# Transactions and Failure Recovery

Instructor: Matei Zaharia

cs245.stanford.edu

# Outline

Recap from last time

Undo/redo logging

External actions

Media failures

# Outline

Recap from last time

Undo/redo logging

External actions

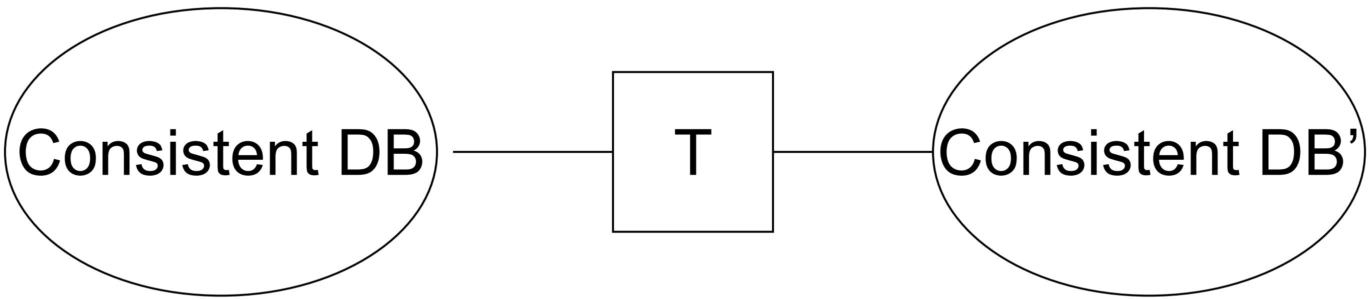
Media failures

# Defining Correctness

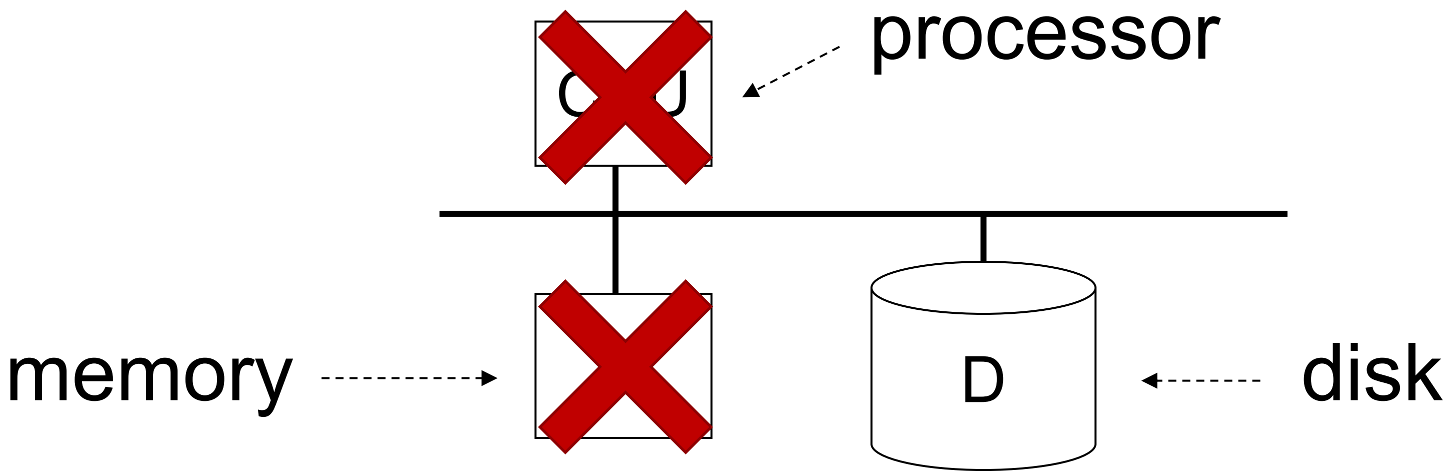
**Constraint:** Boolean predicate about DB state (both logical & physical data structures)

**Consistent DB:** satisfies all constraints

# Transaction: Collection of Actions that Preserve Consistency



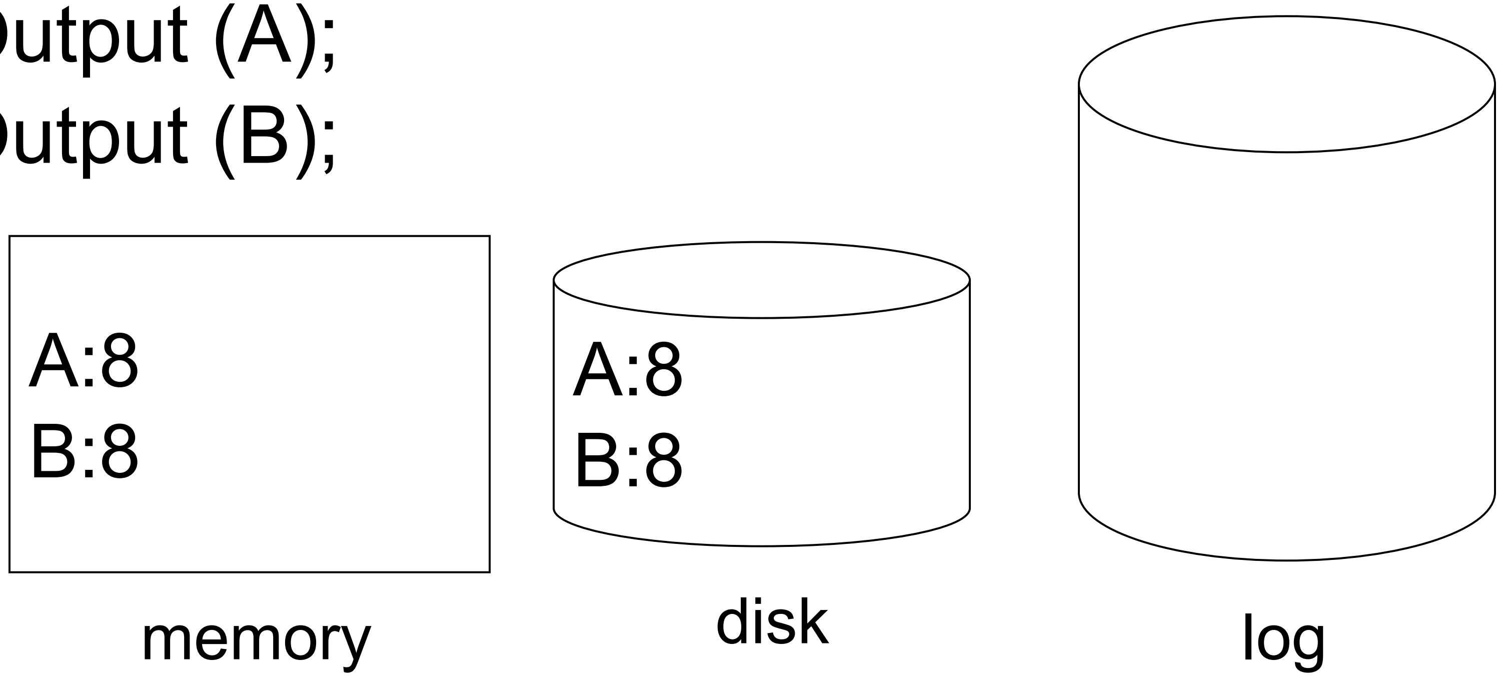
# Our Failure Model




Fail-stop failures of CPU & memory, but disk survives

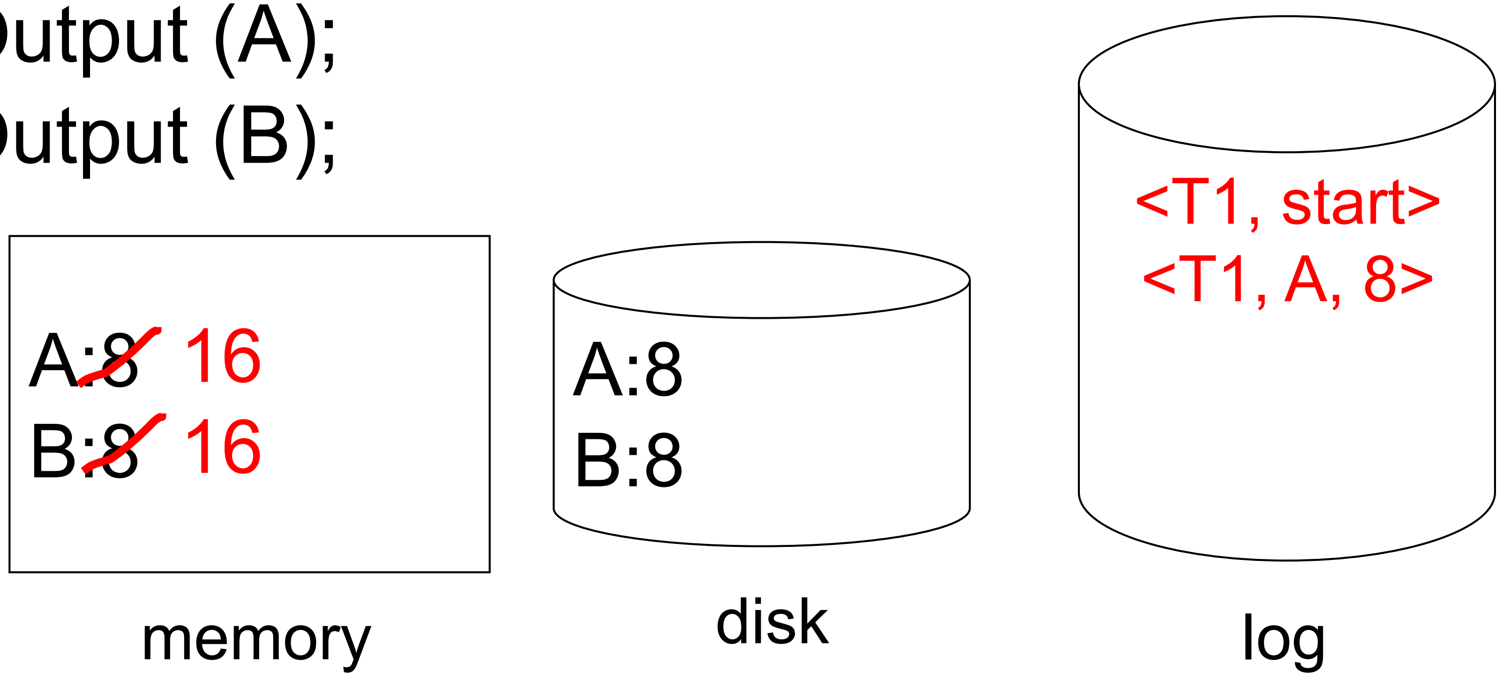
# Undo Logging (Immediate modification)

