

# Transaction: Recovery

Introduction to Databases

CompSci 316 Spring 2020



**DUKE**  
COMPUTER SCIENCE

# Recovery

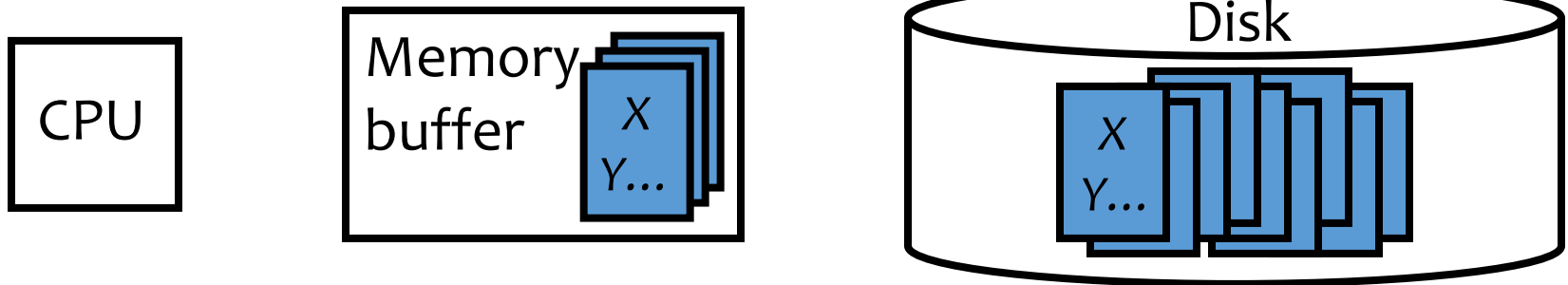
- Goal: ensure “A” (atomicity) and “D” (durability)



# Execution model

To read/write  $X$

- The disk block containing  $X$  must be first brought into memory
- $X$  is read/written in memory
- The memory block containing  $X$ , if modified, must be written back (flushed) to disk eventually



# Failures

Commit  $\neq$  Writing updates to disk!

- System crashes in the middle of a transaction  $T$ ; partial effects of  $T$  were written to disk
  - How do we undo  $T$  (**atomicity**)?
- System crashes right after a transaction  $T$  commits; not all effects of  $T$  were written to disk
  - How do we complete  $T$  (**durability**)?

# Naïve approach

- **Force:** When a transaction commits, all writes of this transaction must be reflected on disk
  - Without force, if system crashes right after  $T$  commits, effects of  $T$  will be lost
  - 👉 **Problem:** Lots of random writes hurt performance
- **No steal:** Writes of a transaction can only be flushed to disk at commit time
  - With steal, if system crashes before  $T$  commits but after some writes of  $T$  have been flushed to disk, there is no way to undo these writes
  - 👉 **Problem:** Holding on to all dirty blocks requires lots of memory

# Logging

- **Log**
  - Sequence of **log records**, recording all changes made to the database
  - Written to stable storage (e.g., disk) during normal operation
  - Used in recovery
- **Hey, one change turns into two—bad for performance?**
  - But writes are sequential (append to the end of log)
  - Can use dedicated disk(s) to improve performance

# Undo/redo logging rules

- When a transaction  $T_i$  starts, log  $\langle T_i, \text{start} \rangle$
- Record values before and after each modification:  
 $\langle T_i, X, \text{old\_value\_of\_X}, \text{new\_value\_of\_X} \rangle$ 
  - $T_i$  is transaction id and  $X$  identifies the data item
- A transaction  $T_i$  is committed when its commit log record  
 $\langle T_i, \text{commit} \rangle$  is written to disk

# WAL

- **Write-ahead logging (WAL)**: Before  $X$  is modified on disk, the log record pertaining to  $X$  must be flushed
  - Without WAL, system might crash after  $X$  is modified on disk but before its log record is written to disk—no way to undo

