

Crash Recovery

Hector Garcia-Molina
Stijn Vansumneren

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

- Examples of predicates data must satisfy:
 - x is key of relation R
 - $x \rightarrow y$ holds in R
 - $\text{Domain}(x) = \{\text{Red}, \text{Blue}, \text{Green}\}$
 - 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

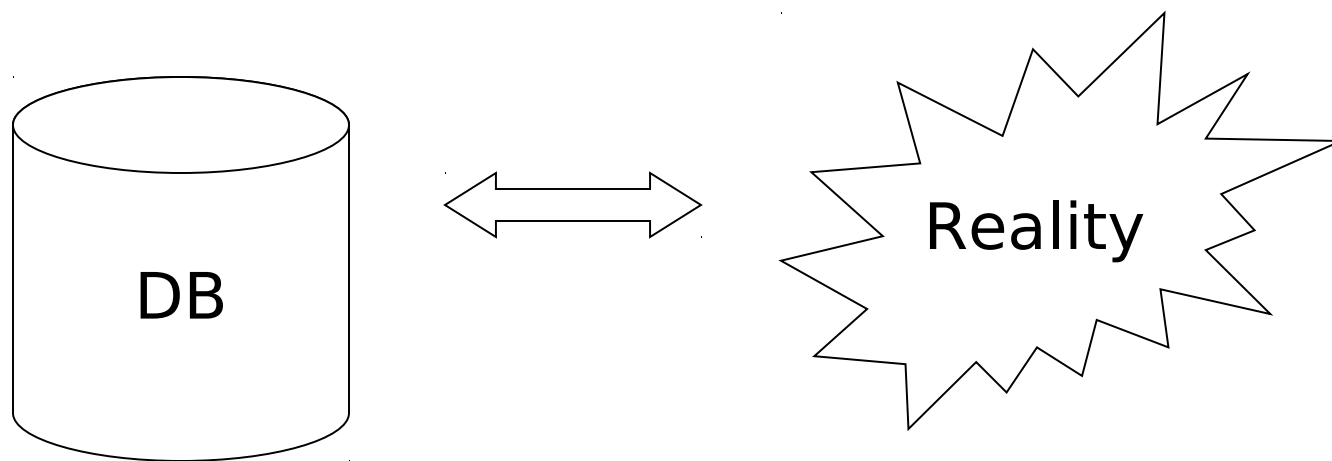
Constraints (as we use here) may
not capture “full correctness”

Example 1 Transaction constraints

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

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!

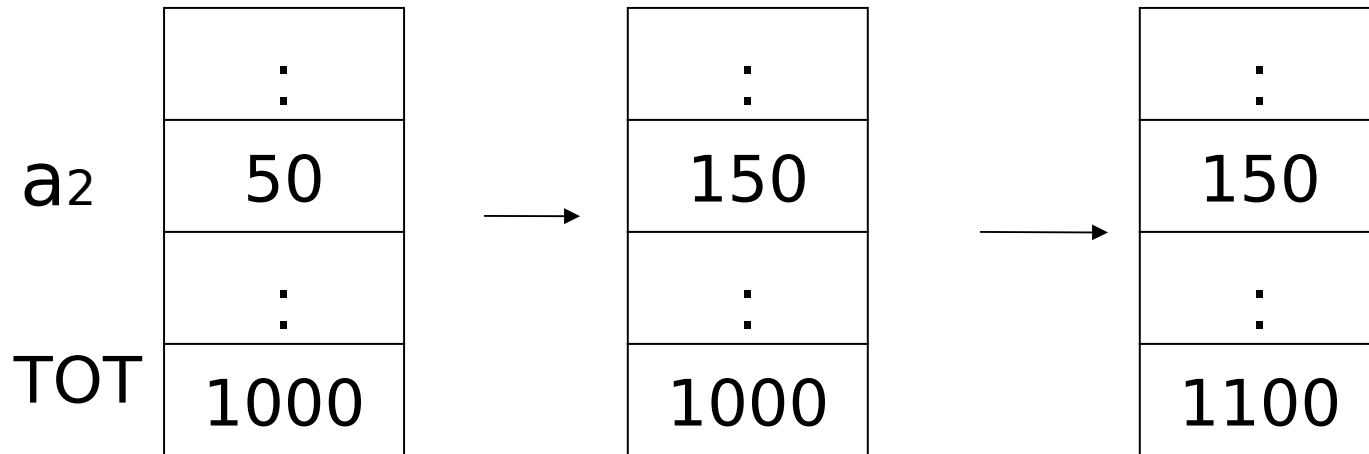
Example: $a_1 + a_2 + \dots + a_n = \text{TOT}$ (constraint)

Deposit \$100 in a_2 : $\left\{ \begin{array}{l} a_2 \leftarrow a_2 + 100 \\ \text{TOT} \leftarrow \text{TOT} + 100 \end{array} \right.$

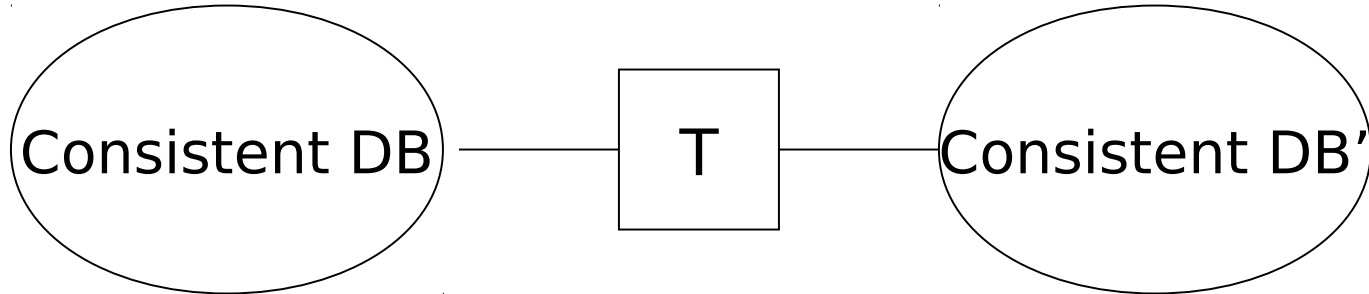
Example: $a_1 + a_2 + \dots + a_n = \text{TOT}$ (constraint)

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

$\text{TOT} \leftarrow \text{TOT} + 100$



Transaction: collection of actions
that preserve consistency



Big assumption:

If transaction T starts with consistent state + T executes in isolation
⇒ T leaves consistent state

Correctness (informally)

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

How can constraints be violated?

- Transaction bug
- DBMS bug
- 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

How can we prevent/fix violations?

- Chapter 17: due to failures only
- Chapter 18: due to data sharing only
- Chapter 19: due to failures and sharing

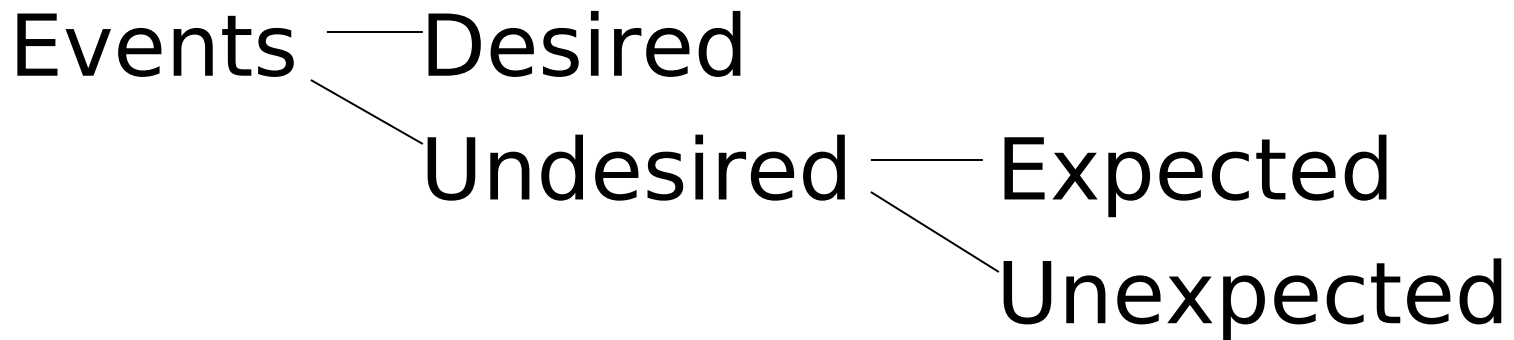
Will not consider:

- How to write correct transactions
- How to write correct DBMS
- Constraint checking & repair

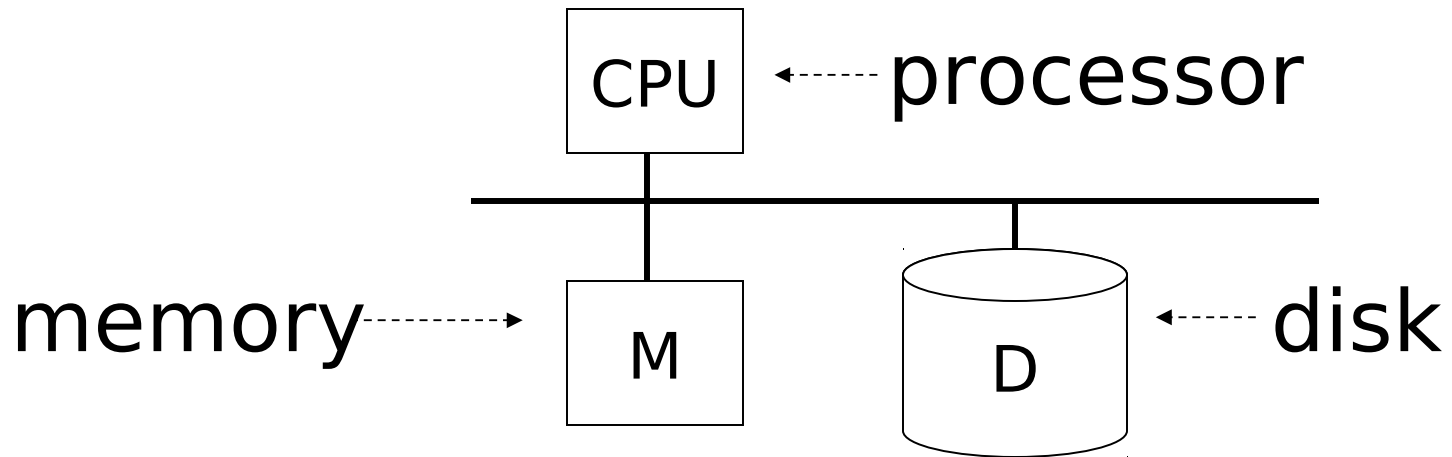
That is, solutions studied here do not need to know constraints

Crash Recovery

- First order of business:
Failure Model



Our failure model



Desired events: see product manuals....

Undesired expected events:

System crash

- memory lost
- cpu halts, resets

Desired events: see product manuals....

Undesired expected events:

System crash

- memory lost
- cpu halts, resets

that's it!!

Undesired Unexpected: Everything else!

Undesired Unexpected: Everything else!

Examples:

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

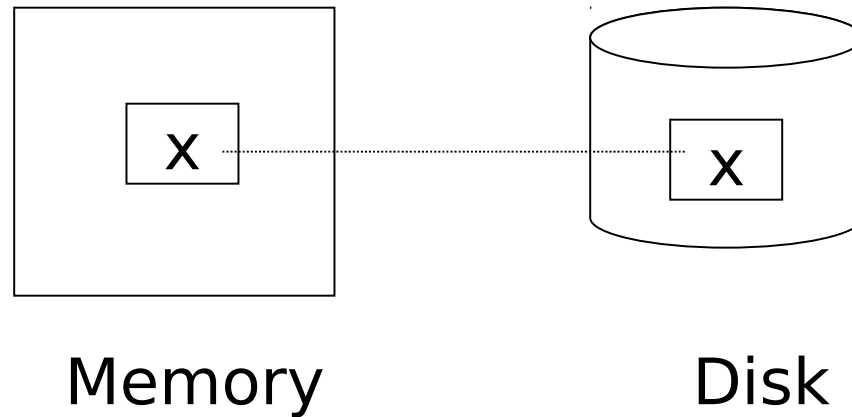
Is this model reasonable?

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

E.g., { Replicate disk storage (stable store)
Memory parity
CPU checks

Second order of business:

Storage hierarchy



Operations:

- Input (x): block containing x \rightarrow memory
- Output (x): block containing x \rightarrow disk

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

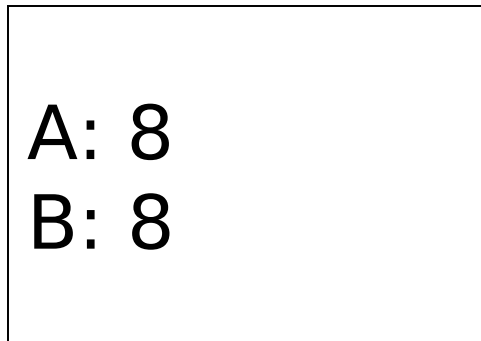
Key problem Unfinished transaction

Example

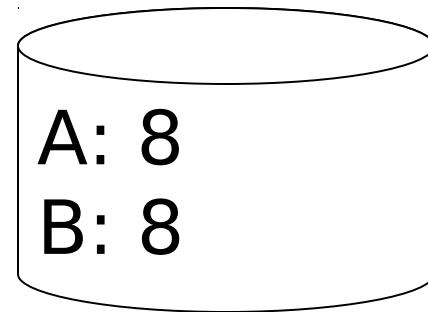
Constraint: $A=B$

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

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);

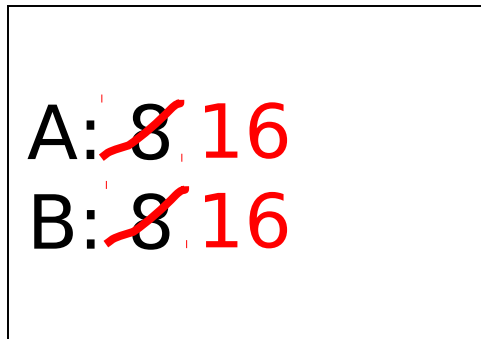


memory

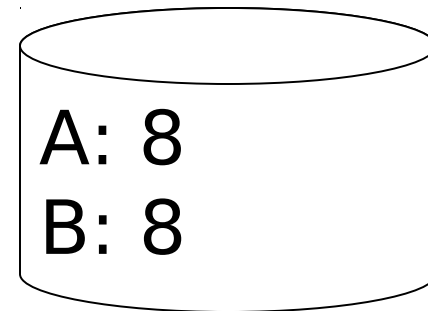


disk

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);

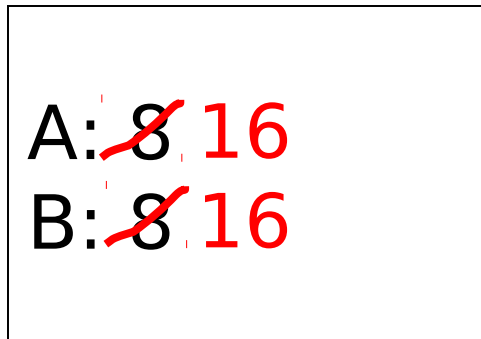


memory

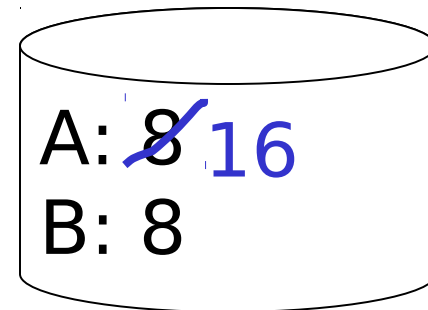


disk

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); **failure!**



memory



disk

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

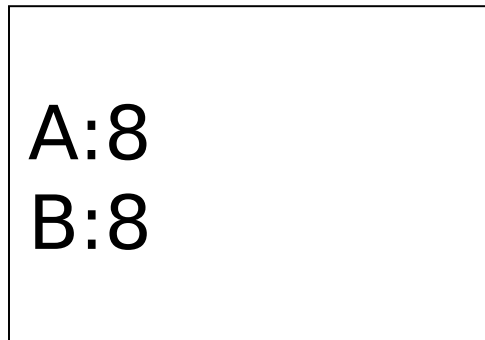
One solution: undo logging (immediate
modification)

essentially due to:

