Failure Recovery

Data Integrity

- Protect data from system failures
  - Key Idea: Logs recording change history.
  - Today. Chapter 17.
- Maintain data integrity, when several queries/modifications are run together.
  - Key Idea: Locks granting controlled access.
  - Next week(s). Chapter 18 and 19.

DB State

- DB state: Value for each db element.
- Consistent DB state: That satisfies all database constraints (key, value constraints etc.)

Transactions

- Transactions: Processes that query and modify database.
  - Executes a number of steps in sequence.
  - State: gives what has been done so far.
- Desirable properties: "ACID"
  - Atomic = Either all work is done, or none of it.
  - Consistent = Database constraints are preserved.
  - Isolated = Appear as if one process executes at a time. Often called *serializable* behavior.
  - Durable = Effects are permanent irrespective of system crashes.
- Commit/Abort actions (next slide)
- Consistency is assumed.

Commit/Abort Decision

Each transaction ends with either:

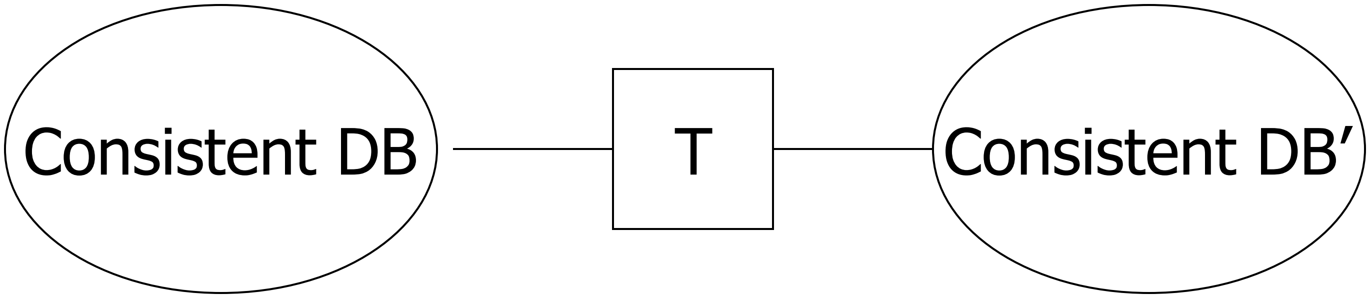
- Commit = Causes the transaction to complete. It's database modifications are now permanent in the database; previously its changes may be invisible to other transactions.
- Abort = no changes by the transaction appear in the database; it is as if the transaction never occurred.
  - ROLLBACK is the term used in SQL and Oracle.
  - Failures like division by 0 can also cause abortion/rollback, even if the user doesn't request it.

Consistency Property (Assumption)

If T starts with consistent state AND

T executes in isolation

$\Rightarrow$ T leaves consistent state



Failure Recovery Motivation

- Transactions should be atomic
  - Whole execution or nothing at all
- Challenge: Concurrent transactions may cause problems unless controlled.

Failure Possibilities

- Transaction bug
- DBMS bug
- Hardware failure

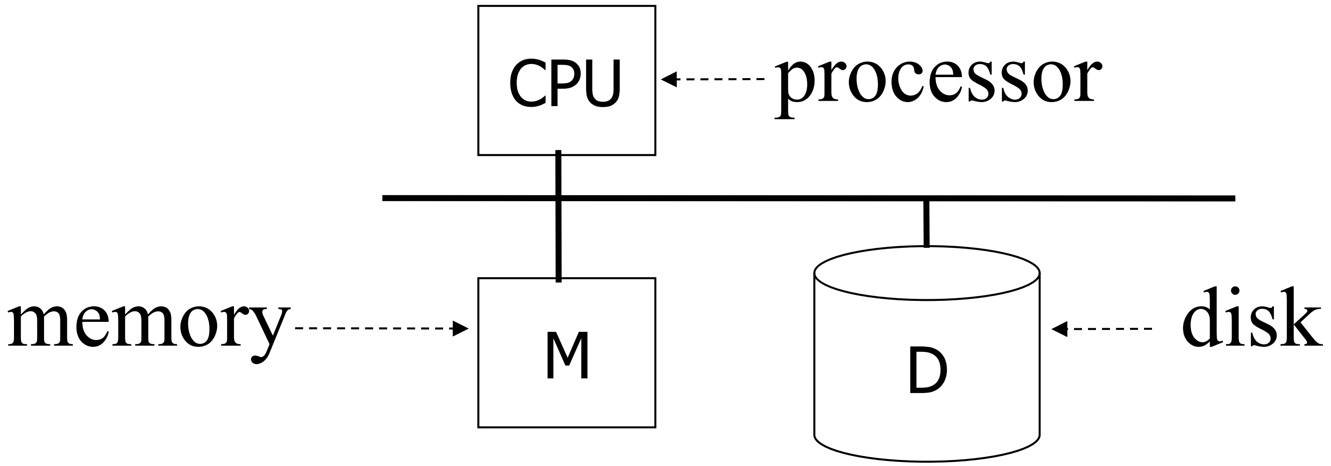
  e.g., disk crash alters balance of account