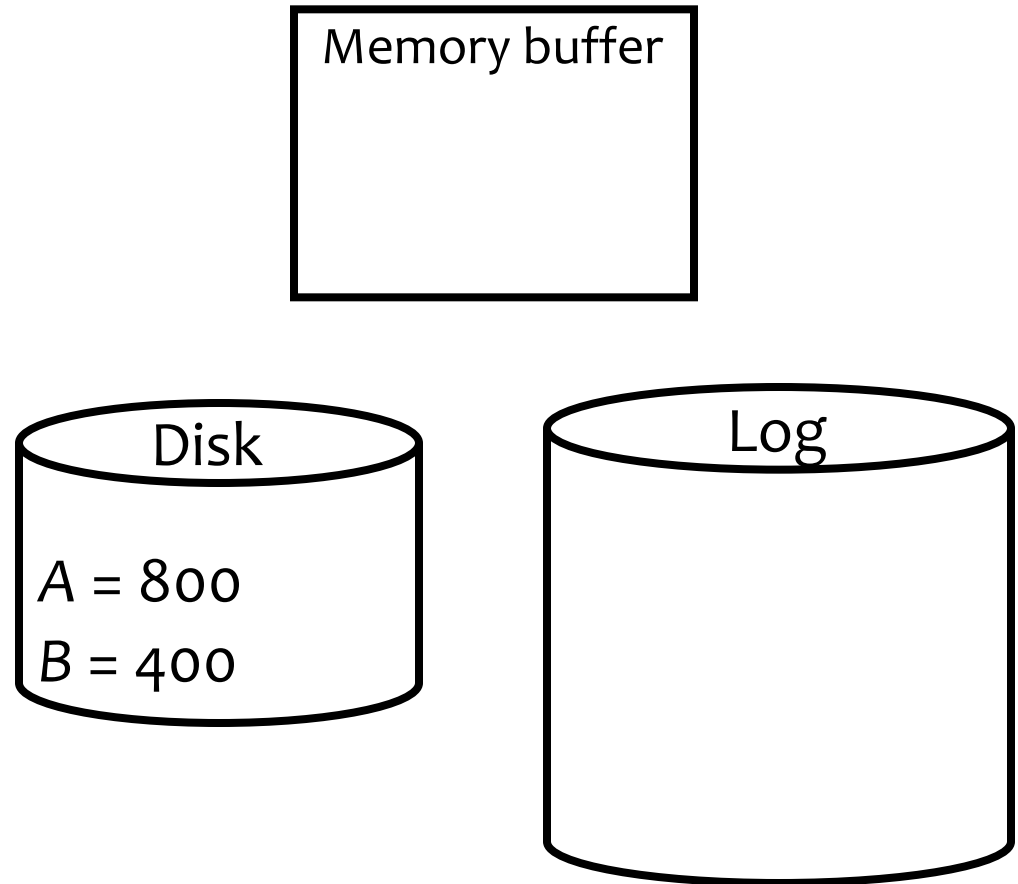
See difference with naïve approach

- **No force**: A transaction can commit even if its modified memory blocks have not be written to disk (since redo information is logged)
- **Steal**: Modified memory blocks can be flushed to disk anytime (since undo information is logged)



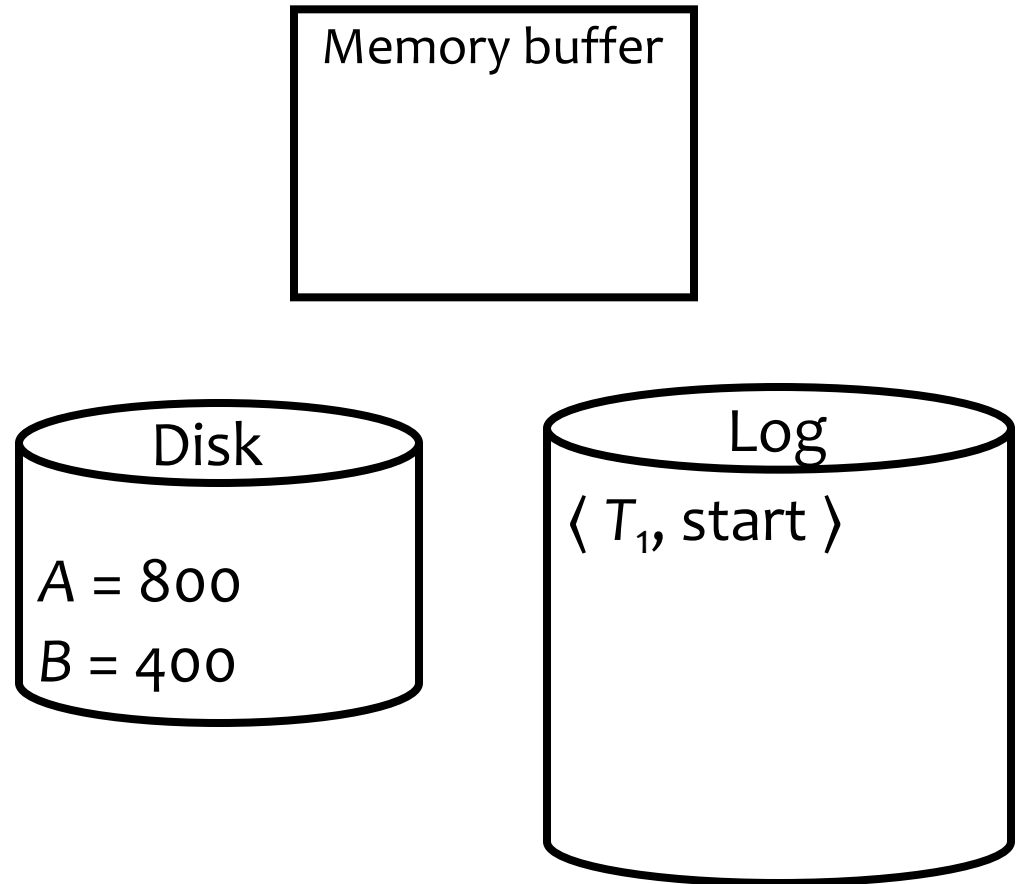
# Undo/redo logging example

$T_1$  (balance transfer of \$100 from A to B)



