Transactions and Failure Recovery

Instructor: Matei Zaharia

cs245.stanford.edu

Outline

Assignment 1 bonus solutions

Defining correctness

Transaction model

Hardware failures

Recovery with logs

Outline

Assignment 1 bonus solutions

Defining correctness

Transaction model

Hardware failures

Recovery with logs

Outline

Assignment 1 bonus solutions

Defining correctness

Transaction model

Hardware failures

Recovery with logs

Focus of This Part of Course

Correctness in case of failures & concurrency

» There's no point running queries quickly if the input data is wrong!

Correctness of Data

Would like all data in our system to be "accurate" or "correct" at all times

» Both logical data model and physical structs

Employees

Name	Age
Smith	52
Green	3421
Chen	1

Idea: Integrity or Consistency Constraints

Predicates that data structures must satisfy

Examples:

» x is field of relation R

» Domain(x) = {student, prof, staff}

» If x=prof in a record then office!=NULL in it

» T is valid B-tree index for attribute x of R

» No staff member 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 All Issues

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 All Issues

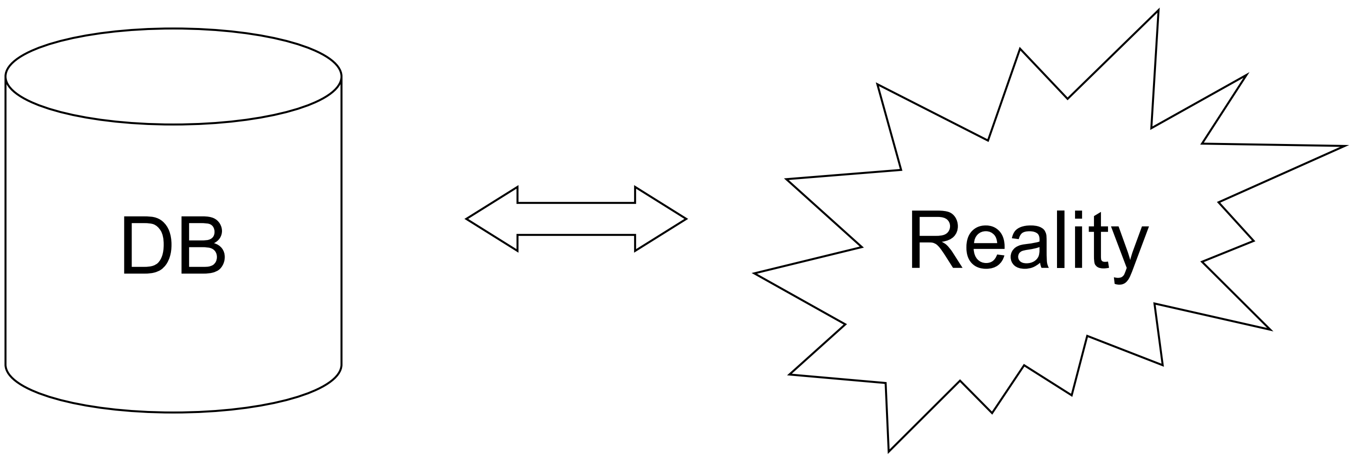
Note: some transaction constraints could be "emulated" by simple constraints, e.g.,

account

acct #	….	balance	deleted?

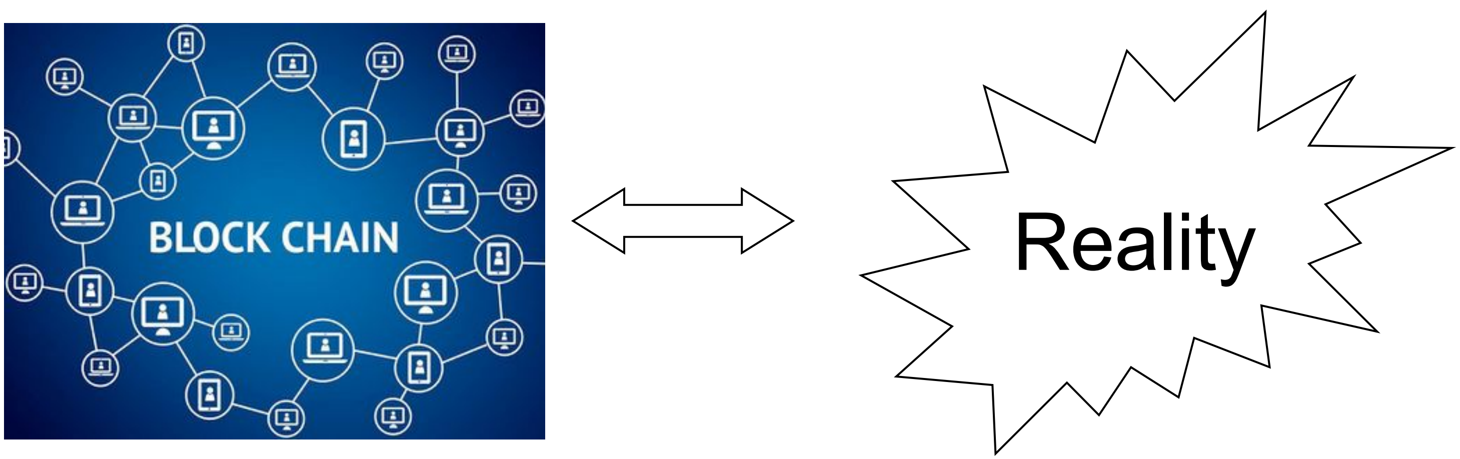
Constraints (As We Use Here) May Not Capture All Issues

Example 2: database should reflect real world



Constraints (As We Use Here) May Not Capture All Issues

Example 2: database should reflect real world



In Any Case, Continue with Constraints...

Observation: DB can't always be consistent!

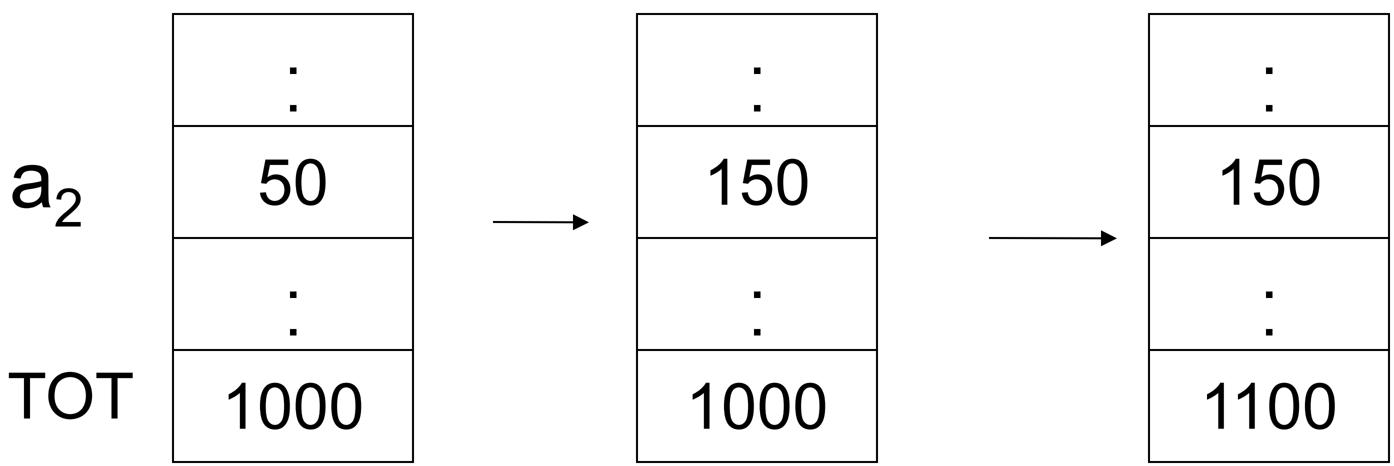
Example: $a_1 + a_2 + \ldots a_n = TOT$ (constraint)