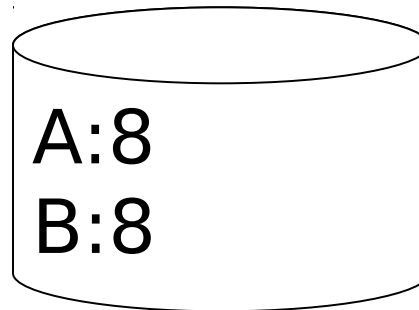
- Hansel and Gretel, 782 AD

Undo logging (Immediate modification)

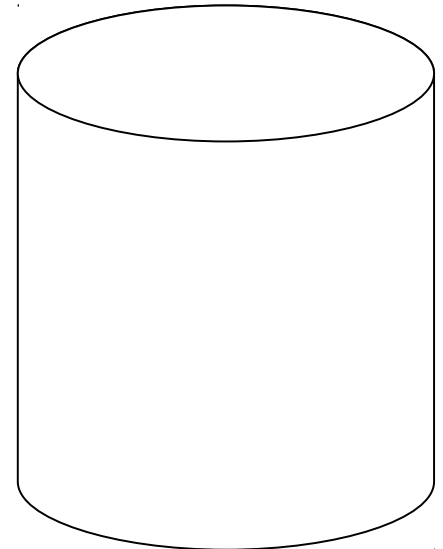
T1: Read (A,t); $t \leftarrow t \times 2$ $A=B$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



memory



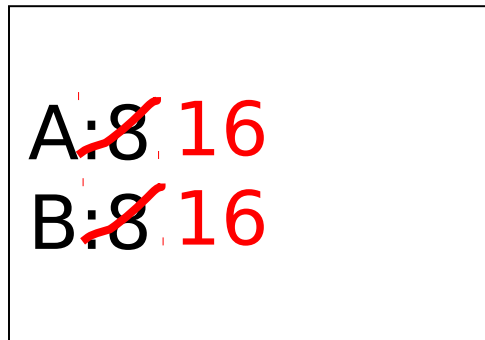
disk



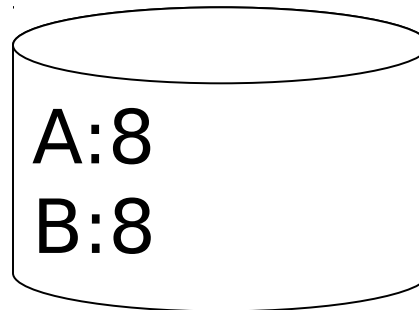
log

Undo logging (Immediate modification)

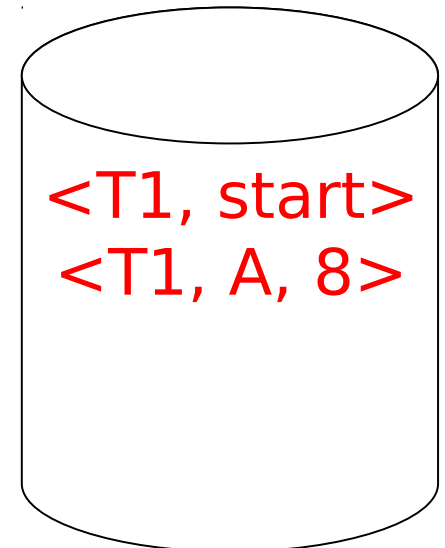
T1: Read (A,t); $t \leftarrow t \times 2$ $A=B$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



memory



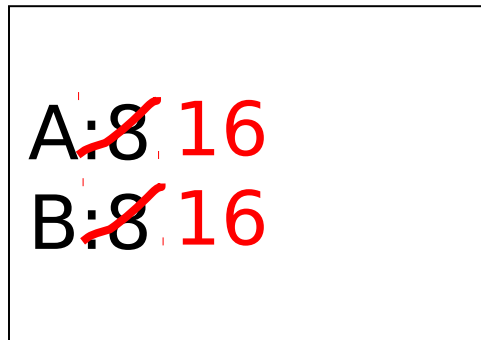
disk



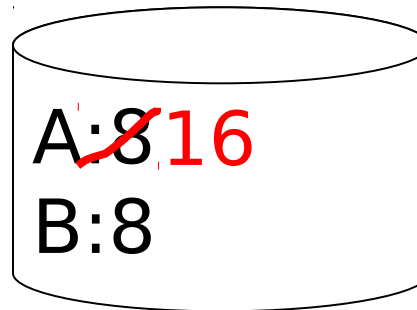
log

Undo logging (Immediate modification)

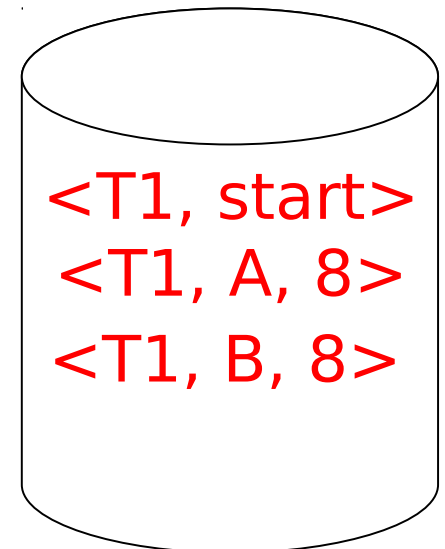
T1: Read (A,t); $t \leftarrow t \times 2$ $A=B$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



memory



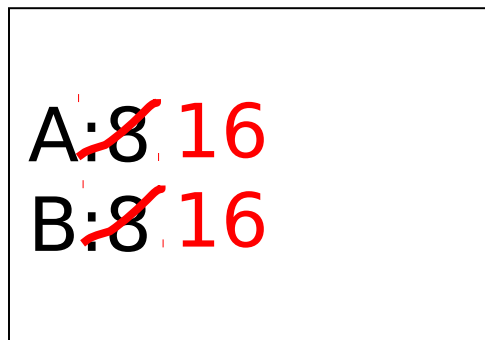
disk



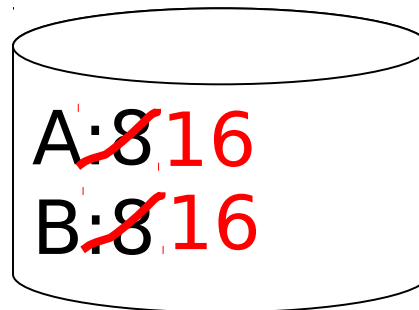
log

Undo logging (Immediate modification)

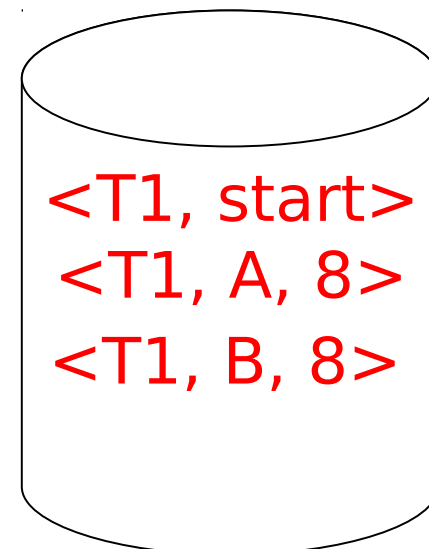
T1: Read (A,t); $t \leftarrow t \times 2$ $A=B$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



memory



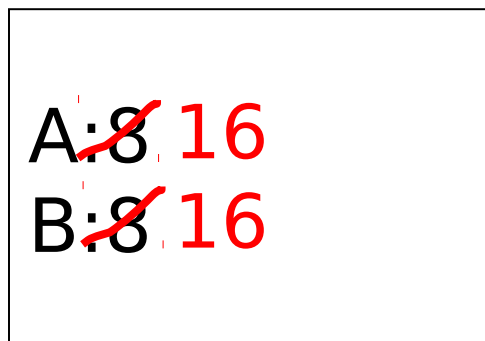
disk



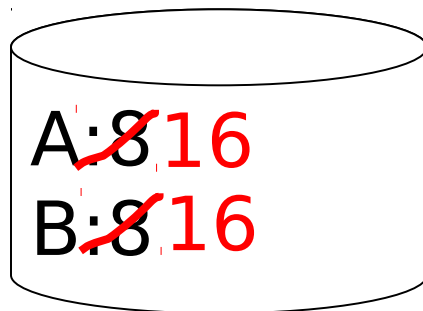
log

Undo logging (Immediate modification)