$T_1$: 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);



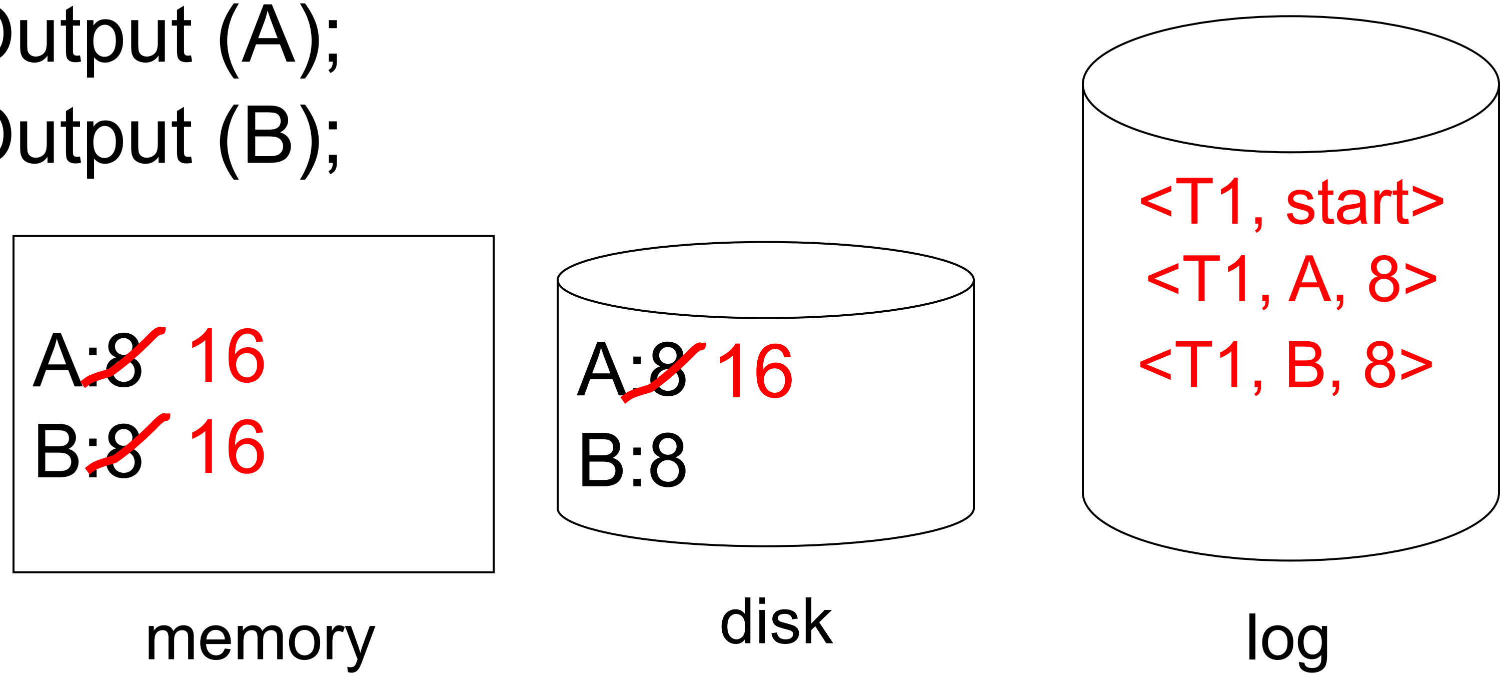
# Undo Logging (Immediate modification)

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);



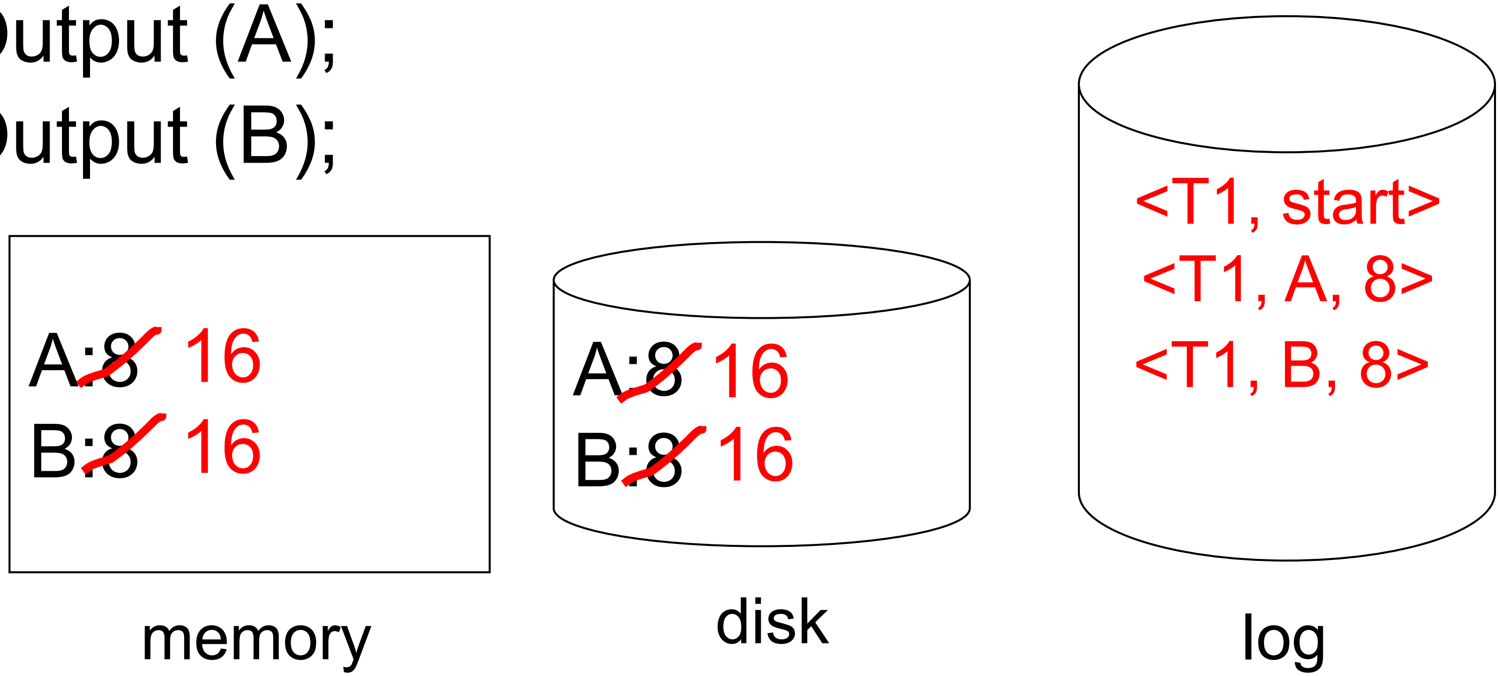
# Undo Logging (Immediate modification)

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);



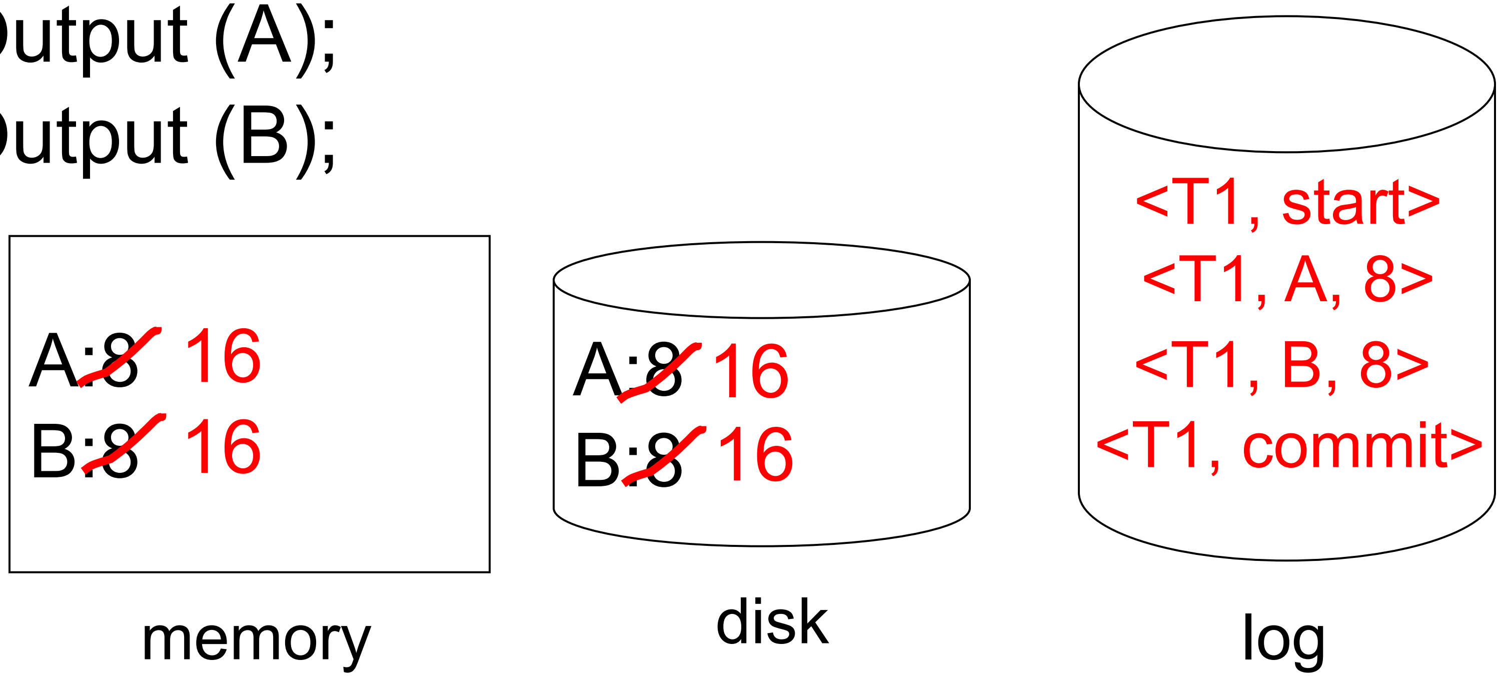
# Undo Logging (Immediate modification)

