CompSci 516
Database Systems

Lecture 15
Transactions – Recovery

Instructor: Sudeepa Roy

Announcements (Thurs 2/24)

- HW2-part 1 due next week 3/3 (Thursday) 12 noon
  - Extended deadline by two days
  - One group submission per pair is needed
  - Part 2 (on cloud) will be released later in the semester if we have time due to change in AWS setups
  - If you are still looking for someone to work with, **send me an email NOW by 12 noon today!**
  - Shweta had a tutorial yesterday, watch the recording
- Midterm Project report due next week 3/4 (Friday) 12 noon
  - Extended by ~3 days
  - Keep working on your project

Reading Material

- [GUW]
  - 17.4: UNDO/REDO
  - Lecture slides will be sufficient for exams

Acknowledgement:
A few of the following slides have been created adapting the instructor material of the [RG] book provided by the authors Dr. Ramakrishnan and Dr. Gehrke.

Today

Recovery

- STEAL/ NO STEAL
- FORCE/NO FORCE
- UNDO/REDO log
- Checkpointing and Recovery

Review: The ACID properties

- **A** tomicity: All actions in the transaction happen, or none happen.
- **C** onsistency: If each transaction is consistent, and the DB starts consistent, it ends up consistent.
- **I** solation: Execution of one transaction is isolated from that of other transactions.
- **D** urability: If a transaction commits, its effects persist.

- Which property did we cover in CC? : Isolation
- Now : Atomicity and Durability by recovery manager

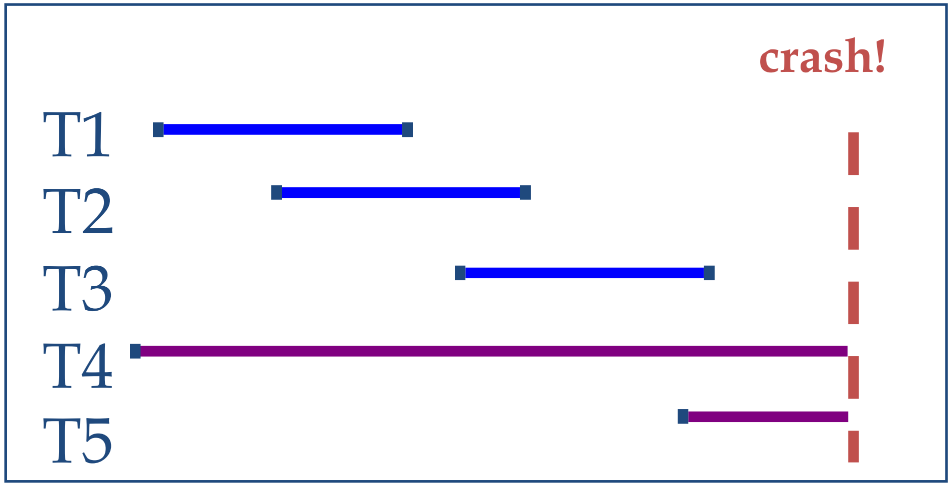
Motivation: A & D

Commit ≠ Disk Write!

Abort ≠ No Disk Write!

Eventually yes, but not necessarily immediately

- Atomicity:
  - Transactions may abort ("Rollback").
- Durability:
  - What if DBMS stops running?
  - (power failure/crash/error/fire-flood etc.)



- Desired Behavior after system restarts:
  - T1, T2 & T3 should be durable.
  - T4 & T5 should be aborted (effects not seen).

Recovery: A & D

- Atomicity
  – by "undo"ing actions of "aborted transactions"

- Durability
  – by making sure that all actions of committed transactions survive crashes and system failure
  – i.e. by "redo"-ing actions of "committed transactions"

Assumptions

- Concurrency control is in effect
- Updates are happening "in place".
  - i.e., data is overwritten on (deleted from) the disk.
- Simple schemes to guarantee Atomicity & Durability (next):
  - NO STEAL
  - FORCE

Handling the Buffer Pool

- Force every write to disk?
- Steal buffer-pool frames from uncommitted transactions?