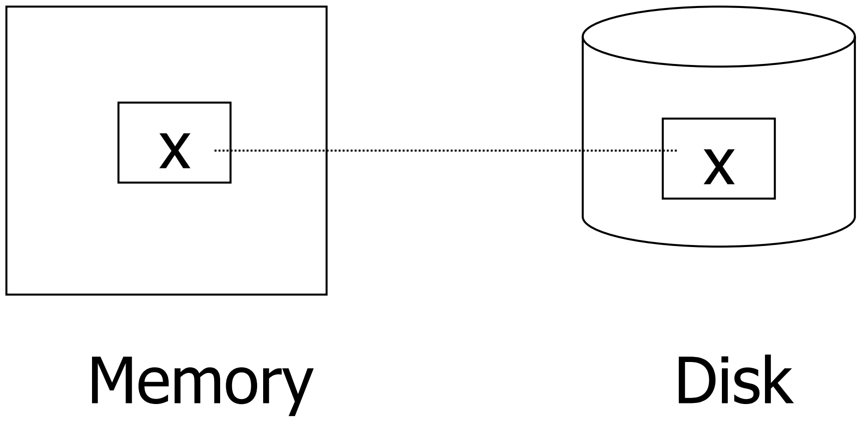
- Data sharing

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

  T2: change programmers $\Rightarrow$ systems analysts

Model





Operations

Let Block(x) be the block containing x

- Input (x): Block(x) $\rightarrow$ memory
- Output (x): Block(x) $\rightarrow$ disk
- Read (x,t): do input(x), if necessary
  $t \leftarrow x$
- Write (x,t): do input(x), if necessary
  $x \leftarrow t$