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



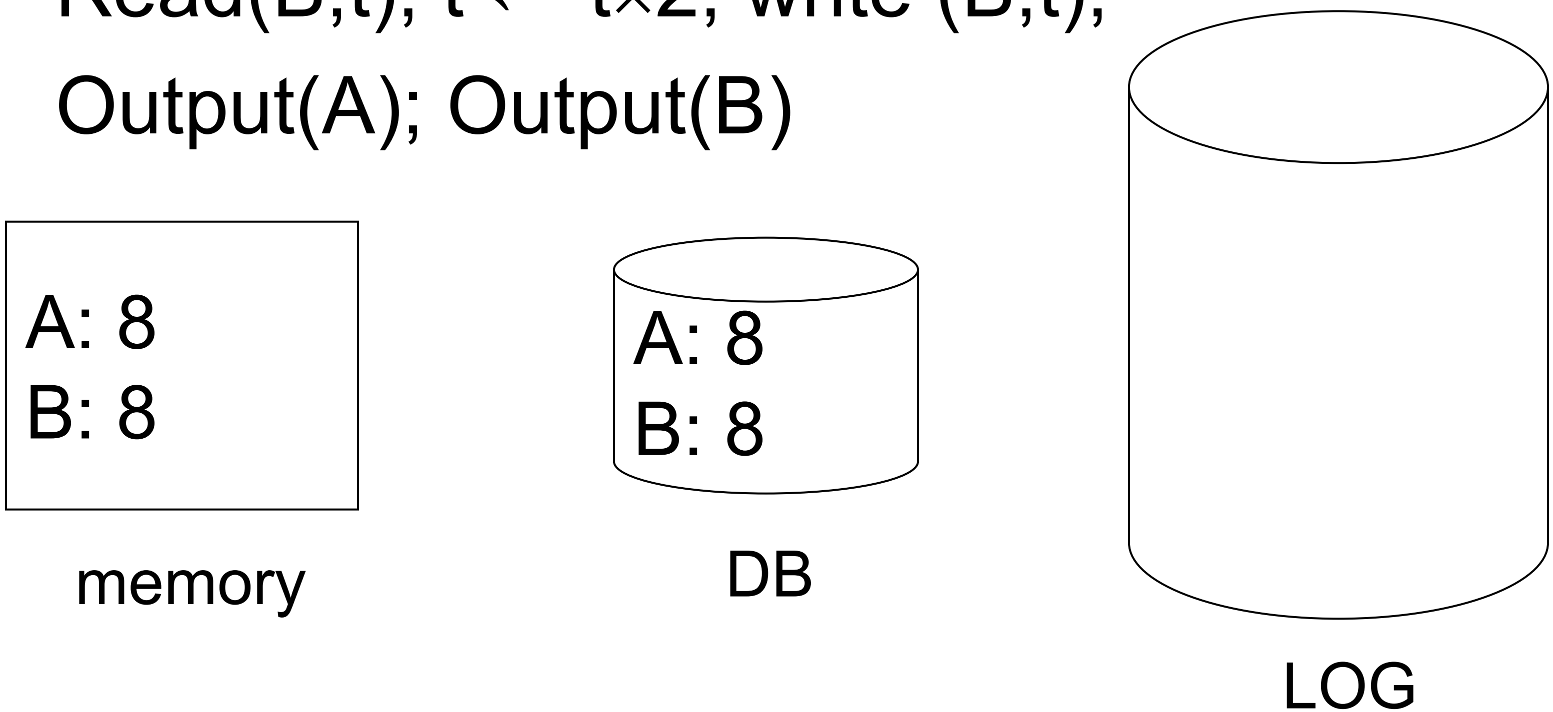
# Undo Logging (Immediate modification)

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



# Redo Logging (deferred modification)

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

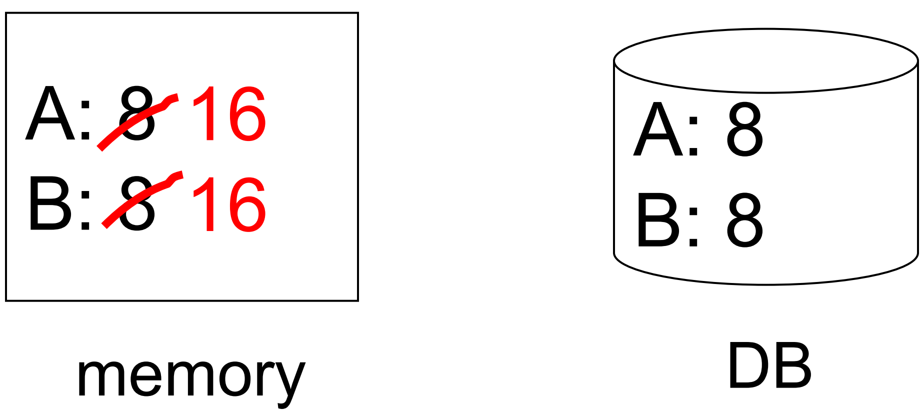


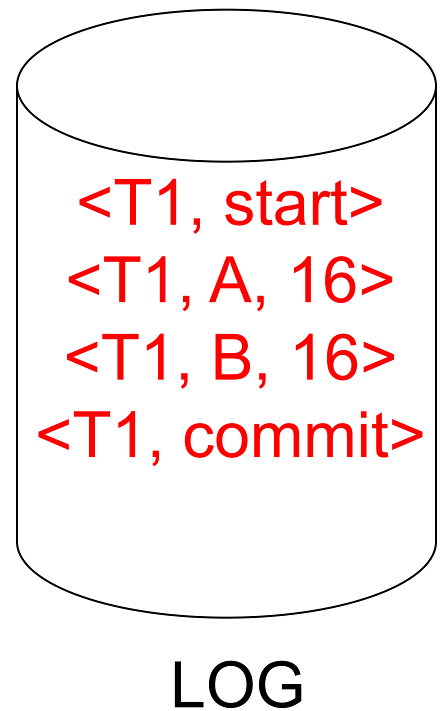
# Redo Logging (deferred modification)

T1: Read(A,t); t ← t×2; write (A,t);

Read(B,t); t ← t×2; write (B,t);

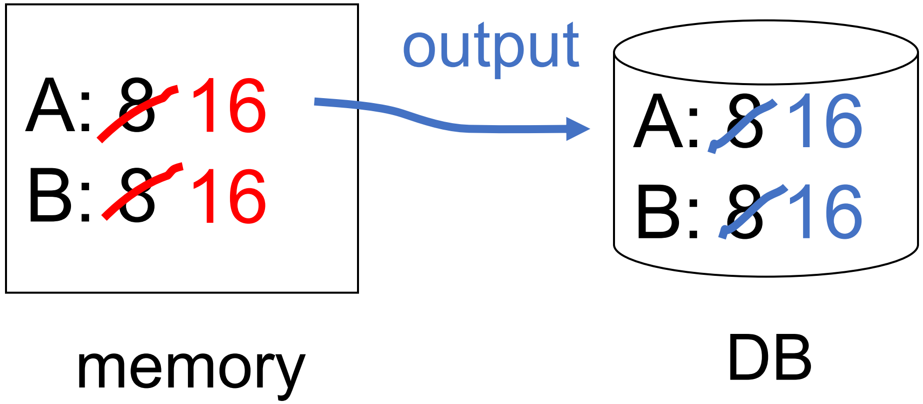
Output(A); Output(B)

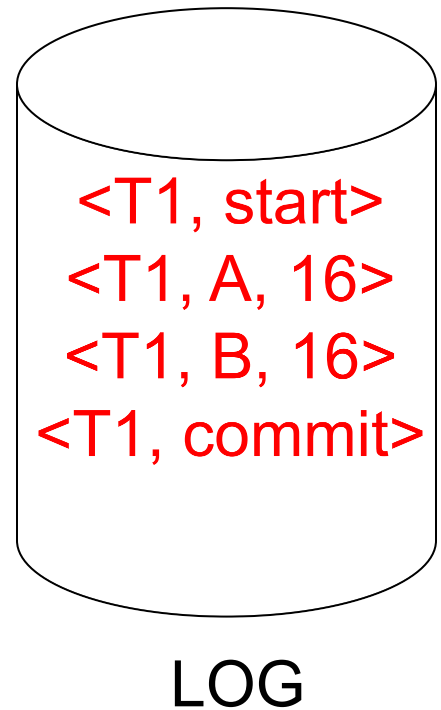


# Redo Logging (deferred modification)

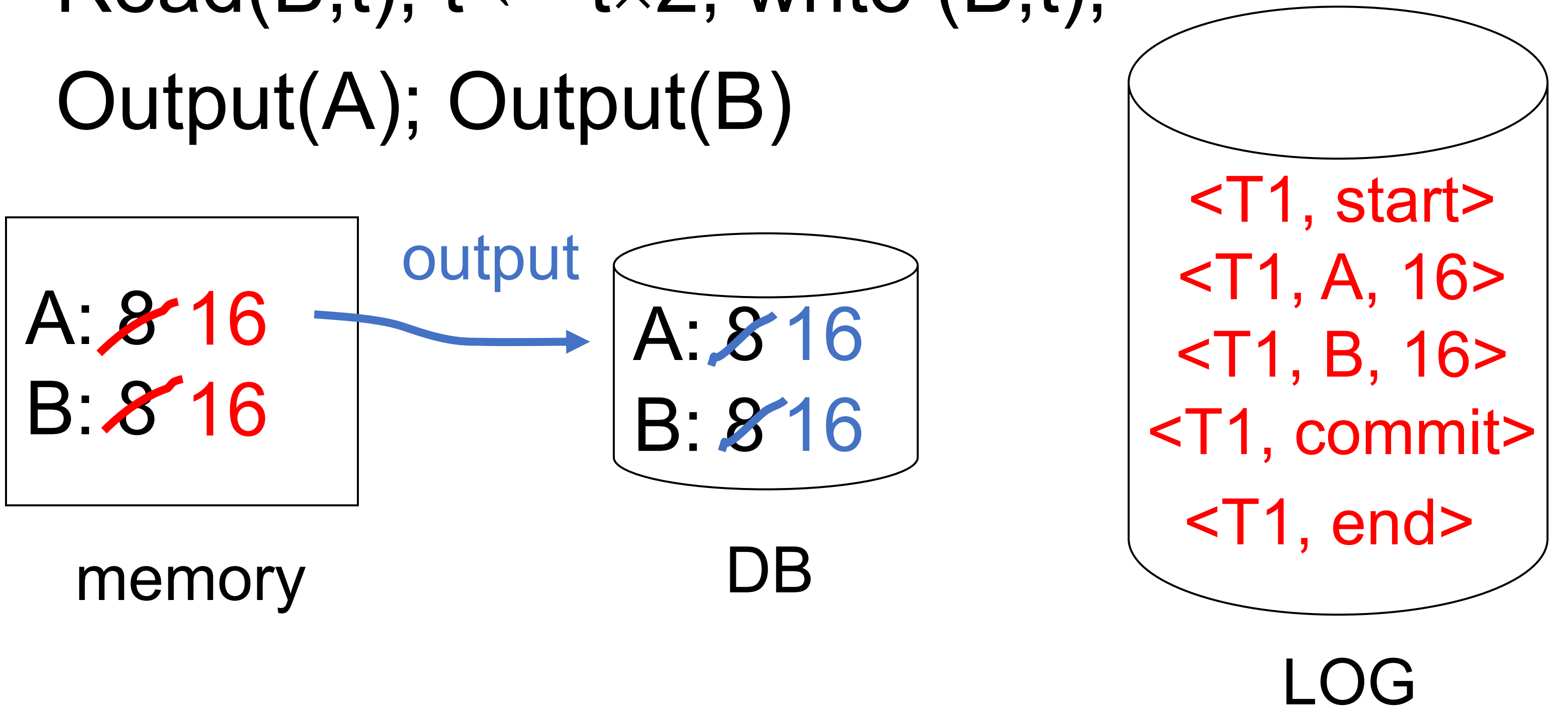
T1: Read(A,t); t ← t×2; write (A,t);
Read(B,t); t ← t×2; write (B,t);
Output(A); Output(B)



# 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)



# Combining <Ti, end> Records

Want to delay DB flushes for hot objects

Say X is branch balance:
T1: ... update X...
T2: ... update X...
T3: ... update X...
T4: ... update X...