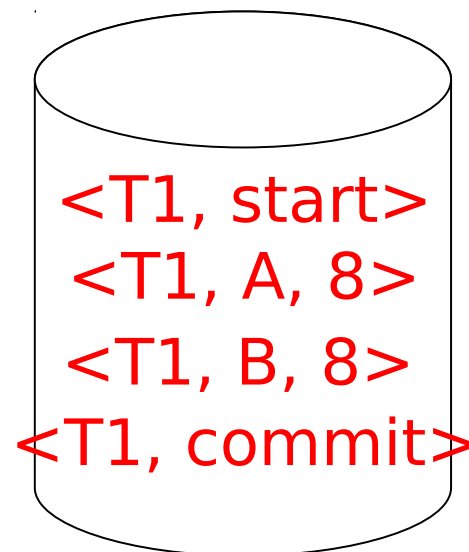
T1: Read (A,t); $t \leftarrow t \times 2$ $A=B$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



memory



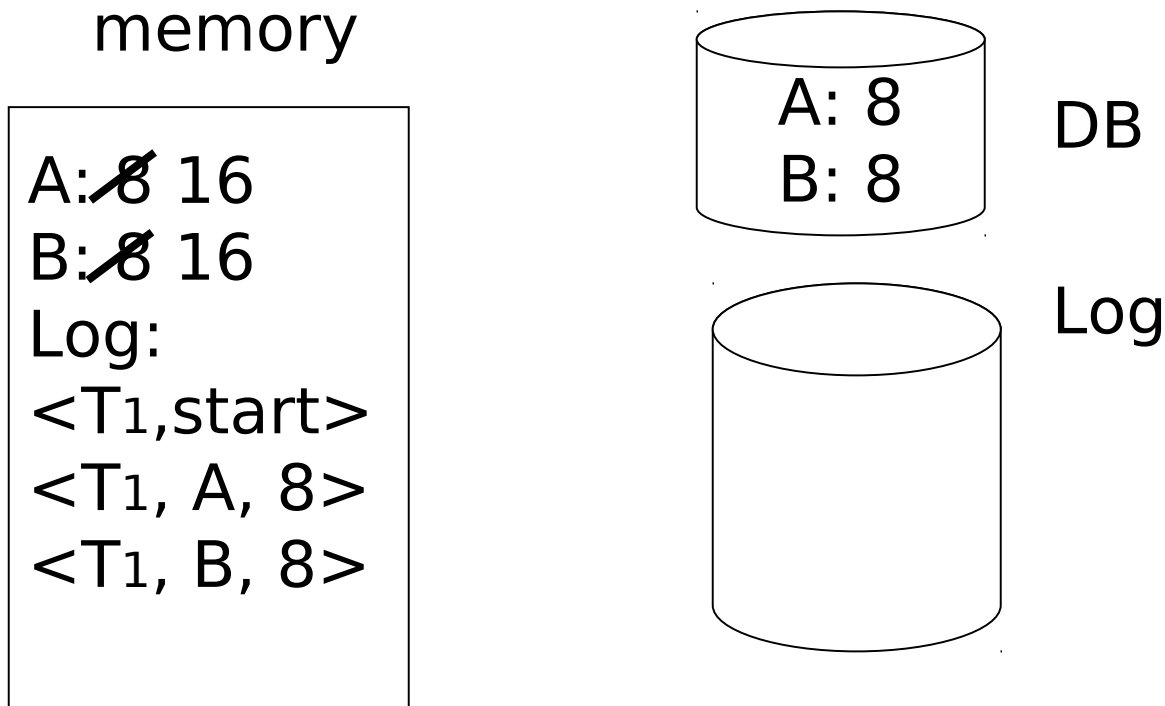
disk



log

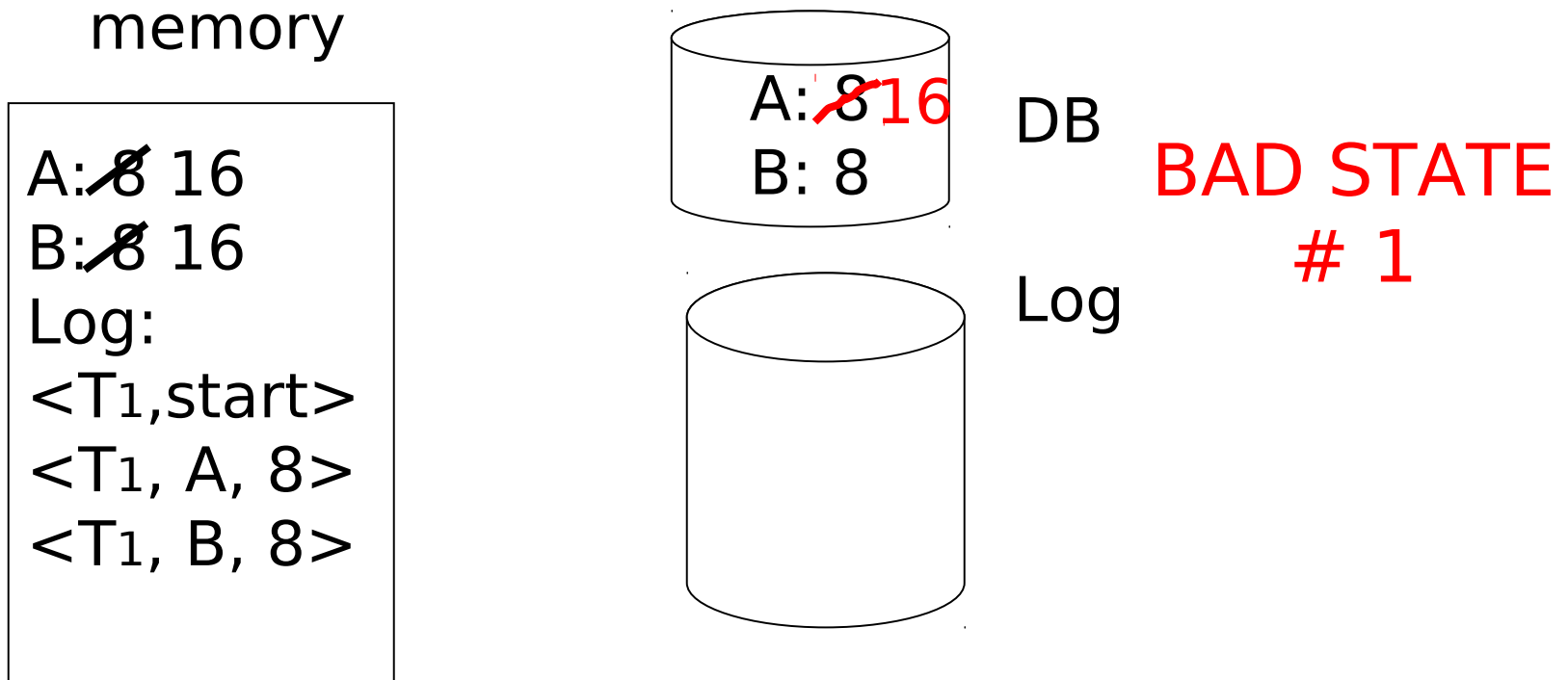
One “complication”