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

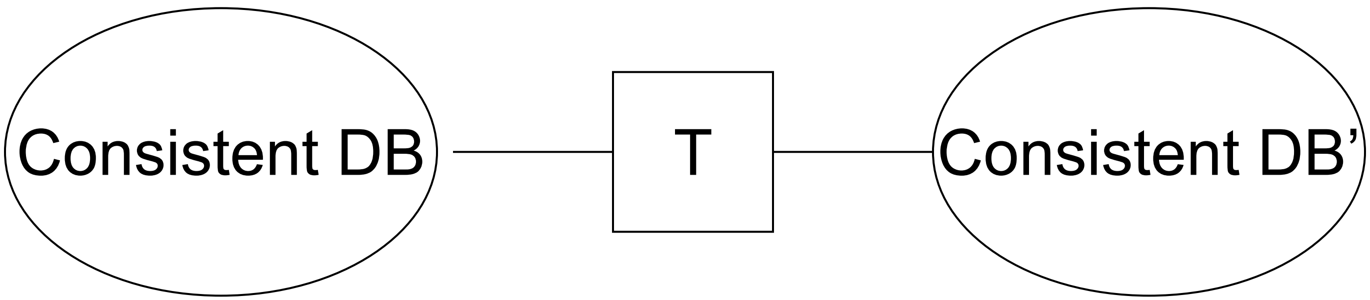
Example: $a_1 + a_2 + \ldots. a_n = TOT$ (constraint)

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 a consistent state

+ T executes in isolation

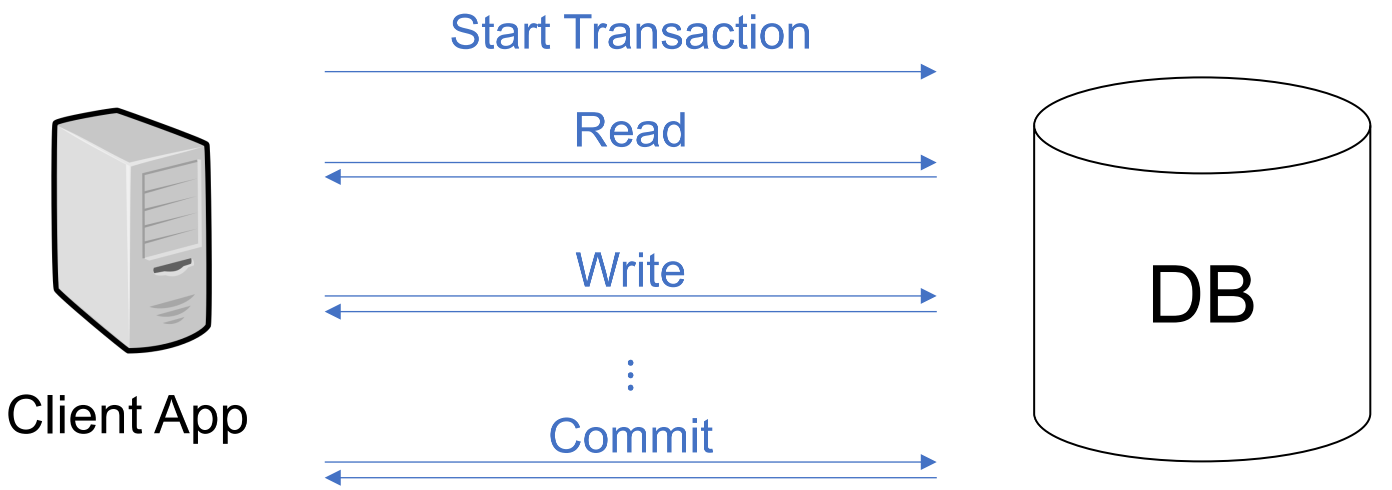
$\Rightarrow$ T leaves a consistent state

Correctness (Informally)

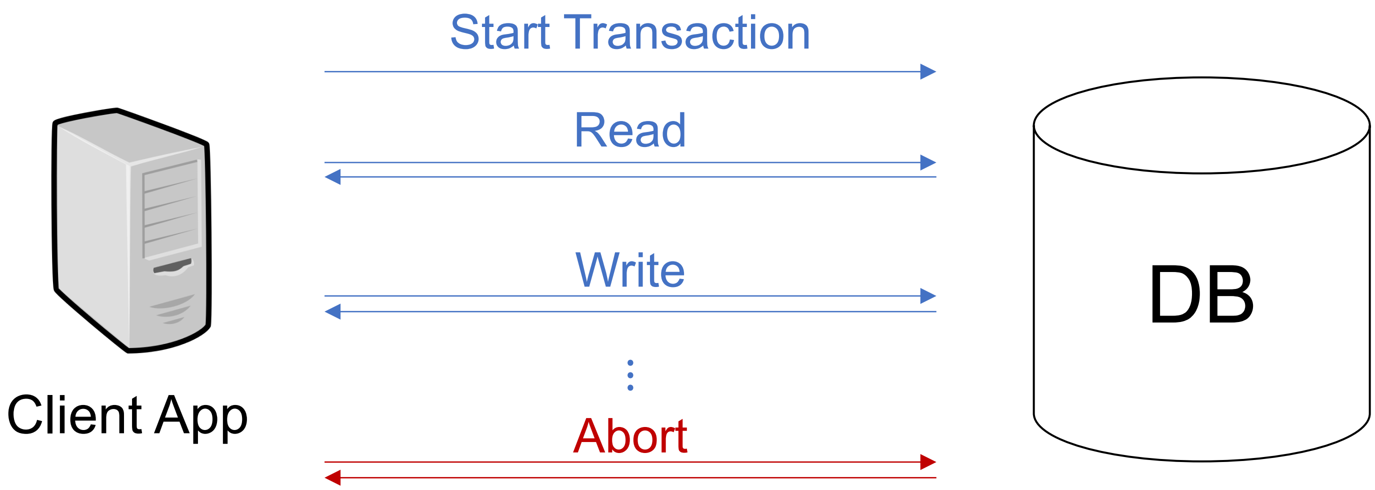
If we stop running transactions, database is left consistent

Each transaction sees a consistent DB

More Detail: Transaction API



More Detail: Transaction API



Both clients and system can abort transactions

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 $\Rightarrow$ marketers

We Won’t Consider:

How to write correct transactions

How to check for DBMS bugs

Constraint verification & repair

» That is, the solutions we’ll study **do not need to know** the constraints!

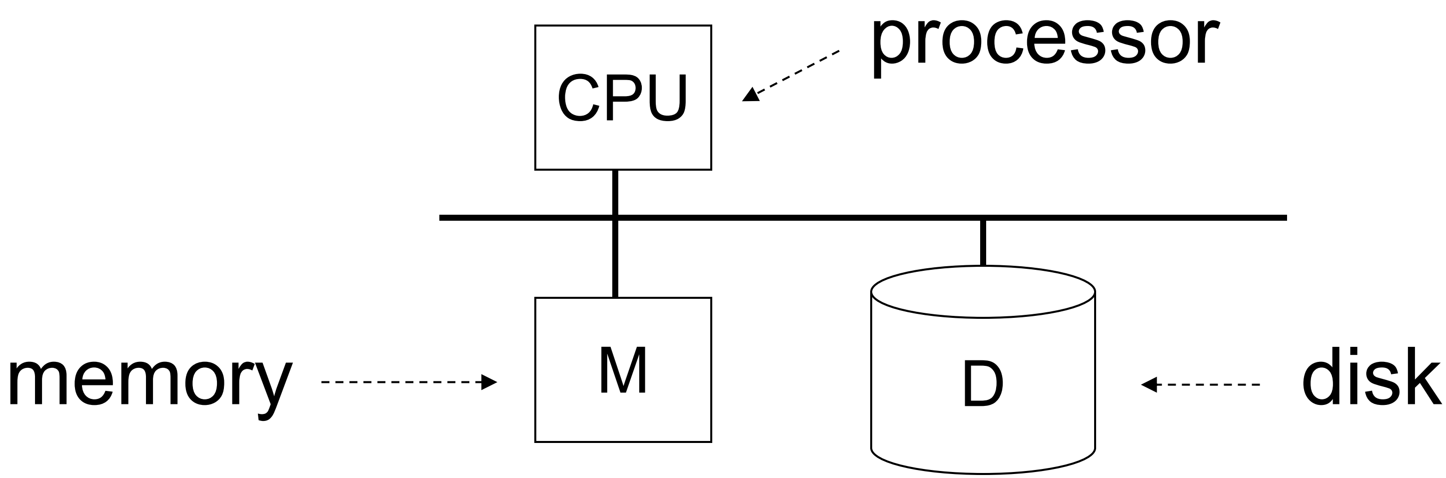
Failure Recovery

First order of business: **Failure Model**

Failure Models

- Events
  - Desired
  - Undesired
    - Expected
    - Unexpected

Our Failure Model



Our Failure Model

Desired Events: see product manuals….