<u>Actions:</u>
write X
output X
write X
output X
write X
output X
write X
output X

# Combining <Ti, end> Records

Want to delay DB flushes for hot objects

Say X is branch balance:
T1: ... update X...
T2: ... update X...
T3: ... update X...
T4: ... update X...

__Actions:__
write X
~~output X~~
write X
~~output X~~
write X
~~output X~~
write X
output X

combined <end> record (checkpoint)

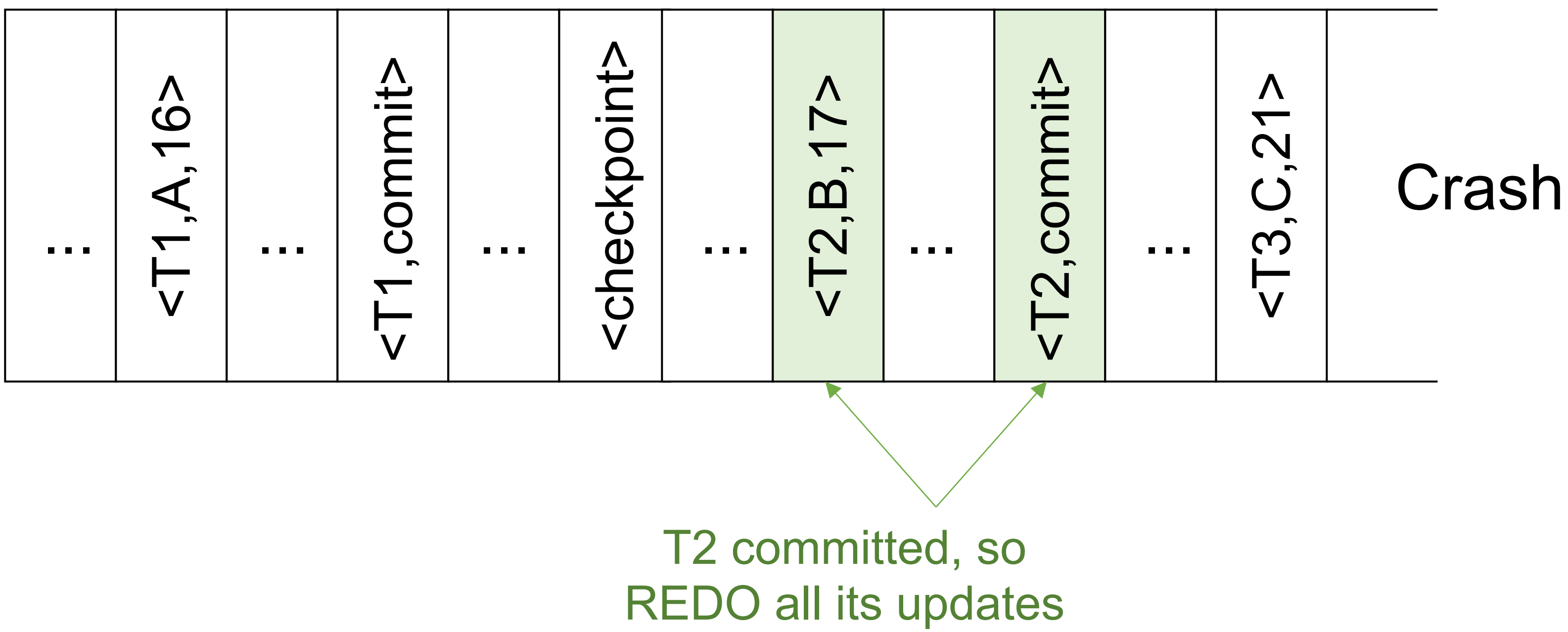
# Redo Logging: What To Do at Recovery?

Redo log (disk):

| | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| ... | <T1,A,16> | ... | <T1,commit> | ... | <checkpoint> | ... | <T2,B,17> | ... | <T2,commit> | ... | <T3,C,21> | Crash |

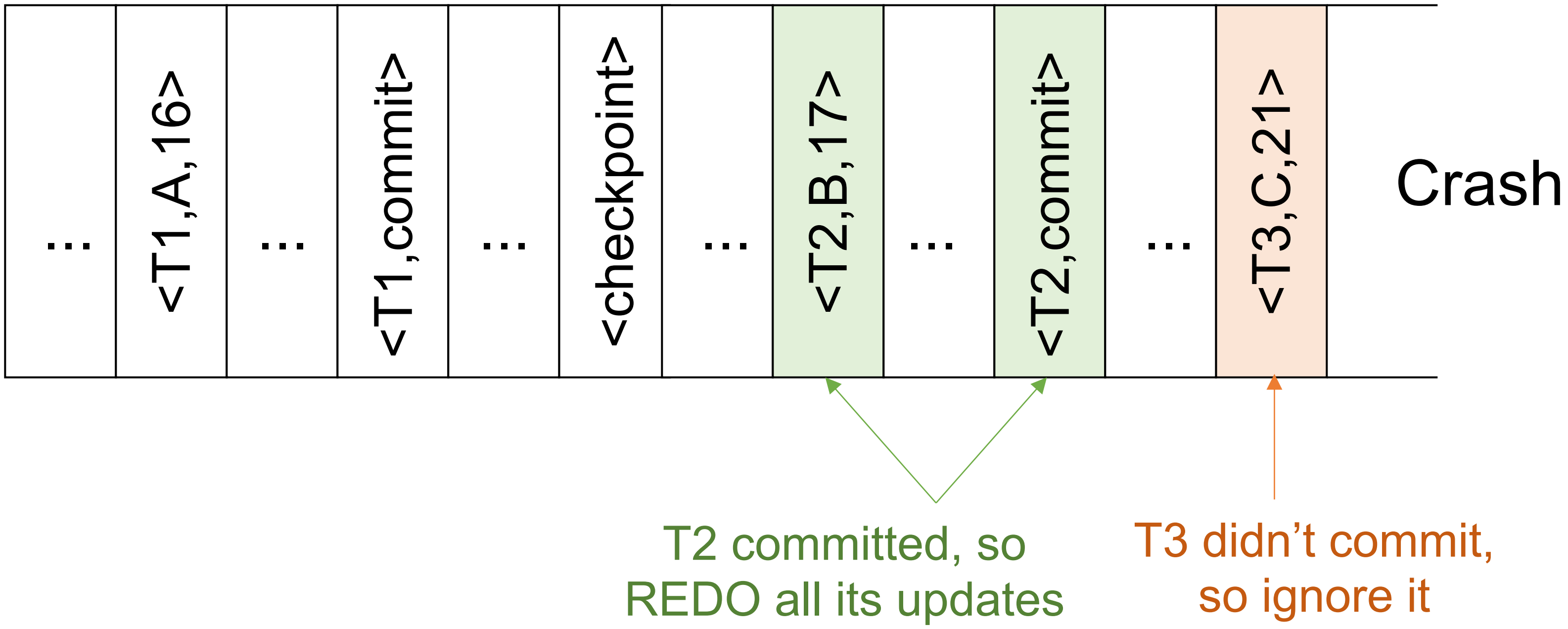
# Redo Logging: What To Do at Recovery?

Redo log (disk):



# Redo Logging:
# What To Do at Recovery?

Redo log (disk):



# Problems with Ideas So Far

**Undo logging:** need to wait for lots of I/O to commit; can't easily have backup copies of DB

**Redo logging:** need to keep all modified blocks in memory until commit

# Solution: Undo/Redo Logging!

Update = <Ti, X, new X val, old X val>

(X is the object updated)

# Undo/Redo Logging Rules

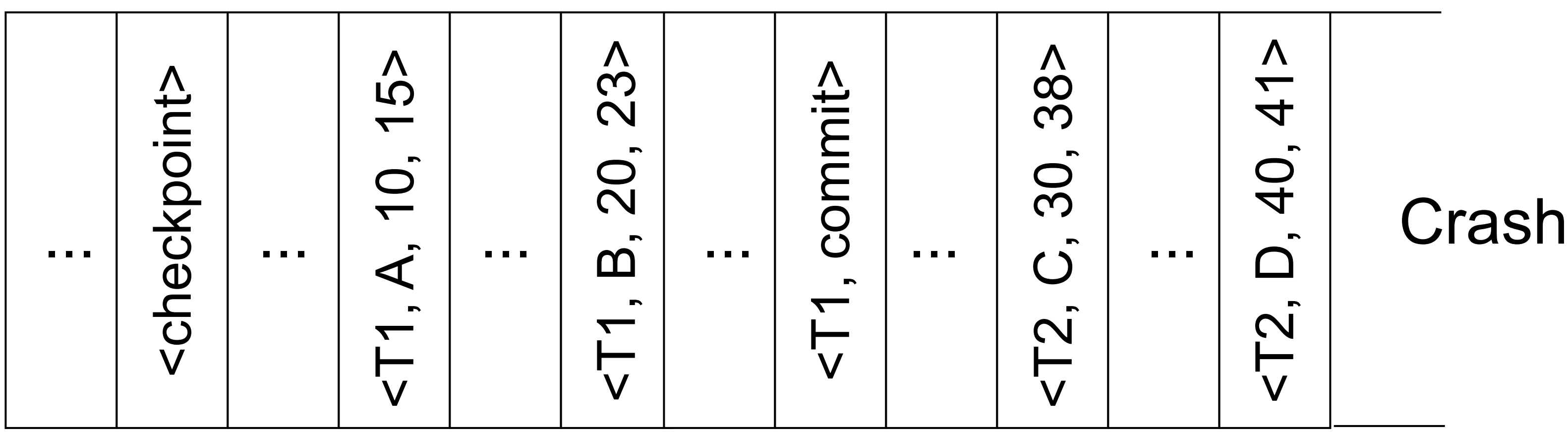
Object X can be flushed **before or after** Ti commits