# Undo/redo logging example

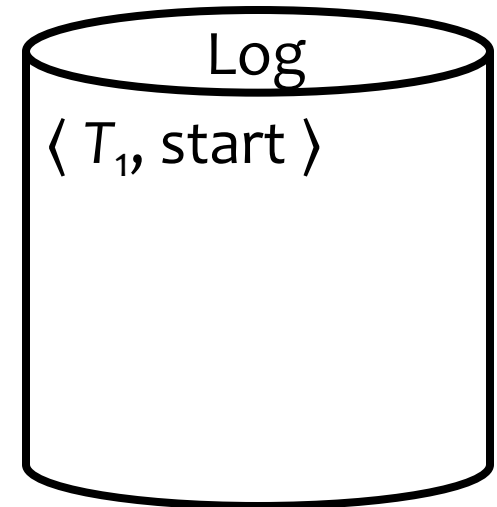
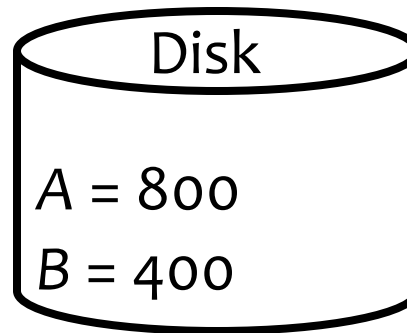
$T_1$  (balance transfer of \$100 from A to B)



# Undo/redo logging example

$T_1$  (balance transfer of \$100 from A to B)

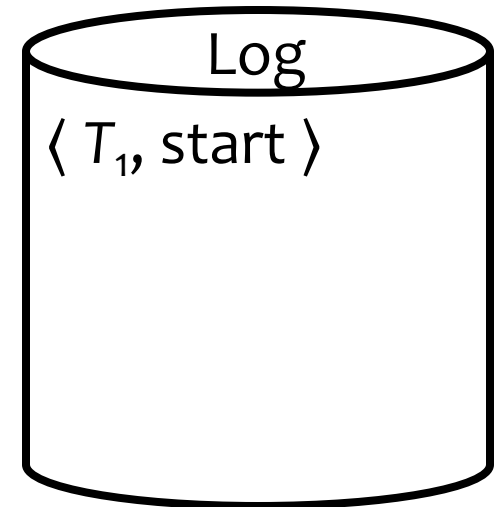
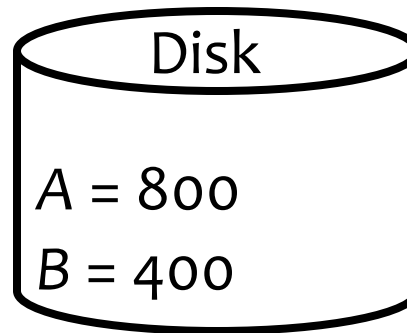
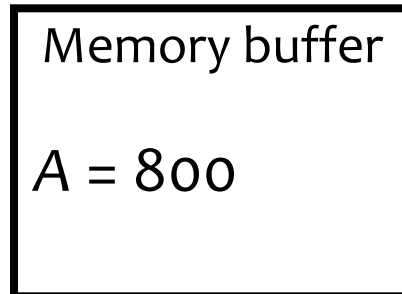
`read(A, a); a = a - 100;`



# Undo/redo logging example

$T_1$  (balance transfer of \$100 from A to B)

`read(A, a); a = a - 100;`

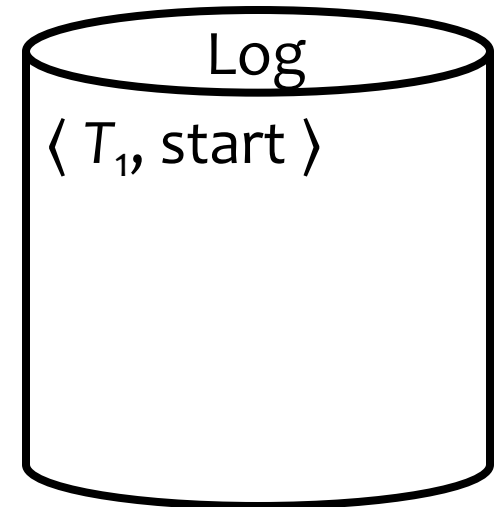
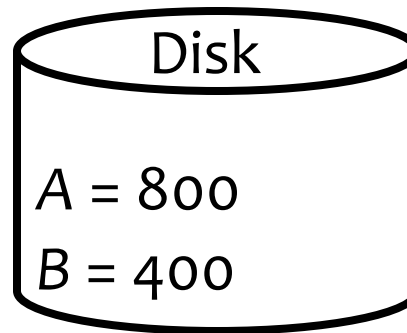
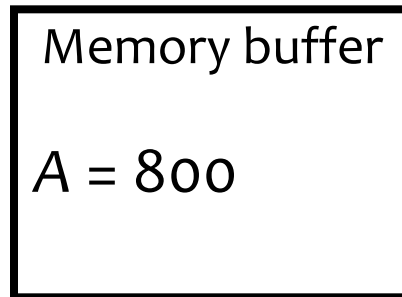


# Undo/redo logging example

$T_1$  (balance transfer of \$100 from A to B)

