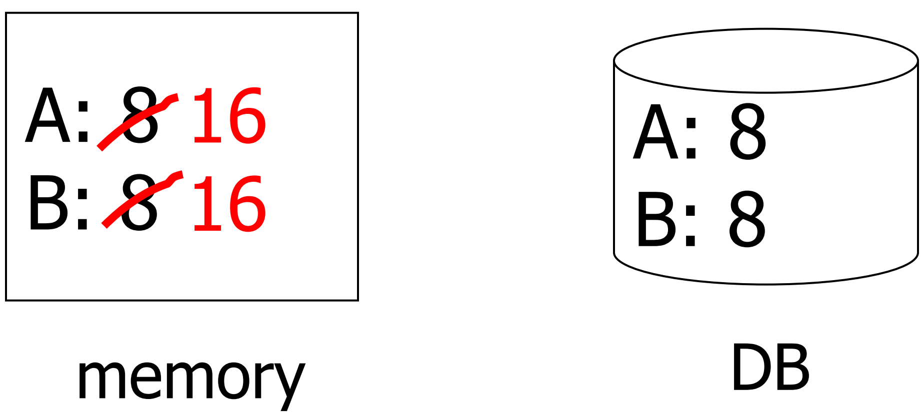
Redo logging (deferred modification)

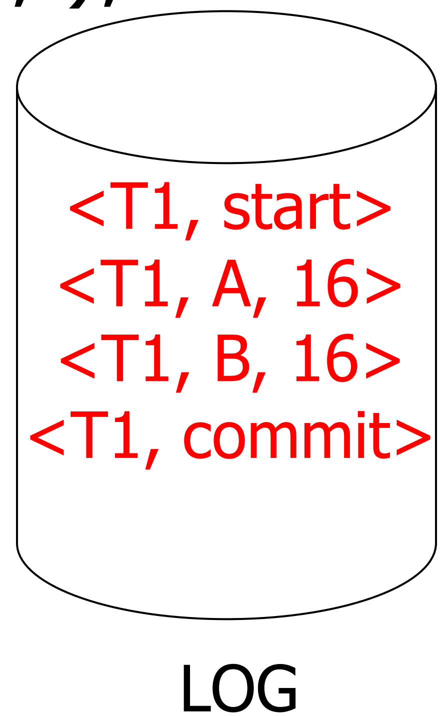
T_1: Read(A,t); $t \leftarrow t \times 2$; write (A,t);
Read(B,t); $t \leftarrow t \times 2$; write (B,t);
Output(A); Output(B)



Redo logging (deferred modification)

T1: 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)



