

CS 525: Advanced Database Organization

13: Failure and Recovery



Boris Glavic

Slides: adapted from a [course](#) taught by [Hector Garcia-Molina](#), Stanford InfoLab

Now

- Crash recovery

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

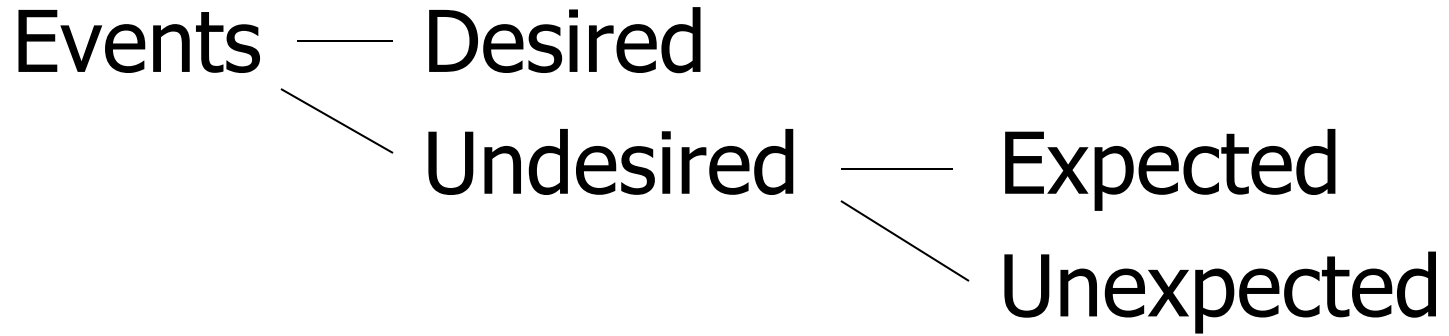
e.g.: T1: give 10% raise to programmers

T2: change programmers \Rightarrow systems analysts

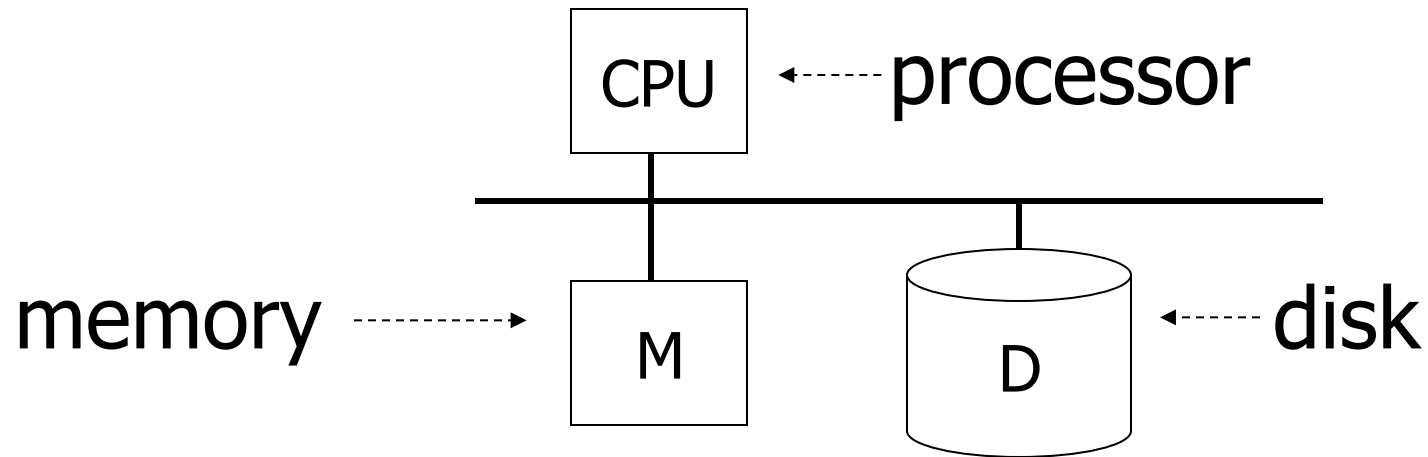
Recovery

- 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....

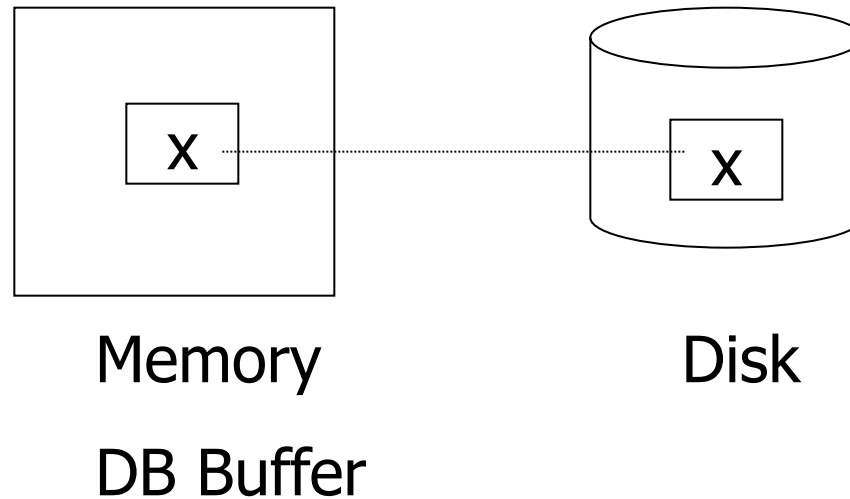
Is this model reasonable?

Approach: Add low level checks +
redundancy to increase
probability 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

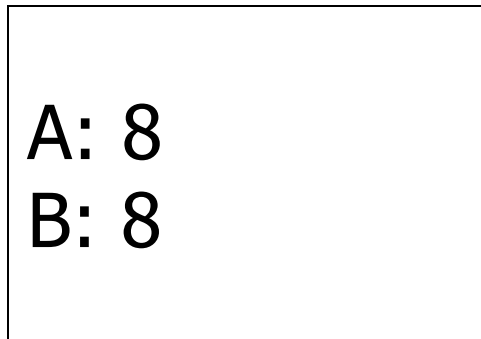
Example

Constraint: $A=B$

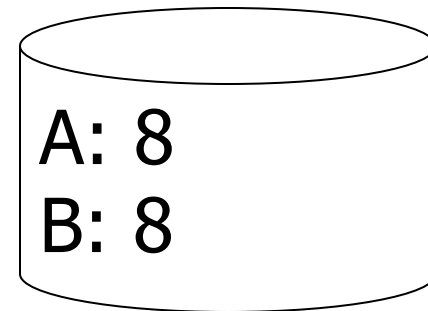
$T_1: A \leftarrow A \times 2$

$B \leftarrow B \times 2$

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

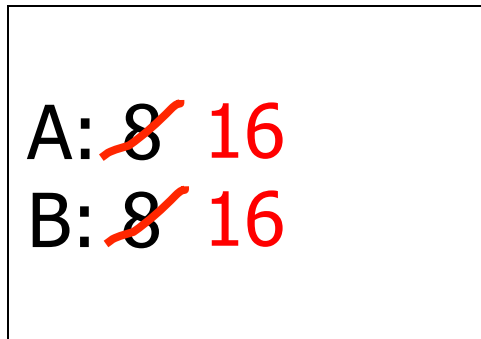


memory

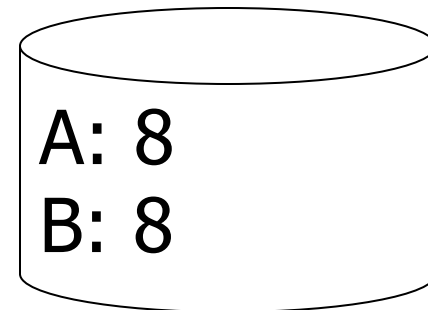


disk

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

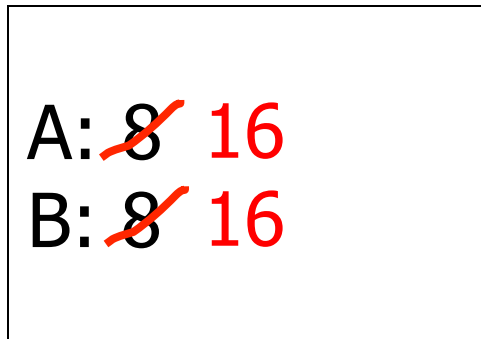


memory

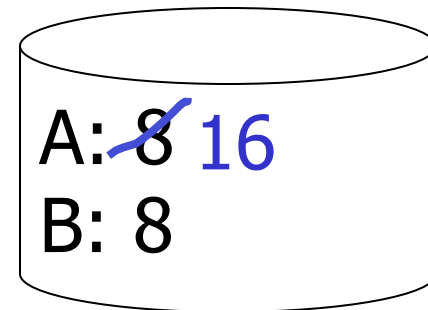


disk

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



memory



disk

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

How to restore consistent state after crash?

- Desired state after recovery:
 - Changes of committed transactions are reflected on disk
 - Changes of unfinished transactions are not reflected on disk
- After crash we need to
 - **Undo** changes of unfinished transactions that have been written to disk
 - **Redo** changes of finished transactions that have not been written to disk

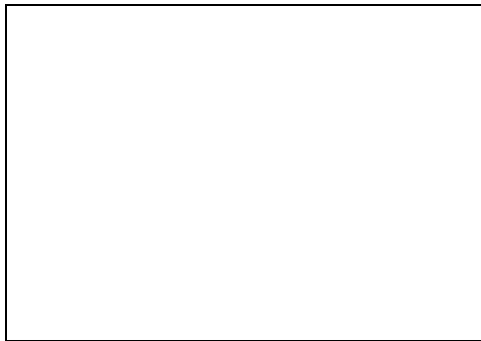
How to restore consistent state after crash?

- After crash we need to
 - **Undo** changes of unfinished transactions that have been written to disk
 - **Redo** changes of finished transactions that have not been written to disk
- We need to either
 - Store additional data to be able to Undo/Redo
 - Avoid ending up in situations where we need to Undo/Redo

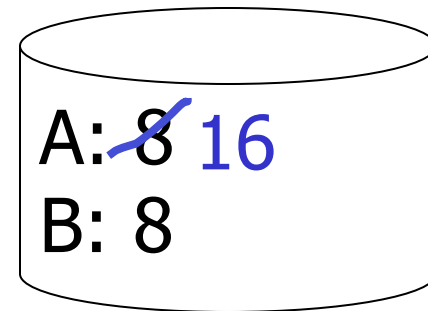
T₁: Read (A,t); t ← t×2
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);

T₁ is unfinished
-> need to undo the
write to A to recover
to consistent state

failure!



memory



disk

Logging

- After crash need to
 - **Undo**
 - **Redo**
- We need to know
 - Which operations have been executed
 - Which operations are reflected on disk
- -> **Log** upfront what is to be done

Buffer Replacement Revisited

- Now we are interested in knowing how buffer replacement influences recovery!

Buffer Replacement Revisited

- **Steal:** all pages with fix count = 0 are replacement candidates
 - Smaller buffer requirements
- **No steal:** pages that have been modified by active transaction -> not considered for replacement
 - No need to undo operations of unfinished transactions after failure

Buffer Replacement Revisited

- **Force:** Pages modified by transaction are flushed to disk at end of transaction
 - No redo required
- **No force:** modified (dirty) pages are allowed to remain in buffer after end of transaction
 - Less repeated writes of same page

Effects of Buffer Replacement

	force	No force
No steal	<ul style="list-style-type: none">• No Undo• No Redo	<ul style="list-style-type: none">• No Undo• Redo
steal	<ul style="list-style-type: none">• Undo• No Redo	<ul style="list-style-type: none">• Redo• Undo

Schedules and Recovery

- Are there certain schedules that are easy/hard/impossible to recover from?

Recoverable Schedules

- We should never have to rollback an already committed transaction (D in ACID)
- **Recoverable** schedules require that
 - A transaction does not commit before every transaction that it has read from has committed
 - A transaction **T** reads from another transaction **T'** if it reads an item X that has last been written by T' and T' has not aborted before the read

$$T_1 = w_1(X), c_1$$

$$T_2 = r_2(X), w_2(X), c_2$$

Recoverable Schedule

$$S_1 = w_1(X), r_2(X), w_2(X), c_1, c_2$$

Nonrecoverable Schedule

$$S_2 = w_1(X), r_2(X), w_2(X), c_2, c_1$$

Cascading Abort

- Transaction **T** has written an item that is later read by **T'** and **T** aborts after that
 - we have to also abort **T'** because the value it read is no longer valid anymore
 - This is called a **cascading abort**
 - Cascading aborts are complex and should be avoided

$S = \dots w_1(X) \dots r_2(X) \dots a_1$

Cascadeless Schedules

- **Cascadeless** schedules guarantee that there are no cascading aborts
 - Transactions only read values written by already committed transactions

$$T_1 = w_1(X), c_1$$

$$T_2 = r_2(X), w_2(X), c_2$$

Cascadeless Schedule

$$S_1 = w_1(X), c_1, r_2(X), w_2(X), c_2$$

Recoverable Schedule

$$S_2 = w_1(X), r_2(X), w_2(X), c_1, c_2$$

Nonrecoverable Schedule

$$S_3 = w_1(X), r_2(X), w_2(X), c_2, c_1$$

$$T_1 = w_1(X), a_1$$

$$T_2 = r_2(X), w_2(X), c_2$$

Cascadeless Schedule

$$S_1 = w_1(X), a_1, r_2(X), w_2(X), c_2$$

Recoverable Schedule

$$S_2 = w_1(X), r_2(X), w_2(X), a_1, a_2$$

Nonrecoverable Schedule

$$S_3 = w_1(X), r_2(X), w_2(X), c_2, a_1$$

Consider what happens if T1 aborts!