	No Steal	Steal
Force	Trivial	
No Force		Desired

Handling the Buffer Pool

- Force every write to disk?
  - Poor response time
  - But provides durability

- Not Steal buffer-pool frames from uncommitted transactions?
  - If not steal, poor throughput, holding on to all dirty blocks requires lots of memory
  - If steal, how can we ensure atomicity?

	No Steal	Steal
Force	Trivial	
No Force		Desired

What if we do "Steal" and "NO Force"

- STEAL (why enforcing Atomicity is hard)
  - To steal frame F: Current page in F (say P) is written to disk; some transaction holds lock on P
  - What if the transaction with the lock on P aborts?
  - Must remember the old value of P at steal time (to support UNDOing the write to page P)

- NO FORCE (why enforcing Durability is hard)
  - What if system crashes before a modified page is written to disk?
  - Write as little as possible, in a convenient place, at commit time, to support REDOing modifications.

Basic Idea: Logging

- Log: An ordered list of REDO/UNDO actions
  - Log record may contain:

    <Tr.ID, pageID, offset, length, old data, new data>

- Record REDO and UNDO information, for every update, in a log
- one change turns into two—bad for performance?
  - Sequential writes to log (put it on a separate disk) – append only
  - Minimal info (diff) written to log, so multiple updates fit in a single log page
  - Log blocks are created and updated in the main memory first, then written to disk
  - Can use dedicated disk(s) to improve performance

Different types of logs

- UNDO (STEAL + FORCE)
- REDO (NO STEAL + NO FORCE)
- UNDO/REDO (STEAL + NO FORCE)

We only talk about UNDO/REDO
In this lecture

GUW 17.4
(Lecture material will be sufficient for Exams)

- ARIES
  – an UNDO/REDO log implementation

Will talk about ARIES if we have time later

UNDO/REDO logging

UNDO/REDO logging

- Simple representation for illustration
  - (actual implementation has more info)

- <T, X, v, w>
  - Transaction T changed the value of element X
  - former value v
  - new value w

UNDO/REDO logging rule

When a transaction *T* starts, log $\langle START\ T \rangle$

Before modifying any element X on disk, <T, X, v, w> must appear on disk

A transaction T_i is committed when its commit log record $\langle \text{COMMIT T} \rangle$ is written to disk

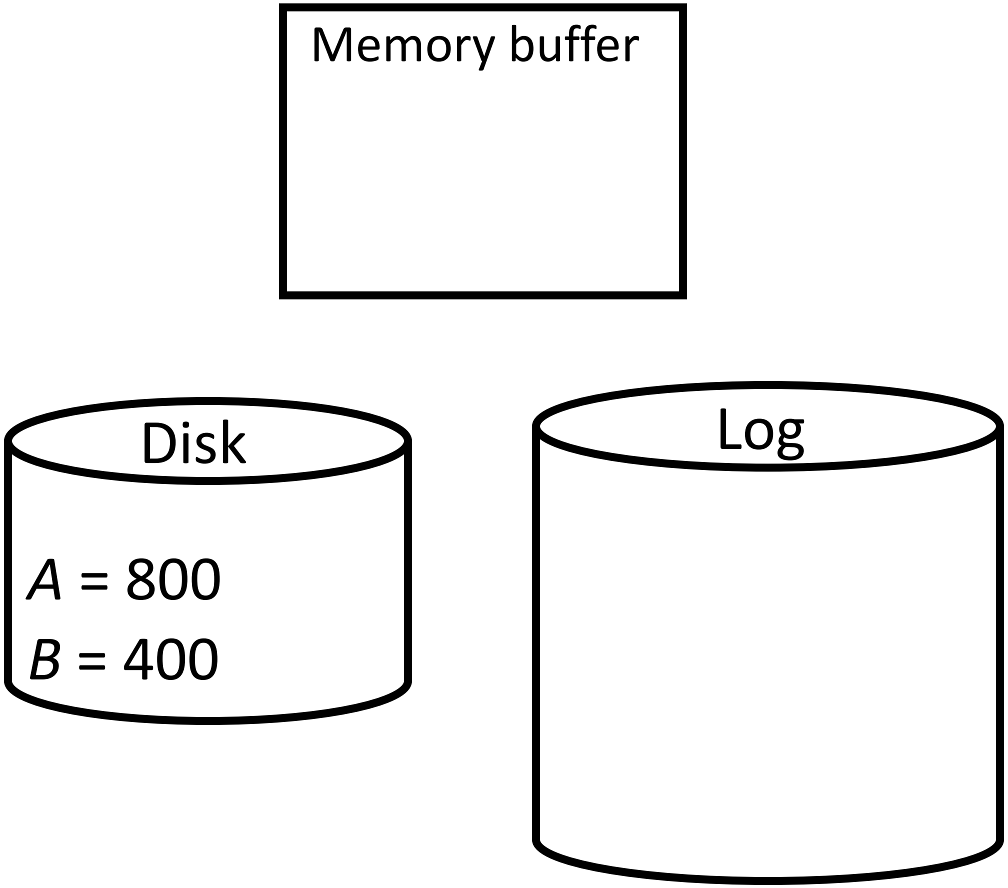
- can precede or follow any of the changes to the db elements on disk