Key problem Unfinished transaction

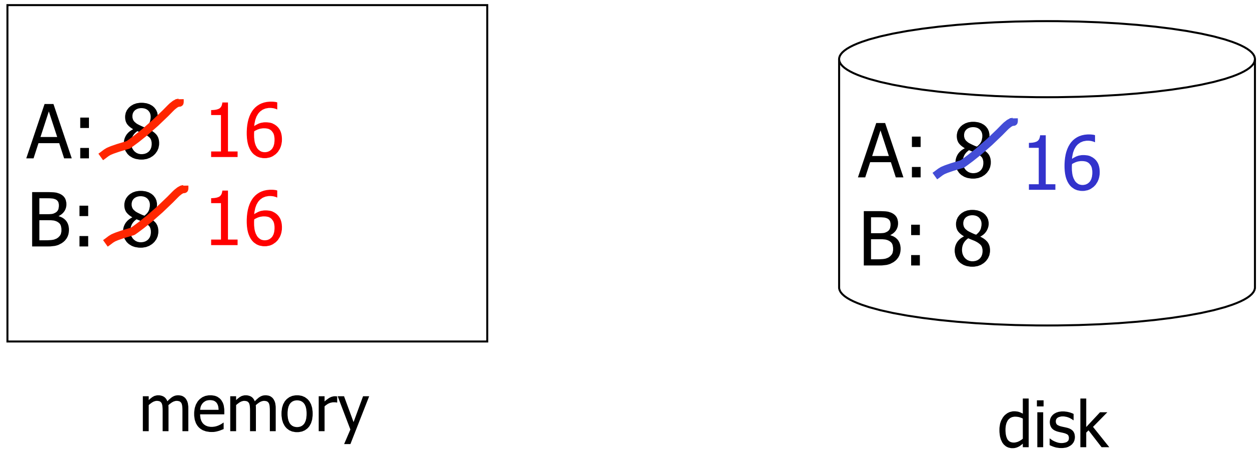
Example Constraint: A=B

T_1: $A \leftarrow A \times 2$

$B \leftarrow B \times 2$

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

failure!



How to ensure atomicity?

Solution: Log Old Values Before Disk-Change

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

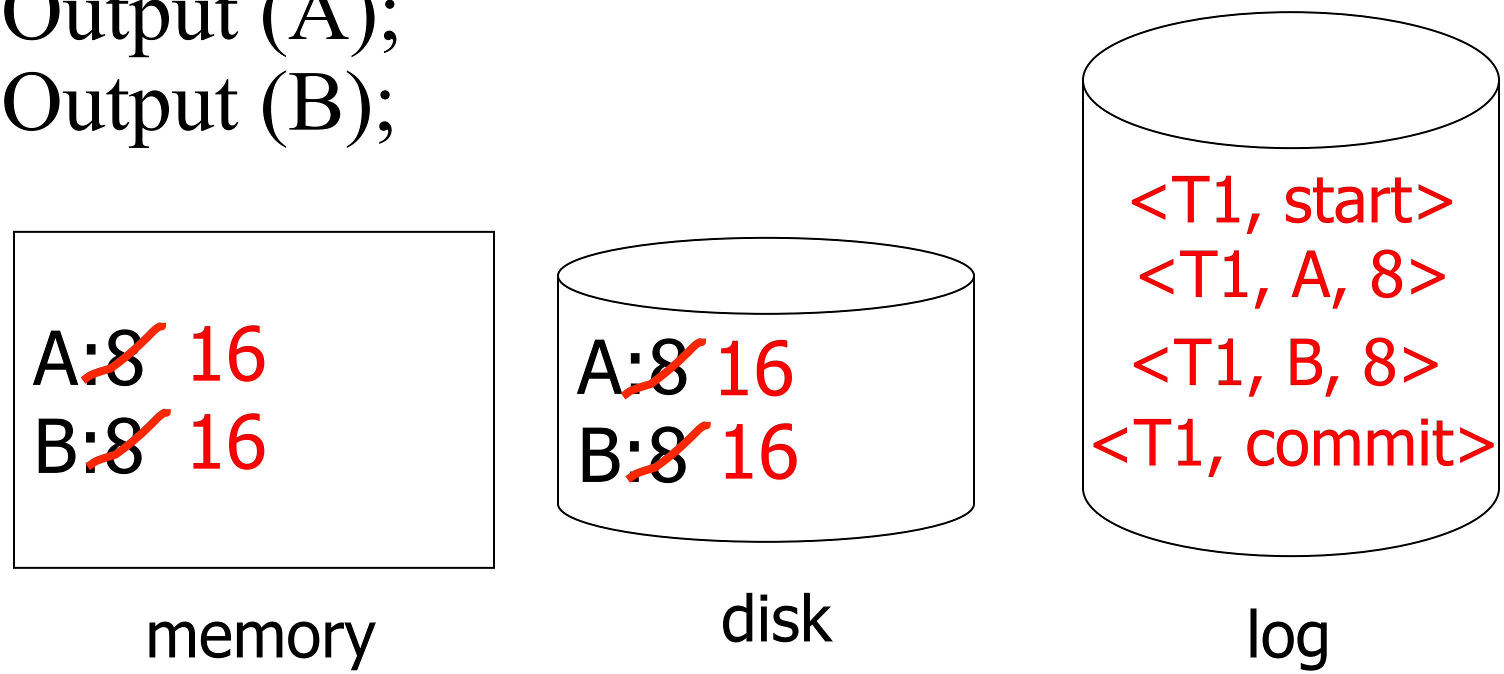
Write (A,t);