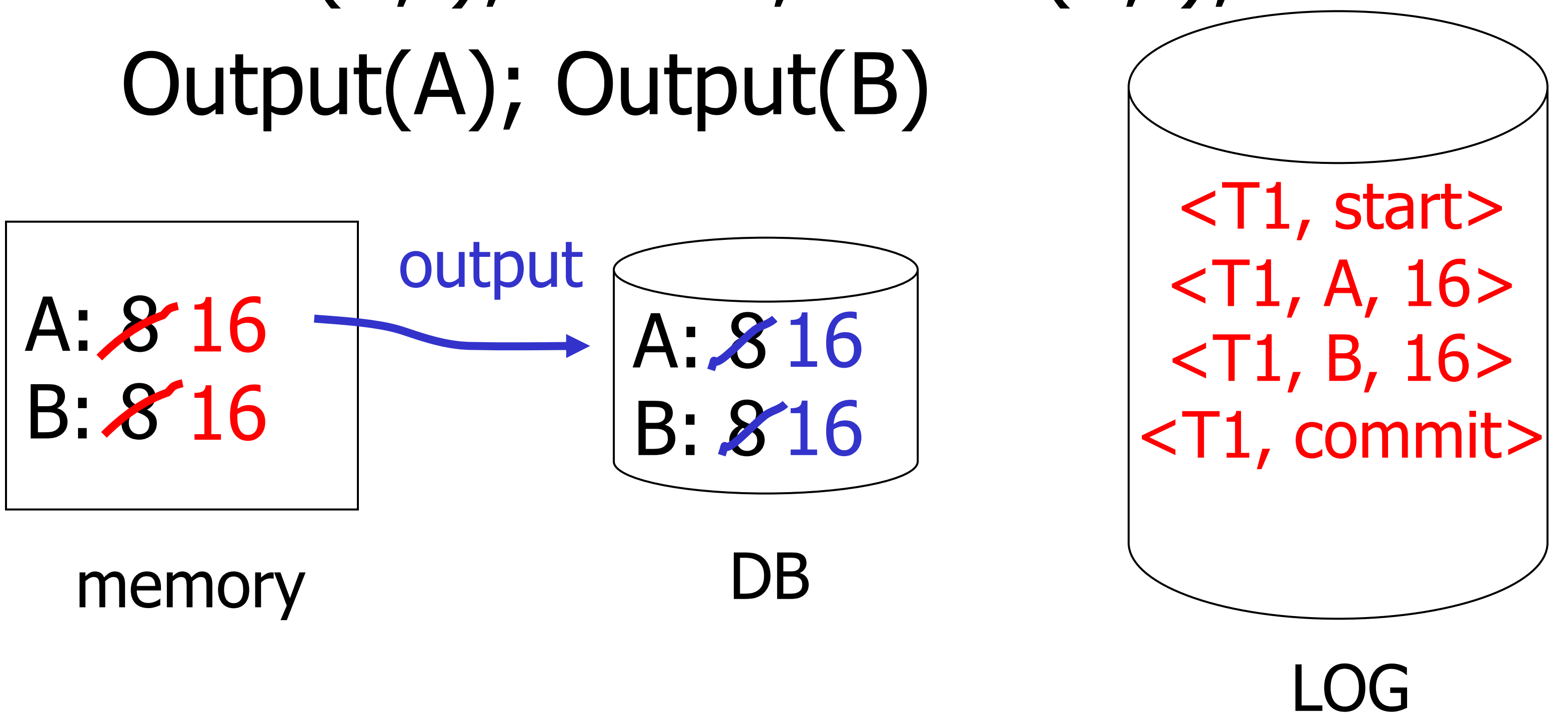


Redo logging (deferred modification)

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)