Undesired Expected Events:

» System crash ("fail-stop failure")

- CPU halts, resets
- Memory lost

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 the universe….

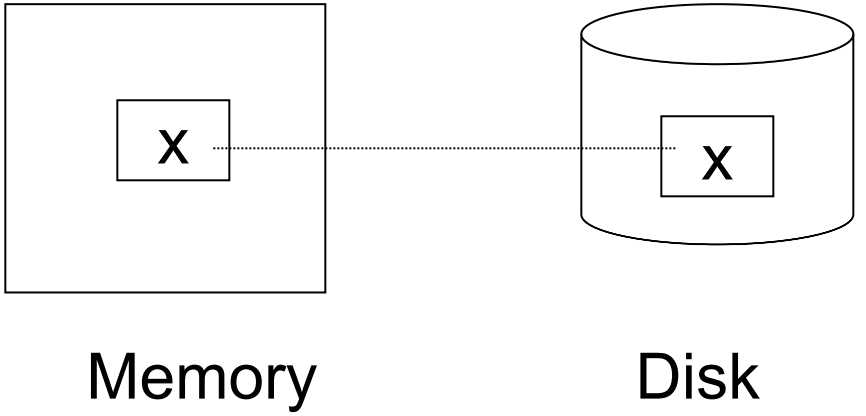
Is This Model Reasonable?

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

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

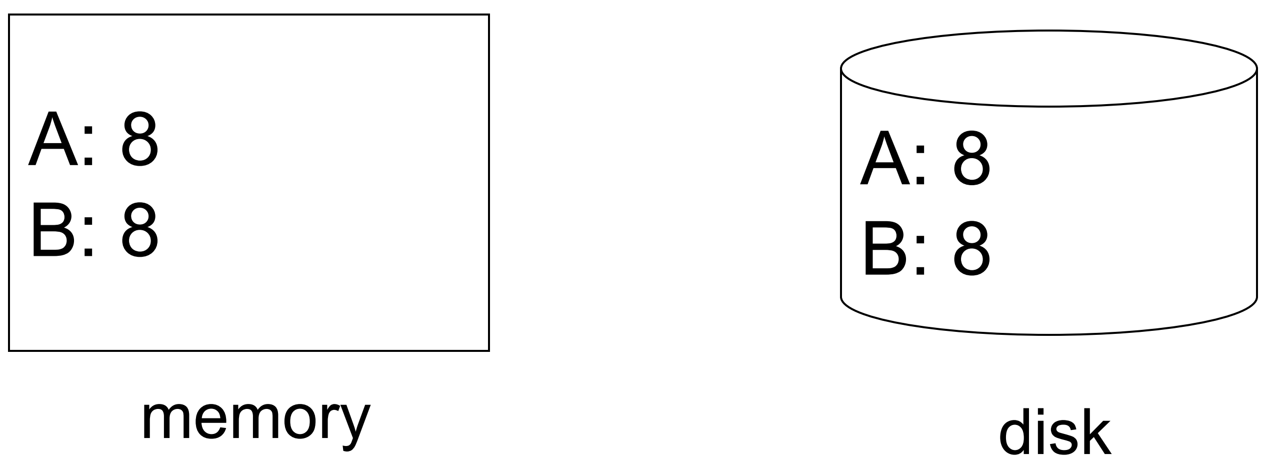
Key Problem: Unfinished Transaction

Example Constraint: A=B

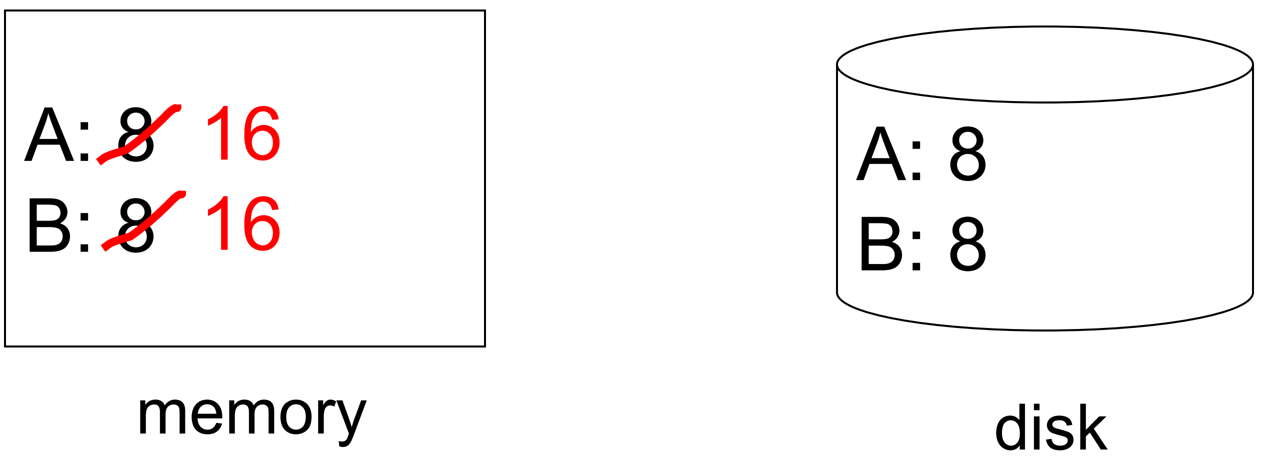
T1: $A \leftarrow A \times 2$

$B \leftarrow B \times 2$

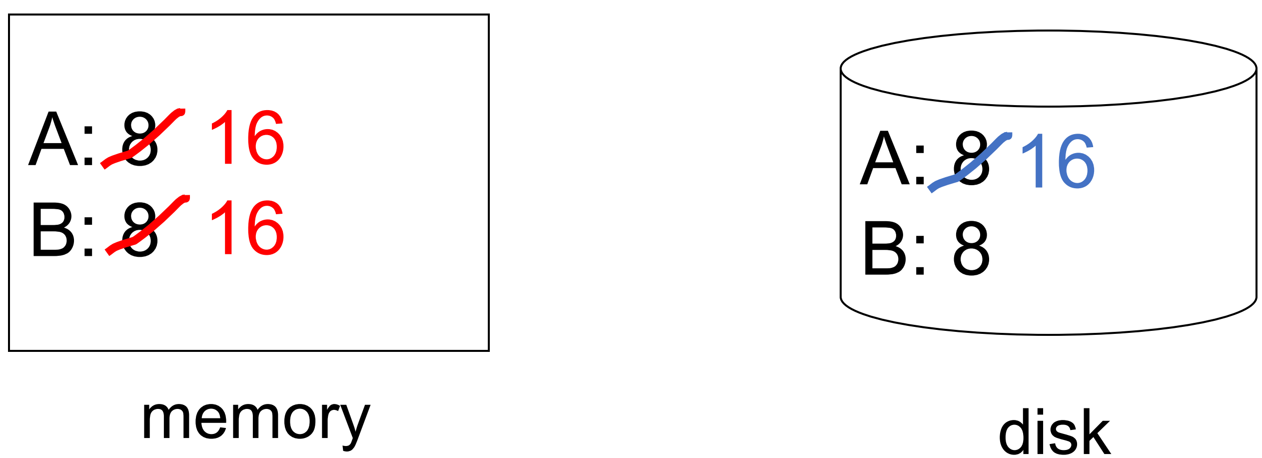
T_1: Read (A,t); $t \leftarrow t \times 2$
Write (A,t);
Read (B,t); $t \leftarrow t \times 2$
Write (B,t);
Output (A);
Output (B);