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



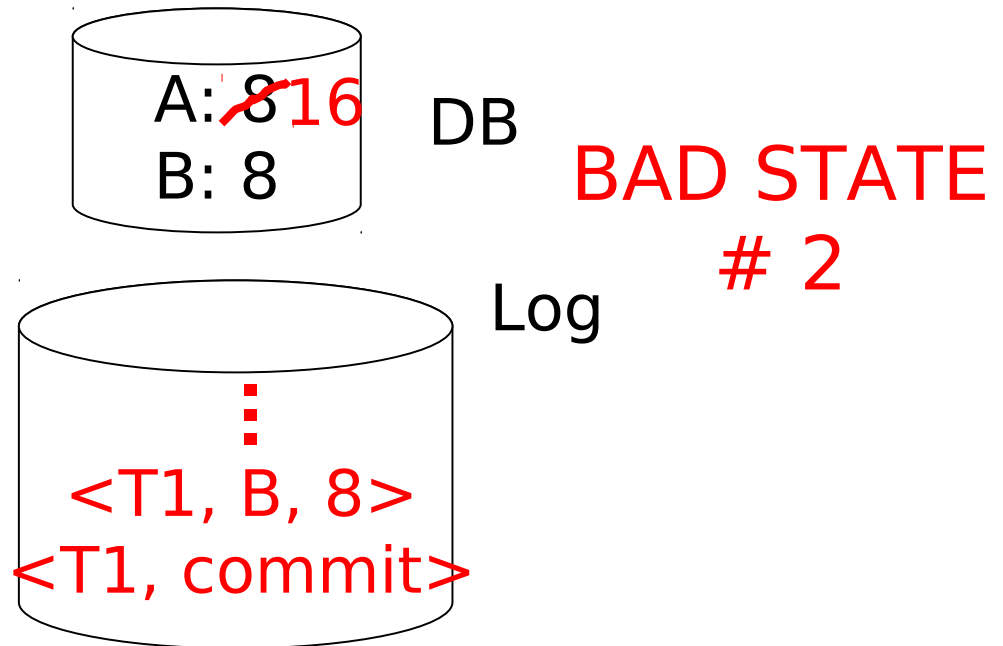
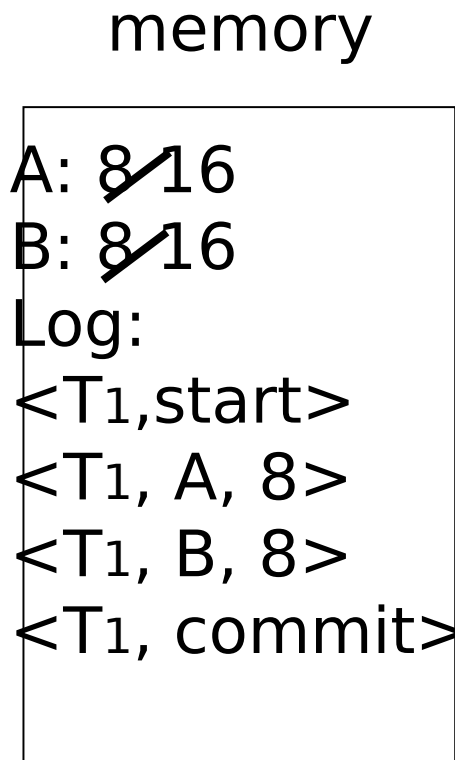
One “complication”

- 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

Recovery rules:

Undo logging

- For every transaction T_i with $\langle T_i, \text{start} \rangle$ in log:
 - If $\langle T_i, \text{commit} \rangle$ or $\langle T_i, \text{abort} \rangle$ in log:
do nothing
 - Else
 - For all $\langle T_i, X, v \rangle$ in log:
write (X, v)
output (X)
 - Write $\langle T_i, \text{abort} \rangle$ to log

Recovery rules:

Undo logging

- For every transaction T_i with $\langle T_i, \text{start} \rangle$ in log:
 - If $\langle T_i, \text{commit} \rangle$ or $\langle T_i, \text{abort} \rangle$ in log:
do nothing
 - Else
 - For all $\langle T_i, X, v \rangle$ in log:
write (X, v)
output (X)
 - Write $\langle T_i, \text{abort} \rangle$ to log
- IS THIS CORRECT??**

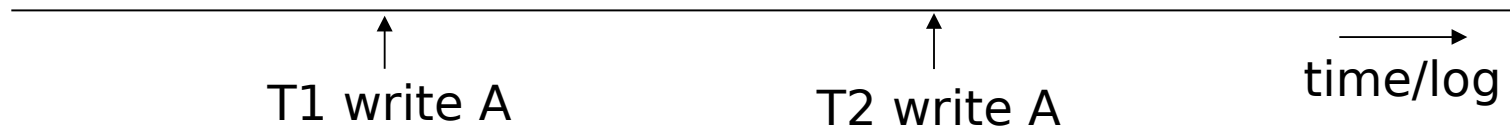
Recovery rules:

Undo logging

- (1) Let S = set of transactions with $\langle T_i, \text{start} \rangle$ in log, but no $\langle T_i, \text{commit} \rangle$ or $\langle T_i, \text{abort} \rangle$ record in log
- (2) For each $\langle T_i, X, v \rangle$ in log,
in reverse order (latest \rightarrow earliest) do:
 - if $T_i \in S$ then $\left\{ \begin{array}{l} - \text{write } (X, v) \\ - \text{output } (X) \end{array} \right.$
- (3) For each $T_i \in S$ do
 - write $\langle T_i, \text{abort} \rangle$ to log

Question

- Can writes of $\langle T_i, \text{abort} \rangle$ records be done in any order (in Step 3)?
 - Example: T1 and T2 both write A
 - T1 executed before T2
 - T1 and T2 both rolled-back
 - $\langle T1, \text{abort} \rangle$ written but NOT $\langle T2, \text{abort} \rangle$?
 - $\langle T2, \text{abort} \rangle$ written but NOT $\langle T1, \text{abort} \rangle$?



What if failure during recovery?

No problem! ▪ Undo idempotent

Can we truncate the log?

- Under a heavy transaction load, the log grows quickly
- Are there parts of the log that we can discard? (i.e. are there parts we know for sure won't be needed again?)
 - E.g., everything before a $\langle T_i, \text{commit} \rangle$?

Solution: (Simple) Checkpoint

Periodically:

- (1) Do not accept new transactions
- (2) Wait until all running transactions have finished and flushed their modifications to disk
- (3) Flush all log records to disk (log)
- (4) Write “checkpoint” record on disk (log)
- (5) Resume accepting transactions

An example undo log with simple checkpoint (disk)

<T1, start>

<T1, A, 5>

<T2, start>

<T2, B, 10>

<T2, C, 15>

<T1, D, 20>

<T1, commit>

<T2, commit>

<CKPT>

<T3, start>

<T3, E, 25>

<T3, F, 30>

 failure!

An example undo log with simple checkpoint (disk)

<T1, start>

<T1, A, 5>

<T2, start>

<T2, B, 10>

<T2, C, 15>

<T1, D, 20>

<T1, commit>

<T2, commit>

<CKPT>

<T3, start>

<T3, E, 25>

<T3, F, 30>

UNDO to latest checkpoint



failure!



An example undo log with simple checkpoint (disk)

<T1, start>
<T1, A, 5>
<T2, start>
<T2, B, 10>
<T2, C, 15>
<T1, D, 20>
<T1, commit>
<T2, commit>
<CKPT>
<T3, start>
<T3, E, 25>
<T3, F, 30>

This part can be removed from the log

UNDO to latest checkpoint

failure!



Non-quiescent checkpoint

Simple checkpoints effectively shut down the system while waiting for the open transactions to commit

Therefore, a more complex technique known as *nonquiescent checkpointing* is normally used, that allows new transactions to enter the system during the checkpoint

Solution: non-quiescent checkpoint

Periodically:

- (1) Write a log record $\langle \text{START CKPT } (T_1, \dots, T_k) \rangle$ and flush the log. $T_1 \dots T_k$ indentify the active transactions (not yet committed and written their changes to disk)
- (2) Wait until all of $T_1 \dots T_k$ commit or abort, but do not prohibit other transactions form starting
- (3) When all of $T_1 \dots T_k$ have completed, write $\langle \text{END CKPT} \rangle$ to log on disk (log)

An example undo log with nonquiescent checkpoint (disk)

<T1, start>

<T1, A, 5>

<T2, start>

<T2, B, 10>

<START CKPT (T1,T2)>

<T2, C, 15>

<START T3>

<T1, D, 20>

<T1, commit>

<T3, E, 25>

<T2, commit>

<END CKPT>

<T3, F, 30>

failure!

An example undo log with nonquiescent checkpoint (disk)

<T1, start>

<T1, A, 5>

<T2, start>

<T2, B, 10>

<START CKPT (T1,T2)>

<T2, C, 15>

<START T3>

<T1, D, 20>

<T1, commit>

<T3, E, 25>

<T2, commit>

<END CKPT>

<T3, F, 30>

UNDO to latest
start checkpoint

failure!

An example undo log with nonquiescent checkpoint (disk)

<T1, start>

<T1, A, 5>

<T2, start>

<T2, B, 10>

<START CKPT (T1,T2)>

<T2, C, 15>

<START T3>

<T1, D, 20>

<T1, commit>

<T3, E, 25>

<T2, commit>

<END CKPT>

<T3, F, 30>

This part can be removed from the log

UNDO to latest start checkpoint

failure!

An example undo log with nonquiescent checkpoint (disk)

<T1, start>

<T1, A, 5>

<T2, start>

<T2, B, 10>

<START CKPT (T1,T2)>

<T2, C, 15>

<START T3>

<T1, D, 20>

<T1, commit>

<T3, E, 25>

 failure!