`read(A, a); a = a - 100;`

`write(A, a);`

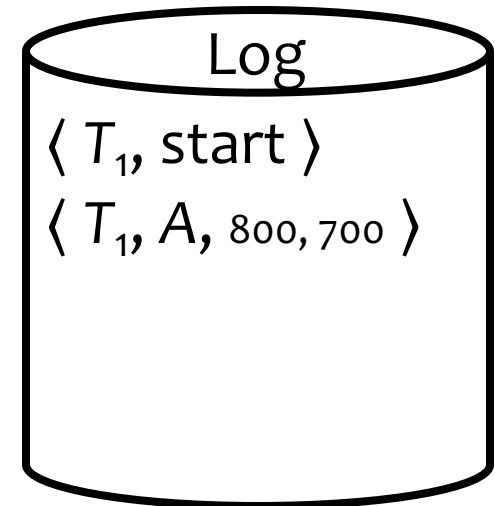
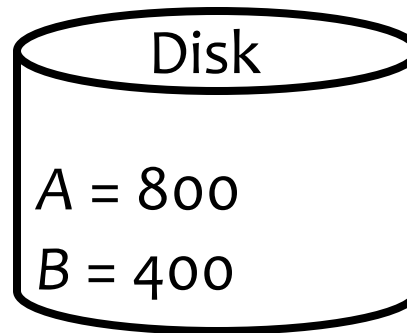
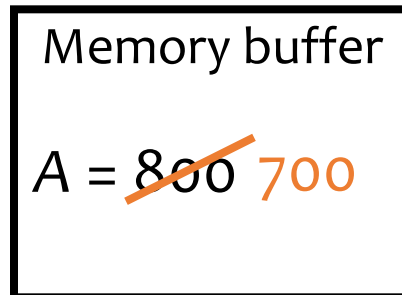


# Undo/redo logging example

$T_1$  (balance transfer of \$100 from A to B)

`read(A, a); a = a - 100;`

`write(A, a);`



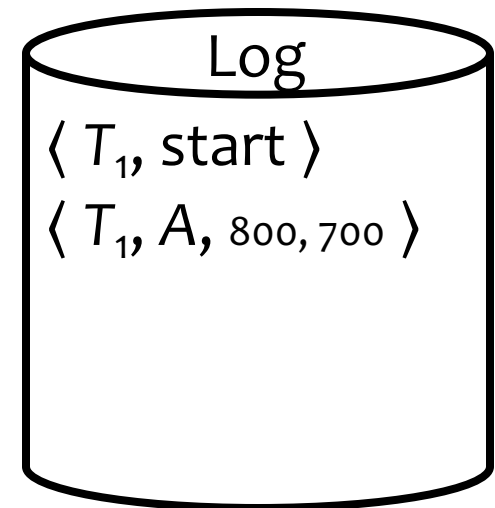
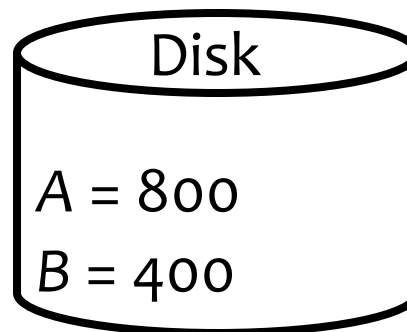
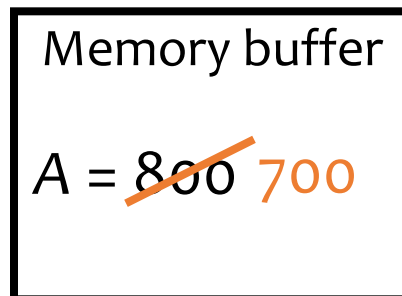
# Undo/redo logging example

$T_1$  (balance transfer of \$100 from A to B)

read(A, a);  $a = a - 100$ ;

write(A, a);

read(B, b);  $b = b + 100$ ;



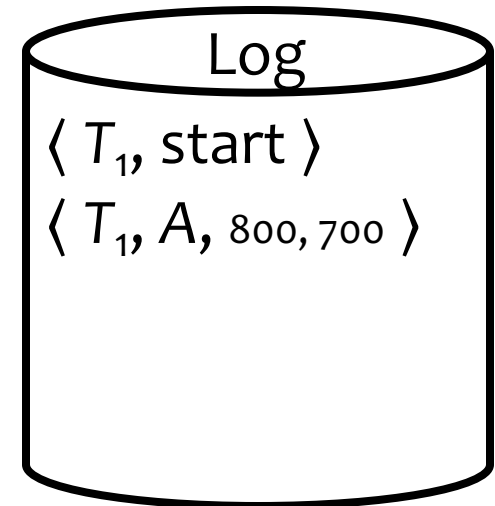
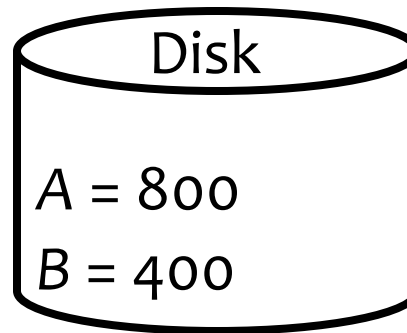
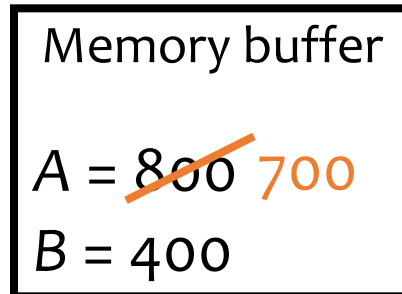
# Undo/redo logging example