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);



T1: Read (A,t); t ← t×2
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A); failure!
Output (B);



Need: Atomicity

Execute **all** actions of a transaction together, or **none** at all

One Solution

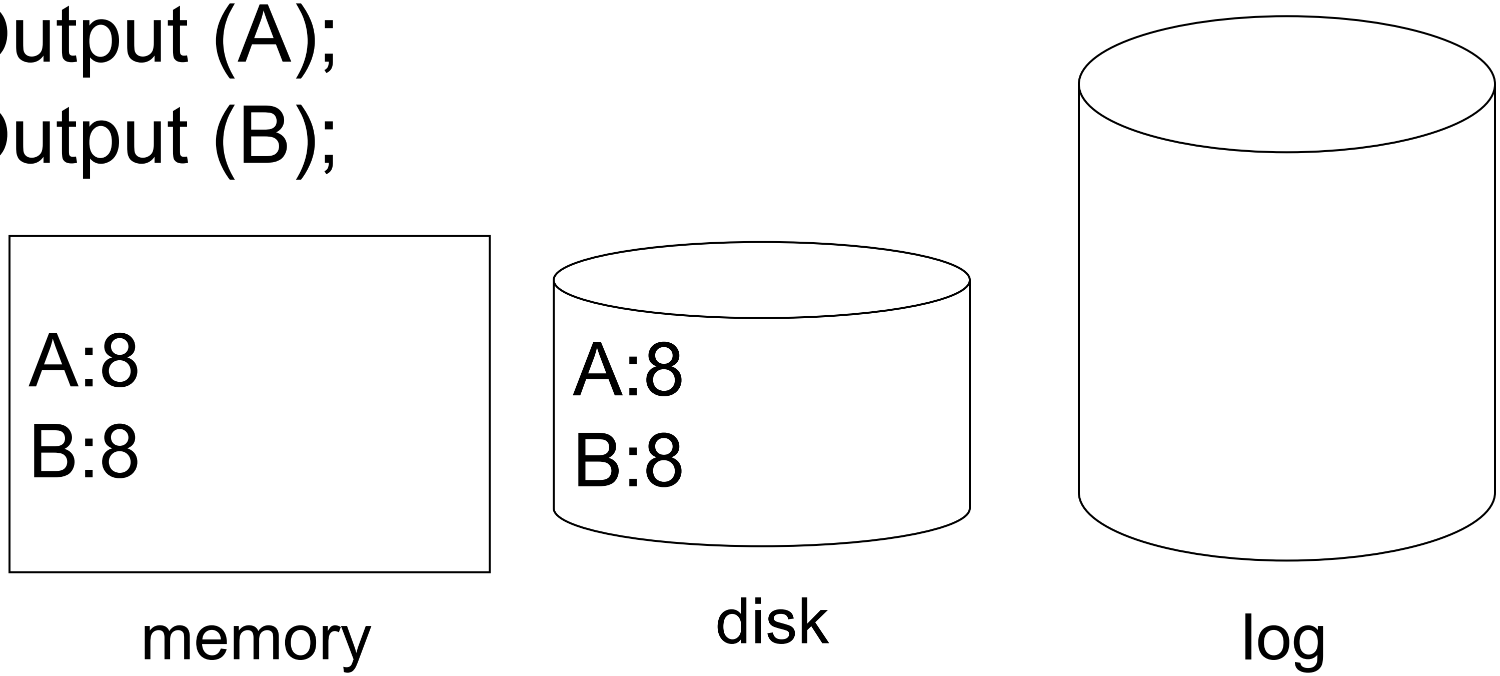
Undo logging (immediate modification)

Due to: Hansel and Gretel, 1812 AD

Updated to durable undo logging in 1813 AD

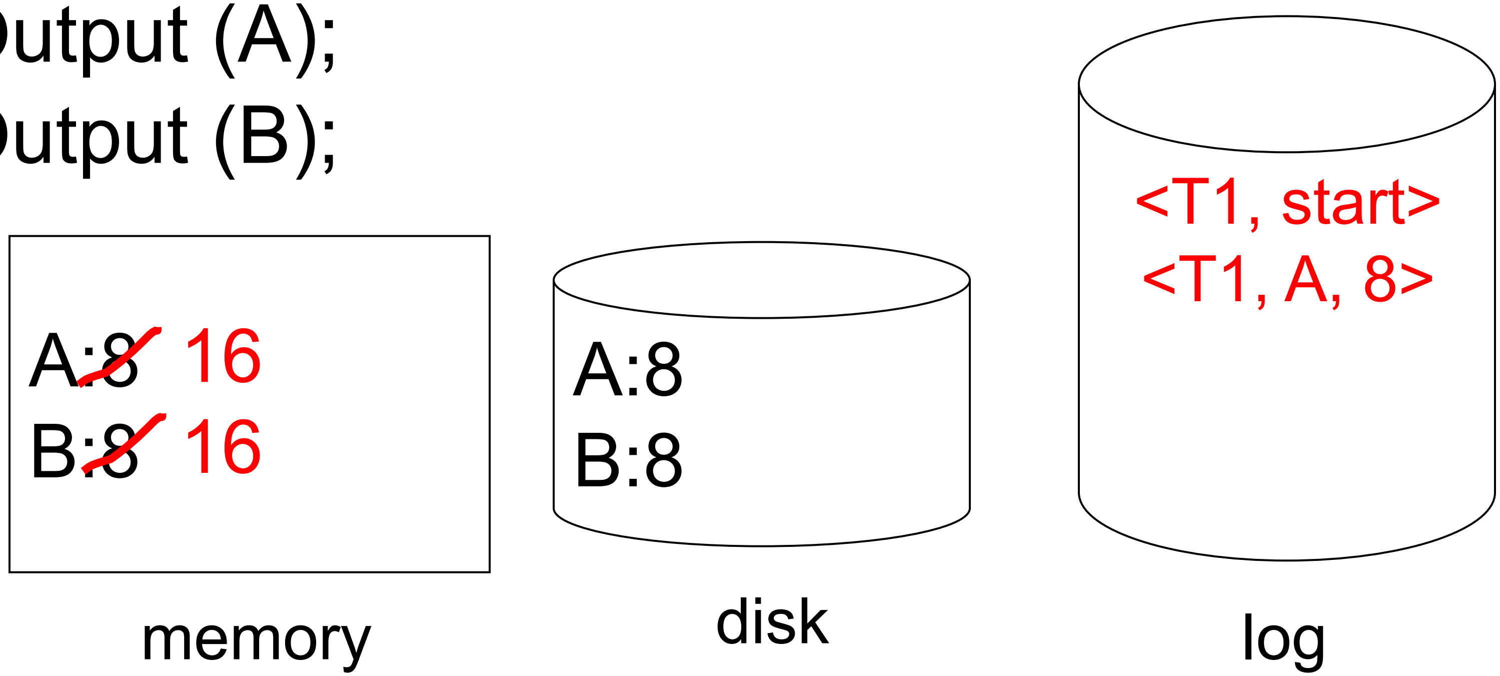
Undo Logging (Immediate modification)

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);