Log record (with undo/redo info) must be flushed before corresponding data (WAL)

Flush only commit record at Ti commit

# Undo/Redo Logging: What to Do at Recovery?

Undo/redo log (disk):

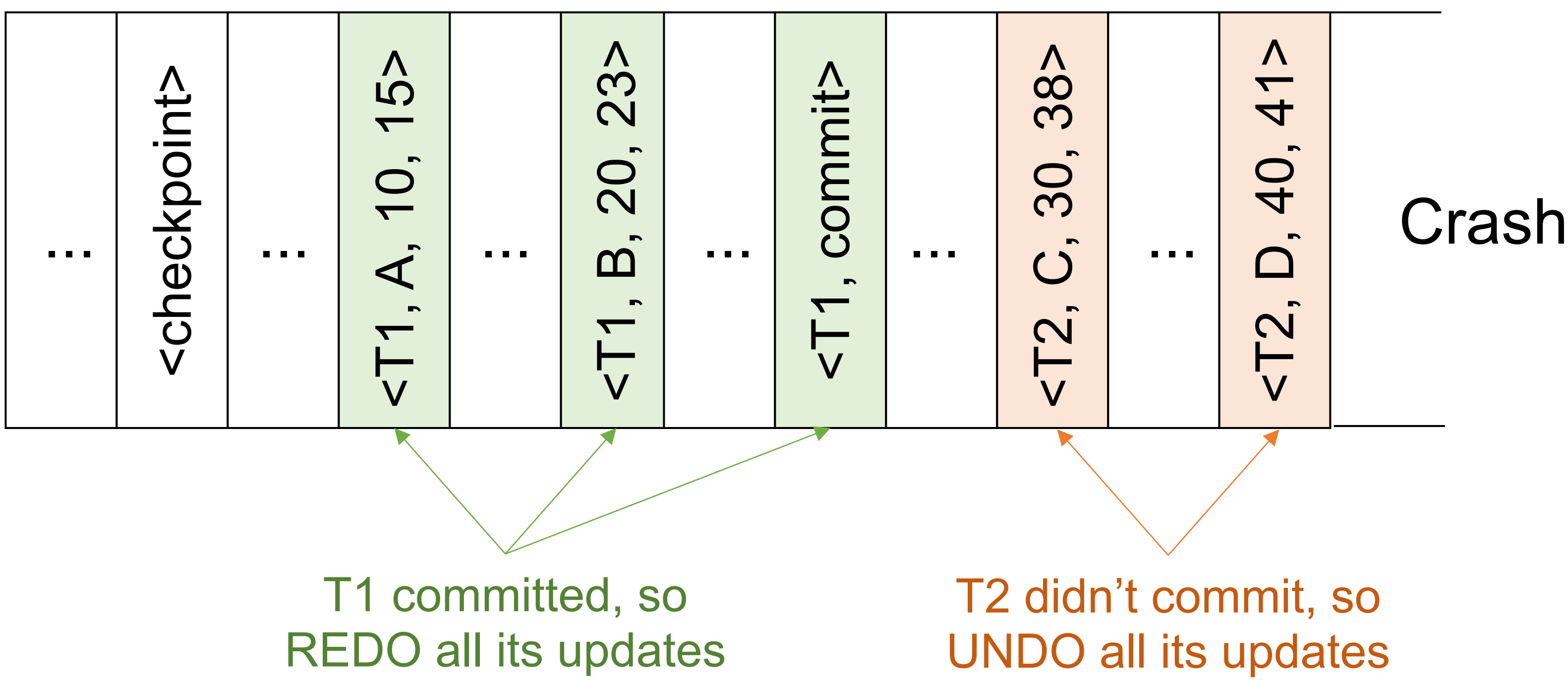


# Undo/Redo Logging: What to Do at Recovery?

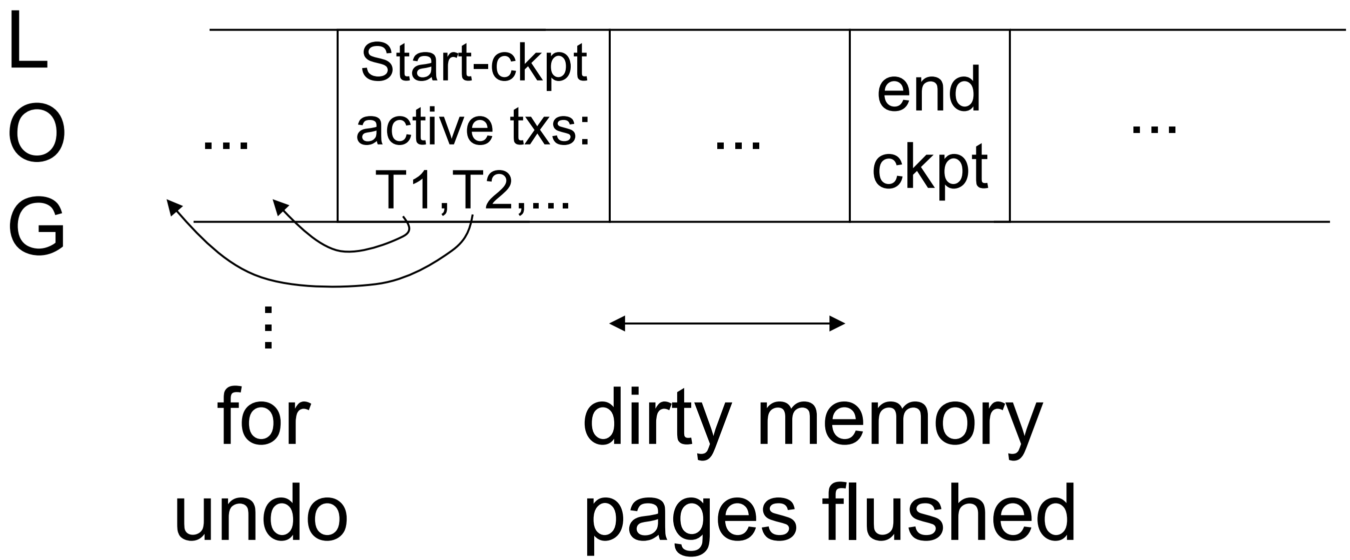
Undo/redo log (disk):

# Undo/Redo Logging: What to Do at Recovery?

Undo/redo log (disk):



# Non-Quiescent Checkpoints

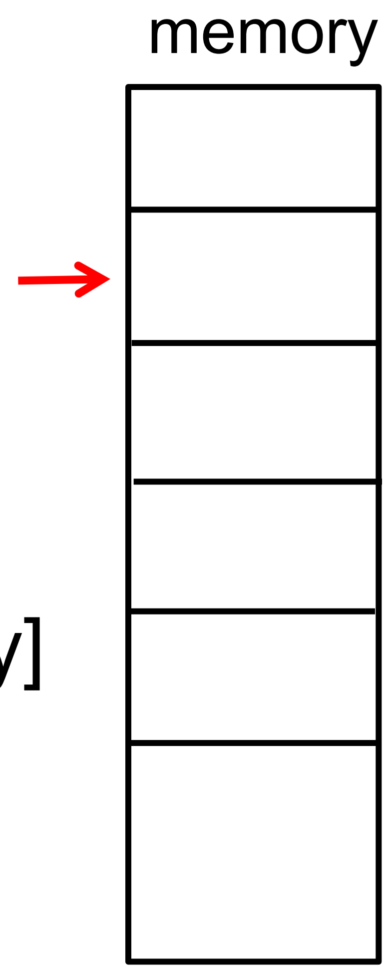


# Non-Quiescent Checkpoints

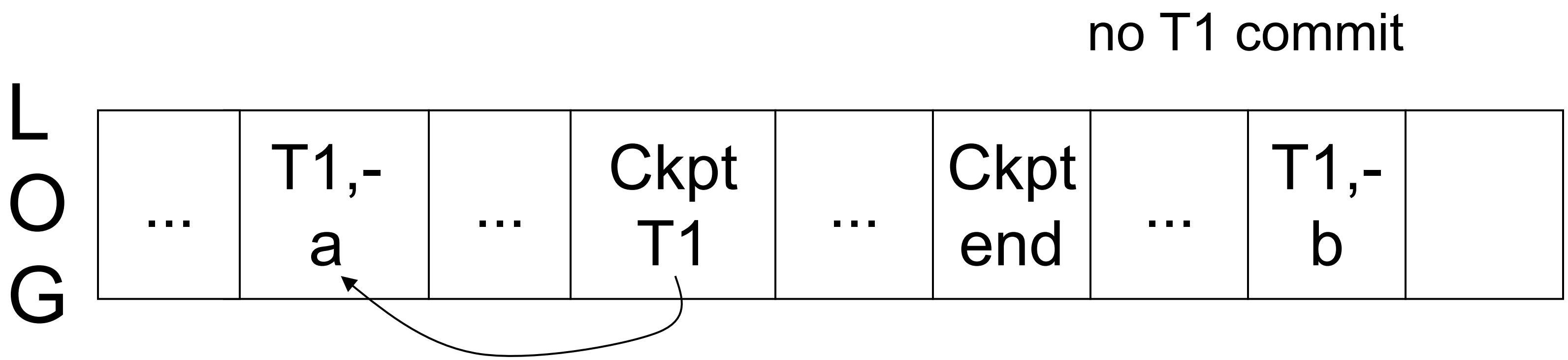
checkpoint process:

for i := 1 to M do
        Output(buffer i)

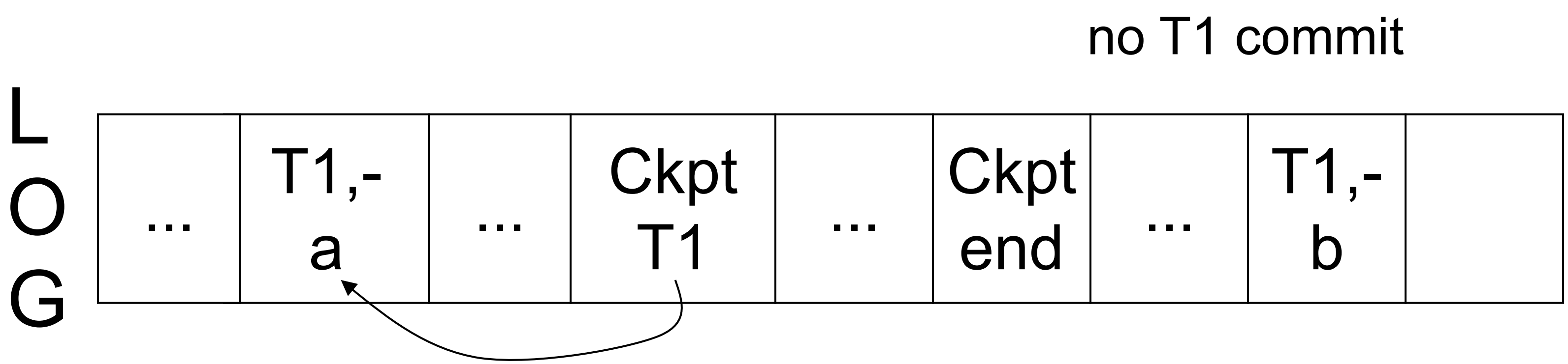
[transactions run concurrently]