$T_1$  (balance transfer of \$100 from A to B)

read(A, a);  $a = a - 100$ ;

write(A, a);

read(B, b);  $b = b + 100$ ;





# Undo/redo logging example

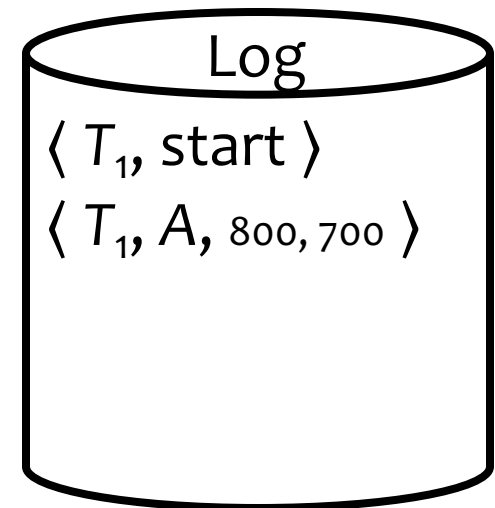
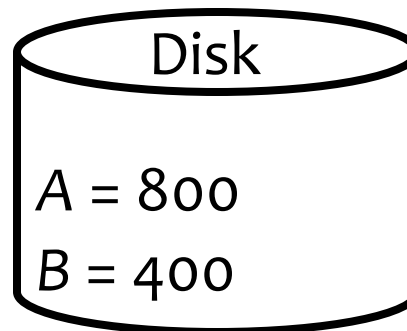
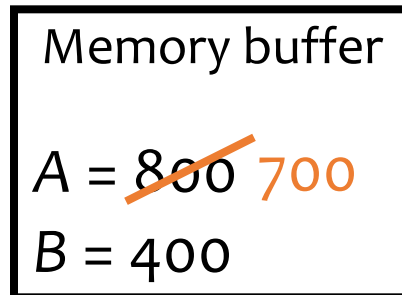
$T_1$  (balance transfer of \$100 from A to B)

read(A, a);  $a = a - 100$ ;

write(A, a);

read(B, b);  $b = b + 100$ ;

write(B, b);



# Undo/redo logging example

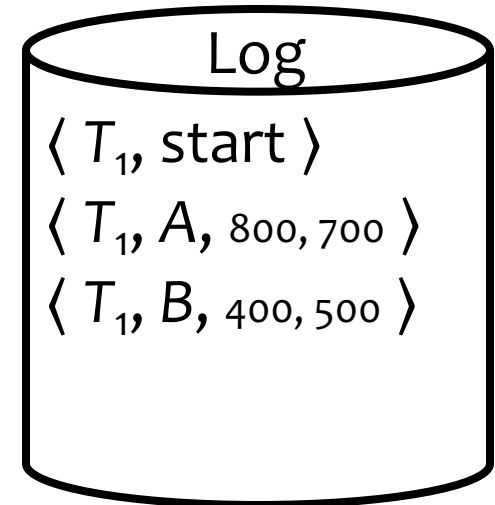
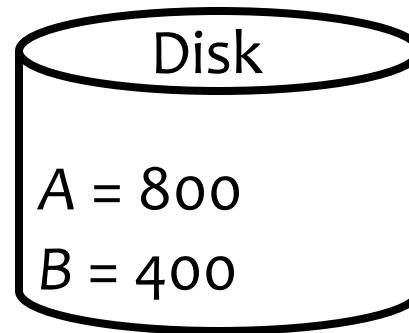
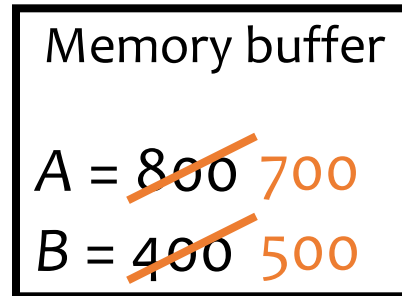
$T_1$  (balance transfer of \$100 from A to B)

read(A, a);  $a = a - 100$ ;

write(A, a);

read(B, b);  $b = b + 100$ ;

write(B, b);



# Undo/redo logging example

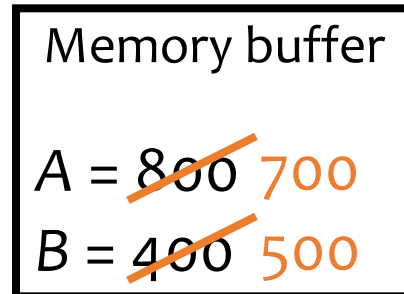
$T_1$  (balance transfer of \$100 from A to B)

read(A, a);  $a = a - 100$ ;