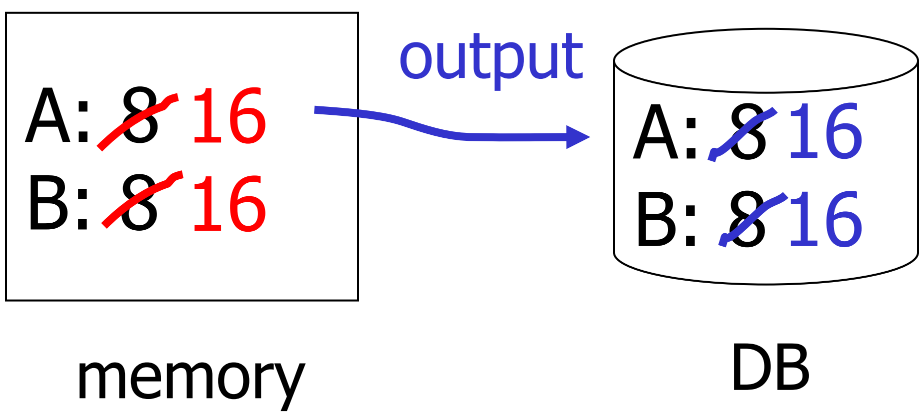


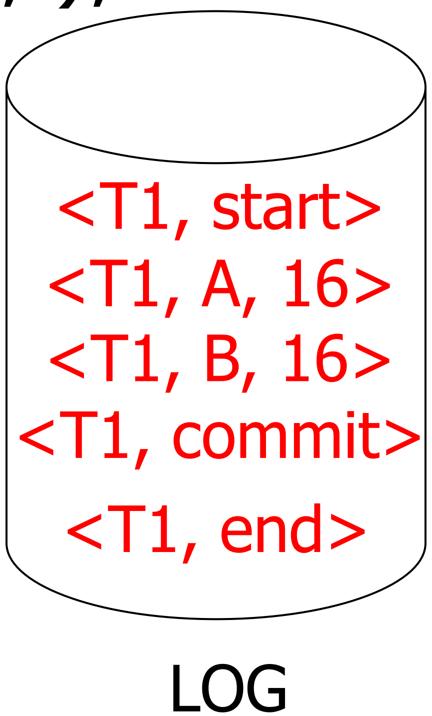
Redo logging (deferred modification)

T1: 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)





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

Must write to disk in the following order (REDO LOG)

1. The log records indicating changed database elements.

2. The COMMIT log record.

3. The changed database elements themselves.

REDO logging rules

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	
12)							<T,END>
13)	FLUSH LOG						

<u>Recovery rules:</u> 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 $\begin{cases} \text{Write(X, v)} \\ \text{Output(X)} \end{cases}$

(3) For each Ti $\in$ S, write <Ti, end> to Log (plus FLUSH LOG)

Modified REDO log

We don't use <Ti,end> record for completed transactions, but use <Ti,abort> for incomplete ones.

In the Textbook you find this modified version.

This way we use the same log records as in UNDO logging.

In the following we use this version.

This is synchronized with UNDO log.

Recovery rules: Modified Redo log

(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 $\begin{cases} \text{Write(X, v)} \\ \text{Output(X)} \end{cases}$

(3) For each Ti NOT in S, write <Ti, abort> to Log (plus FLUSH LOG)

Checkpoint

• non-quiescent checkpoint

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

   T1 … Tk are active (uncommitted) transactions, and flush log.

2. 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. (dirty buffers)
3. When all Ti-s have completed, write a log record <END CKPT> and flush the log.

Example: what to do at recovery?

Redo log (disk):

...	<T1,A,16>	...	<T1,commit>	...	Checkpoint	...	<T2,B,17>	...	<T2,commit>	...	<T3,C,21>	Crash

When we scan the log backwards

If we first meet an <END CKPT> record, we need to scan no further back than the earliest of <START Ti> records (among corresponding <START CKPT(T1, T2, … Tk)>.

If we first meet a record < START CKPT (T1, ... , *Tk)>,* then the crash occurred during the checkpoint.

We cannot be sure that committed transactions prior to the start of this checkpoint had their changes written to disk.

Thus, we must search back to the previous <END CKPT> record, find its matching < START CKPT (T1, ... , *Tk)>.*

Key drawbacks:

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

Solution: undo/redo logging!

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

Rules

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

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

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

Example: Undo/Redo logging what to do at recovery?