# Example 1: How to Recover?

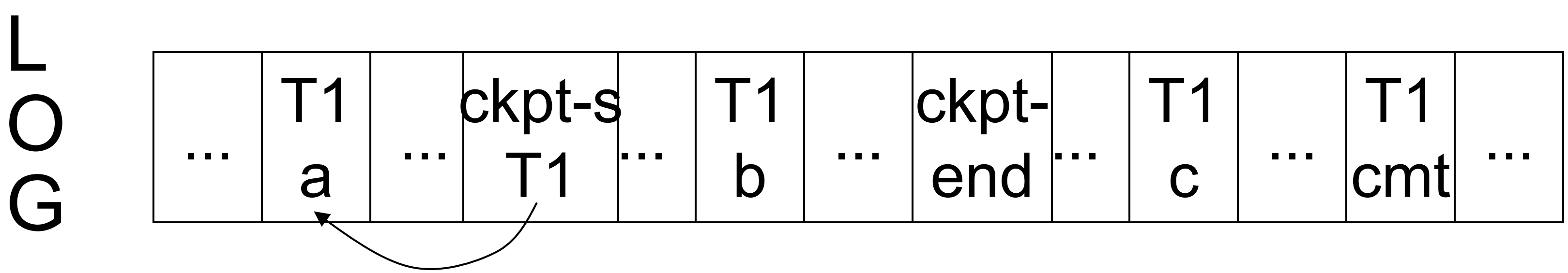


# Example 1: How to Recover?

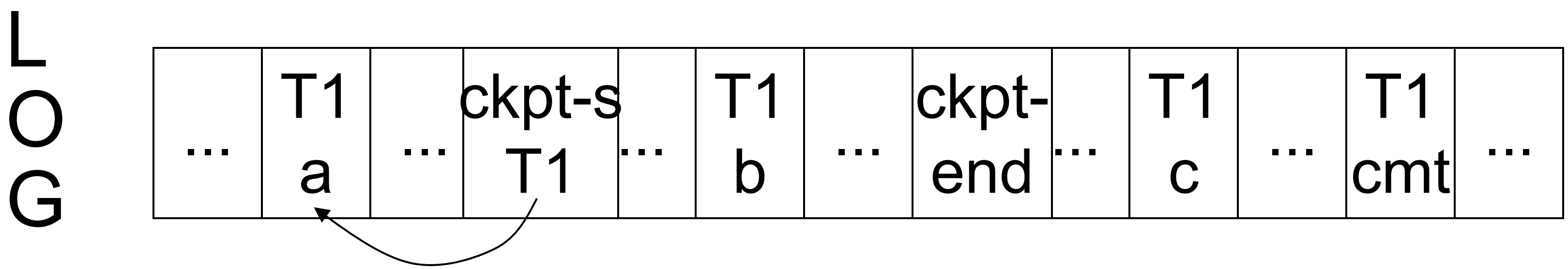



Undo T1  (undo a,b)

# Example 2: How to Recover?

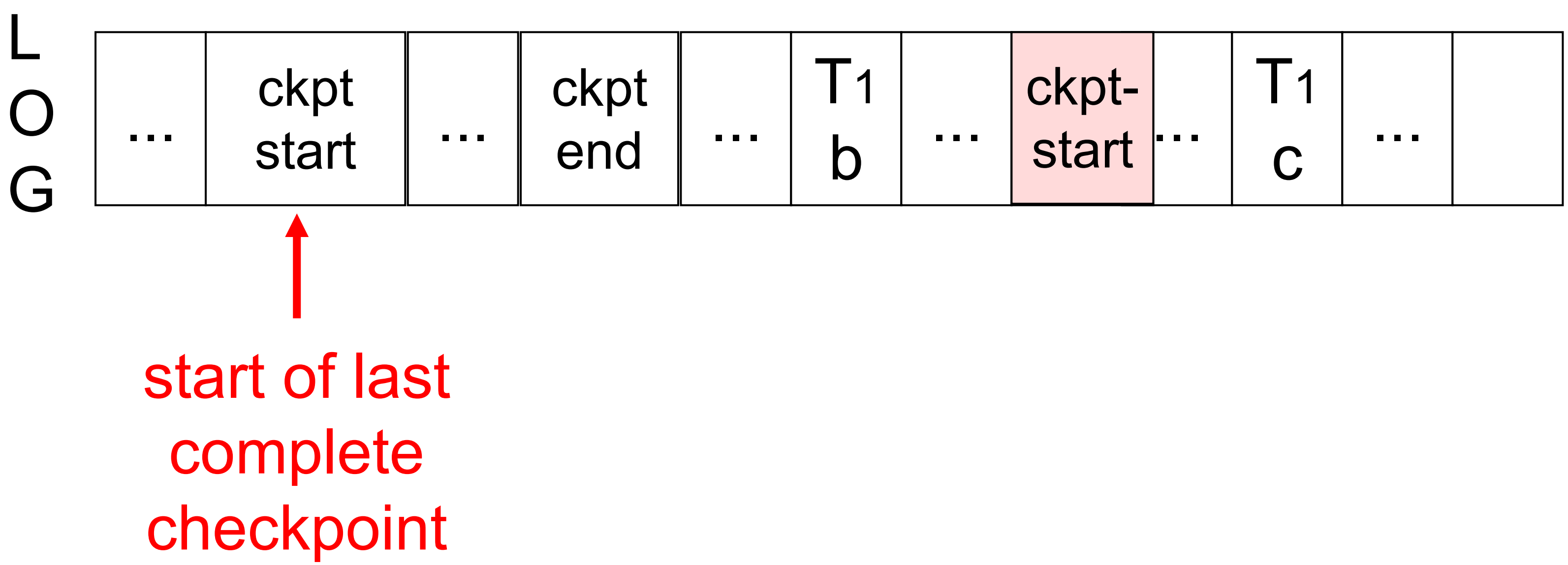


# Example 2: How to Recover?




Redo T1  (redo b,c)

# What if a Checkpoint Did Not Complete?




Start recovery from last **complete** checkpoint

# Undo/Redo Recovery Algorithm

Backward pass (end of log → latest valid checkpoint start)

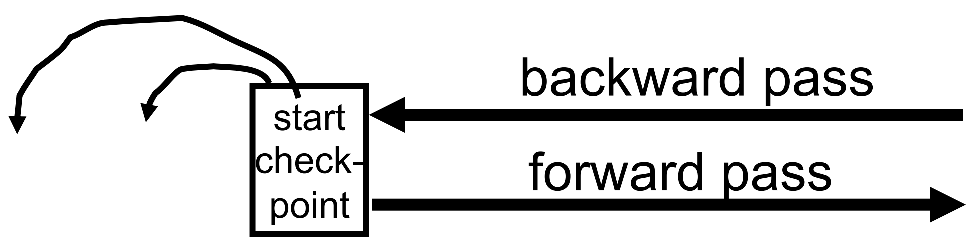
- » construct set S of committed transactions
- » undo actions of transactions not in S

Undo pending transactions

- » follow undo chains for transactions in (checkpoint's active list) - S

Forward pass (latest checkpoint start → end of log)

- » redo actions of all transactions in S



# Outline

Recap from last time

Undo/redo logging

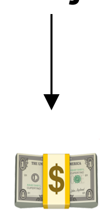
External actions

Media failures

# External Actions

E.g., dispense cash at ATM

$$Ti = a_1\ a_2 \ldots\ldots\ a_j \ldots\ldots\ a_n$$



# Solution

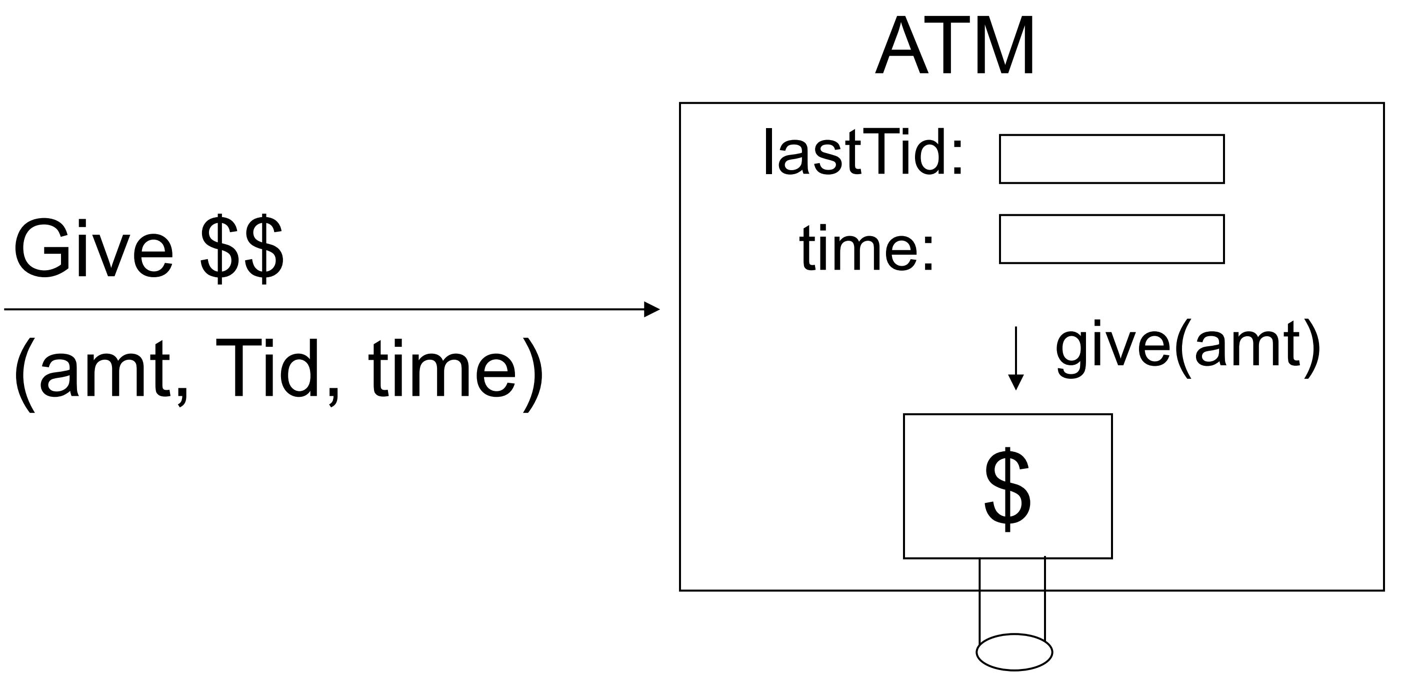
(1) Execute real-world actions after commit