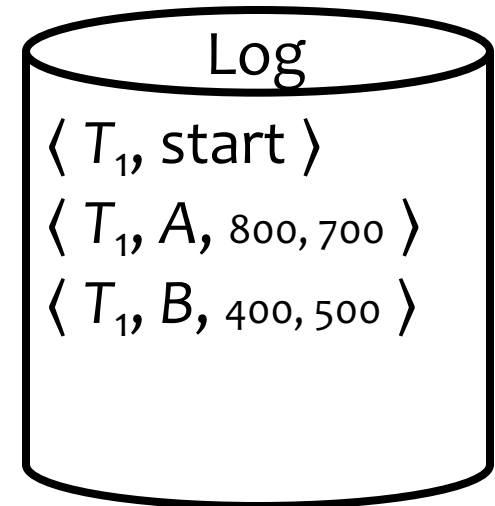
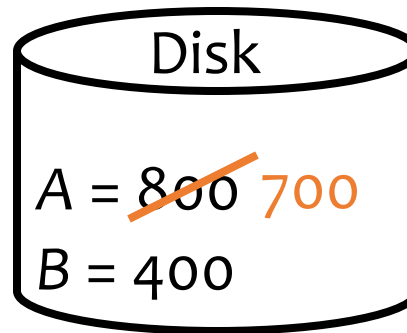
write(A, a);

read(B, b);  $b = b + 100$ ;

write(B, b);



Steal: can flush  
before commit



# Undo/redo logging example

$T_1$  (balance transfer of \$100 from A to B)

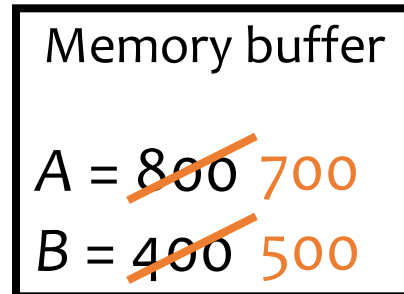
read(A, a);  $a = a - 100$ ;

write(A, a);

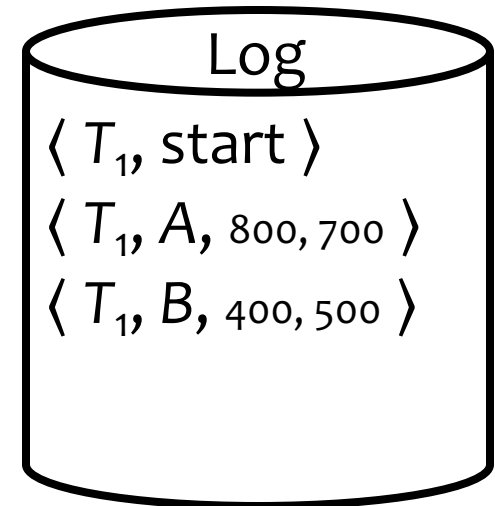
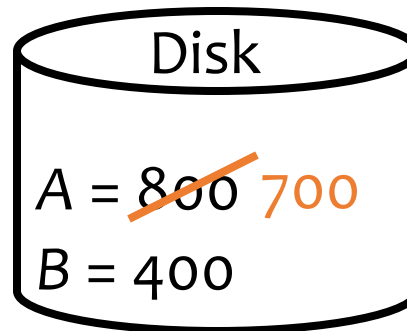
read(B, b);  $b = b + 100$ ;

write(B, b);

commit;



Steal: can flush  
before commit



# Undo/redo logging example

$T_1$  (balance transfer of \$100 from A to B)

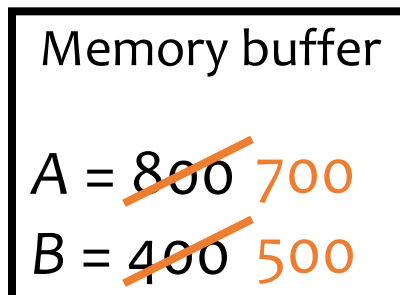
read(A, a);  $a = a - 100$ ;

write(A, a);

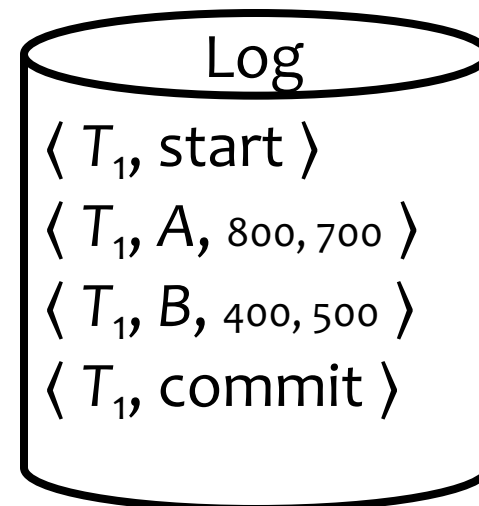
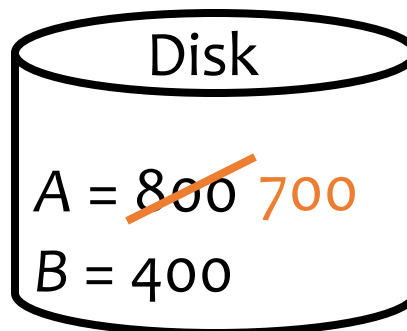
read(B, b);  $b = b + 100$ ;

write(B, b);

commit;



Steal: can flush  
before commit



# Undo/redo logging example

$T_1$  (balance transfer of \$100 from A to B)

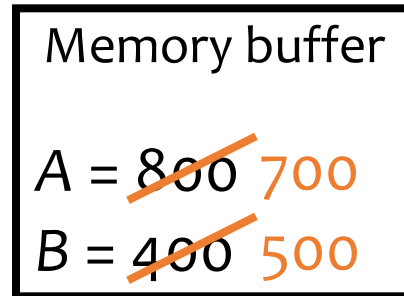
read(A, a);  $a = a - 100$ ;

write(A, a);