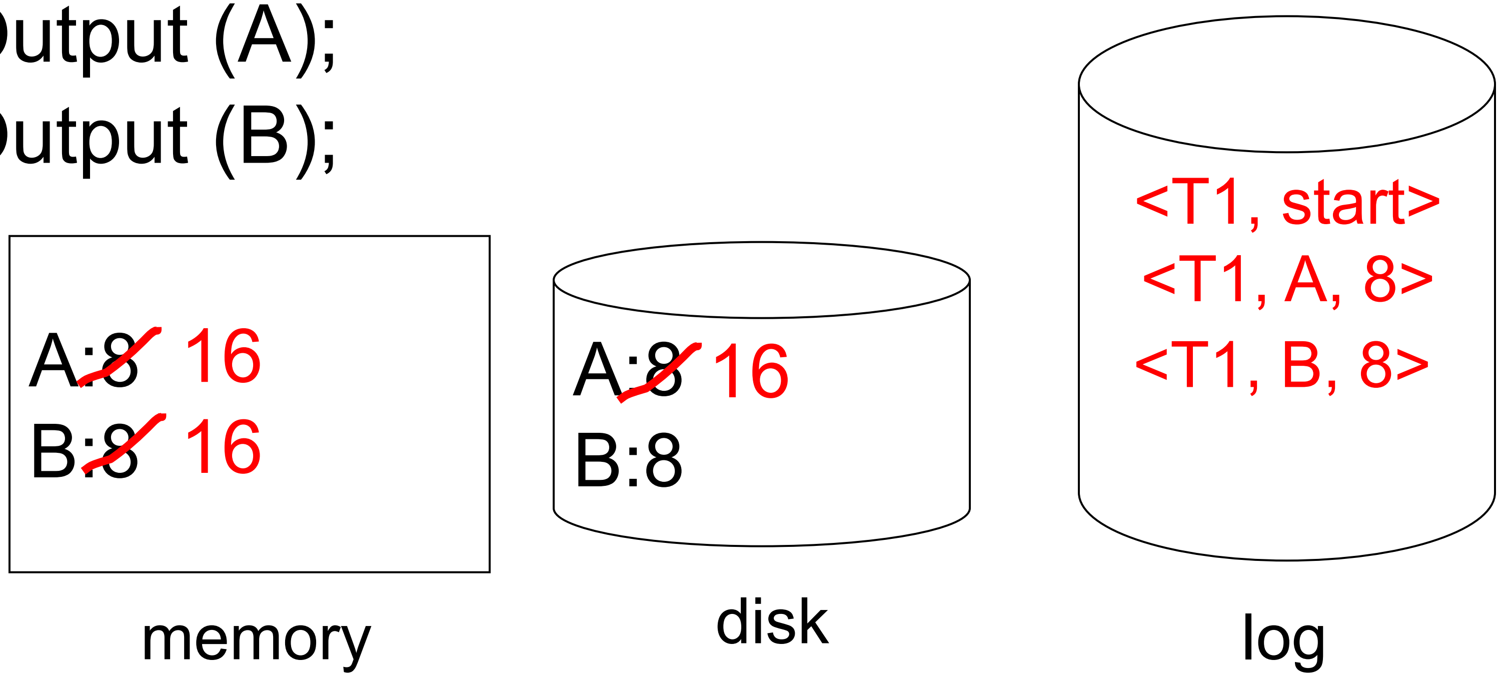
Undo Logging (Immediate modification)

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



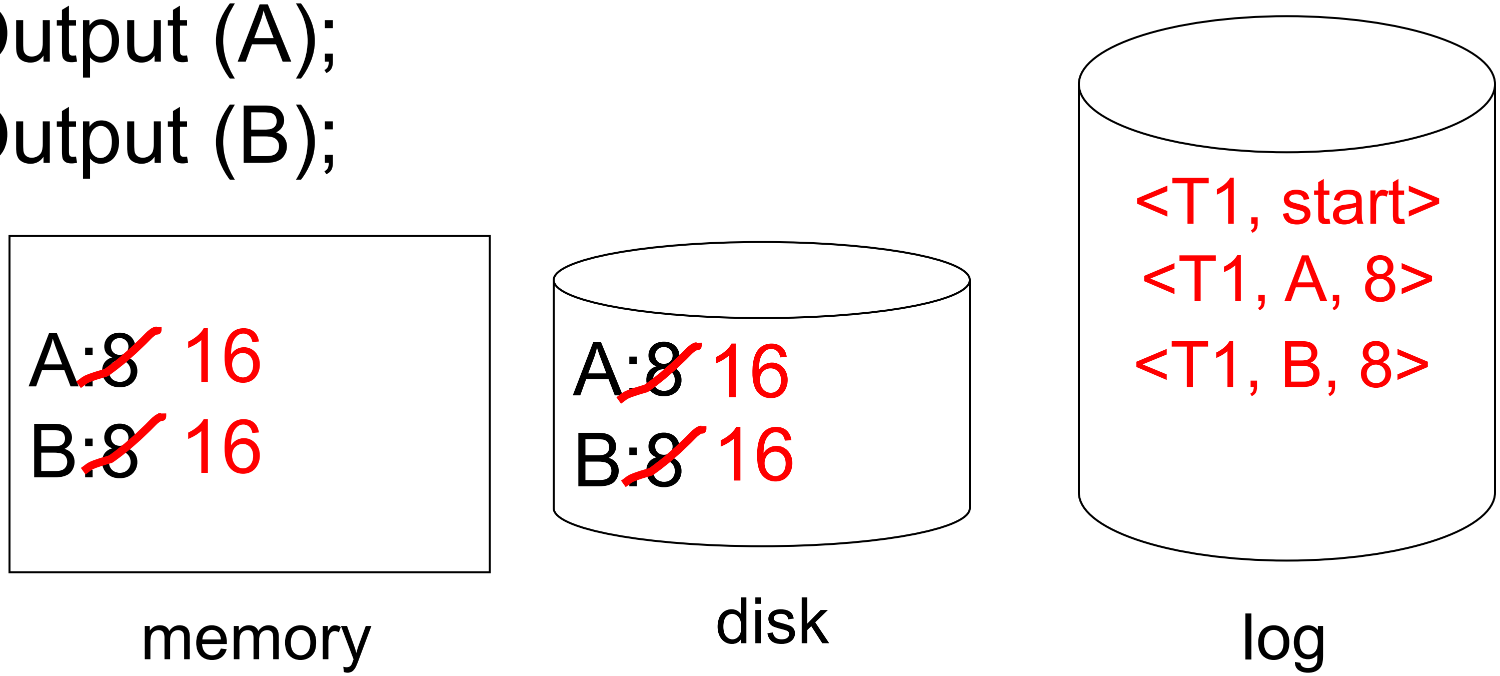
Undo Logging (Immediate modification)

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



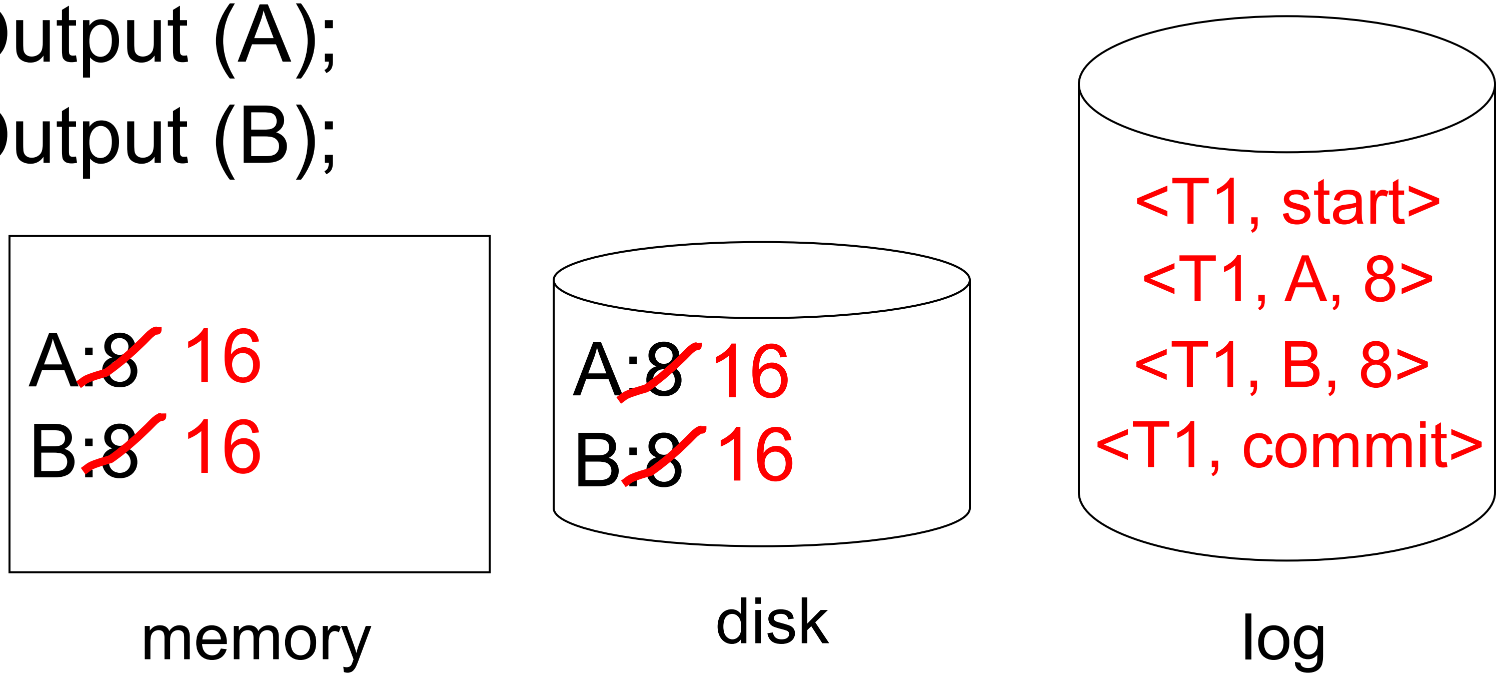
Undo Logging (Immediate modification)