(2) Try to make idempotent

# Solution

(1) Execute real-world actions after commit

(2) Try to make idempotent



# How Would You Handle These Other External Actions?

Charge a customer’s credit card

Cancel someone’s hotel room

Send data into a streaming system

# Outline

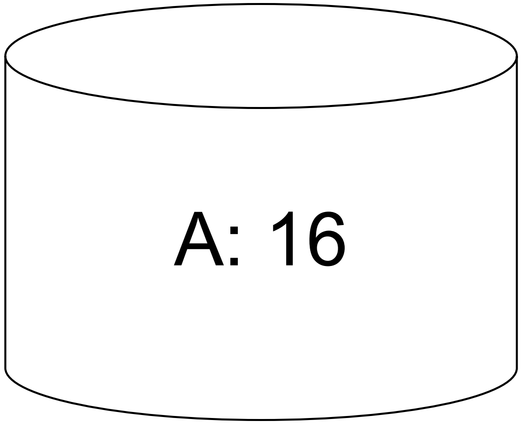
Recap from last time

Undo/redo logging

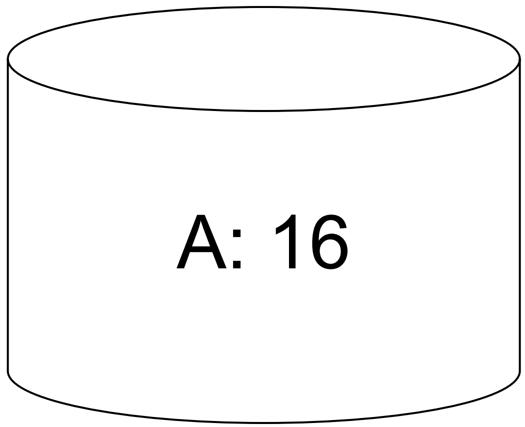
External actions

Media failures

# Media Failure (Loss of Nonvolatile Storage)



# Media Failure (Loss of Nonvolatile Storage)



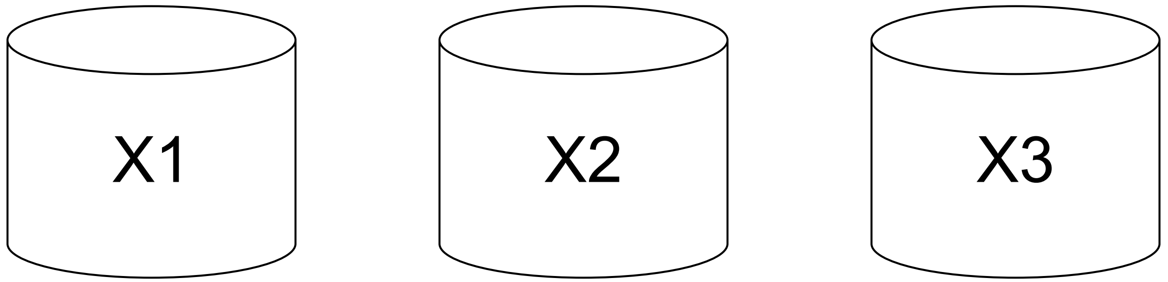

**Solution:** Make copies of data!

# Example 1: 3-Way Redundancy

Keep 3 copies on separate disks

Output(X) → three outputs

Input(X) → three inputs + vote



# Example 2: Redundant Writes, Single Reads

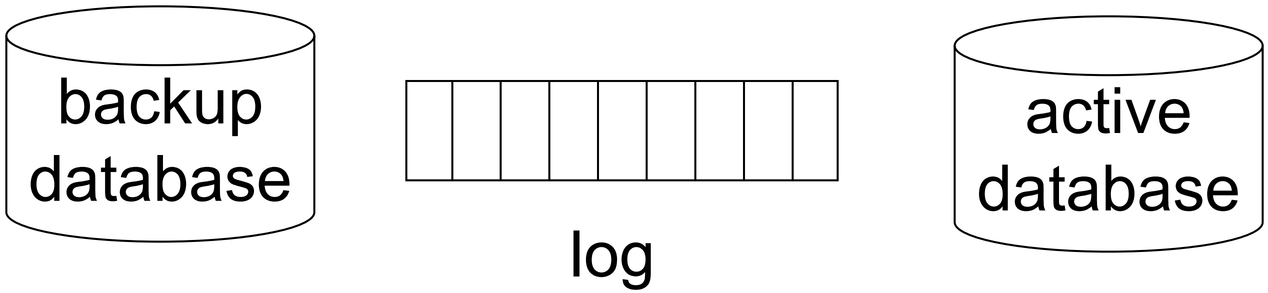
Keep N copies on separate disks

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

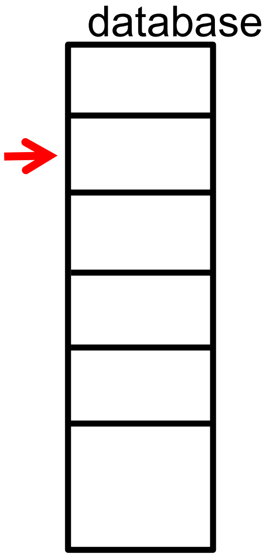
# Backup Database

Just like a checkpoint, except that we write the full database

create backup database:

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

[transactions run concurrently]



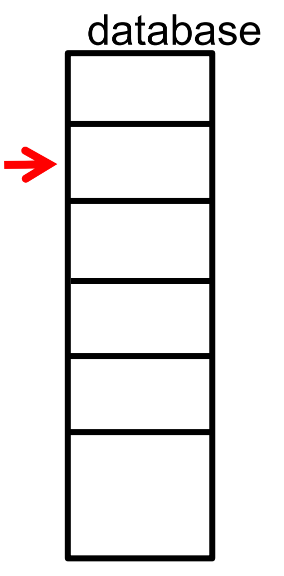
# Backup Database

Just like a checkpoint, except that we write the full database

create backup database:

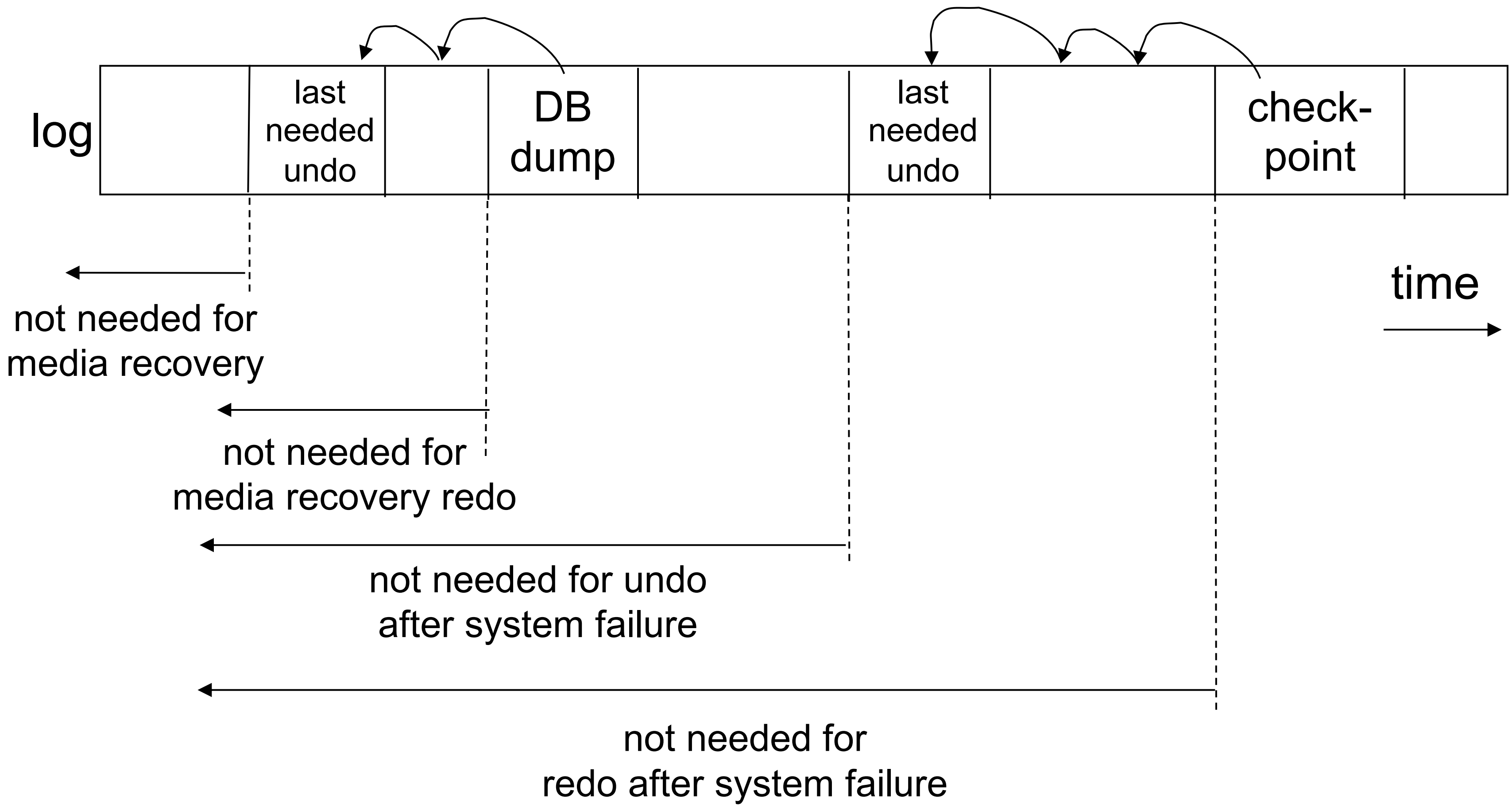
for i := 1 to DB_Size do
   [read DB block i; write to backup]

[transactions run concurrently]




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

# When Can Logs Be Discarded?



# Summary

Consistency of data: maintain constraints

One source of problems: failures

» Logging

» Redundancy

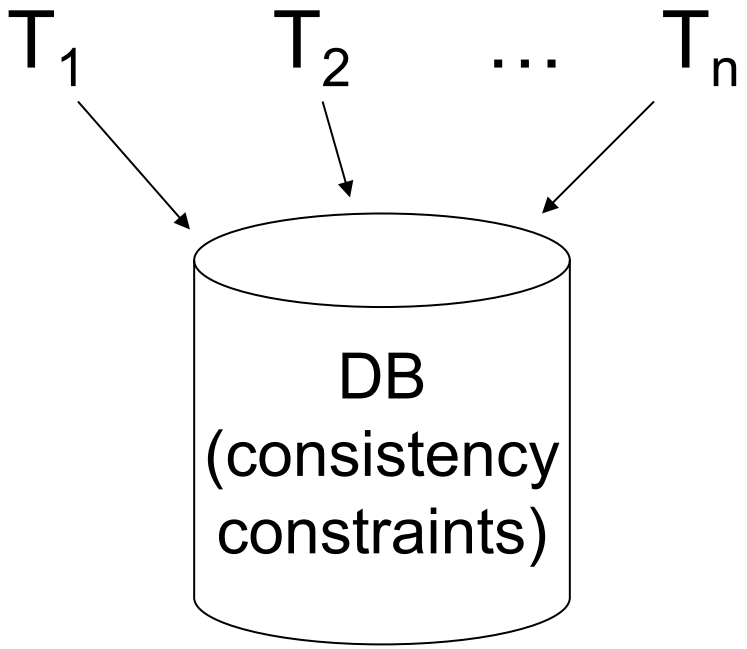
Another source of problems: data sharing

» We'll cover this next!