Strict Schedules

- **Strict** schedules guarantee that to Undo the effect of an transaction we simply have to undo each of its writes
 - Transactions do not read nor write items written by uncommitted transactions

$$T_1 = w_1(X), c_1$$

$$T_2 = r_2(X), w_2(X), c_2$$

Cascadeless Schedule

$$S_1 = w_1(X), c_1, r_2(X), w_2(X), c_2$$

Recoverable Schedule

$$S_2 = w_1(X), r_2(X), w_2(X), c_1, c_2$$

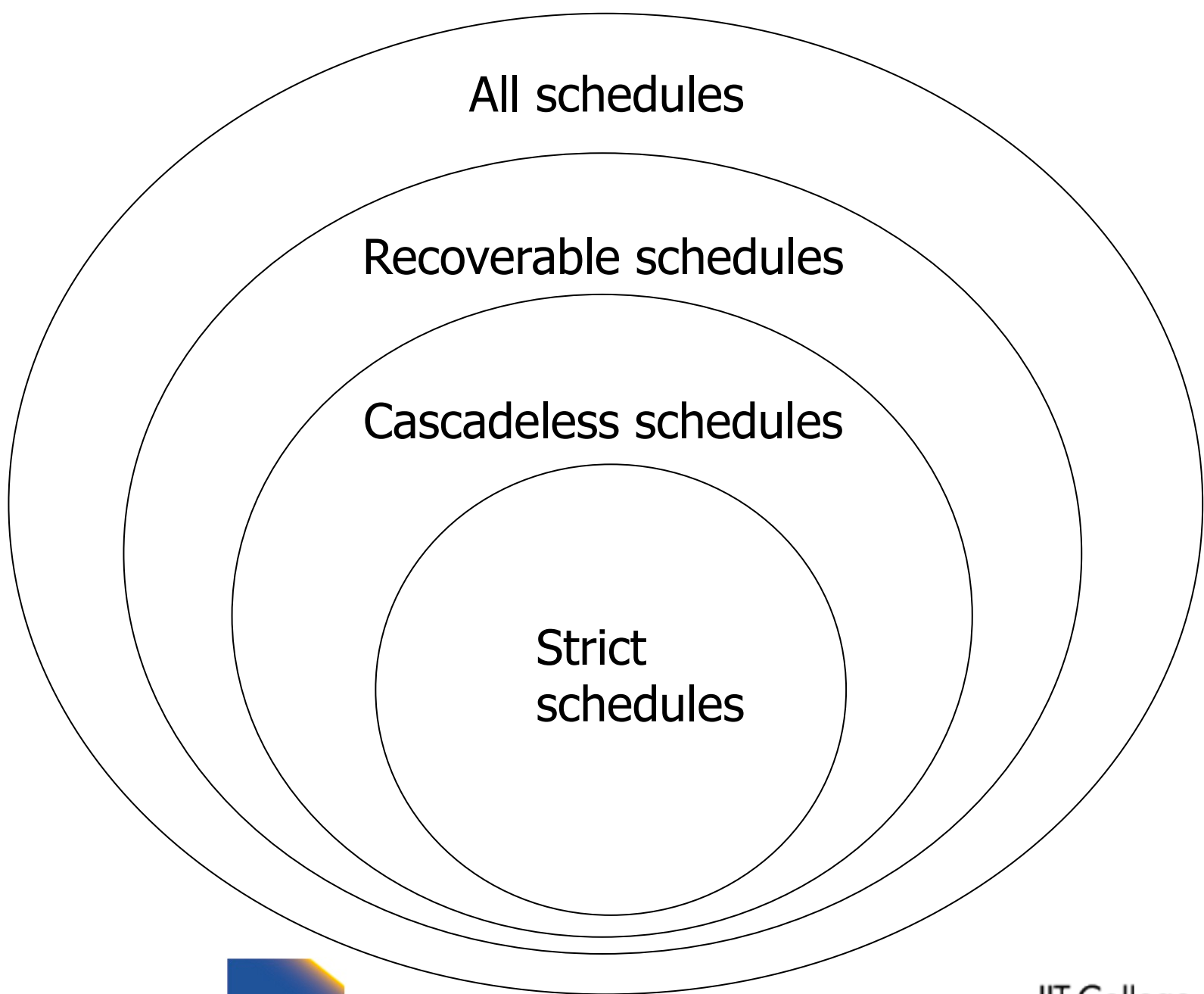
Nonrecoverable Schedule

$$S_3 = w_1(X), r_2(X), w_2(X), c_2, c_1$$

Compare Classes

ST \subset CL \subset RC \subset ALL





Logging and Recovery

- We now discuss approaches for logging and how to use them in recovery

One solution: undo logging (immediate
modification)

due to: Hansel and Gretel, 782 AD

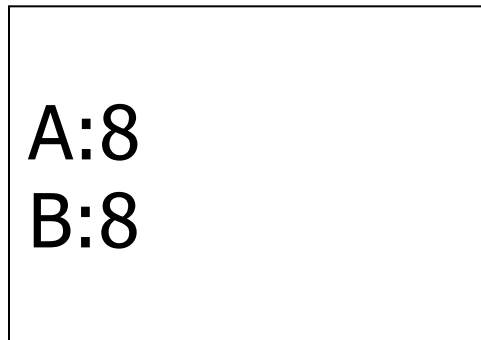
One solution: undo logging (immediate
modification)

due to: Hansel and Gretel, 782 AD

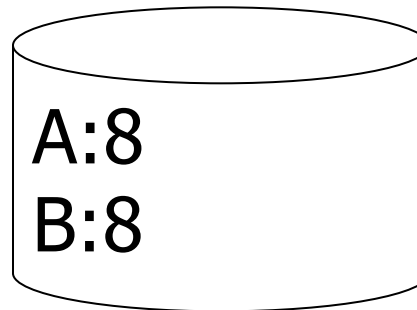
- Improved in 784 AD to durable
undo logging

Undo logging (Immediate modification)

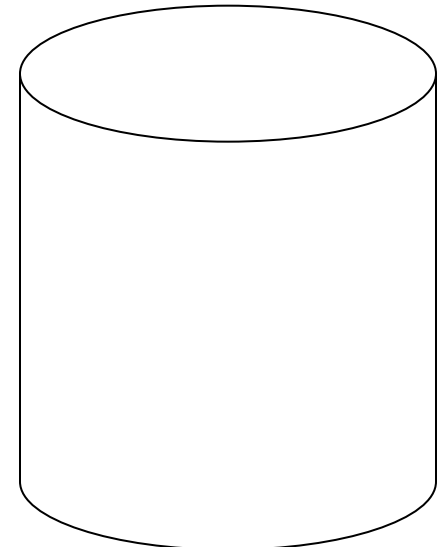
T₁: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);



memory



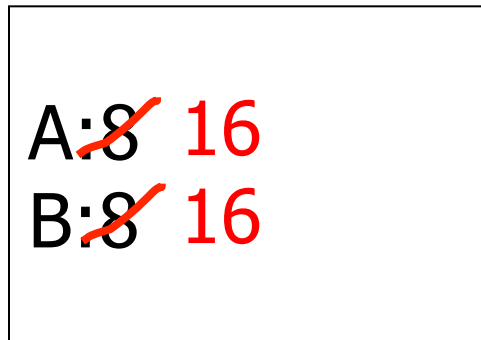
disk



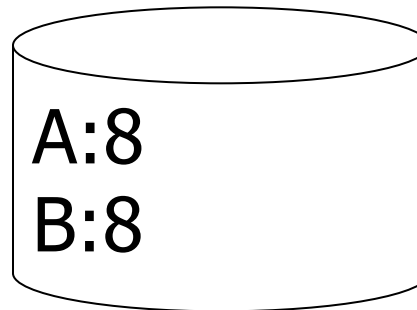
log

Undo logging (Immediate modification)

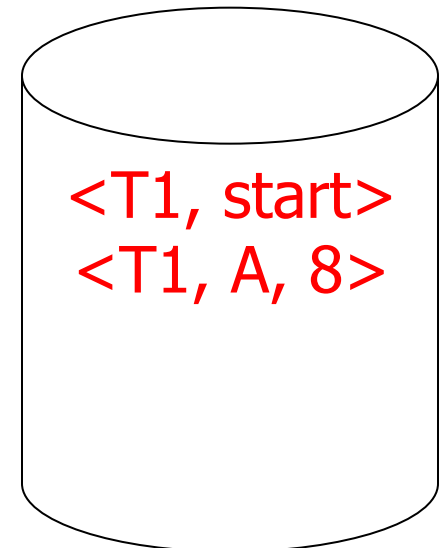
T₁: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);



memory



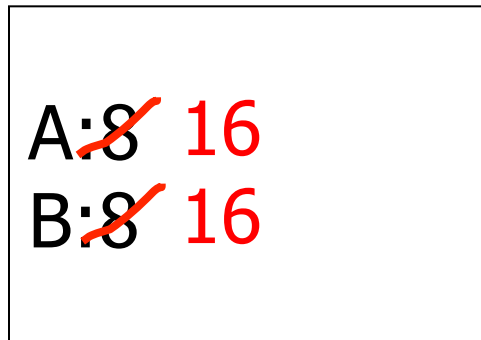
disk



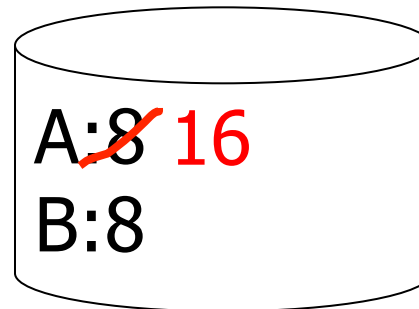
log

Undo logging (Immediate modification)

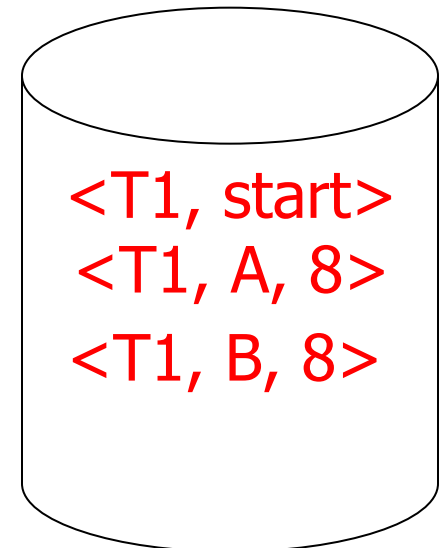
T₁: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);



memory



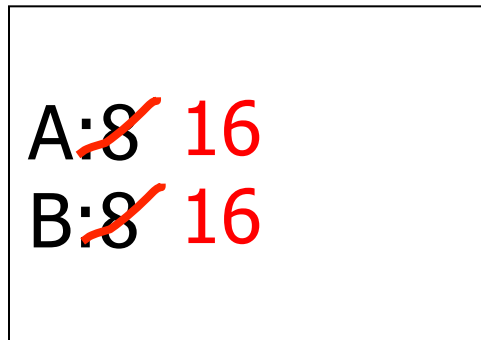
disk



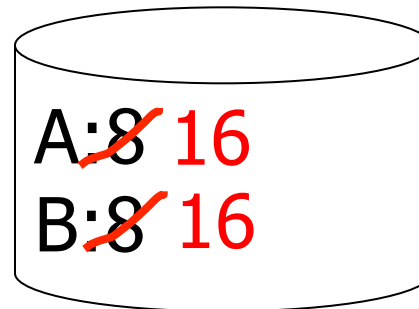
log

Undo logging (Immediate modification)

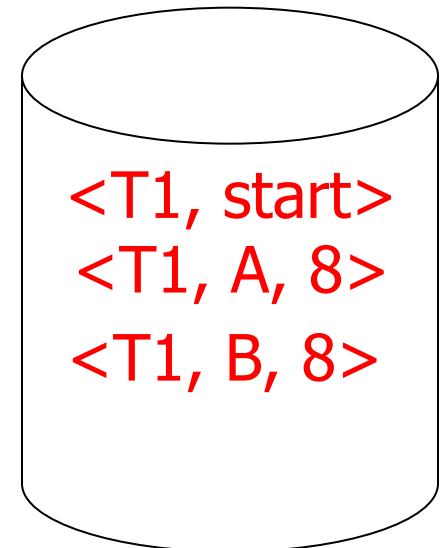
T₁: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);



memory



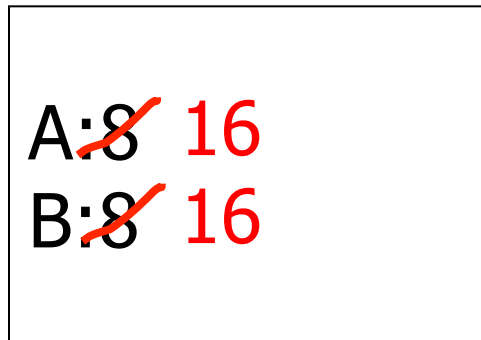
disk



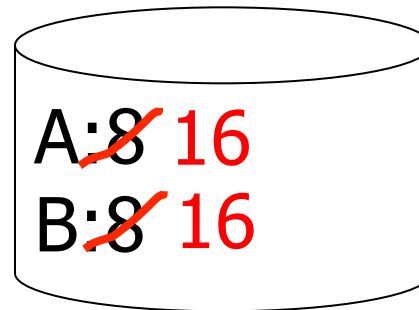
log

Undo logging (Immediate modification)

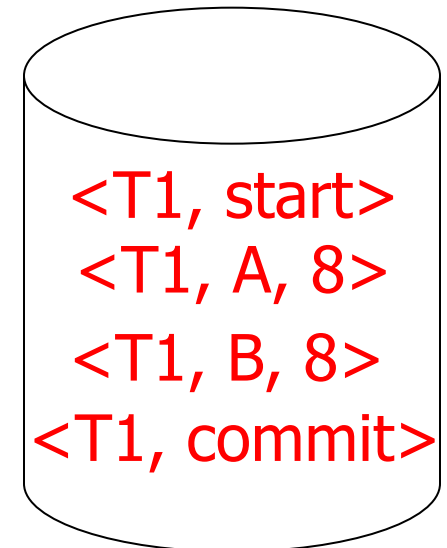
T₁: Read (A,t); t ← t×2 A=B
Write (A,t);
Read (B,t); t ← t×2
Write (B,t);
Output (A);
Output (B);