Read (B,t); $t \leftarrow t \times 2$

Write (B,t);

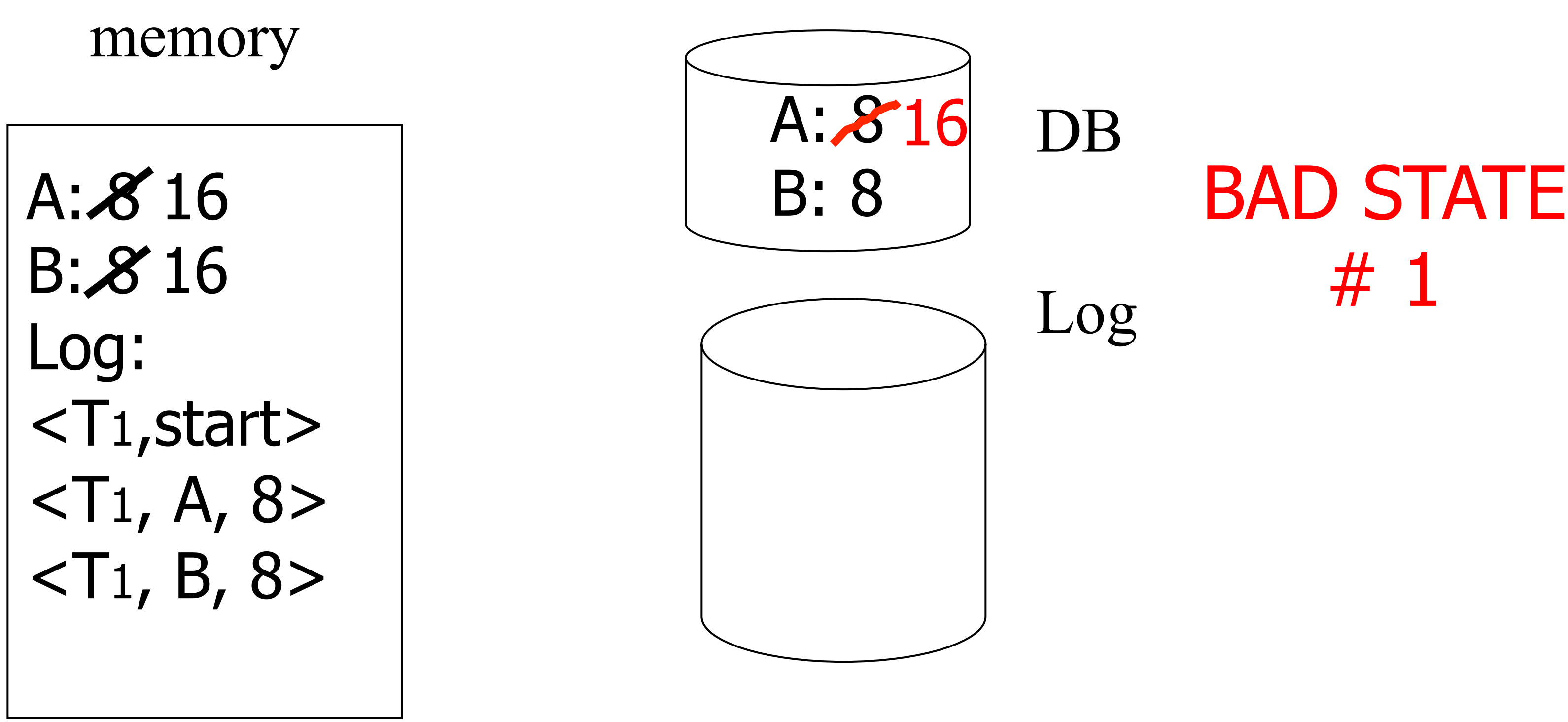
Output (A);

Output (B);