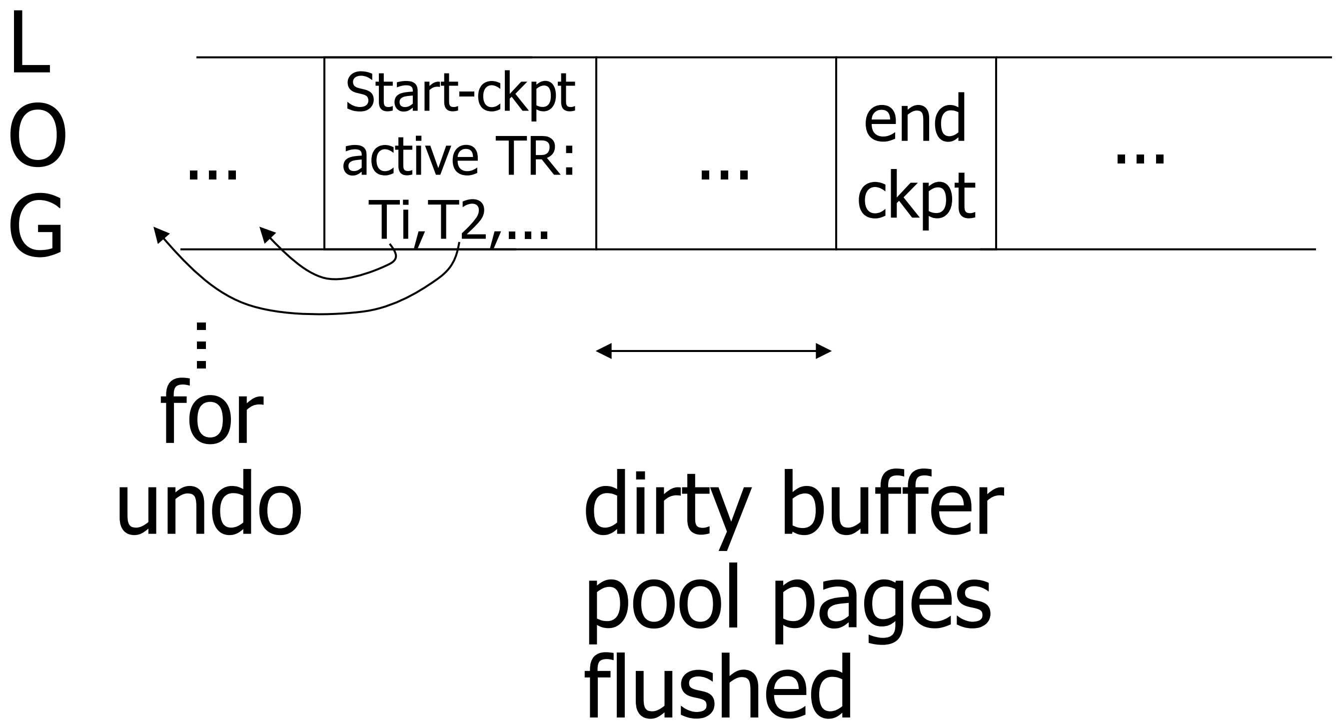
log (disk):

…	<checkpoint>	…	<T1, A, 10, 15>	…	<T1, B, 20, 23>	…	<T1, commit>	…	<T2, C, 30, 38>	…	<T2, D, 40, 41>	Crash

The undo/redo recovery policy is:

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

Non-quiescent checkpoint

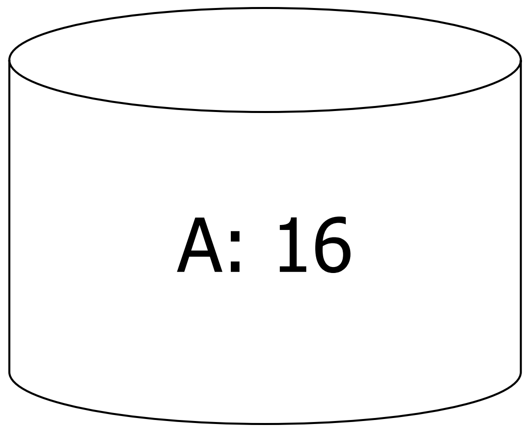


Checkpoint

- non-quiescent checkpoint

1. Write log record <START CKPT(T1, …Tk)>
   T1 … Tk are active (uncommitted) transactions, and flush log.
2. Write to disk all the buffers that are *dirty;* i.e., they contain one or more changed database elements. Unlike redo logging, we flush all dirty buffers, not just those written by committed transactions.
3. When all Ti-s have completed, write a log record <END CKPT> and flush the log.