memory



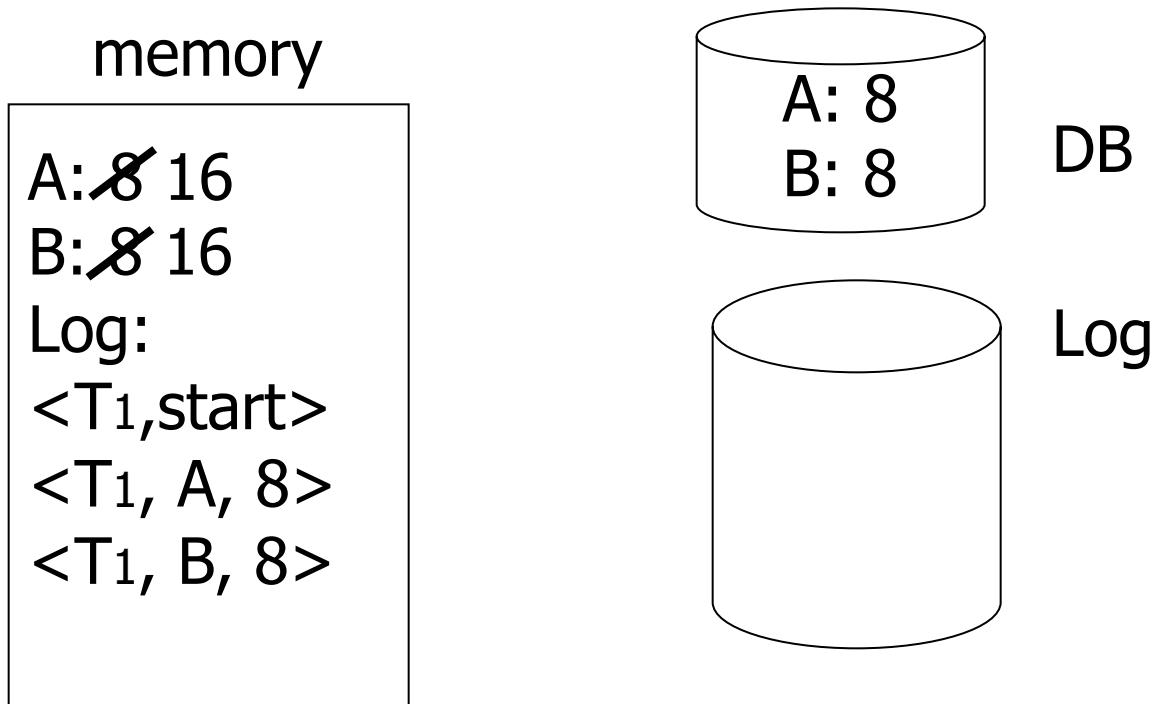
disk



log

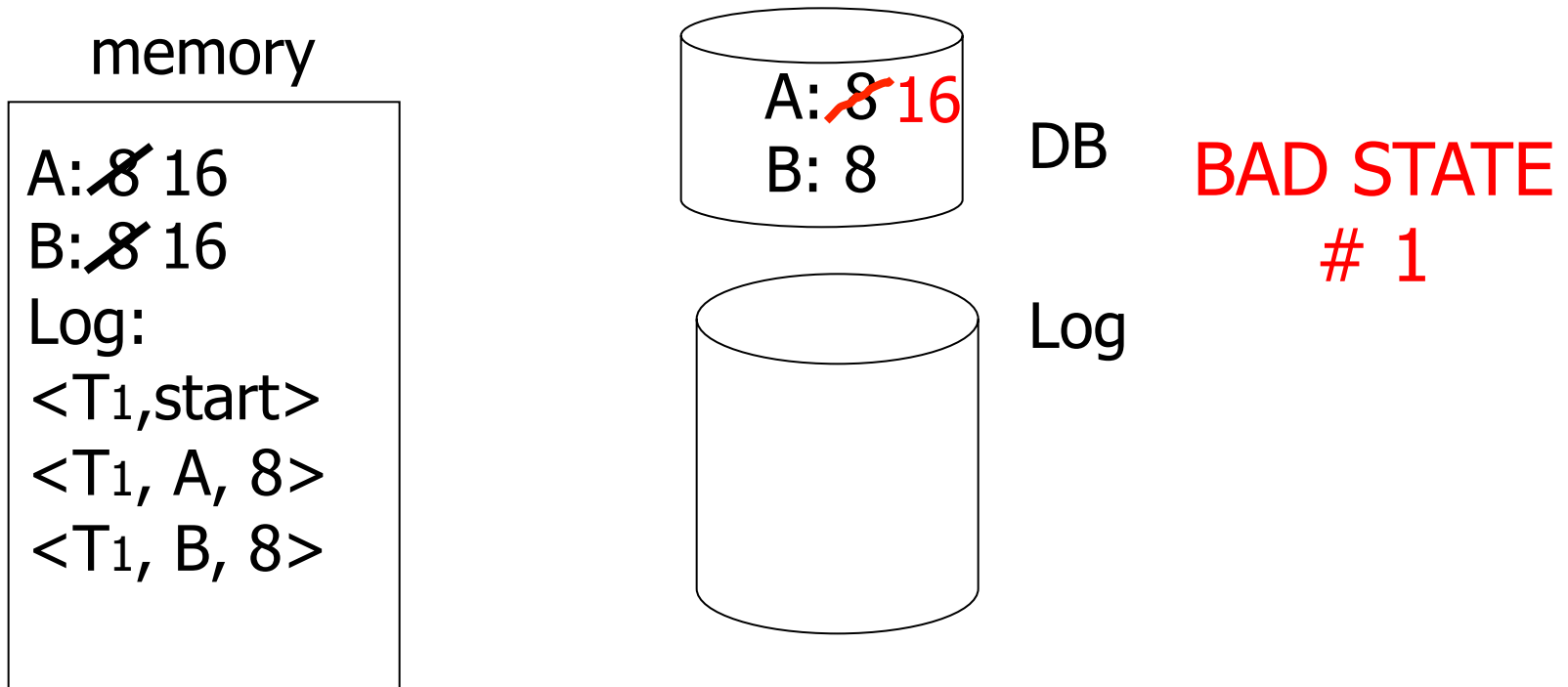
One “complication”

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



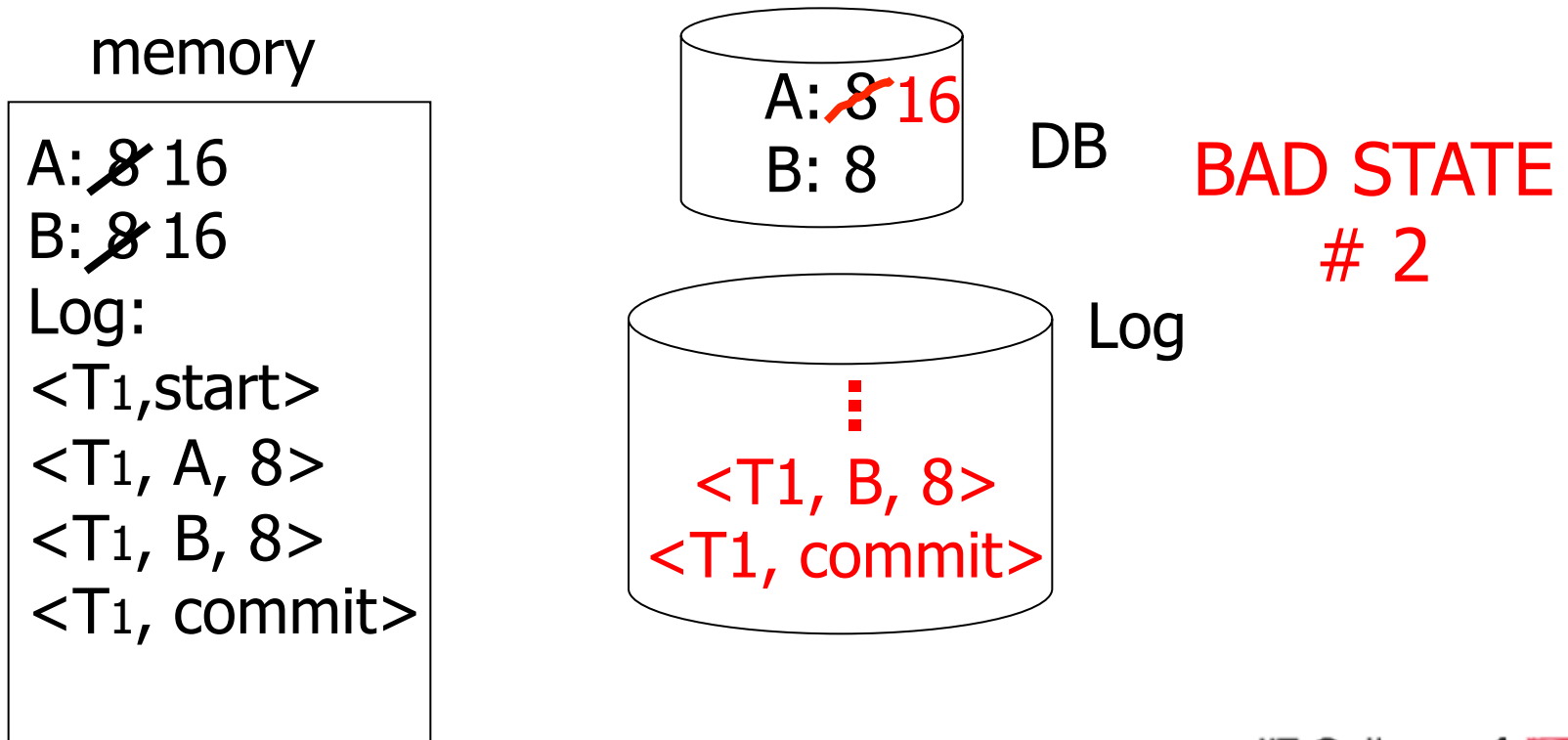
One “complication”

- 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 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 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 $\left\{ \begin{array}{l} \text{For all } \langle T_i, X, v \rangle \text{ in log:} \\ \quad \left\{ \begin{array}{l} \text{write } (X, v) \\ \text{output } (X) \end{array} \right. \\ \text{Write } \langle T_i, \text{abort} \rangle \text{ to log} \end{array} \right.$

 **IS THIS CORRECT??**

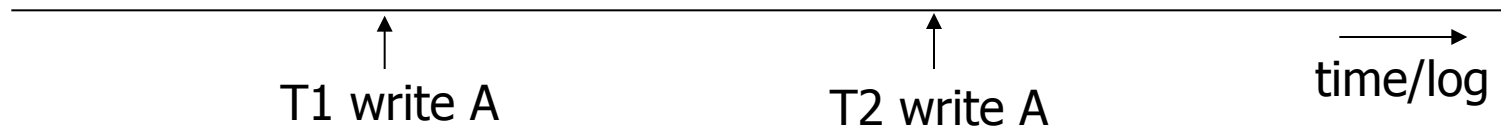
Recovery rules:

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

- An operation is called **idempotent** if the number of times it is applied do not effect the result
- For Undo:
 - $\text{Undo}(\log) = \text{Undo}(\text{Undo}(\dots (\text{Undo}(\log)) \dots))$

Undo is idempotent

- We store the values of data items before the operation
- Undo can be executed repeatedly without changing effects
 - idempotent

Physical vs. Logical Logging

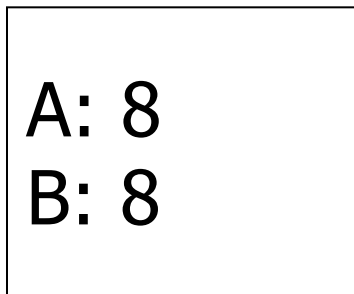
- How to represent values in log entries?
- Physical logging
 - Content of pages before and after
- Logical operations
 - Operation to execute for undo/redo
 - E.g., delete record x
- Hybrid (Physiological)
 - Delete record x from page y

To discuss:

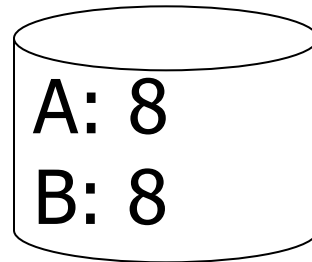
- Redo logging
- Undo/redo logging, why both?
- Real world actions
- Checkpoints
- Media failures

Redo logging (deferred modification)

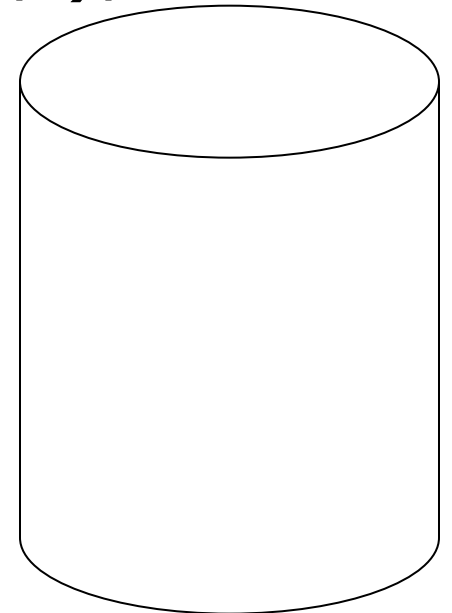
T₁: Read(A,t); t ← t×2; write (A,t);
Read(B,t); t ← t×2; write (B,t);
Output(A); Output(B)



memory



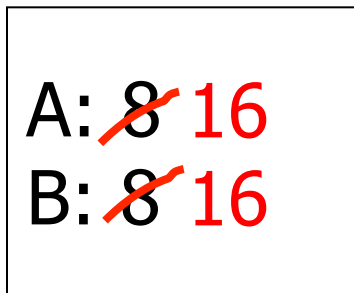
DB



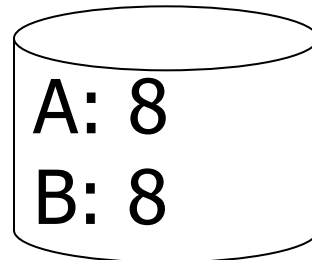
LOG

Redo logging (deferred modification)

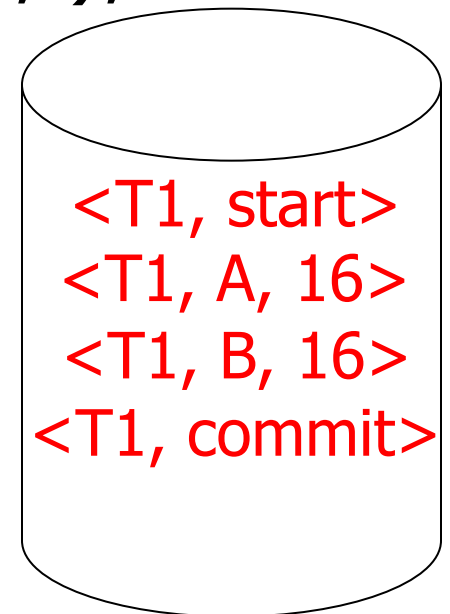
T₁: Read(A,t); t ← t×2; write (A,t);
Read(B,t); t ← t×2; write (B,t);
Output(A); Output(B)



memory



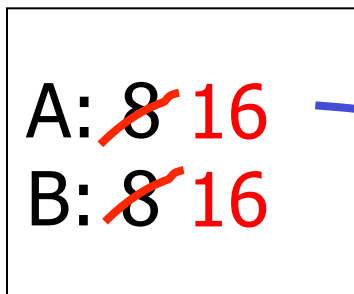
DB



LOG

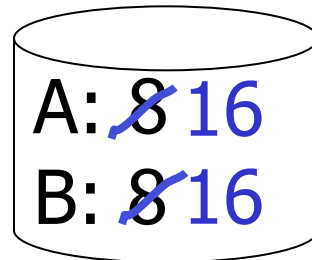
Redo logging (deferred modification)

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

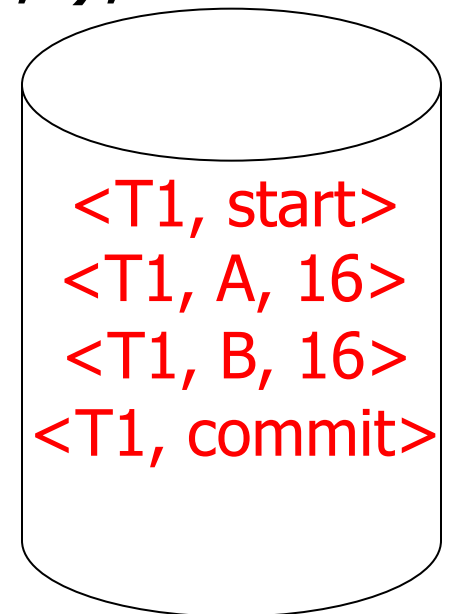


memory

output



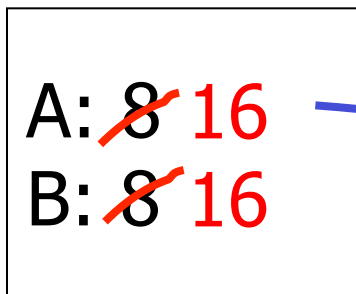
DB



LOG

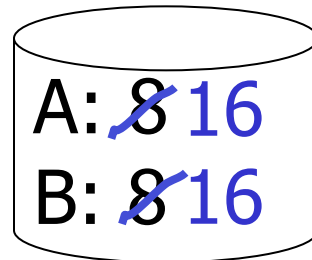
Redo logging (deferred modification)