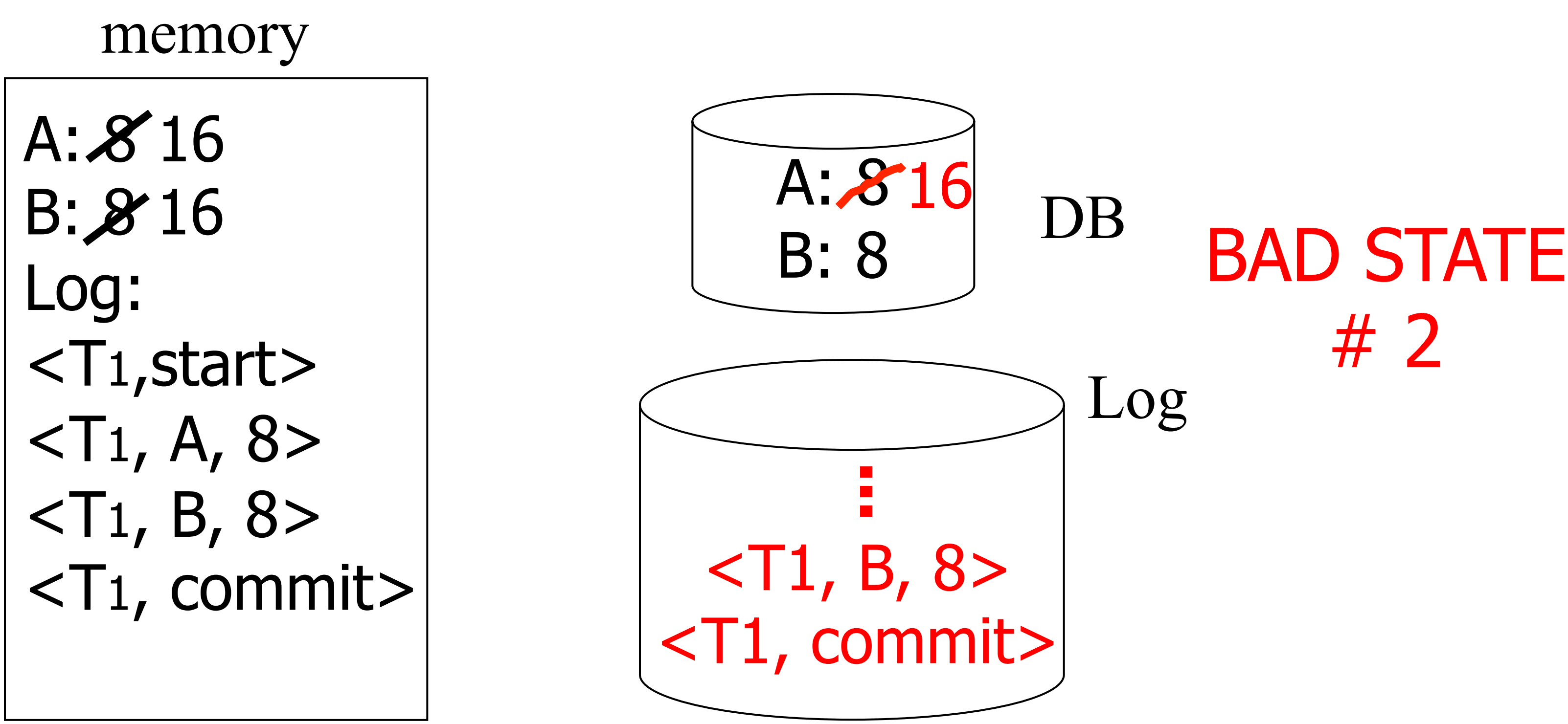
First "complication"

- When should be log be written onto disk?



Second "complication"

- What if Write(x) is not written to disk before "commit" log?



Solution: Undo Logging Rules

For a Write(x,V) in Ti

1. Generate log(x) = <Ti, x, oldValue(x)>
2. Output(log(x)) before Output(x)

   [WAL: Write Ahead Logging]

Output(x) before Output(<Ti,Commit>)

Recovery in Undo logging

- Basic Idea: For every transaction that hasn't been committed in the log, abort it.
- Commit check: <Ti, Commit> in log.
- Abort: Write back old values on disk using the logs, and write <Ti, Abort> in log.