An example undo log with nonquiescent checkpoint (disk)

<T1, start>

<T1, A, 5>

<T2, start>

<T2, B, 10>

<START CKPT (T1,T2)>

<T2, C, 15>

<START T3>

<T1, D, 20>

<T1, commit>

<T3, E, 25>




UNDO to latest start checkpoint

failure!

An example undo log with nonquiescent checkpoint (disk)

<T1, start>
<T1, A, 5>
<T2, start>
<T2, B, 10>
<START CKPT (T1,T2)>
<T2, C, 15>
<START T3>
<T1, D, 20>
<T1, commit>
<T3, E, 25>



UNDO to latest
COMPLETED
start checkpoint

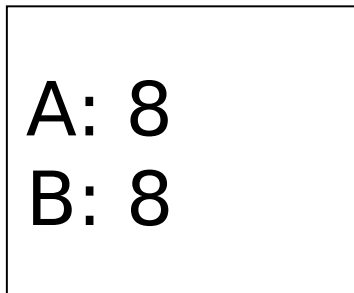
failure!

To discuss:

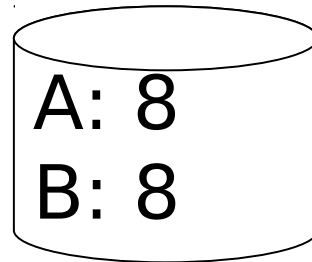
- Redo logging
- Undo/redo logging, why both?
- Real world actions
- Media failures

Redo logging (deferred modification)

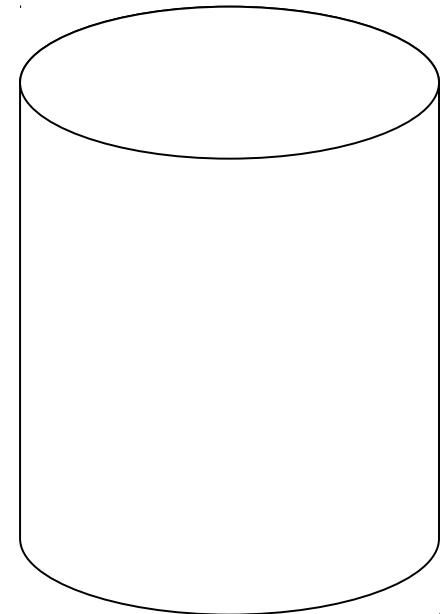
T1: Read(A,t); $t \leftarrow t \times 2$; write (A,t);
Read(B,t); $t \leftarrow t \times 2$; write (B,t);
Output(A); Output(B)



memory



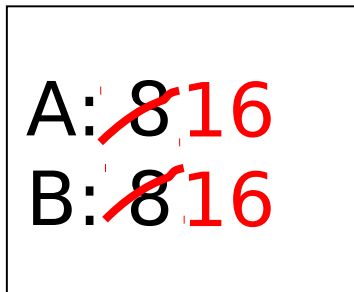
DB



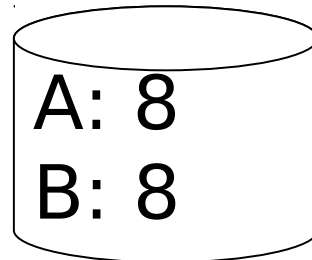
LOG

Redo logging (deferred modification)

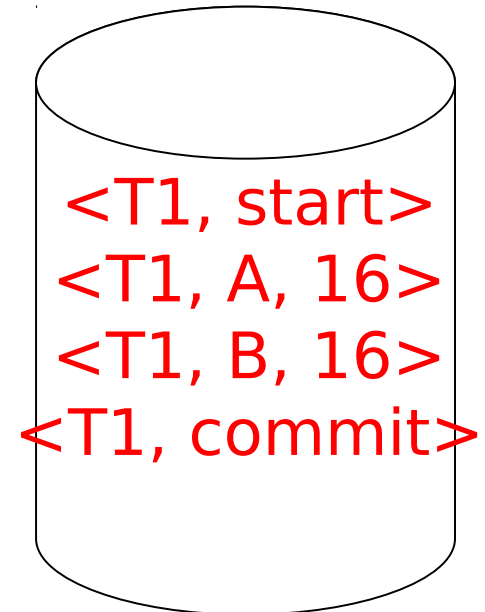
T₁: Read(A,t); t ← t×2; write (A,t);
Read(B,t); t ← t×2; write (B,t);
Output(A); Output(B)



memory



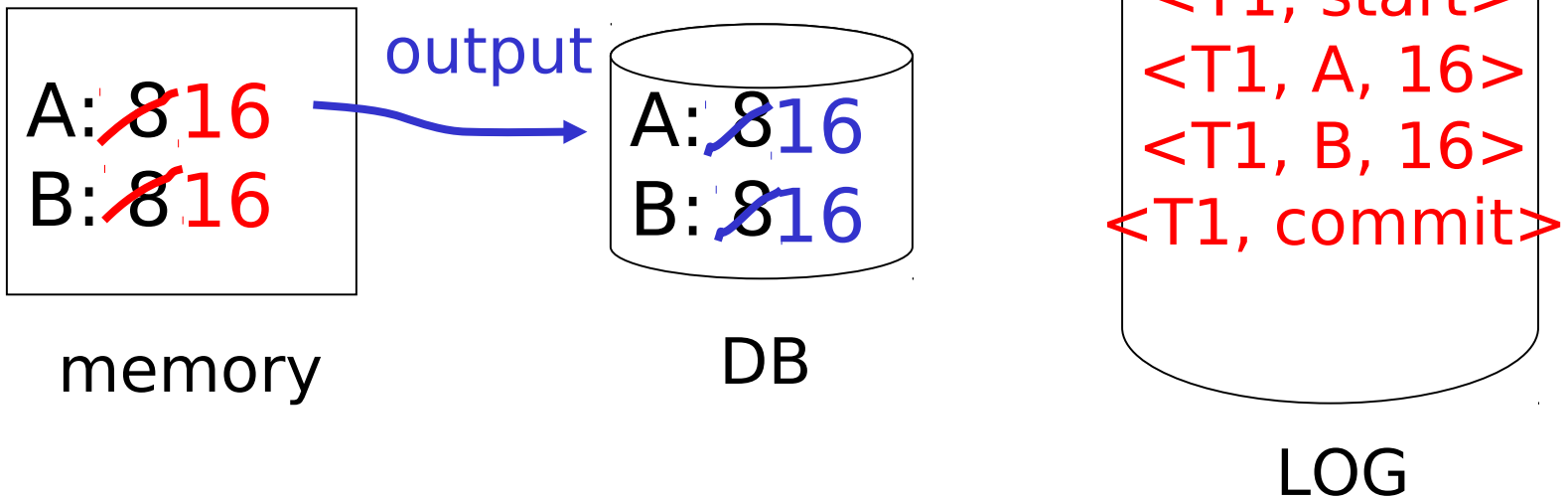
DB



LOG

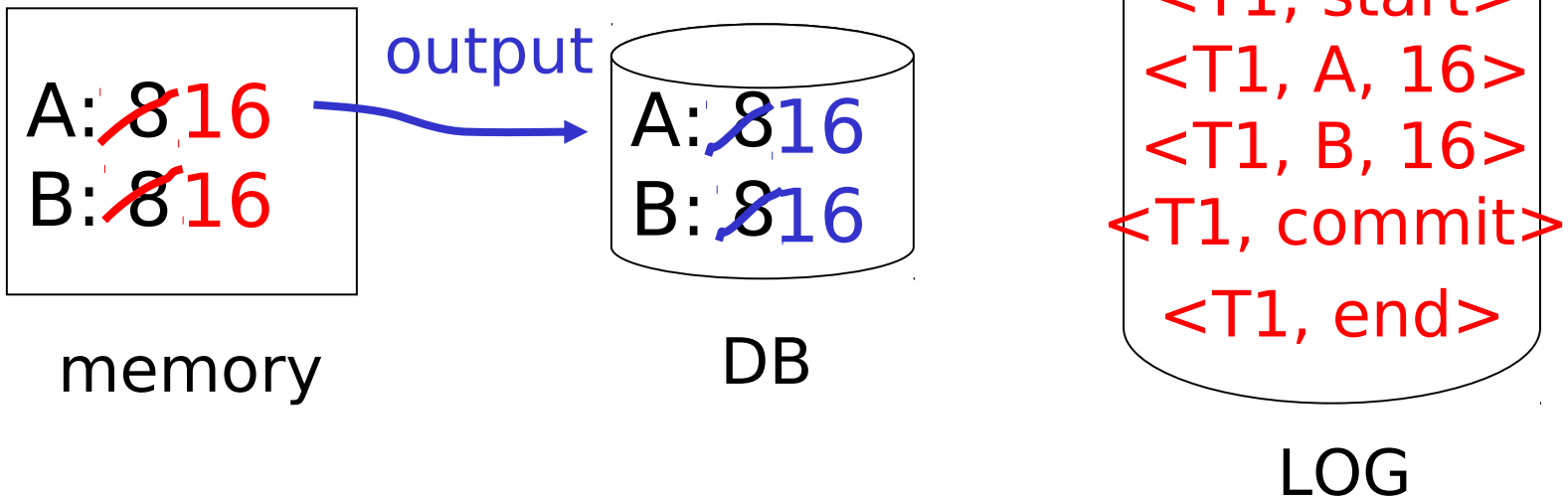
Redo logging (deferred modification)