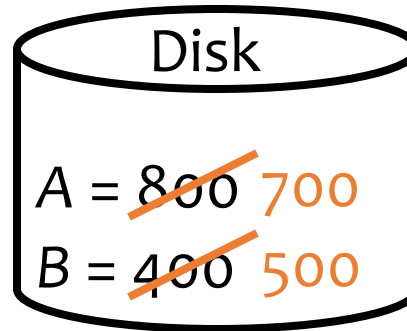
read(B, b);  $b = b + 100$ ;

write(B, b);

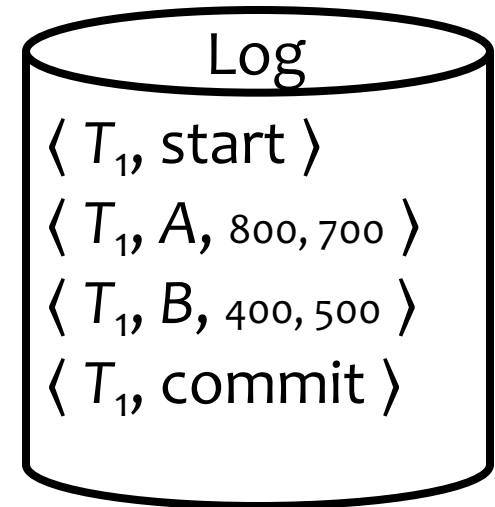
commit;



Steal: can flush  
before commit



No force: can flush  
after commit



# Undo/redo logging example

$T_1$  (balance transfer of \$100 from A to B)

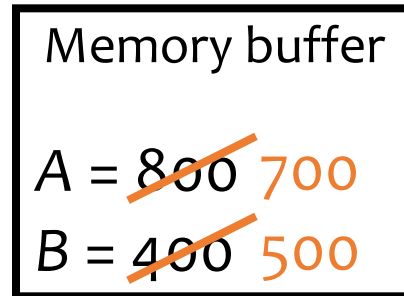
read(A, a);  $a = a - 100$ ;

write(A, a);

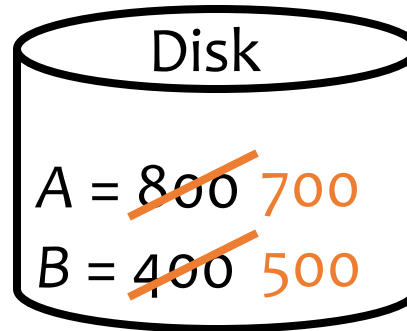
read(B, b);  $b = b + 100$ ;

write(B, b);

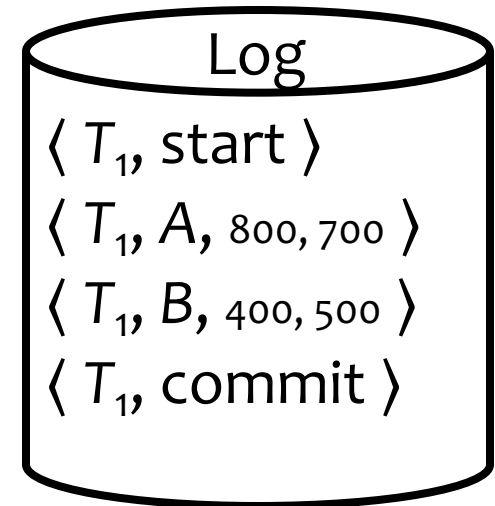
commit;



Steal: can flush  
before commit



No force: can flush  
after commit



No restriction (except WAL) on when memory blocks can/should be flushed

# Checkpointing

- Where does recovery start? Beginning of very large log file?
  - No – use checkpointing

Naïve approach:

- To checkpoint:
  - Stop accepting new transactions (**lame!**)
  - Finish all active transactions
  - Take a database dump
- To recover:
  - Start from last checkpoint





# Fuzzy checkpointing

- Add to log records `<START CKPT S>` and `<END CKPT>`
  - Transactions normally proceed and new transactions can start during checkpointing (between `START CKPT` and `END CKPT`)
- Determine  $S$ , the set of (ids of) **currently active transactions**, and log `< START CKPT S >`
- Flush all blocks (dirty at the time of the checkpoint) at your leisure
- Log `<END CKPT    START-CKPT_location >`
  - To easily access `<START CKPT>` of an `<END CKPT>` otherwise can read the log backward to find it

# An UNDO/REDO log with checkpointing

Log records
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>
<T2, B, 9, 10>
<START CKPT( T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

- T2 is active, T1 already committed
  - So <START CKPT (T2)>
- During CKPT,
  - flush A to disk if it is not already there (dirty buffer)
  - flush B to disk if it is not already there (dirty buffer)
  - Assume that the DBMS keeps track of dirty buffers

# Recovery using Log and CKPT: Three steps at a glance

## 1. Analysis

- Runs backward, from end of log, to the <START CKPT> of the last <END CKPT> record found (note this would be encountered “first” when reading backwards)
- Goal: Reach the relevant <START CKPT> record