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);



Undo Logging (Immediate modification)

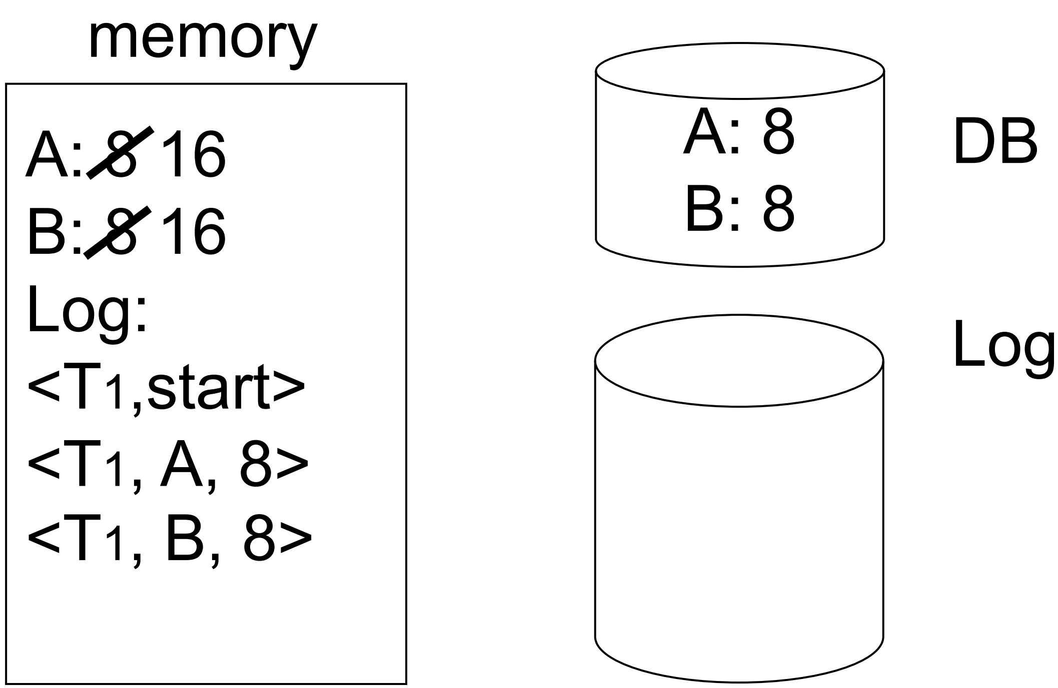
T1: Read (A,t); t ← t×2   A=B
    Write (A,t);
    Read (B,t); t ← t×2
    Write (B,t);
    Output (A);
    Output (B);



One "Complication"

Log is first written in memory

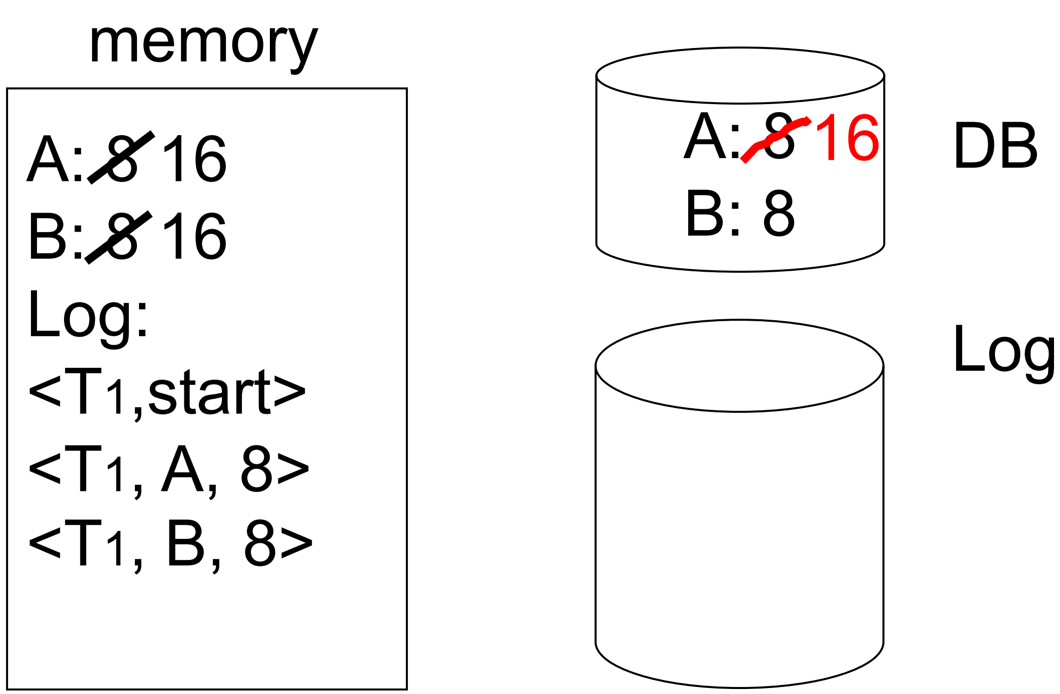
Not written to disk on every action



One “Complication”

Log is first written in memory

Not written to disk on every action

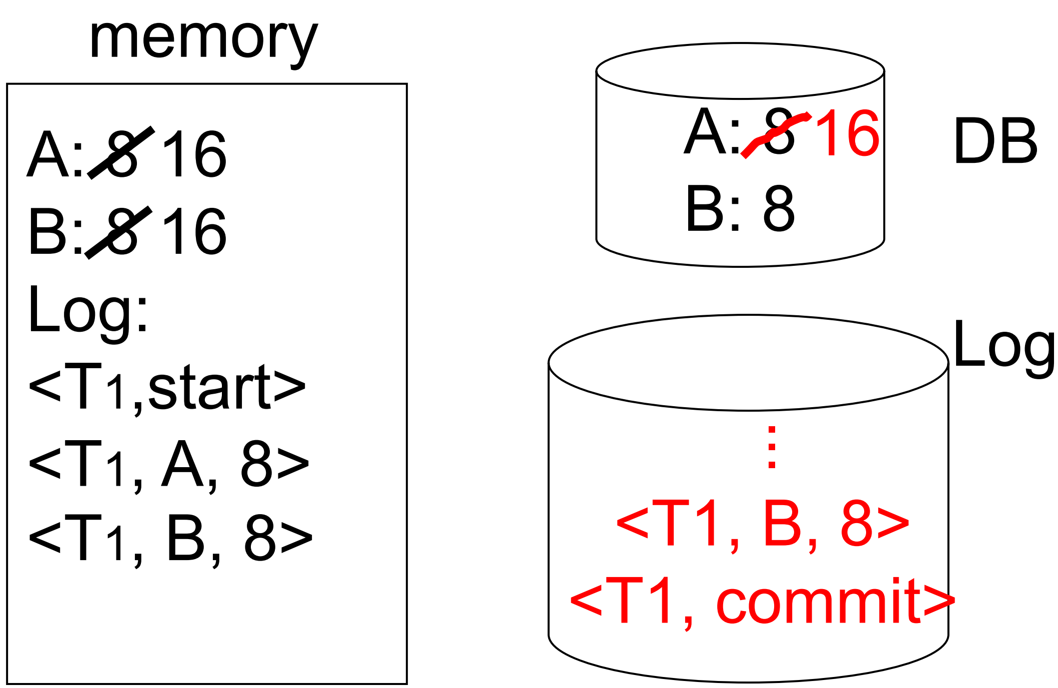


BAD STATE
1

One "Complication"