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)

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

Recovery rules:

Redo logging

- For every T_i with $\langle T_i, \text{commit} \rangle$ in log:
 - For all $\langle T_i, X, v \rangle$ in log:
 - { Write(X, v)
 - Output(X)

Recovery rules:

Redo logging

- For every T_i with $\langle T_i, \text{commit} \rangle$ in log:
 - For all $\langle T_i, X, v \rangle$ in log:
 - { Write(X, v)
 - Output(X)

IS THIS CORRECT??

Recovery rules:

Redo logging

- (1) Let S = set of transactions with $\langle T_i, \text{commit} \rangle$ (and no $\langle T_i, \text{end} \rangle$) in log
- (2) For each $\langle T_i, X, v \rangle$ in log, in forward order (earliest \rightarrow latest) do:
 - if $T_i \in S$ then $\left\{ \begin{array}{l} \text{Write}(X, v) \\ \text{Output}(X) \end{array} \right.$
- (3) For each $T_i \in S$, write $\langle T_i, \text{end} \rangle$

Non-quietescent checkpointing a redo log

Periodically:

- (1) Write a log record $\langle \text{START CKPT } (T_1, \dots, T_k) \rangle$ where T_1, \dots, T_k are all the active (uncommitted) transactions, and flush the log.
- (2) Write to disk all database elements written to buffers but not yet to disk by transactions that had already committed when the start ckpt record was written to the log
- (3) Write the $\langle \text{END CKPT} \rangle$ record and flush the log

An example redo log with nonquiescent checkpoint (disk)

<T1, start>
<T1, A, 5>
<T2, start>
<T1, commit>
<T2, B, 10>
<START CKPT (T2)>
<T2, C, 15>
<START T3>
<T3, D, 20>
<T1, end>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

failure!

An example redo log with nonquiescent checkpoint (disk)

<T1, start>

<T1, A, 5>

<T2, start>

<T1, commit>

<T2, B, 10>

<START CKPT (T2)>

<T2, C, 15>

<START T3>

<T3, D, 20>

<T1, end>

<END CKPT>

<COMMIT T2>

<COMMIT T3>

REDO all committed transactions
that were active (uncommitted)
when the checkpoint began,
or started later:
T2 and T3

failure!

An example redo log with nonquiescent checkpoint (disk)

<T1, start>
<T1, A, 5>
<T2, start>
<T1, commit>
<T2, B, 10>
<START CKPT (T2)>
<T2, C, 15>
<START T3>
<T3, D, 20>
<T1, end>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

failure!

An example redo log with nonquiescent checkpoint (disk)

<T1, start>

<T1, A, 5>

<T2, start>

<T1, commit>

<T2, B, 10>

<START CKPT (T2)>

<T2, C, 15>

<START T3>

<T3, D, 20>

<T1, end>

<END CKPT>

<COMMIT T2>

<COMMIT T3>

REDO all committed transactions
that were active (uncommitted)
when the checkpoint began,
or started later:
Only T2

failure!

An example redo log with nonquiescent checkpoint (disk)

<T1, start>

<T1, A, 5>

<T2, start>

<T1, commit>

<T2, B, 10>

<START CKPT (T2)>

<T2, C, 15>

<START T3>

<T3, D, 20>

<T1, end>

<END CKPT>

<COMMIT T2>

<COMMIT T3>

failure!

An example redo log with nonquiescent checkpoint (disk)

<T1, start>

<T1, A, 5>

<T2, start>

<T1, commit>

<T2, B, 10>

<START CKPT (T2)>

<T2, C, 15>

<START T3>

<T3, D, 20>

<T1, end>

<END CKPT>

<COMMIT T2>

<COMMIT T3>



REDO until the previous
complete

<START CKPT>

(or to the beginning of the log)

failure!

Note:

- In the presence of non-quietuscent checklogging, the $\langle T_i, \text{end} \rangle$ log records are redundant (the checkpoint gives the same information). The book hence does **not** use such log records.
- The exercises do **not** use such records

Key drawbacks:

- *Undo logging*: cannot bring backup DB copies up to date
- *Redo logging*: need to keep all modified blocks in memory until commit

Solution: undo/redo logging!

Update \Rightarrow $\langle T_i, X_{id}, \text{New } X \text{ val}, \text{Old } X \text{ val} \rangle$
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)

Non-quiescent checkpointing an undo/redo log

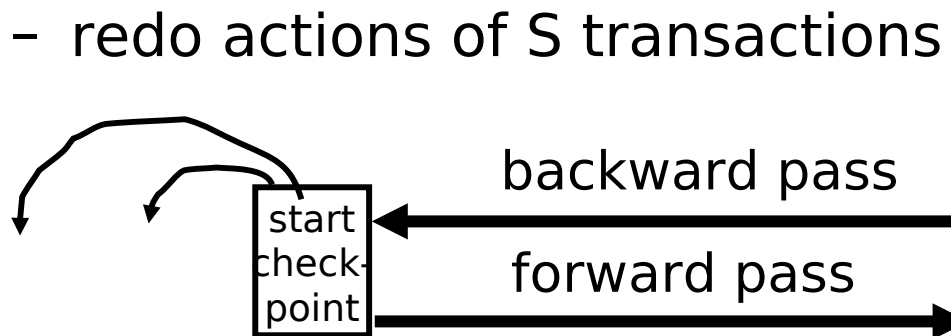
Periodically:

- (1) Write a log record $\langle \text{START CKPT } (T_1, \dots, T_k) \rangle$ where T_1, \dots, T_k are all the active (uncommitted) transactions, and flush the log.
- (2) Write to disk all buffers that are dirty, i.e., they contain one or more changed database elements.
- (3) Write the $\langle \text{END CKPT} \rangle$ record and flush the log

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 valid checkpoint start -> end of log)



Real world actions

E.g., dispense cash at ATM

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




\$

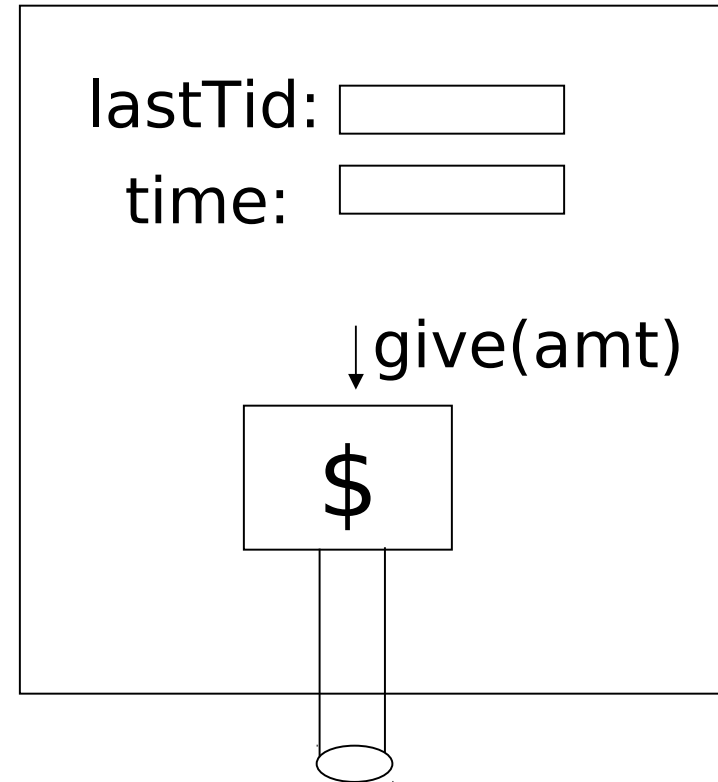
Solution

- (1) execute real-world actions after commit
- (2) try to make idempotent

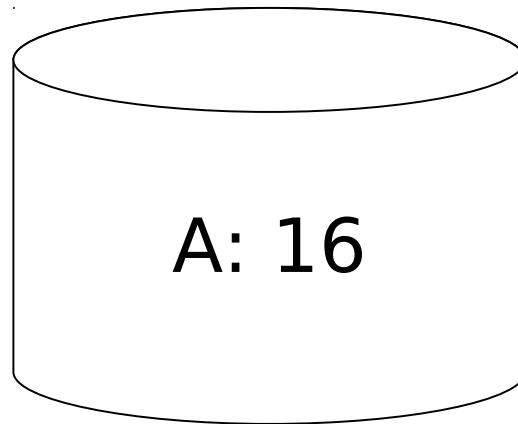
Give\$\$
(amt, Tid, time)