# Concurrency Control

Instructor: Matei Zaharia
cs245.stanford.edu

# The Problem




Different transactions may need to access data items at the same time, violating constraints

# Example

Constraint: all interns have equal salaries

$T_1$: add $1000 to each intern's salary

$T_2$: double each intern's salary

| | | | | | |
|---|---|---|---|---|---|
| Salaries: | ~~2000~~ | ~~2000~~ | ~~2000~~ | ~~2000~~ | ~~2000~~ |
| | ~~3000~~ | ~~3000~~ | ~~3000~~ | ~~4000~~ | ~~4000~~ |
| | 6000 | 6000 | 6000 | 5000 | 5000 |

😱

# The Problem

Even if each transaction maintains constraints by itself, interleaving their actions does not

Could try to run just one transaction at a time (**serial schedule**), but this has problems

- » **Too slow!** Especially with external clients & IO

# High-Level Approach

Define **isolation levels**: sets of guarantees about what transactions may experience

Strongest level: **serializability** (result is same as some serial schedule)

Many others possible: **snapshot isolation**, **read committed**, **read uncommitted**, …

# Fundamental Tradeoff

Stronger isolation level

Weaker isolation level

Easier to reason about (can’t see others’ changes)

See others’ changes, but more concurrency

# Interesting Fact

SQL standard defines serializability as "same as a serial schedule", but then also lists 3 types of "anomalies" to define levels:

| Isolation Level | Dirty Reads | Unrepeatable Reads | Phantom Reads |
|---|---|---|---|
| Read uncommitted | Y | Y | Y |
| Read committed | N | Y | Y |
| Repeatable read | N | N | Y |
| Serializable | N | N | N |

# Interesting Fact

There are isolation levels other than serializability that meet the last definition!

» I.e. don't exhibit those 3 anomalies

Virtually no commercial DBs do serializability by default, and some can't do it at all

Time to call the lawyers?

# In This Course

We’ll first discuss how to offer serializability

- » Many ideas apply to other isolation levels

We’ll see other isolation levels after

# Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking

» Shared and exclusive locks

» Lock tables and multi-level locking

Optimistic concurrency with validation

# Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking

- » Shared and exclusive locks
- » Lock tables and multi-level locking

Optimistic concurrency with validation

# Example

$T_1$: Read(A)

$A \leftarrow A+100$

Write(A)

Read(B)

$B \leftarrow B+100$

Write(B)

$T_2$: Read(A)

$A \leftarrow A \times 2$

Write(A)

Read(B)

$B \leftarrow B \times 2$

Write(B)

Constraint: A=B

# Schedule A

| $T_1$ | $T_2$ |
|---|---|
| Read(A); $A \leftarrow A+100$ | |
| Write(A); | |
| Read(B); $B \leftarrow B+100$; | |
| Write(B); | |
| | Read(A); $A \leftarrow A \times 2$; |
| | Write(A); |
| | Read(B); $B \leftarrow B \times 2$; |
| | Write(B); |

# Schedule A

| $T_1$ | $T_2$ | A | B |
|---|---|---|---|
| | | 25 | 25 |
| Read(A); A ← A+100 | | | |
| Write(A); | | 125 | |
| Read(B); B ← B+100; | | | |
| Write(B); | | | 125 |
| | Read(A); A ← A×2; | | |
| | Write(A); | 250 | |
| | Read(B); B ← B×2; | | |
| | Write(B); | | 250 |
| | | 250 | 250 |

# Schedule B

| $T_1$ | $T_2$ |
|---|---|
| | Read(A); $A \leftarrow A\times 2$; |
| | Write(A); |
| | Read(B); $B \leftarrow B\times 2$; |
| | Write(B); |
| Read(A); $A \leftarrow A+100$ | |
| Write(A); | |
| Read(B); $B \leftarrow B+100$; | |
| Write(B); | |

# Schedule B

| $T_1$ | $T_2$ | A | B |
|---|---|---|---|
| | | 25 | 25 |
| | Read(A); A $\leftarrow$ A$\times$2; | | |
| | Write(A); | 50 | |
| | Read(B); B $\leftarrow$ B$\times$2; | | |
| | Write(B); | | 50 |
| Read(A); A $\leftarrow$ A+100 | | | |
| Write(A); | | 150 | |
| Read(B); B $\leftarrow$ B+100; | | | |
| Write(B); | | | 150 |
| | | 150 | 150 |

# Schedule C

| $T_1$ | $T_2$ |
|---|---|
| Read(A); $A \leftarrow A+100$ | |
| Write(A); | |
| | Read(A); $A \leftarrow A\times2$; |
| | Write(A); |
| Read(B); $B \leftarrow B+100$; | |
| Write(B); | |
| | Read(B); $B \leftarrow B\times2$; |
| | Write(B); |

# Schedule C

| T1 | T2 | A | B |
|---|---|---|---|
| | | 25 | 25 |
| Read(A); $A \leftarrow A+100$ | | | |
| Write(A); | | 125 | |
| | Read(A); $A \leftarrow A \times 2$; | | |
| | Write(A); | 250 | |
| Read(B); $B \leftarrow B+100$; | | | |
| Write(B); | | | 125 |
| | Read(B); $B \leftarrow B \times 2$; | | |
| | Write(B); | | 250 |
| | | 250 | 250 |

# Schedule D

| T1 | T2 |
| --- | --- |
| Read(A); A $\leftarrow$ A+100 | |
| Write(A); | |
| | Read(A); A $\leftarrow$ A$\times$2; |
| | Write(A); |
| | Read(B); B $\leftarrow$ B$\times$2; |
| | Write(B); |
| Read(B); B $\leftarrow$ B+100; | |
| Write(B); | |

# Schedule D

| T1 | T2 | A | B |
|---|---|---|---|
| | | 25 | 25 |
| Read(A); A $\leftarrow$ A+100 | | | |
| Write(A); | | 125 | |
| | Read(A); A $\leftarrow$ A$\times$2; | | |
| | Write(A); | 250 | |
| | Read(B); B $\leftarrow$ B$\times$2; | | |
| | Write(B); | | 50 |
| Read(B); B $\leftarrow$ B+100; | | | |
| Write(B); | | | 150 |
| | | 250 | 150 |

# Schedule E

Same as Schedule D
but with new T2'

| T1 | T2 |
|---|---|
| Read(A); A ← A+100 | |
| Write(A); | |
| | Read(A); A ← A+50; |
| | Write(A); |
| | Read(B); B ← B+50; |
| | Write(B); |
| Read(B); B ← B+100; | |
| Write(B); | |

# Schedule E

Same as Schedule D
but with new $T_2'$

| $T_1$ | $T_2'$ | A | B |
|---|---|---|---|
| | | 25 | 25 |
| Read(A); A $\leftarrow$ A+100 | | | |
| Write(A); | | 125 | |
| | Read(A); A $\leftarrow$ A+50; | | |
| | Write(A); | 175 | |
| | Read(B); B $\leftarrow$ B+50; | | |
| | Write(B); | | 75 |
| Read(B); B $\leftarrow$ B+100; | | | |
| Write(B); | | | 175 |
| | | 175 | 175 |

# Our Goal

Want schedules that are "good", regardless of

» initial state and

» transaction semantics ← We don't know the logic in external client apps!

Only look at **order of read & write operations**

Example:

$$S_C = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$$

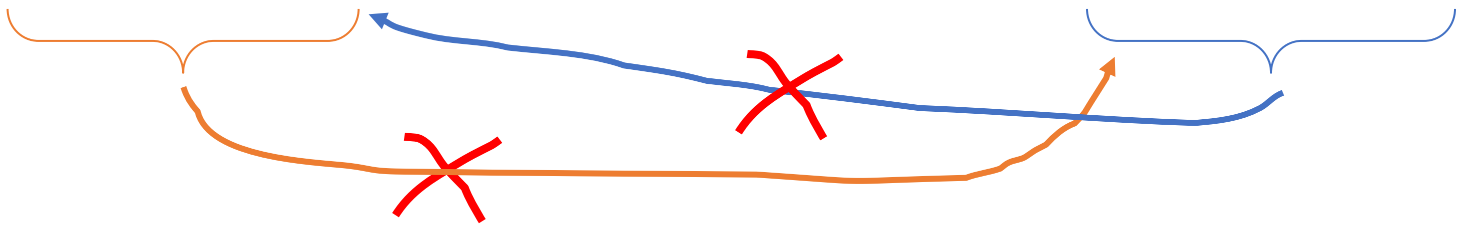
# Example:

$$S_C = r_1(A)w_1(A)r_2(A)w_2(A)r_1(B)w_1(B)r_2(B)w_2(B)$$

$$S_C' = \underbrace{r_1(A)w_1(A)r_1(B)w_1(B)}_{T_1}\underbrace{r_2(A)w_2(A)r_2(B)w_2(B)}_{T_2}$$

However, for $S_D$:

$$S_D = r_1(A)w_1(A)r_2(A)w_2(A)r_2(B)w_2(B)r_1(B)w_1(B)$$



Another way to view this:

- » $r_1(B)$ after $w_2(B)$ means $T_1$ should be after $T_2$ in an equivalent serial schedule ($T_2 \rightarrow T_1$)
- » $r_2(A)$ after $w_1(A)$ means $T_2$ should be after $T_1$ in an equivalent serial schedule ($T_1 \rightarrow T_2$)
- » Can't have both of these!

# Outline

What makes a schedule serializable?

Conflict serializability

Precedence graphs

Enforcing serializability via 2-phase locking

» Shared and exclusive locks

» Lock tables and multi-level locking

Optimistic concurrency with validation