Log is first written in memory

Not written to disk on every action



BAD STATE
2

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 record is flushed to log, all writes of transaction must be on disk

Recovery Rules: Undo Logging

(1) Let S = set of transactions with
<Ti, start> in log,
but no <Ti, commit> or <Ti, abort> in log

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

- if Ti $\in$ S then
  - write (X, v)
  - output (X)

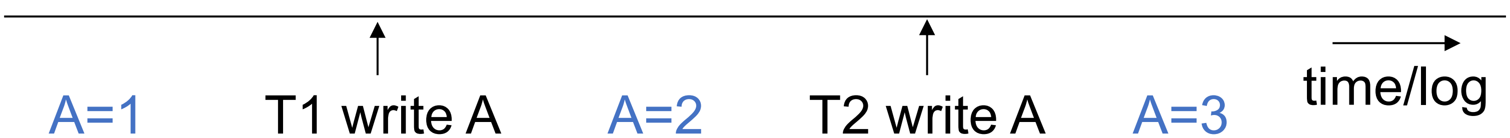
(3) For each Ti $\in$ S do

- write <Ti, abort> to log

Question

Can our writes of <Ti, abort> 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
- » <T1, abort> written but NOT <T2, abort>?
- » <T2, abort> written but NOT <T1, abort>?



What If Crash During Recovery?

No problem! → Undo is **idempotent**

(same effect if you do it twice)

Any Downsides to Undo Logging?

Any Downsides to Undo Logging?

Have to do a lot of I/O to commit (write all updated objects to disk first)

Hard to replicate database to another disk (must push **all** changes across the network)

To Discuss

Redo logging

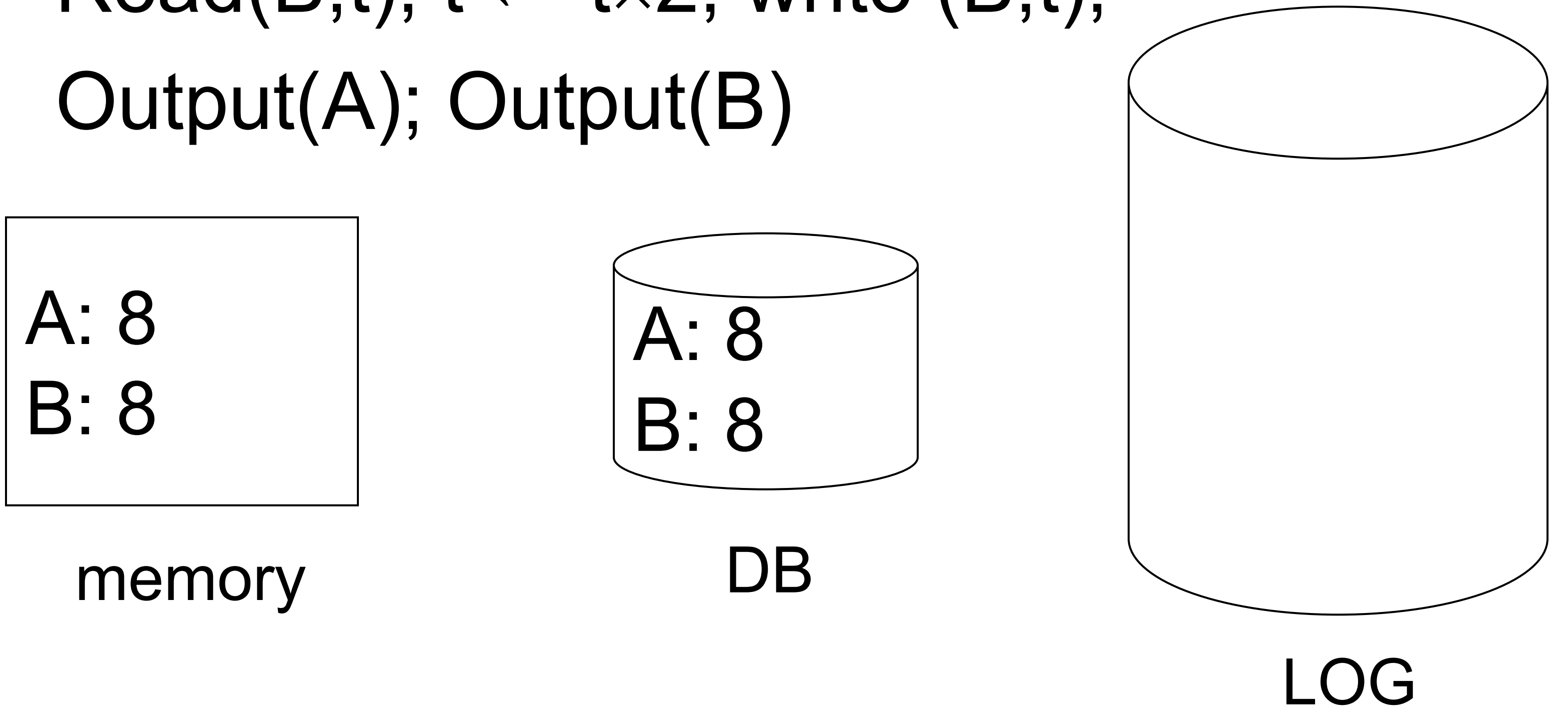
Undo/redo logging

Redo Logging

First send Gretel up with no rope,
then Hansel goes up safely with rope!

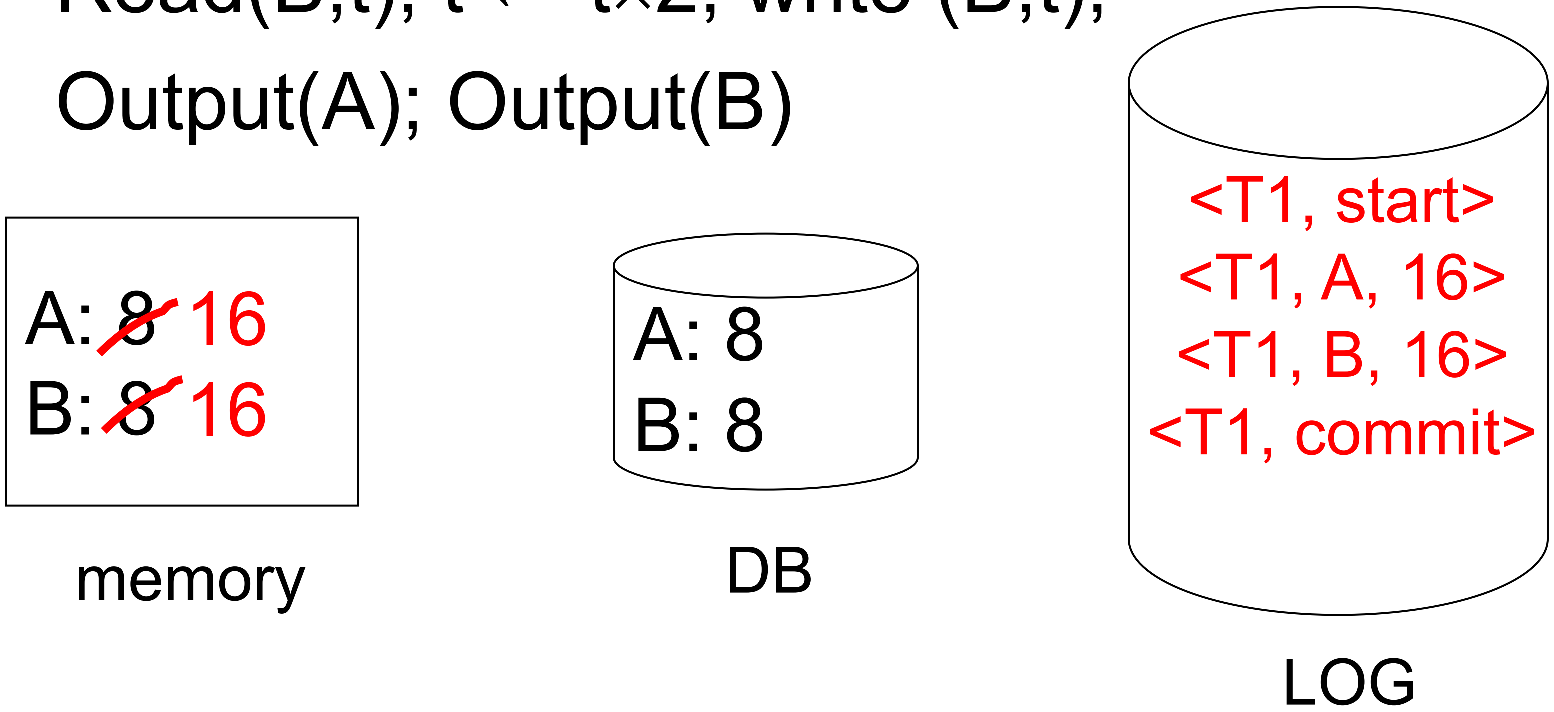
Redo Logging (deferred modification)