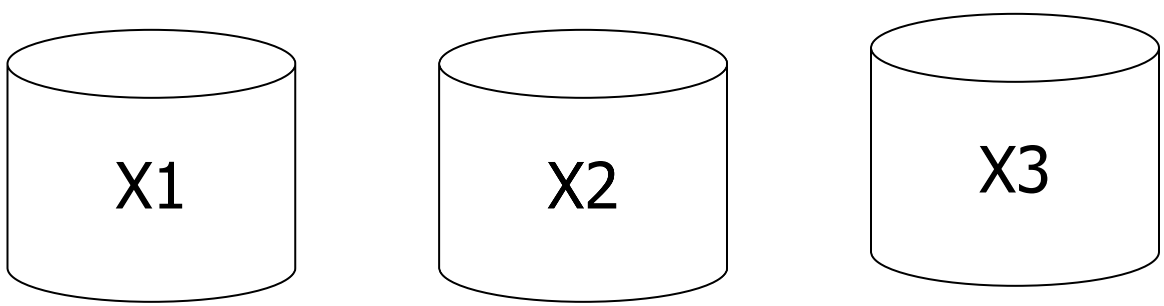
<u>Media failure</u> (loss of non-volatile storage)



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

Example 1 Triple modular redundancy

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

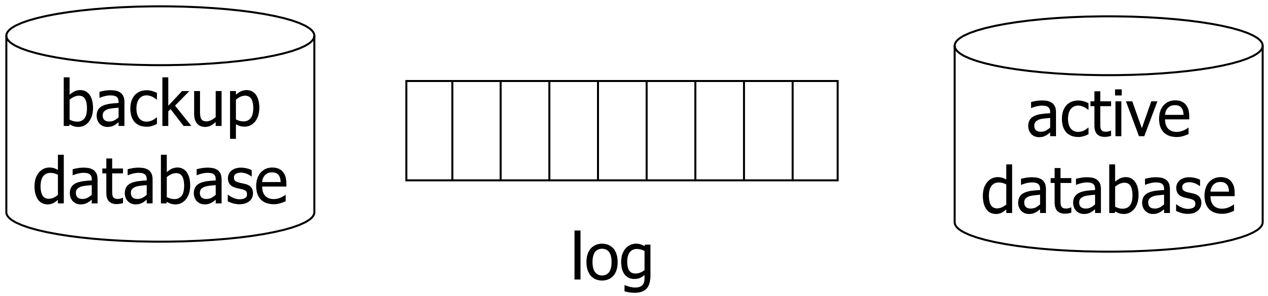


Example #2 Redundant writes, Single reads

- 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



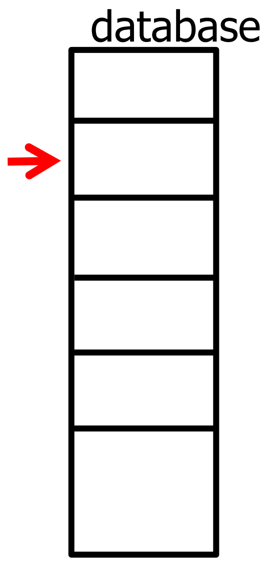
- If active database is lost,
  - restore active database from backup
  - bring up-to-date using redo entries in log

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]