Recovery rules: 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 → earliest) do:
- if Ti ∈ S then { - write (X, v)
- output (X) }

(3) For each Ti ∈ S do
- write <Ti, abort> to log

What if failure during recovery?

No problem!

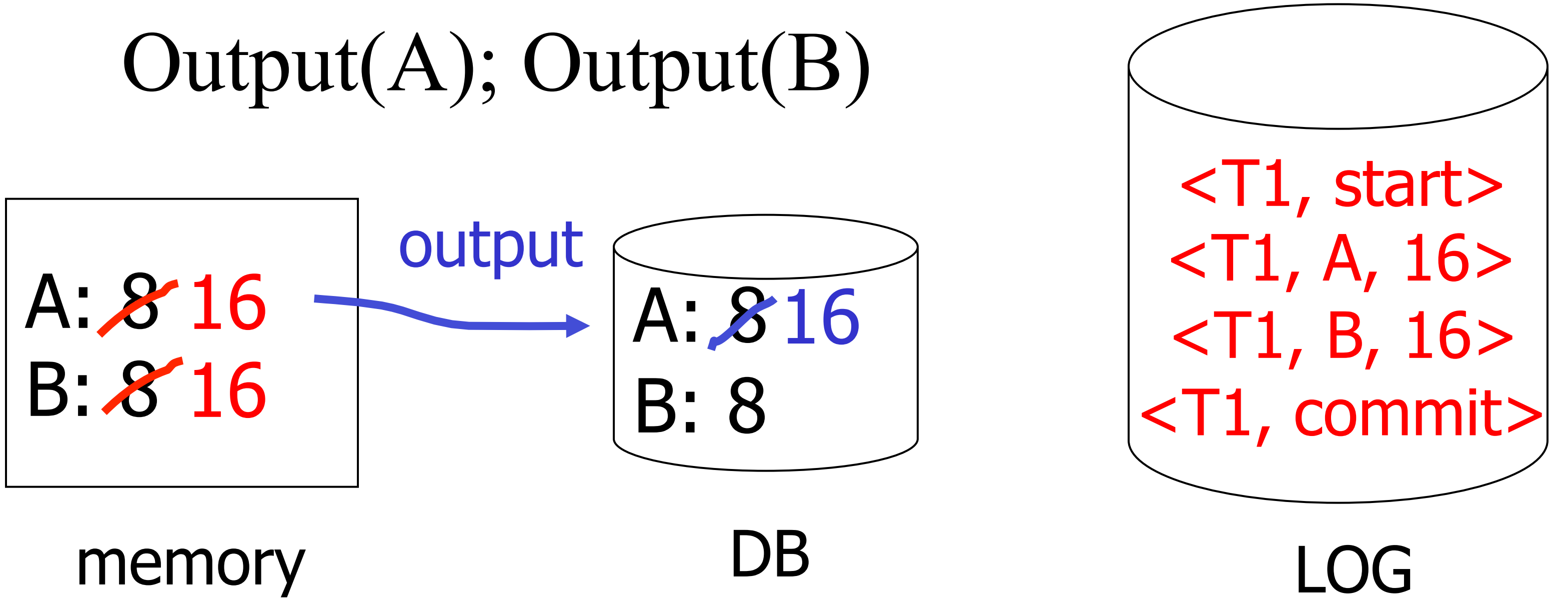
Undo is idempotent

To discuss:

- Redo logging
- Undo/redo logging, why both?
- Checkpoints

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

For a Write(x,V) in Ti

1. Generate log(x) = <Ti, x, newValue(x)>

To commit:

- Flush log including <Ti, commit>
- Output(x) for all x in Ti.

Recovery rules: Redo logging

(1) Let S = set of transactions with
<Ti, commit> in log

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

- if Ti ∈ S then { Write(X, v)
  Output(X) ← optional

Undo vs. Redo Logging

Order of writes

Output(<Ti, x, old/new>)   (old for undo)

…

← Undo: Output(x).

…

Output(<Ti, commit>)

← Redo: Output(x).

Undo: <Ti, commit> on disk implies Ti completed.