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

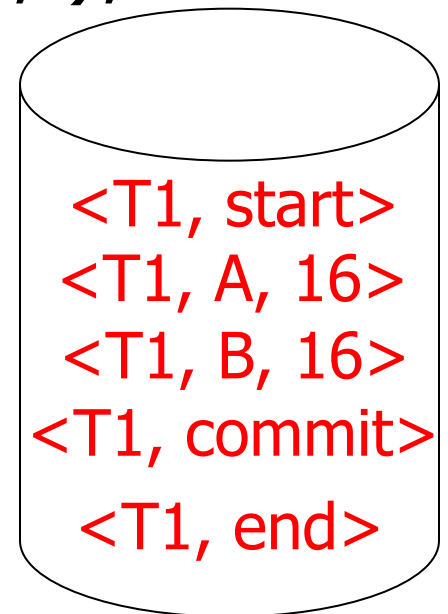


memory

output



DB



LOG

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

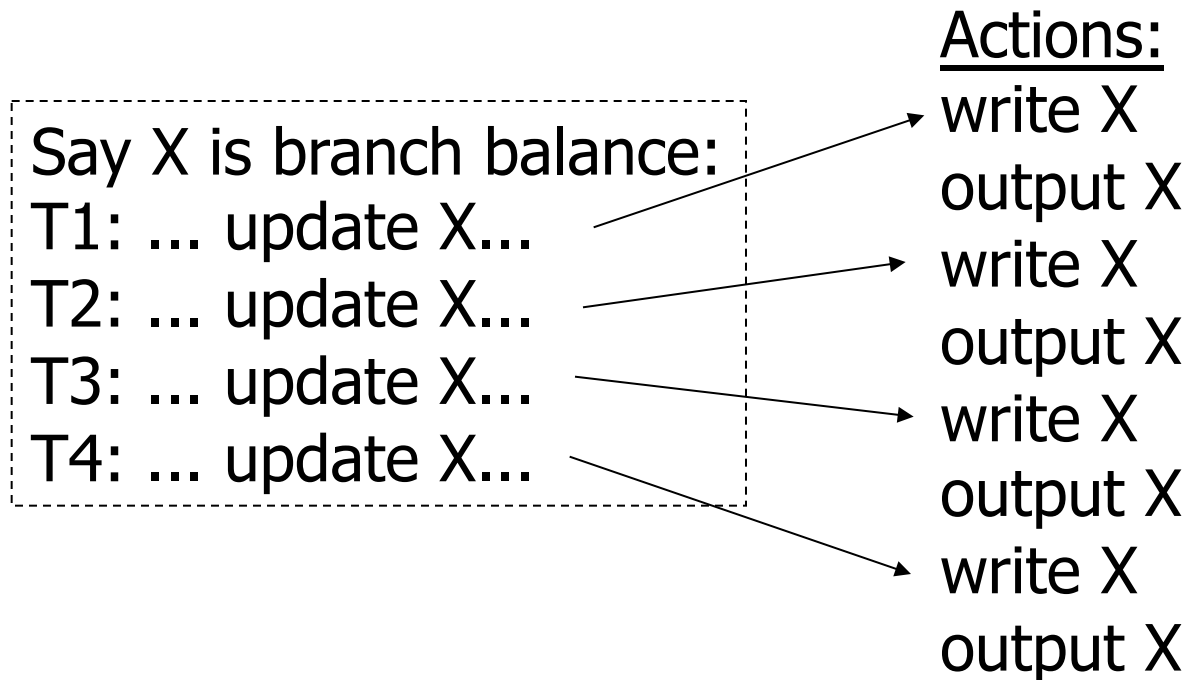
- (1) Let S = set of transactions with $\langle T_i, \text{commit} \rangle$ (and no $\langle T_i, \text{end} \rangle$) in log
- (2) For each $\langle T_i, X, v \rangle$ in log, in forward order (earliest \rightarrow latest) do:
 - if $T_i \in S$ then $\left\{ \begin{array}{l} \text{Write}(X, v) \\ \text{Output}(X)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

Crash During Redo

- Since Redo log contains values after writes, repeated application of a log entry does not change result
 - -> idempotent

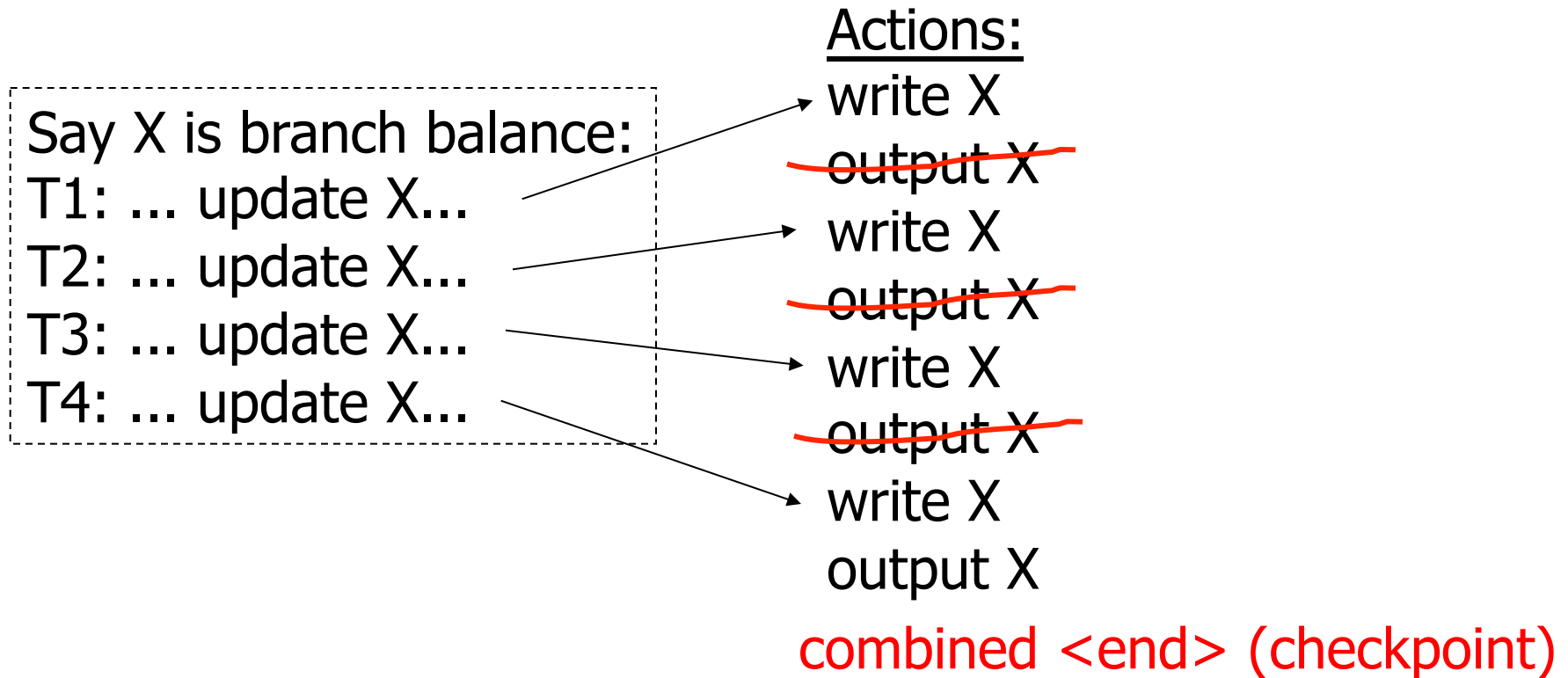
Combining $\langle T_i, \text{end} \rangle$ Records

- Want to delay DB flushes for hot objects



Combining $\langle T_i, \text{end} \rangle$ Records

- Want to delay DB flushes for hot objects



Solution: Checkpoint

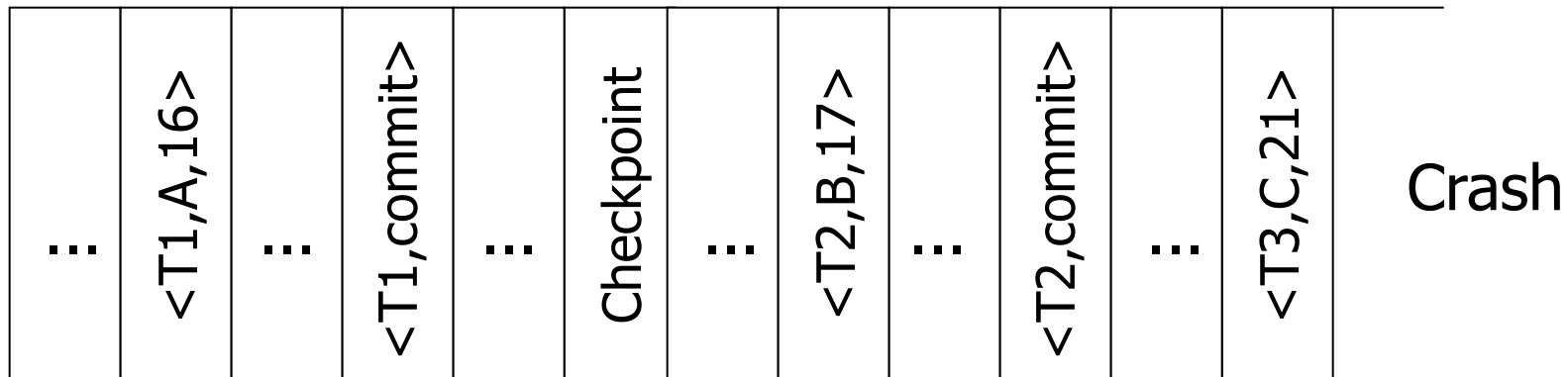
- no $\langle ti, end \rangle$ actions
- simple checkpoint

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

Example: what to do at recovery?

Redo log (disk):



Advantage of Checkpoints

- Limits recovery to parts of the log after the checkpoint
 - Think about system that has been online for months
 - ->Analyzing the whole log is too expensive!
- Source of backups
 - If we backup checkpoints we can use them for media recovery!

Checkpoints Justification

- Checkpoint should be consistent DB state
 - No active transactions
 - Do not accept new transactions
 - Wait until all transactions finish
 - DB state reflected on disk
 - Flush log
 - Flush buffers

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 T_i commit
- Log record flushed before corresponding updated page (WAL)
- Flush at commit (log only)

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

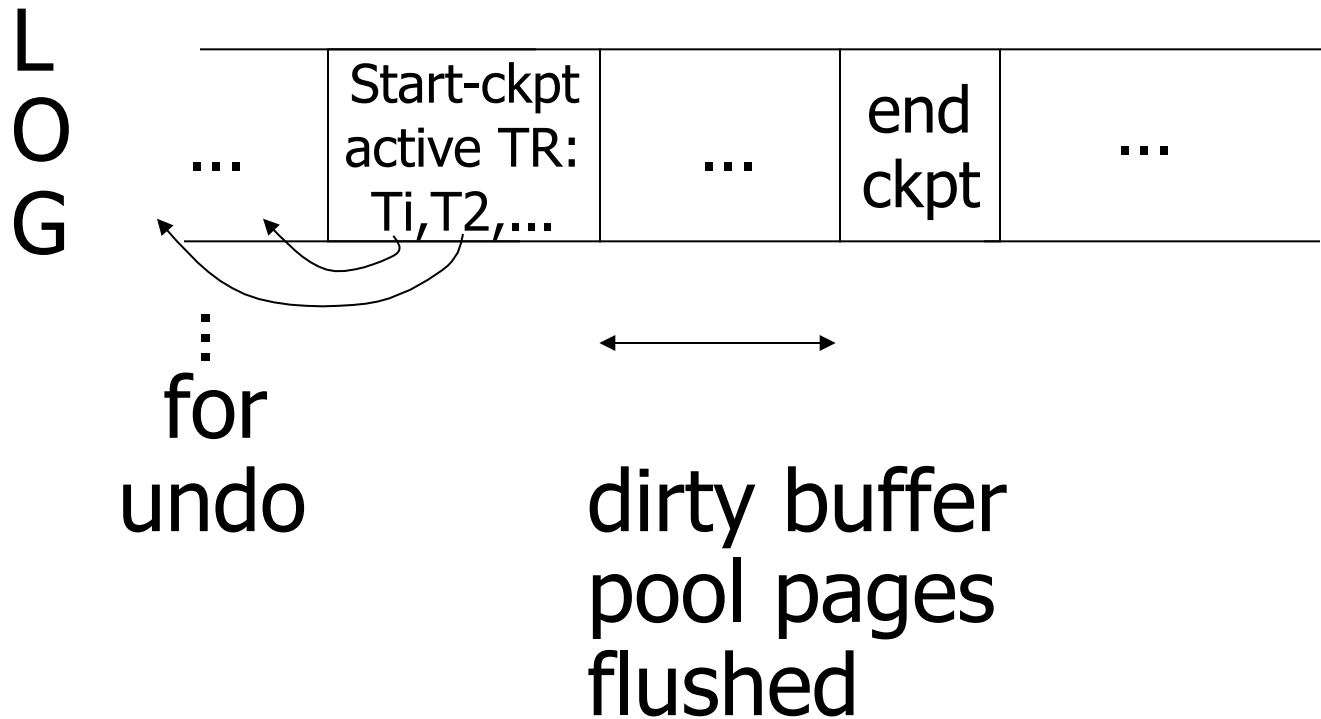
log (disk):

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

Checkpoint Cost

- Checkpoints are expensive
 - No new transactions can start
 - A lot of I/O
 - Flushing the log
 - Flushing dirty buffer pages

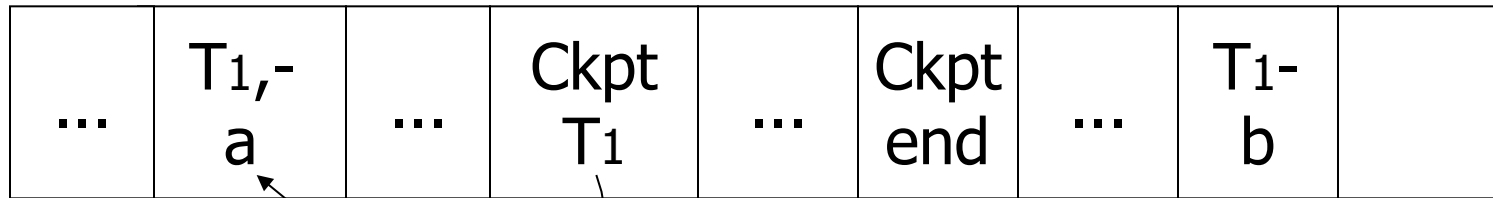
Non-quietse checkpoint



Examples what to do at recovery time?

no T1 commit

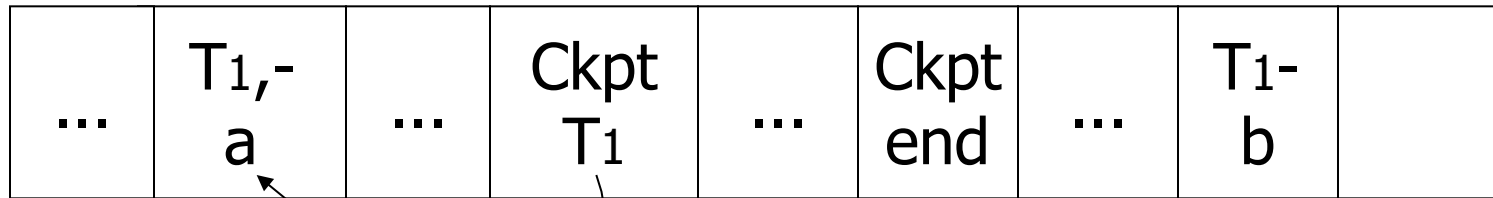
L
O
G



Examples what to do at recovery time?