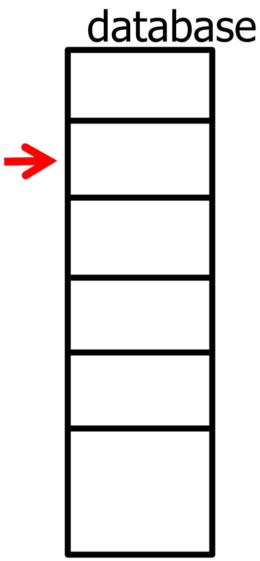


Backup Database

- Just like checkpoint,
  except that we write full database

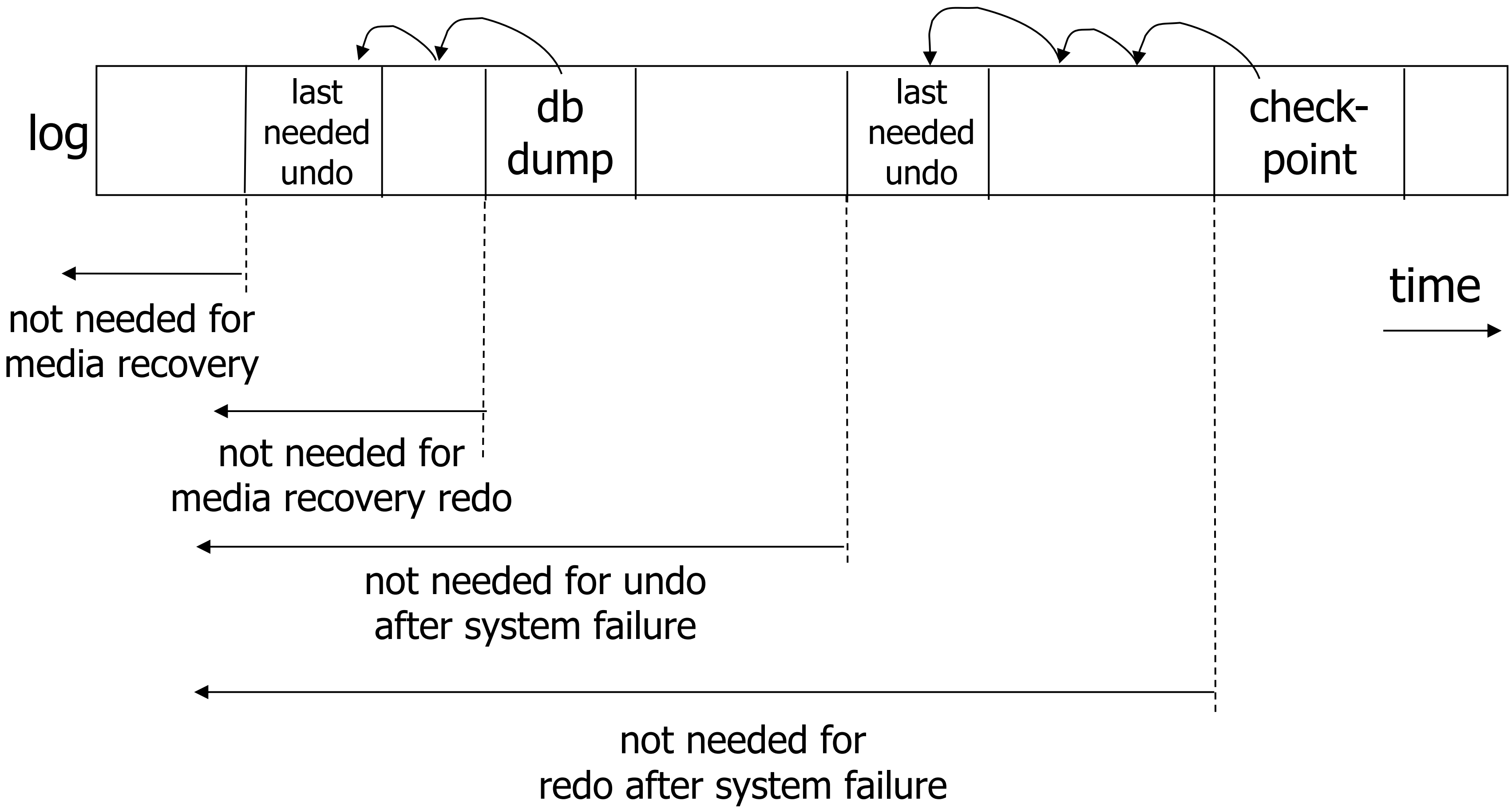
create backup database:
for i := 1 to DB_Size do
    [read DB block i; write to backup]

[transactions run concurrently]



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

When can log be discarded?



Summary

- Consistency of data
- One source of problems: failures
  - Logging
  - Redundancy
- Another source of problems:

  Data Sharing..... next