Redo: No <Ti,commit> implies Ti did nothing.

Redo Recovery is very, very SLOW !

Redo log:

First Record (1 year ago)

T1 wrote A,B Committed a year ago

Last Record

Crash

--> STILL, Need to redo after crash!!

Solution: Checkpoint (simple version)

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

Example: what to do at recovery?

Redo log (disk):

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

Crash

Key drawbacks:

- *Undo logging:* need to flush all Write(x)'s before commit.
- *Redo logging:* can't flush Write(x)'s until commit

Solution: Undo/Redo Logging

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

ONLY Rule:

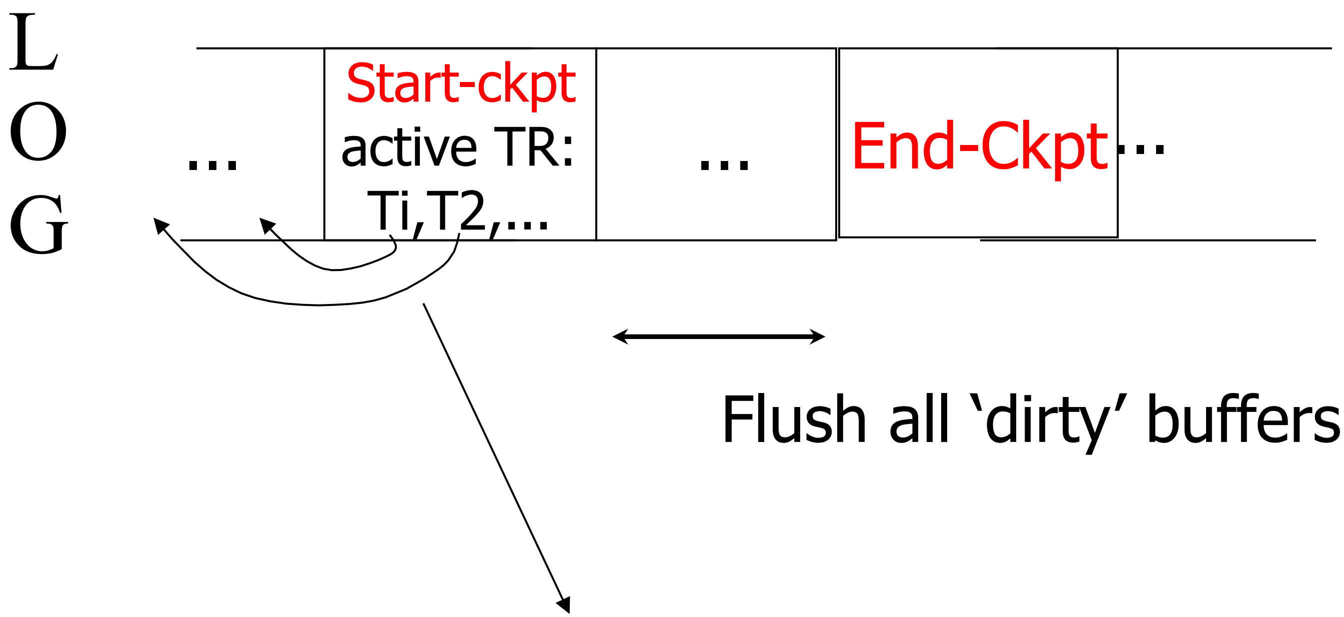
Output(log(x)) before Output(x)

Recovery:

Redo all committed TRs (earliest first)