no T1 commit

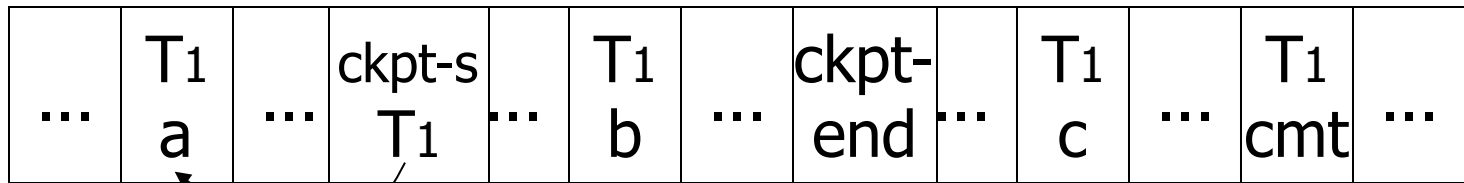
L
O
G



➡ Undo T₁ (undo a,b)

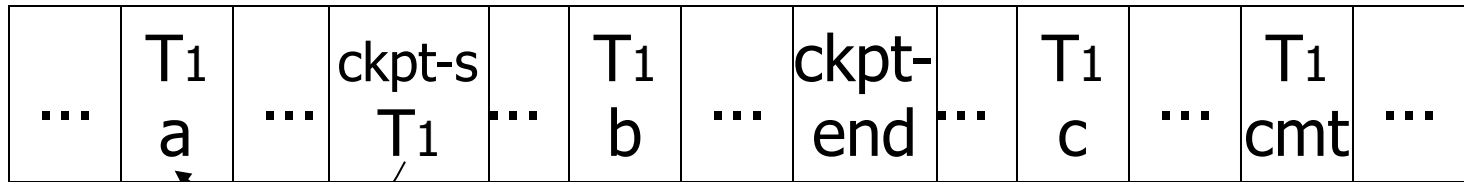
Example

L
O
G



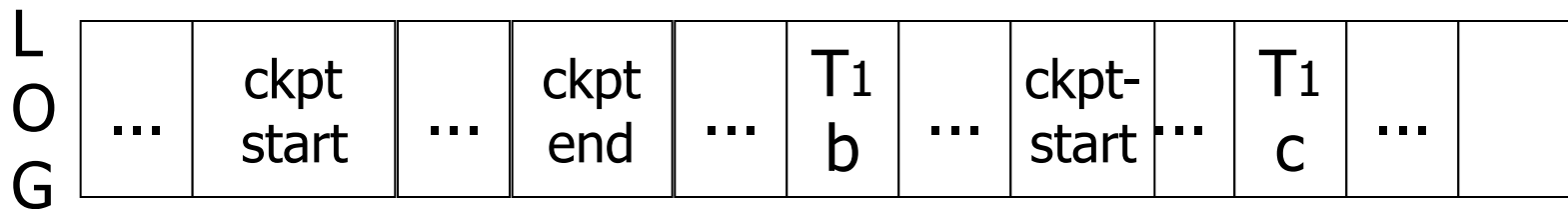
Example

L
O
G



➡ Redo T1: (redo b,c)

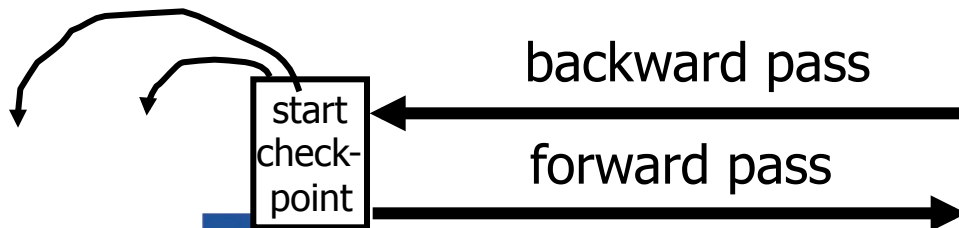
Recover From Valid Checkpoint:



↑
start
of latest
valid
checkpoint

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 checkpoint start → end of log)
 - redo actions of S transactions



Real world actions

E.g., dispense cash at ATM

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

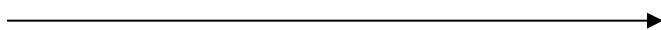


\$

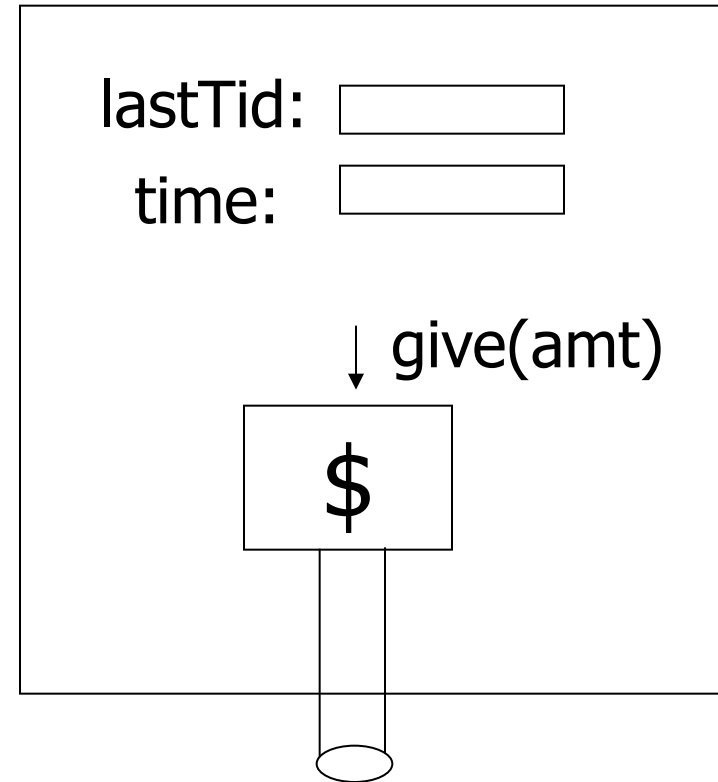
Solution

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

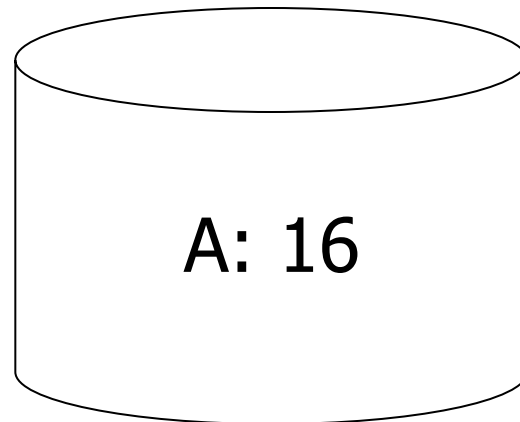
Give\$\$
(amt, Tid, time)



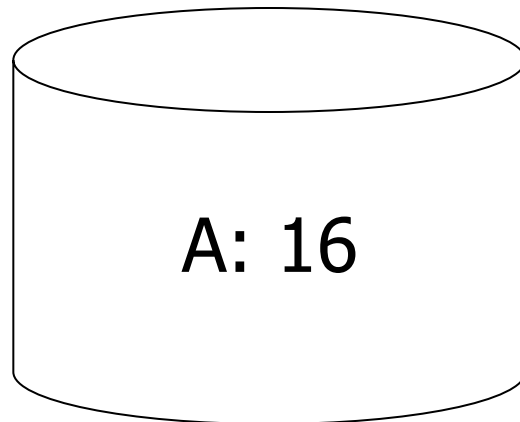
ATM



Media failure (loss of non-volatile storage)



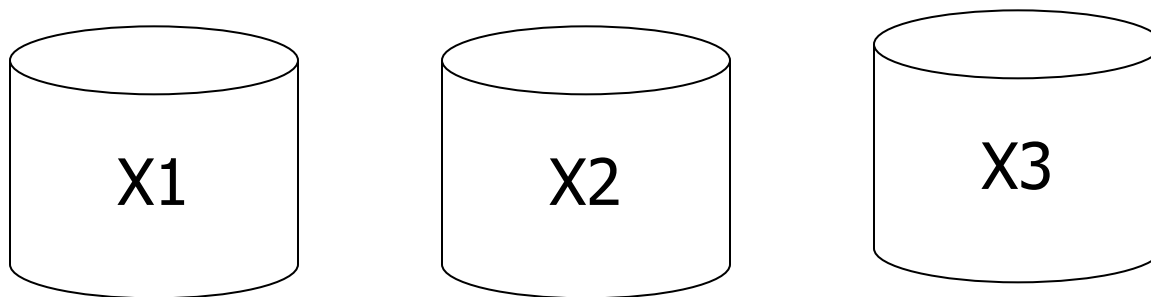
Media failure (loss of non-volatile storage)



Solution: Make copies of data!

Example 1 Triple modular redundancy

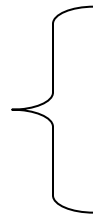
- Keep 3 copies on separate disks
- Output(X) --> three outputs
- Input(X) --> three inputs + vote



Example #2 Redundant writes, Single reads

- Keep N copies on separate disks
- Output(X) --> N outputs
- Input(X) --> Input one copy

done

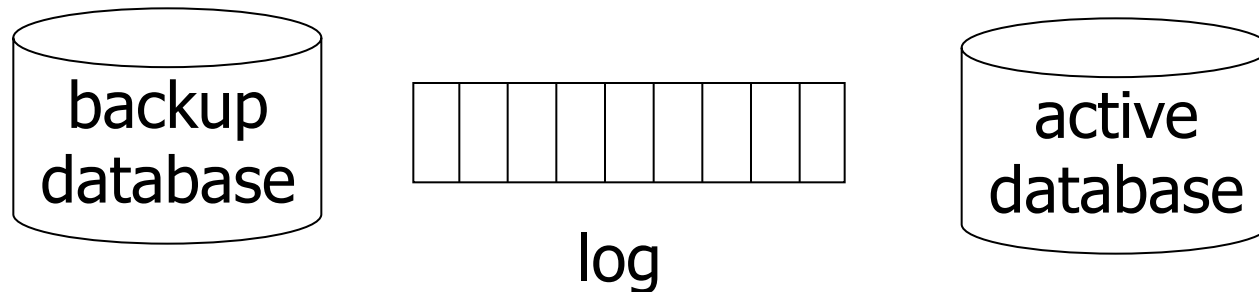


- if ok,

- else try another one

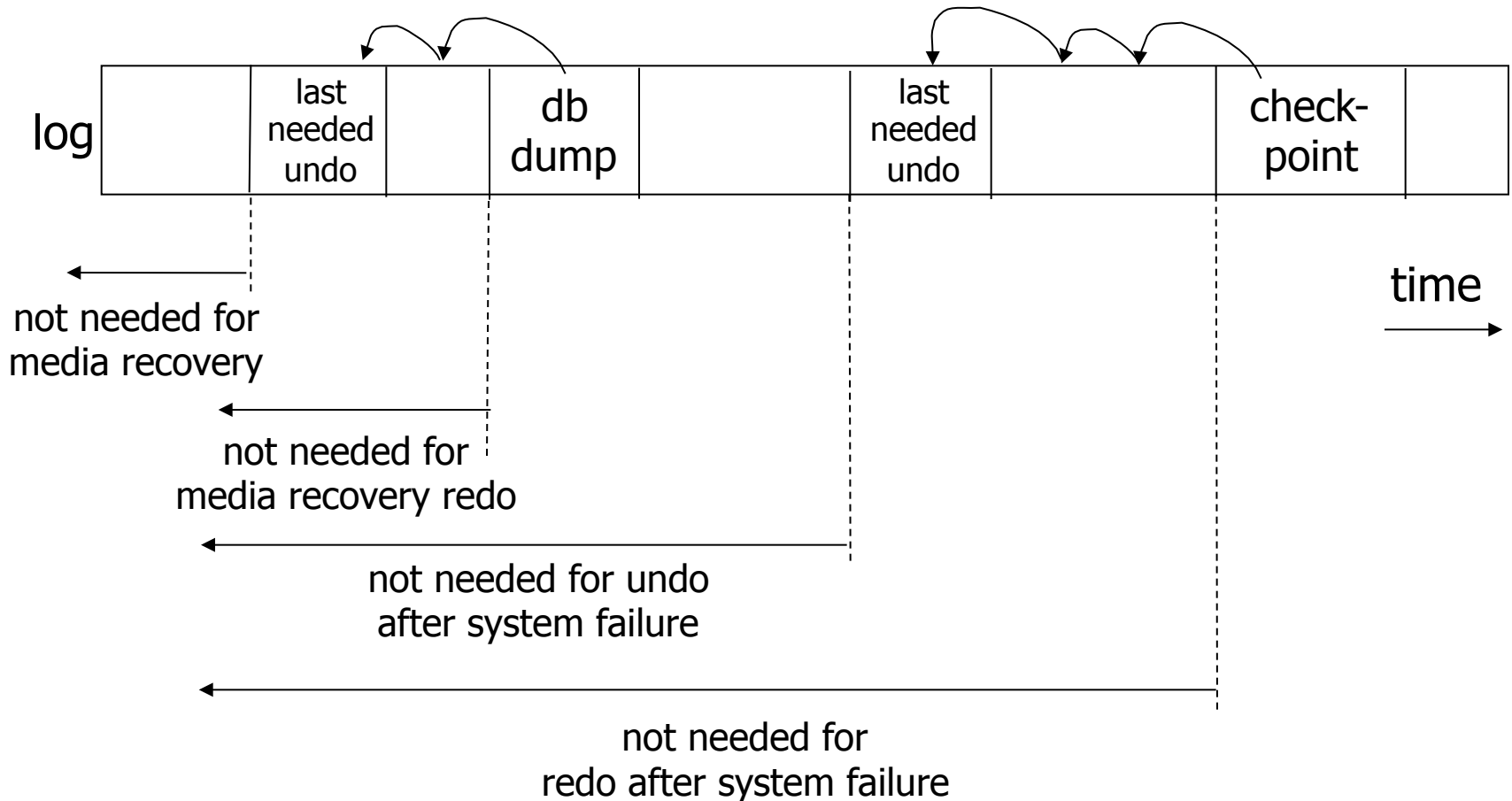
⇒ Assumes bad data can be detected

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?