## 2. Repeating history (also completes REDO for committed transactions)

- Runs forward, from START CKPT, to the end of log
- Goal: (1) Repeat all updates from START CKPT (whether or not they already went to the disk, whether or not they are from committed transactions), (2) Build set U of uncommitted transaction to be used in UNDO step below

## 3. UNDO

- Runs backward, from end of log, to the earliest <START T> of the uncommitted transactions stored in set U (note this may be before or after the <START CKPT> found in analysis step)
- Goal: UNDO the actions of uncommitted transactions

# Recovery: (1) analysis and (2) repeating history/REDO phase

- Need to determine  $U$ , the set of **active transactions at time of crash**
  - Scan log backward to find the **last <END CKPT> record** and follow the pointer to find the **corresponding <START CKPT S>**
  - Initially, let  $U$  be  $S$
  - Scan **forward** from that start-checkpoint to end of the log
    - For a log record  $\langle T, \text{start} \rangle$ , add  $T$  to  $U$
    - For a log record  $\langle T, \text{commit} \mid \text{abort} \rangle$ , remove  $T$  from  $U$
    - For a log record  $\langle T, X, \text{old}, \text{new} \rangle$ , issue  $\text{write}(X, \text{new})$
- ☞ **Basically repeats history!**

REDO is done and committed transactions are all in good shape now!  
Still need to do UNDO for aborted/uncommitted transactions

# Recovery: (3) UNDO phase

- Scan log **backward**
  - Undo the effects of transactions in  $U$
  - That is, for each log record  $\langle T, X, old, new \rangle$  where  $T$  is in  $U$ , issue  $write(X, old)$ , and log this operation too (part of the “repeating-history” paradigm)
  - Log  $\langle T, abort \rangle$  when all effects of  $T$  have been undone

## An optimization

- Each log record stores a pointer to the previous log record for the same transaction; follow the pointer chain during undo

# Recovery: Example 1

Log records
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>
<T2, B, 9, 10>
<START CKPT( T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>

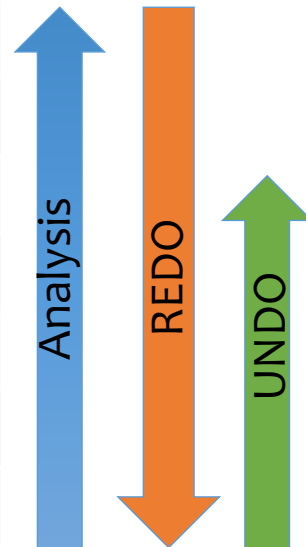


- T1 has committed and writes are already on disk
- After analysis,  $U = S = \{T2\}$
- **REDO all actions**
- Write C = 15 (T2)
- UPDATE U to  $\{T2, T3\}$
- Write D = 20 (T3)
- <COMMIT T2> found:  $U = \{T3\}$
- <COMMIT T3> found:  $U = \{\}$
- At the end  $U = \text{empty}$ , do nothing (**NO UNDO PHASE**)

Assume every log record before crash is on disk

# Recovery: Example 2

Log records
<START T1>
<T1, A, 4, 5>
<START T2>
<COMMIT T1>
<T2, B, 9, 10>
→ <START CKPT( T2)>
<T2, C, 14, 15>
<START T3>
<T3, D, 19, 20>
<END CKPT>
<COMMIT T2>
<COMMIT T3>



- T1 has committed and writes are already on disk
- After analysis,  $U = S = \{T2\}$
- **REDO all actions**
- Write  $C = 15$  (T2)
- UPDATE U to  $\{T2, T3\}$
- Write  $D = 20$  (T3)
- <COMMIT T2> found:  $U = \{T3\}$ 
  - not necessary to set B to 10 (before END CKPT – already on disk)
- **UNDO actions of T3 until its start**
- Write  $D = 19$  (T3)

Assume every log record before crash is on disk

# Recovery: Example 3

Log records	
<START T1>	
<T1, A, 4, 5>	
<START T2>	
<COMMIT T1>	
<T2, B, 9, 10>	
<START CKPT( T2)>	
<T2, C, 14, 15>	
<START T3>	
<T3, D, 19, 20>	
<END CKPT>	
<COMMIT T3>	
<COMMIT T2>	

Analysis

REDO

UNDO

- T1 has committed and writes are already on disk
- After analysis,  $U = S = \{T2\}$
- **REDO all actions**
- Write C = 15 (T2)
- UPDATE U to  $\{T2, T3\}$
- Write D = 20 (T3)
- <COMMIT T3> found:  $U = \{T2\}$
- **UNDO actions of T2 until its start**
  - Beyond <START CKPT>!
  - Those changes already went to disk
- Write C = 14 (T2)
- Write B = 9 (T2)

Assume every log record before crash is on disk



# Summary: Transactions

- **Concurrency control**
  - Serial schedule: no interleaving
  - Conflict-serializable schedule: no cycles in the precedence graph; equivalent to a serial schedule
  - 2PL: guarantees a conflict-serializable schedule
  - Strict 2PL: also guarantees recoverability
- **Recovery: undo/redo logging with fuzzy checkpointing**
  - Normal operation: write-ahead logging, no force, steal
  - Recovery: first redo (forward), and then undo (backward)