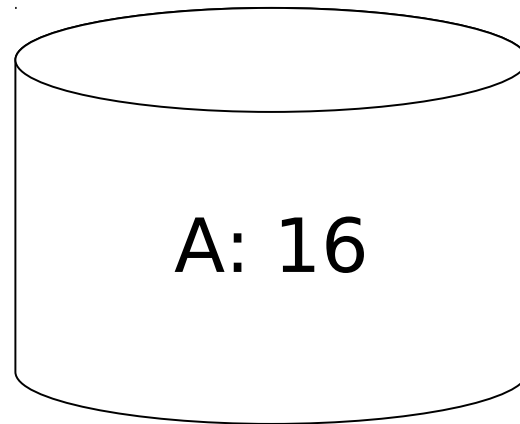
ATM



Media failure (loss of non-volatile storage)



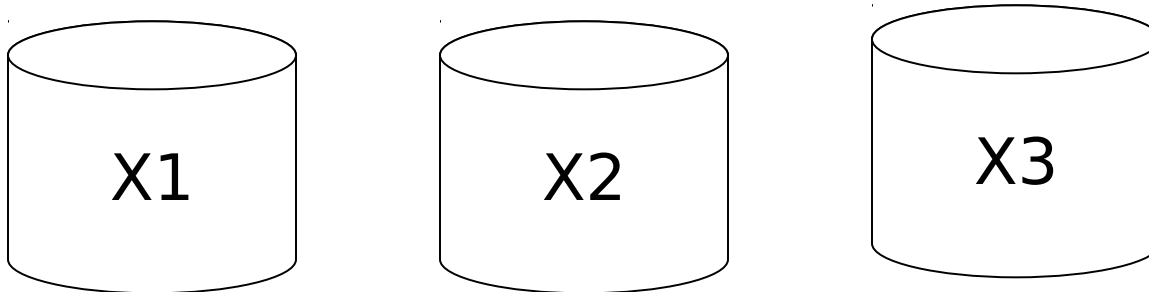
Media failure (loss of non-volatile storage)



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

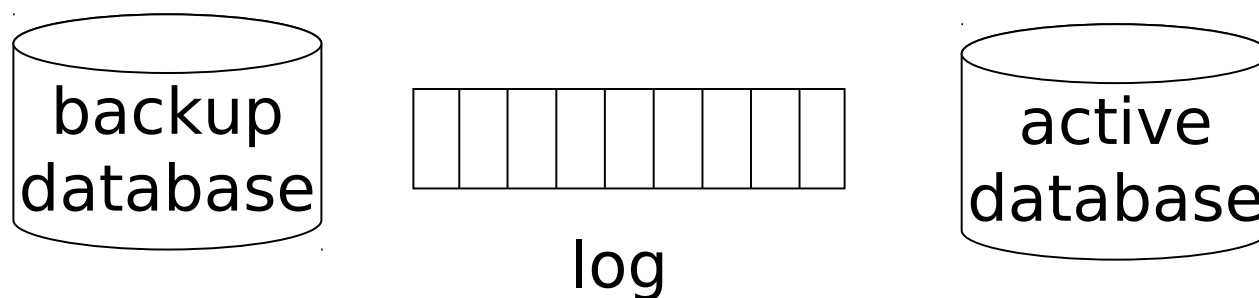


Example #2 Redundant writes, Single reads

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

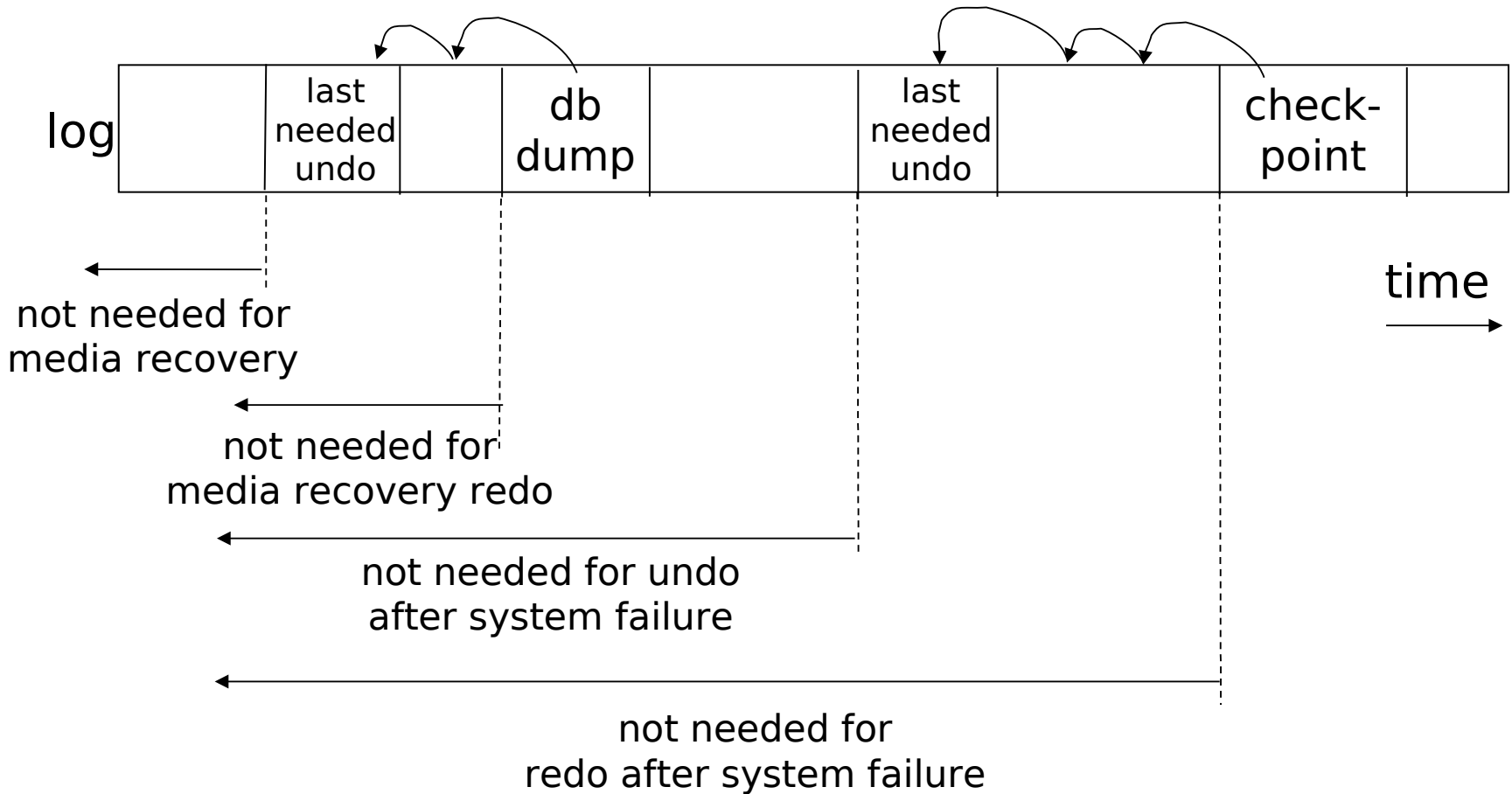
Assumes bad data can be detected

Example #3: DB Dump + Log



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

When can log be discarded?



Summary

- Consistency of data
- One source of problems: failures
 - Logging
 - Redundancy
- Another source of problems:
Data Sharing..... next