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



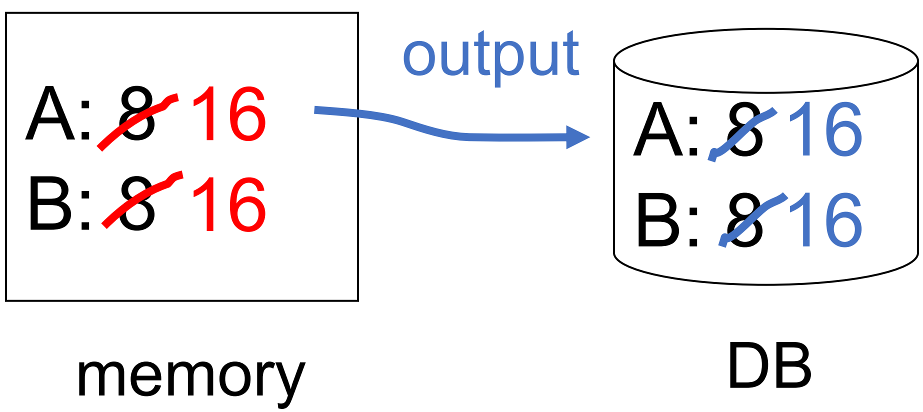
Redo Logging (deferred modification)

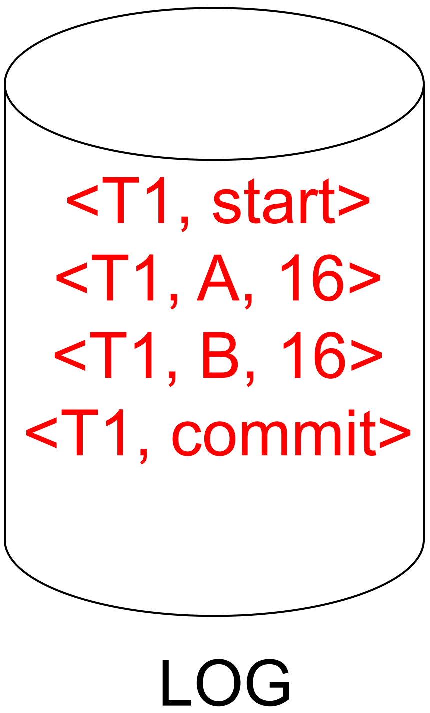
T1: Read(A,t); t ← t×2; write (A,t);
Read(B,t); t ← t×2; write (B,t);
Output(A); Output(B)



Redo Logging (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)



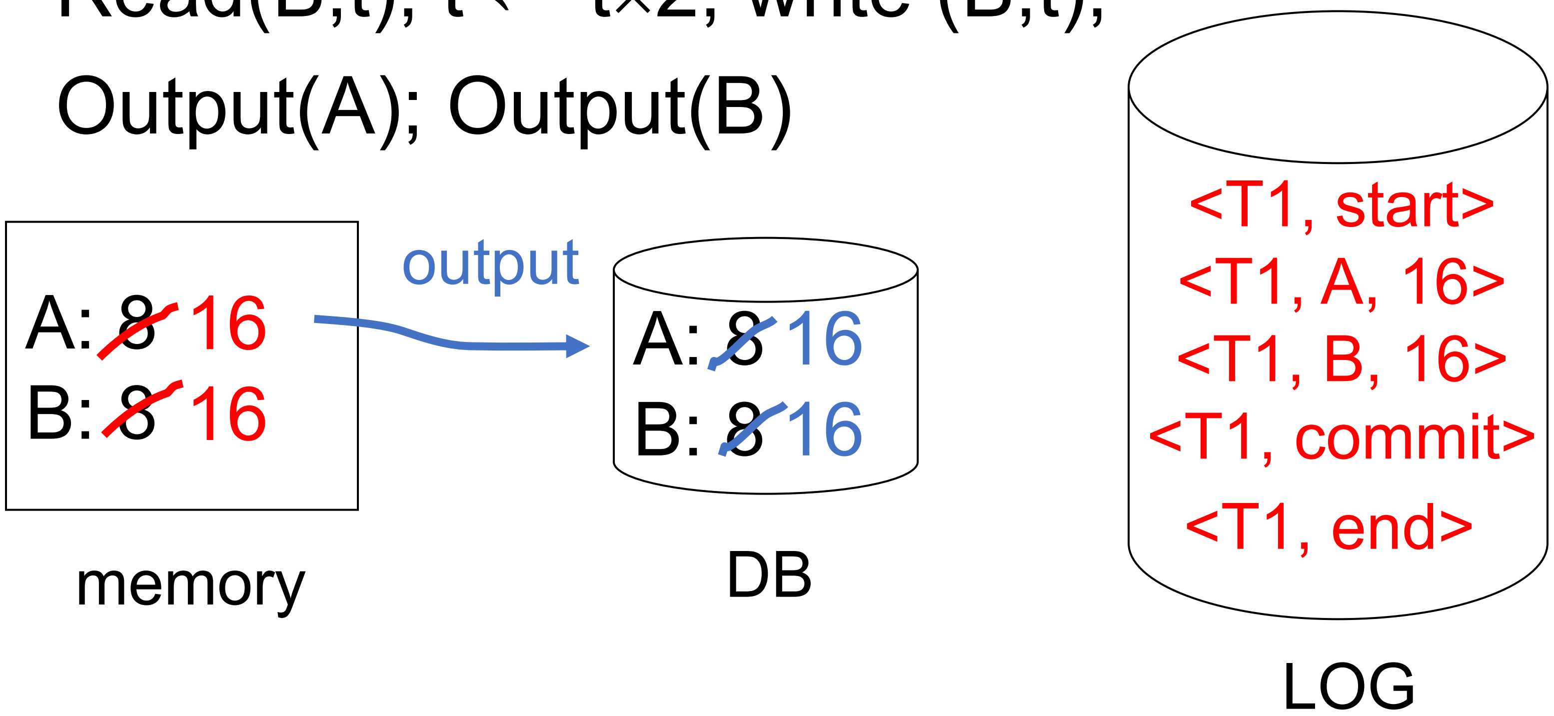


Redo Logging (deferred modification)

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

(1) Let S = set of transactions with
<Ti, commit> (and no <Ti, end>) in log

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

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

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