Practical Recovery with ARIES

- **ARIES**
 - **A**lgorithms for **R**ecovery and **I**solation
Exploiting **S**emantics
- Implemented in, e.g.,
 - DB2
 - MSSQL

Underlying Ideas

- Keep track of state of pages by relating them to entries in the log
- **WAL**
- Recovery in **three phases**
 - Analysis, Redo, Undo
- Log entries to track state of Undo for repeated failures
- **Redo**: page-oriented -> efficient
- **Undo**: logical -> permits higher level of concurrency

Log Entry Structure

- **LSN**
 - **Log sequence number**
 - Order of entries in the log
 - Usually **log file id** and **offset** for direct access

- **LSN**
- **Entry type**
 - Update, compensation, commit, ...
- **TID**
 - Transaction identifier
- **PrevLSN**
 - LSN of previous log record for same transaction
- **UndoNxtLSN**
 - Next undo operation for CLR (later!)
- **Undo/Redo data**
 - Data needed to undo/redo the update

Page Header Additions

- **PageLSN**

- **LSN** of the last update that modified the page
- Used to know which changes have been applied to a page

Forward Processing

- Normal operations when no ROLLBACK is required
 - WAL: write redo/undo log record for each action of a transaction
- Buffer manager has to ensure that
 - changes to pages are not persisted before the corresponding log record has been persisted
 - Transactions are not considered committed before all their log records have been flushed

Dirty Page Table

- **PageLSN**

- Entries **<PageID,RecLSN>**
- Whenever a page is first fixed in the buffer pool with indention to modify
 - Insert **<PageId,RecLSN>** with **RecLSN** being the current end of the log
- Flushing a page removes it from the Dirty page table

Dirty Page Table

- Used for checkpointing
- Used for recovery to figure out what to redo

Transaction Table

- TransID
 - Identifier of the transaction
- State
 - Commit state
- LastLSN
 - LSN of the last update of the transaction
- UndoNxtLSN
 - If last log entry is a CLR then UndoNxtLSN from that record
 - Otherwise = LastLSN

Transaction Table:

$\langle 1, U, -, - \rangle$

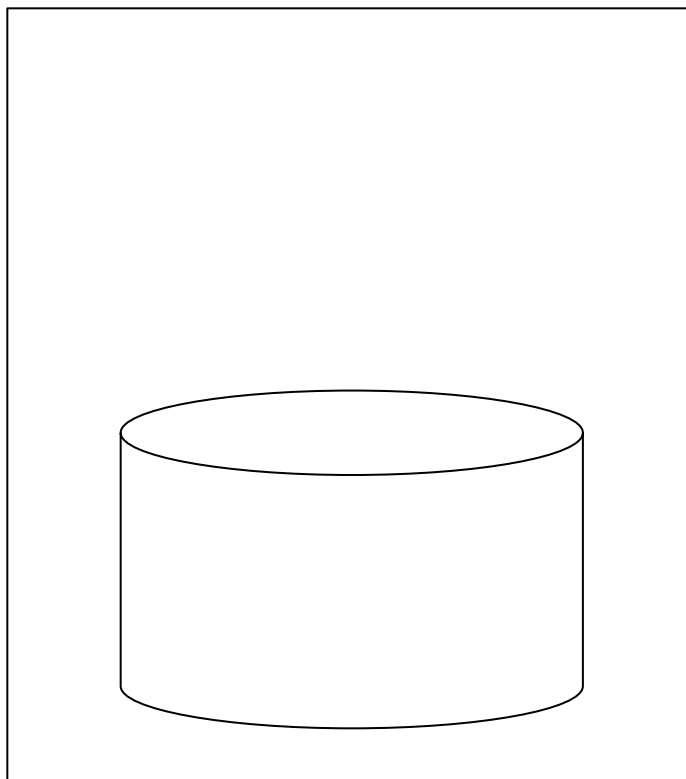
Dirty Page Table:

$T_1 = r_1(A), A = A * 2, w_1(A)$

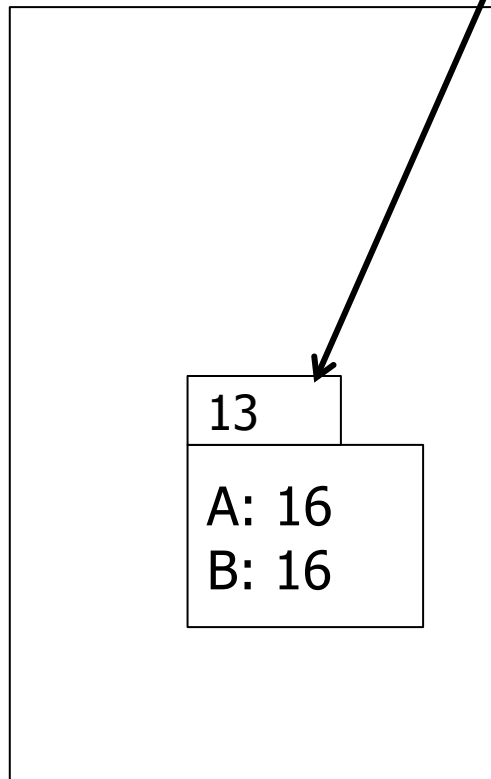
Page_LSN:

LSN of last modification to page

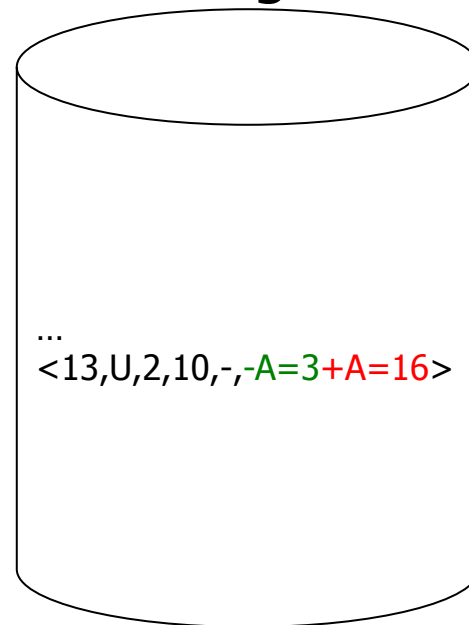
buffer



disk



Persistent log



Transaction Table:

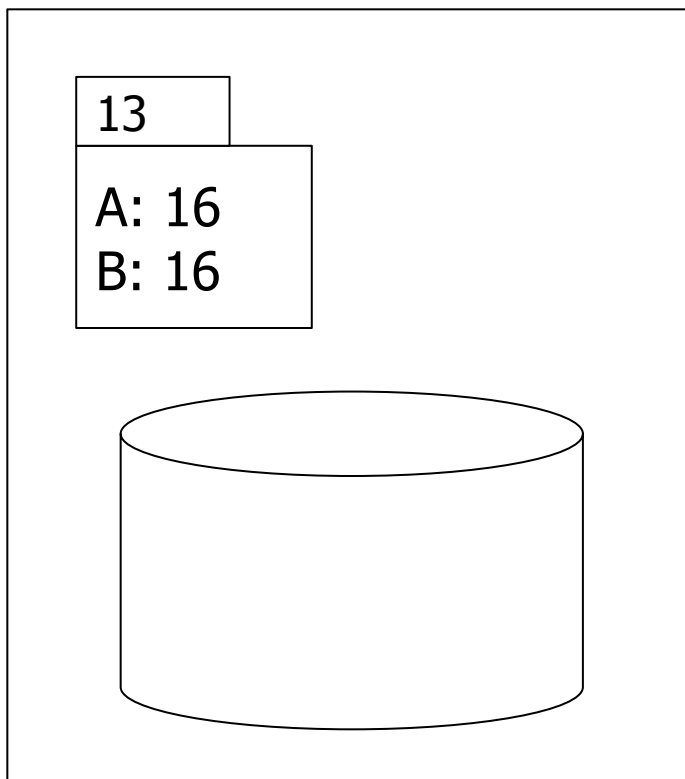
$\langle 1, U, -, - \rangle$

Dirty Page Table:

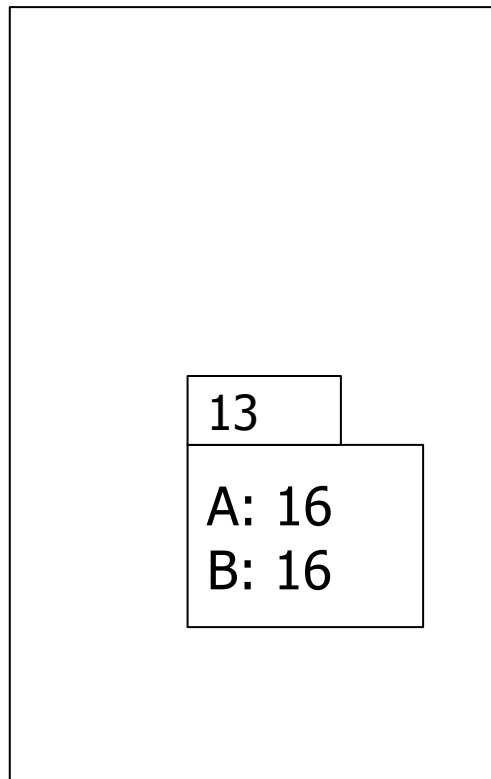
$\langle 100, 14 \rangle$

$T_1 = r_1(A), A = A * 2, w_1(A)$

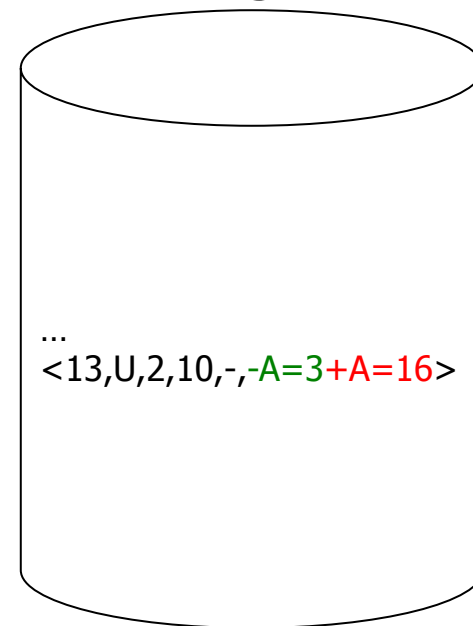
buffer



disk



Persistent
log



Transaction Table:

<1, U, -, ->

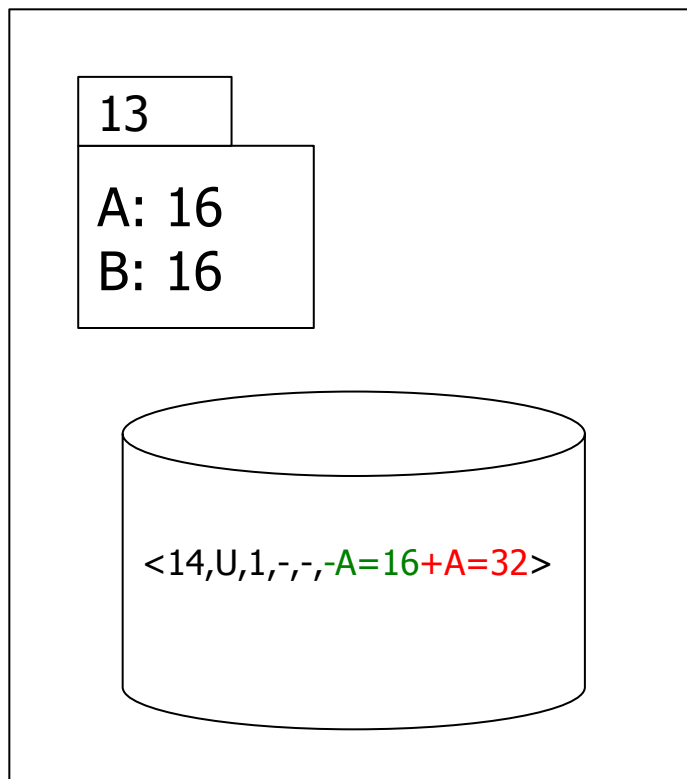
Dirty Page Table:

<100, 14>

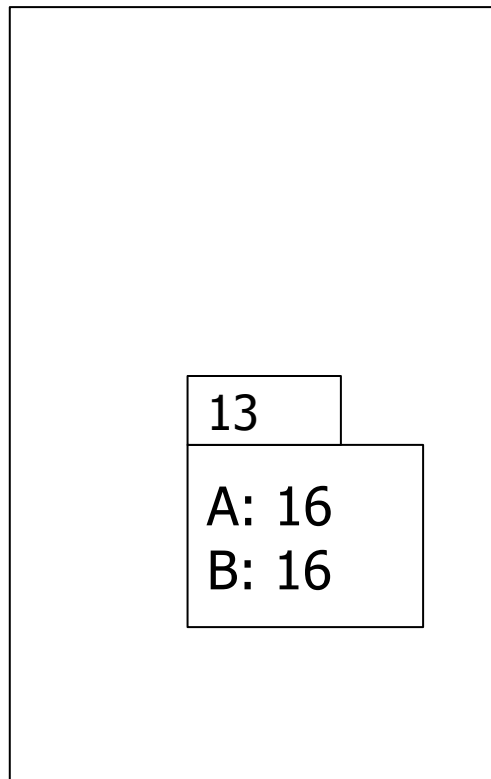
Write log entry

$T_1 = r_1(A), A = A * 2, w_1(A)$

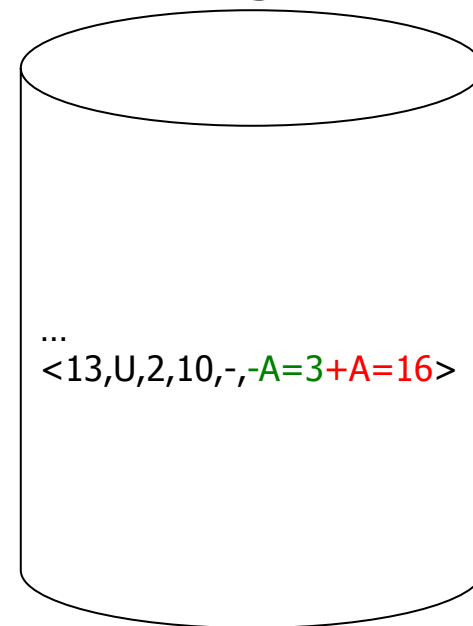
buffer



disk



Persistent
log



Transaction Table:

$\langle 1, U, 14, 14 \rangle$

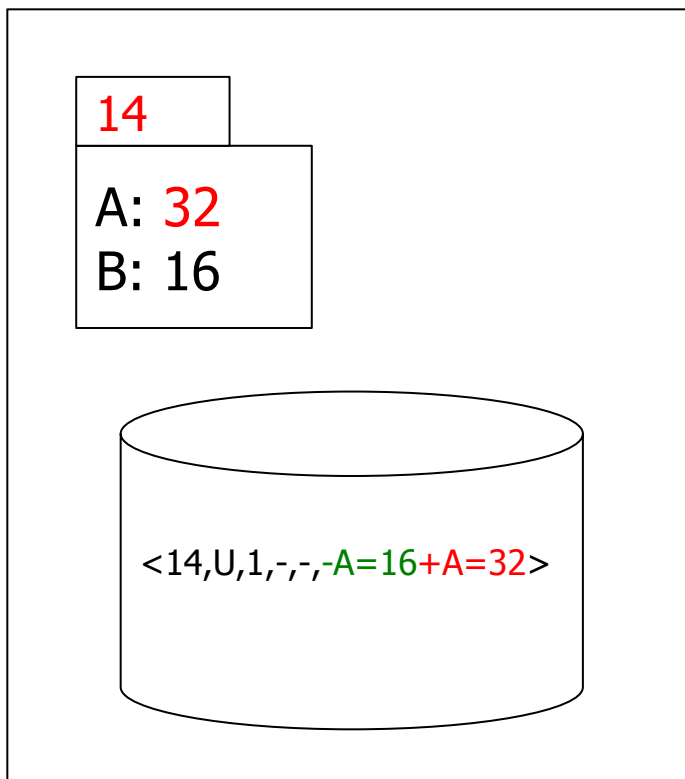
Dirty Page Table:

$\langle 100, 14 \rangle$

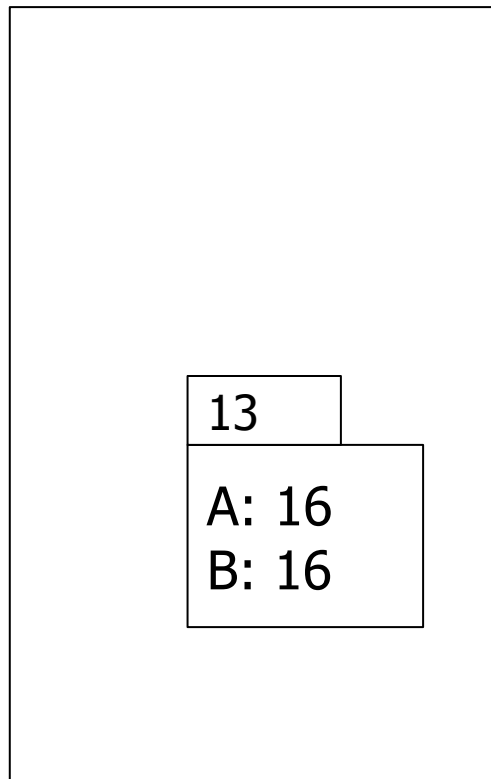
Update page

$T_1 = r_1(A), A = A * 2, w_1(A)$

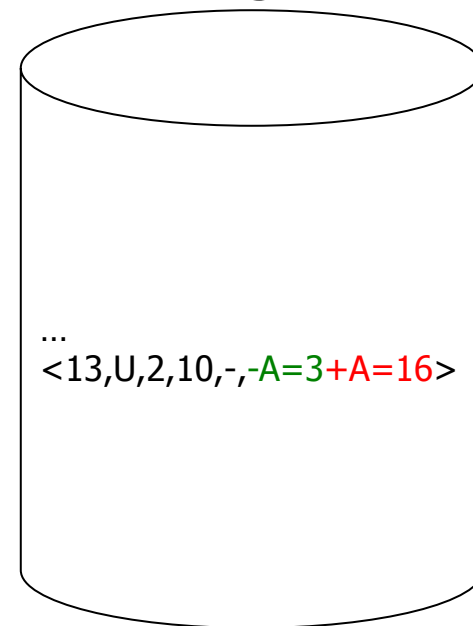
buffer



disk



Persistent
log



Transaction Table:

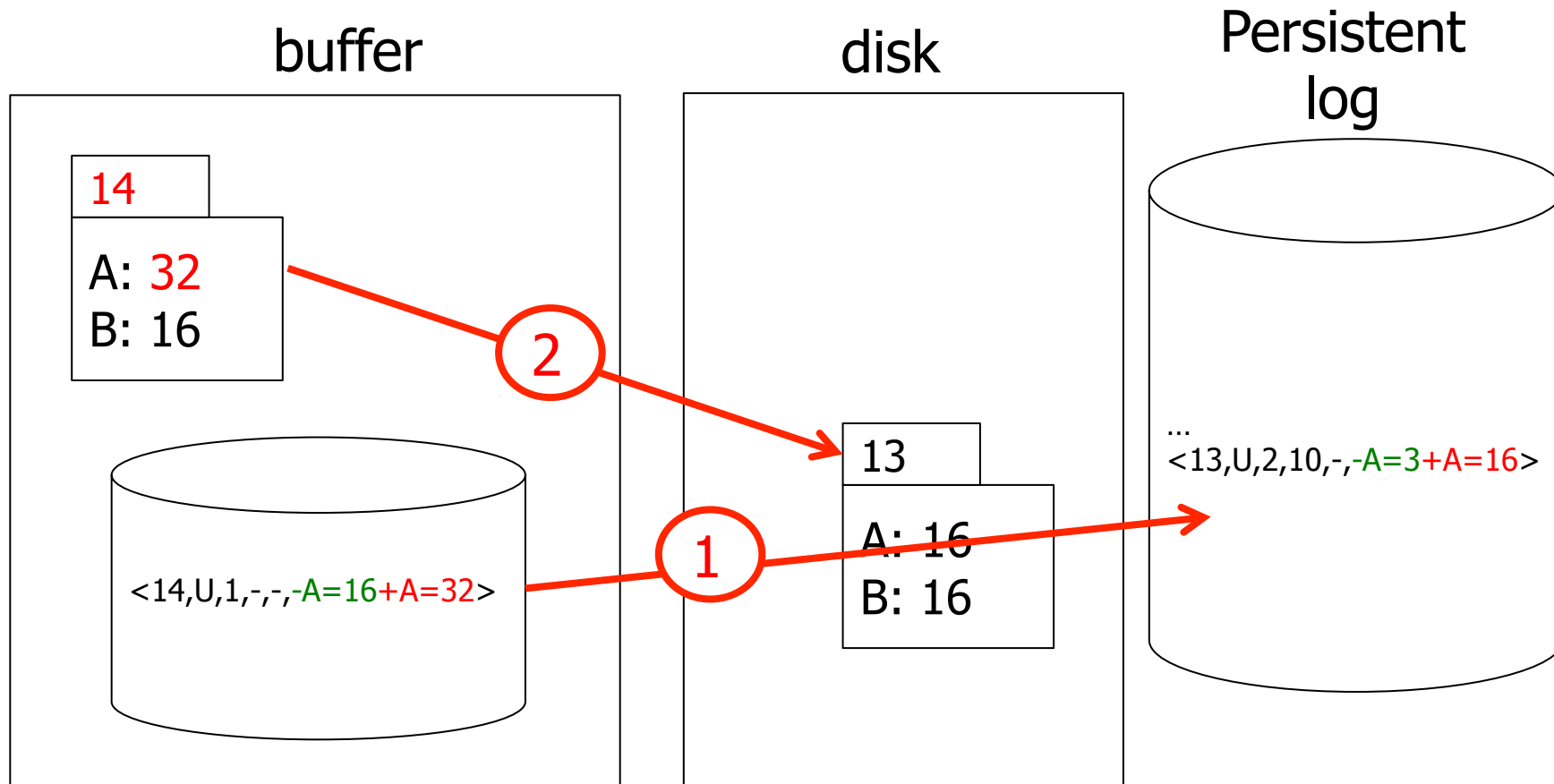
$\langle 1, U, 14, 14 \rangle$

Dirty Page Table:

$\langle 100, 14 \rangle$

can wait with
flushing page, but log
has to be flushed
first!

$T_1 = r_1(A), A = A * 2, w_1(A)$



Undo during forward processing

- Transaction was rolled back
 - User aborted, aborted because of error, ...
- Need to undo operations of transaction
- During Undo
 - Write log entries for every undo
 - **Compensation Log Records (CLR)**
 - Used to avoid repeated undo when failures occur

Undo during forward processing

- Starting with the LastLSN of transaction from transaction table
 - Traverse log entries of transaction last to first using PrevLSN pointers
 - For each log entry use undo information to undo action
 - **<LSN, Type, TID, PrevLSN, -, Undo/Redo data>**
 - Before modifying data write an CLR that stores redo-information for the undo operation
 - **UndoNxtLSN = PrevLSN** of log entry we are undoing
 - **Redo data** = How to redo the undo

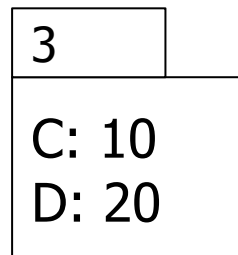
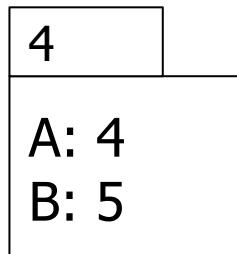
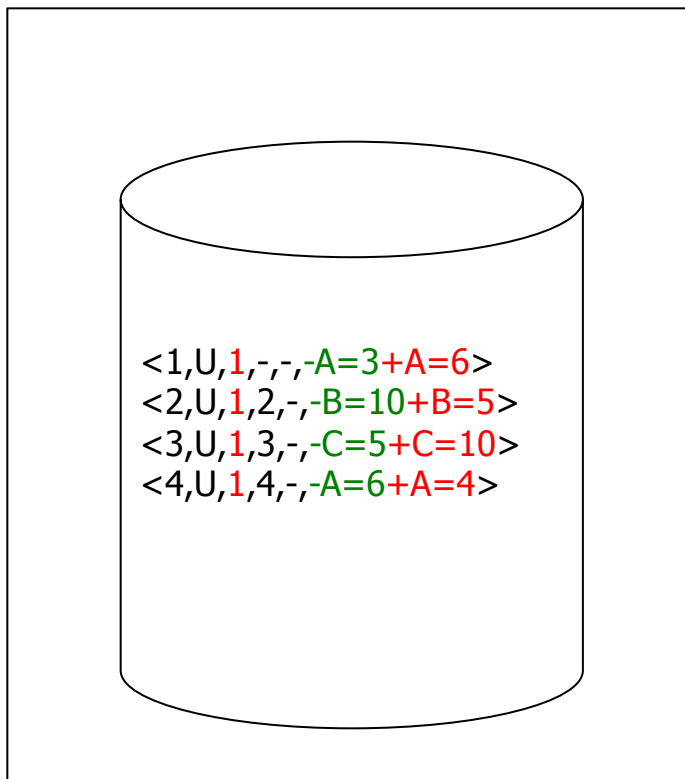
Transaction Table:

$\langle 1, U, 4, 4 \rangle$

Undo T_1

$T_1 = w_1(A), w_1(B), w_1(C), w_1(A), a_1$

buffer



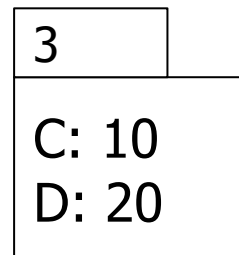
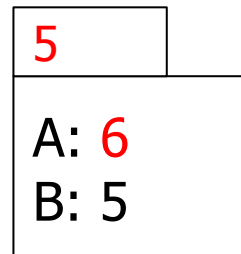
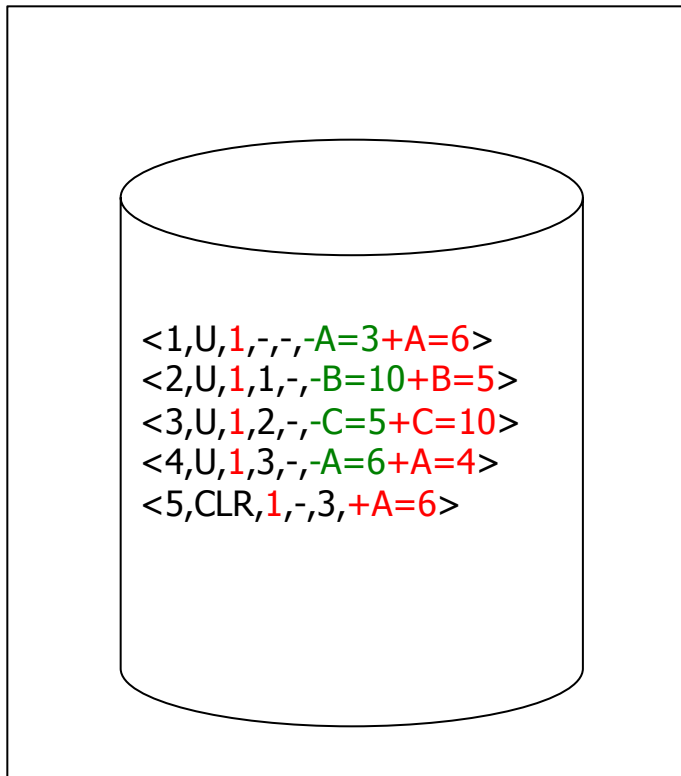
Transaction Table:

$\langle 1, U, 5, 3 \rangle$

Undo T_1

$T_1 = w_1(A), w_1(B), w_1(C), w_1(A), a_1$

buffer



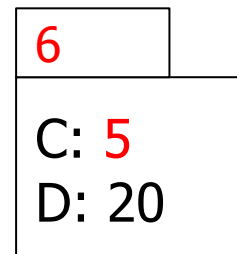
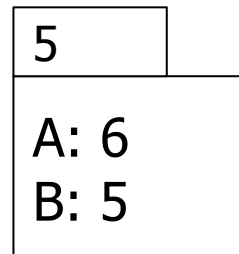
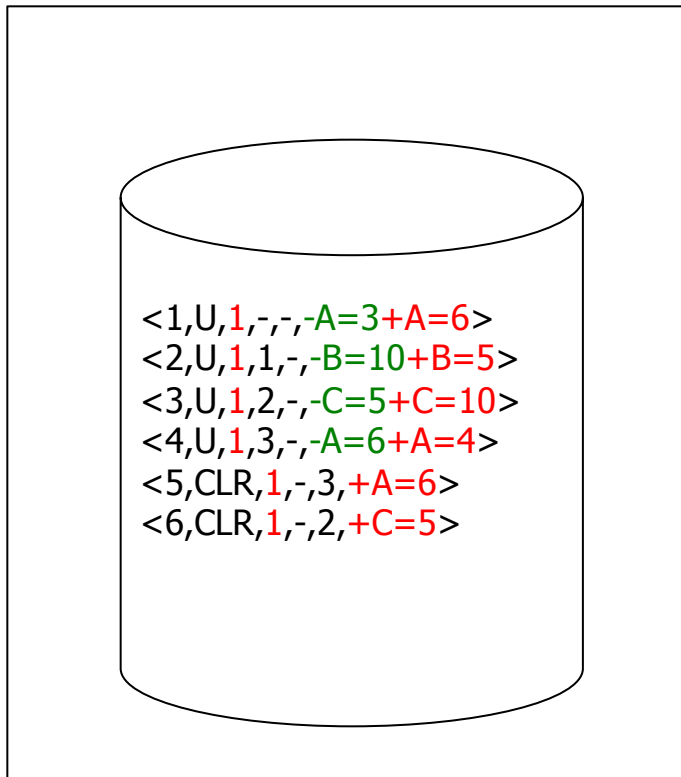
Transaction Table:

$\langle 1, U, 6, 2 \rangle$

Undo T_1

$T_1 = w_1(A), w_1(B), w_1(C), w_1(A), a_1$

buffer



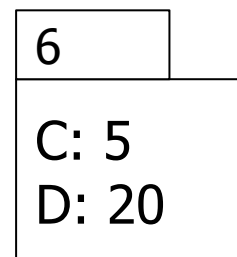
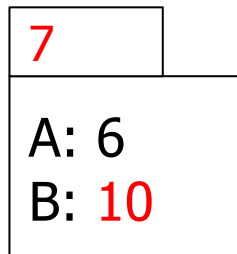
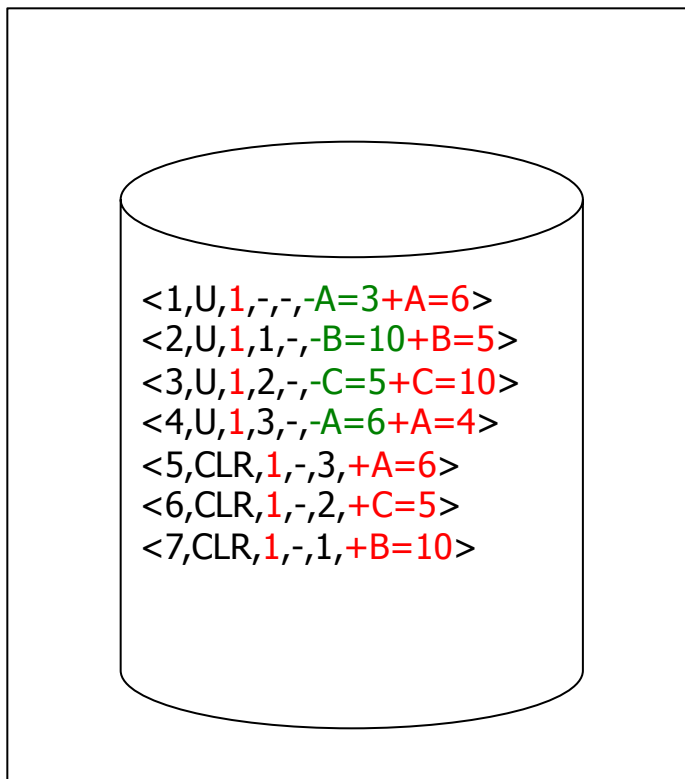
Transaction Table:

$\langle 1, U, 7, 1 \rangle$

Undo T_1

$T_1 = w_1(A), w_1(B), w_1(C), w_1(A), a_1$

buffer



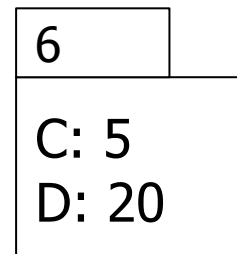
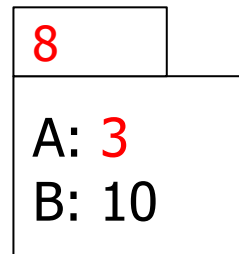
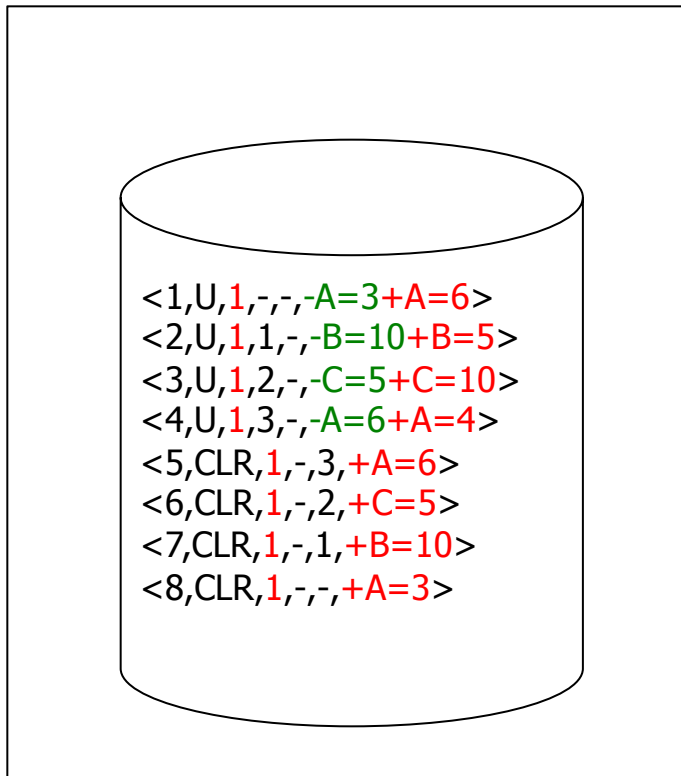
Transaction Table:

$\langle 1, U, 8, - \rangle$

Undo T_1

$T_1 = w_1(A), w_1(B), w_1(C), w_1(A), a_1$

buffer



Fuzzy Checkpointing in ARIES

- Begin of checkpoint
 - Write **begin_cp** log entry
 - Write **end_cp** log entry with
 - Dirty page table
 - Transaction table
- **Master Record**
 - LSN of begin_cp log entry of last complete checkpoint

Restart Recovery

1. Analysis Phase
2. Redo Phase
3. Undo Phase

Analysis Phase

- 1)** Determine LSN of last checkpoint using Master Record
- 2)** Log Dirty Page Table and Transaction Table from checkpoint record
- 3)** **RedoLSN** = $\min(\text{RecLSN})$ from Dirty Page Table or checkpoint LSN if no dirty page

Analysis Phase

4) Scan log forward starting from RedoLSN

- Update log entry from transaction
 - If necessary: Add to Page to Dirty Page Table
 - Add Transaction to Transaction Table or update LastLSN
- Transaction end entry
 - Remove transaction from Transaction Table

Analysis Phase

- Result
 - Transaction Table
 - Transactions to be later undone
 - RedoLSN
 - Log entry to start Redo Phase
 - Dirty Page Table
 - Pages that may not have been written back to disk

Redo Phase

- Start at RedoLSN scan log forward
- Unconditional Redo
 - Even redo actions of transactions that will be undone later
- One redo once
 - Only redo operations that have not been reflected on disk (PageLSN)

Redo Phase

- For each update log entry
 - If affected page is not in Dirty Page Table or $\text{RecLSN} > \text{LSN}$
 - skip log entry
 - Fix page in buffer
 - If $\text{PageLSN} \geq \text{LSN}$ then operation already reflected on disk
 - Skip log entry
 - Otherwise apply update

Redo Phase

- Result
 - State of DB before Failure

Undo Phase

- Scan log backwards from end using Transaction Table
 - Repeatedly take log entry with max LSN from all the current action to be undone for each transaction
 - Write CLR
 - Update Transaction Table

Undo Phase

- All unfinished transactions have been rolled back

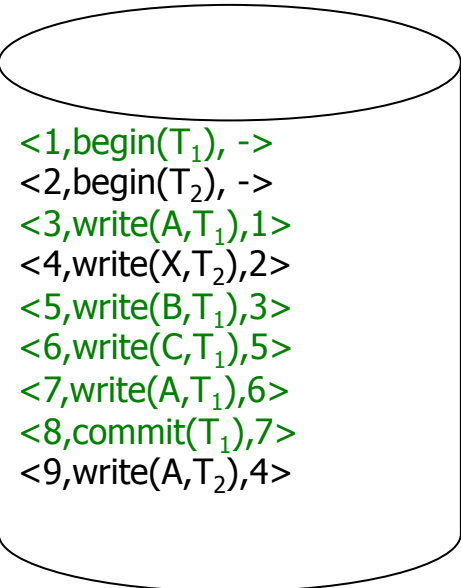
Idempotence?

- Redo
 - We are not logging during Redo so repeated Redo will result in the same state
- Undo
 - If we see CLR's we do not undo this action again

$T_1 = w_1(A), w_1(B), w_1(C), w_1(A), c_1$

$T_2 = w_1(X), r(A), w(A)$

Log



<1,begin(T_1), ->
<2,begin(T_2), ->
<3,write(A, T_1), 1>
<4,write(X, T_2), 2>
<5,write(B, T_1), 3>
<6,write(C, T_1), 5>
<7,write(A, T_1), 6>
<8,commit(T_1), 7>
<9,write(A, T_2), 4>

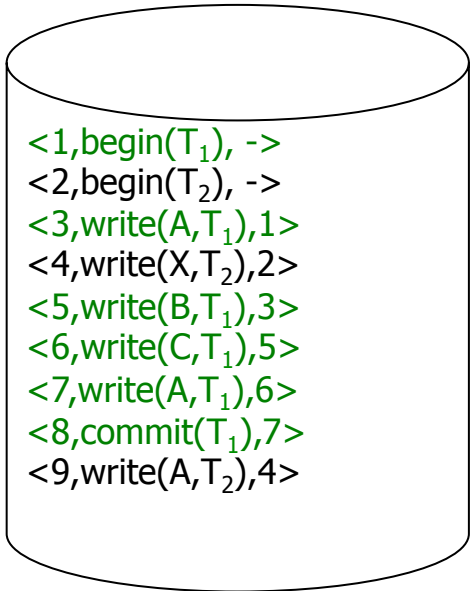
$T_1 = w_1(A), w_1(B), w_1(C), w_1(A), c_1$

$T_2 = w_1(X), r(A), w(A)$

Analysis Phase:

- start at log entry 1
- add T_1 to transaction table (rec. 1)
- add T_2 to transaction table (rec. 2)
- add A to page table (RecLSN 3)
- add X to page table (RecLSN 4)
- add B to page table (RecLSN 5)
- add C to page table (RecLSN 6)
- remove T_1 from Transaction Table (rec. 8)

Log



```
<1,begin(T1), ->  
<2,begin(T2), ->  
<3,write(A,T1),1>  
<4,write(X,T2),2>  
<5,write(B,T1),3>  
<6,write(C,T1),5>  
<7,write(A,T1),6>  
<8,commit(T1),7>  
<9,write(A,T2),4>
```


$T_1 = w_1(A), w_1(B), w_1(C), w_1(A), c_1$

$T_2 = w_1(X), r(A), w(A)$

Analysis Phase Result:

- Transaction Table:

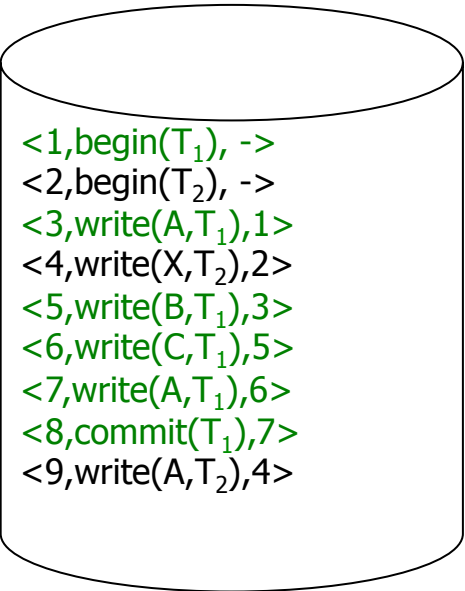
$\langle T_2, 9 \rangle$

- Dirty Page Table:

$\langle A, 3 \rangle, \langle B, 5 \rangle, \langle C, 6 \rangle, \langle X, 4 \rangle$

- RedoLSN = $\min(3, 5, 6, 4) = 3$

Log



$\langle 1, \text{begin}(T_1), - \rangle$
 $\langle 2, \text{begin}(T_2), - \rangle$
 $\langle 3, \text{write}(A, T_1), 1 \rangle$
 $\langle 4, \text{write}(X, T_2), 2 \rangle$
 $\langle 5, \text{write}(B, T_1), 3 \rangle$
 $\langle 6, \text{write}(C, T_1), 5 \rangle$
 $\langle 7, \text{write}(A, T_1), 6 \rangle$
 $\langle 8, \text{commit}(T_1), 7 \rangle$
 $\langle 9, \text{write}(A, T_2), 4 \rangle$

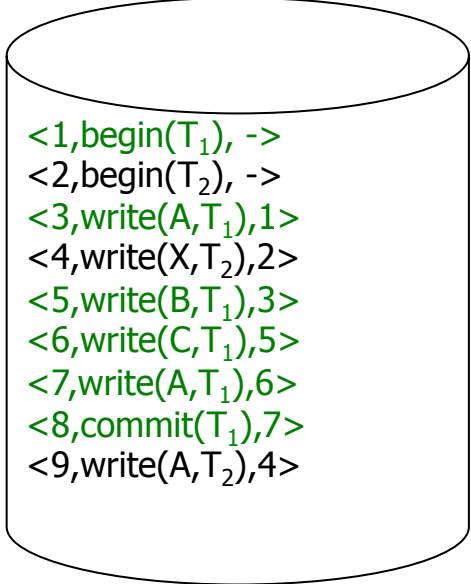
$T_1 = w_1(A), w_1(B), w_1(C), w_1(A), c_1$

$T_2 = w_1(X), r(A), w(A)$

Redo Phase (RedoLSN 3):

- Read A if PageLSN < 3 apply write
- Read X if PageLSN < 4 apply write
- Read B if PageLSN < 5 apply write
- Read C if PageLSN < 6 apply write
- Read A if PageLSN < 7 apply write
- Read A if PageLSN < 9 apply write

Log



<1,begin(T_1), ->
<2,begin(T_2), ->
<3,write(A,T_1),1>
<4,write(X,T_2),2>
<5,write(B,T_1),3>
<6,write(C,T_1),5>
<7,write(A,T_1),6>
<8,commit(T_1),7>
<9,write(A,T_2),4>

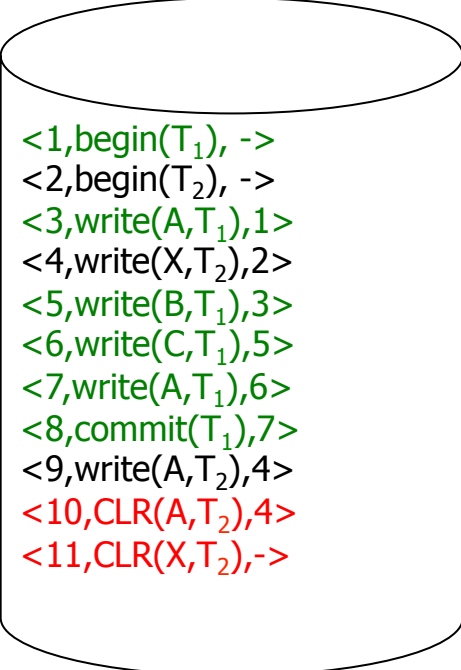
$T_1 = w_1(A), w_1(B), w_1(C), w_1(A), c_1$

$T_2 = w_1(X), r(A), w(A)$

Undo Phase (T_2):

- Undo entry 9
 - write CLR with UndoNxtLSN = 4
 - modify page A
- Undo entry 4
 - write CLR with UndoNxtLSN = 2
 - modify page X
- Done

Log



<1,begin(T_1), ->
<2,begin(T_2), ->
<3,write(A, T_1),1>
<4,write(X, T_2),2>
<5,write(B, T_1),3>
<6,write(C, T_1),5>
<7,write(A, T_1),6>
<8,commit(T_1),7>
<9,write(A, T_2),4>
<10,CLR(A, T_2),4>
<11,CLR(X, T_2),->

ARIES take away messages

- Provide good performance by
 - Not requiring complete checkpoints
 - Linking of log records
 - Not restricting buffer operations (no-force/steal is ok)
- Logical Undo and Physical (Physiological) Redo
- Idempotent Redo and Undo
 - Avoid undoing the same

Media Recovery

- What if disks where log or DB is stored failes
 - ->keep backups of log + DB state

Log Backup

- Split log into several files
- Is append only, backup of old files cannot interfere with current log operations

Backup DB state

- Copy current DB state directly from disk
- May be inconsistent
- -> Use log to know which pages are up-to-date and redo operations not yet reflected

Summary

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