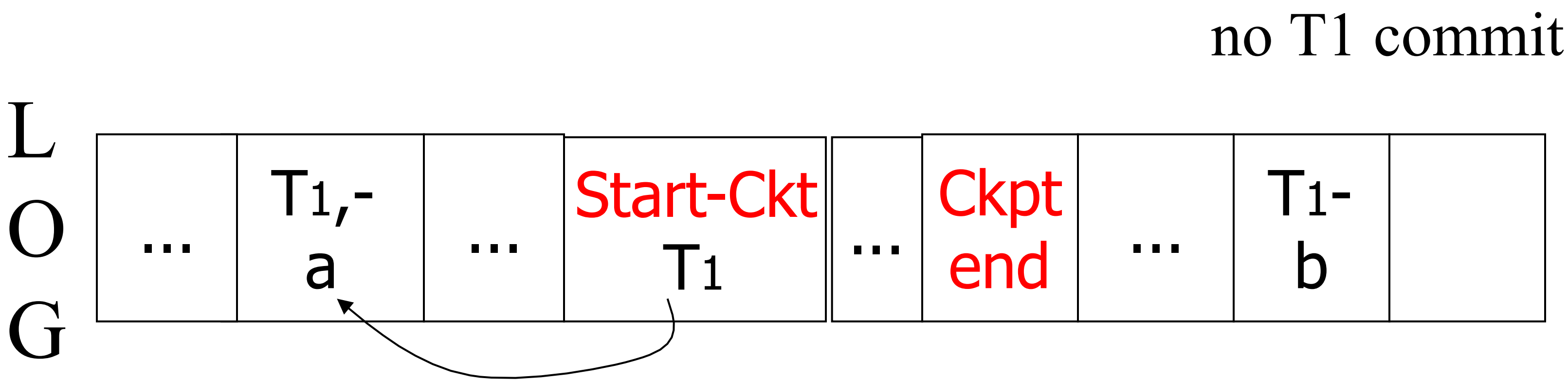
Undo all incomplete TRs (latest first)

Nonquiescent checkpoint



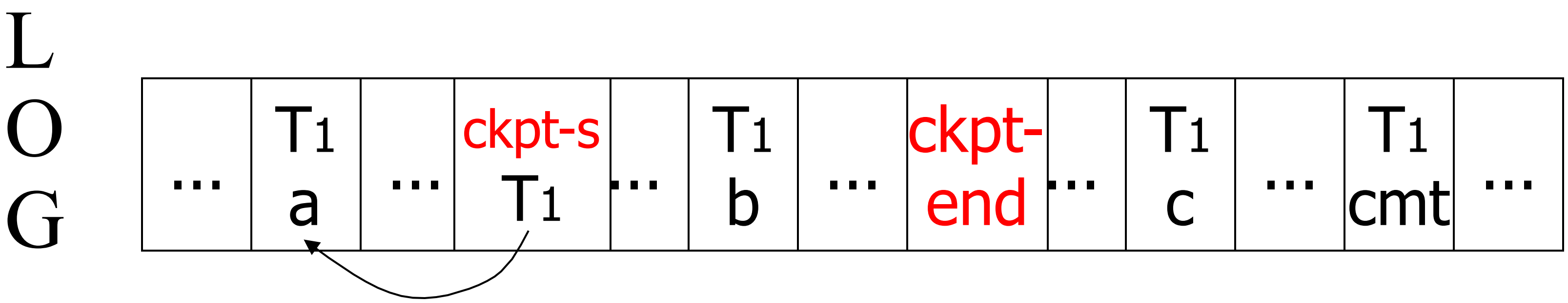
Log chains of active transactions.
Needed for ‘Undo’

Examples what to do at recovery time?



- Undo T_1 (undo a,b)

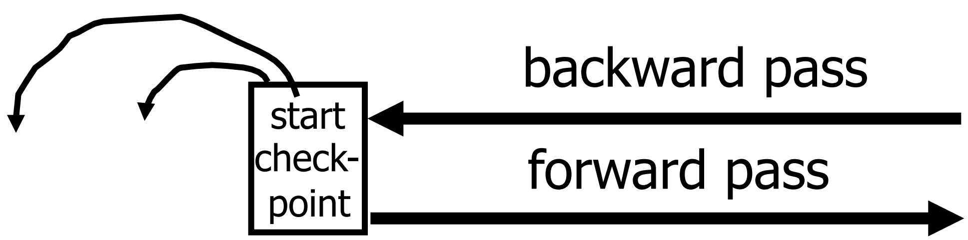
Example



- Redo T1: (redo b,c). No need to (redo a).

Recovery process:

- Backwards pass (end of log ➜ latest 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 checkpoint start ➜ end of log)
  - redo actions of S transactions



Summary

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