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

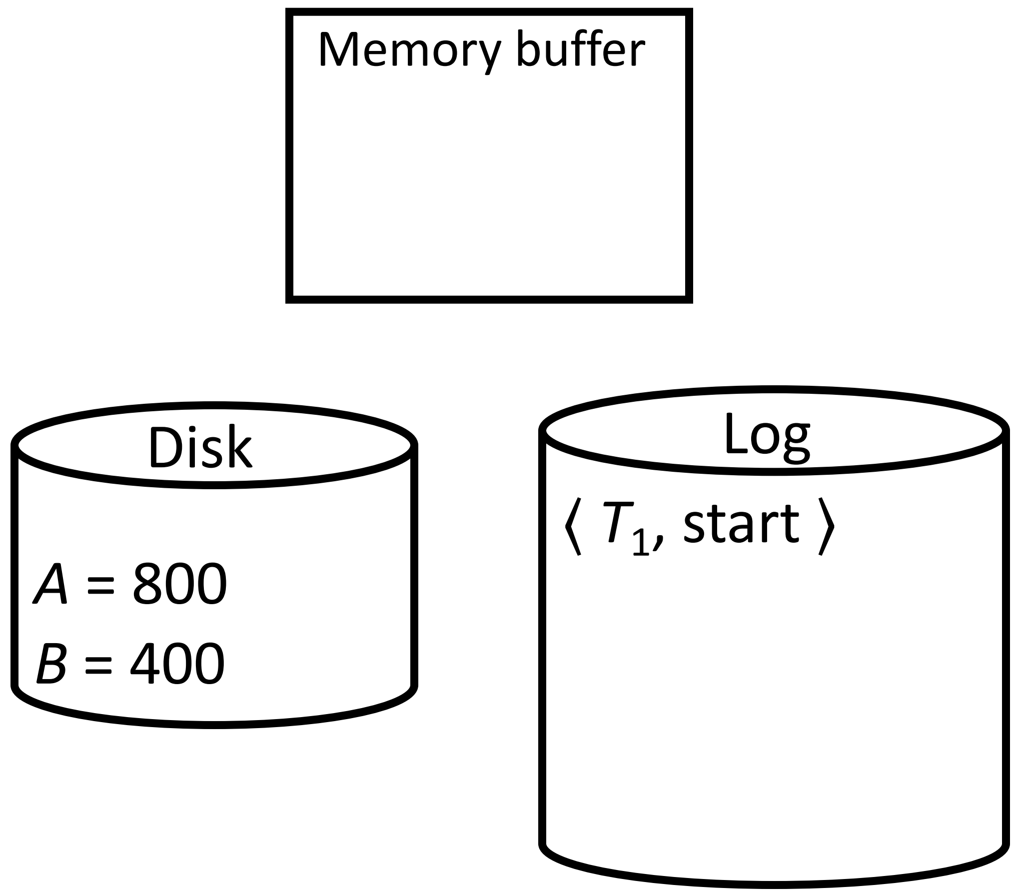
Undo/redo logging example

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



Undo/redo logging example

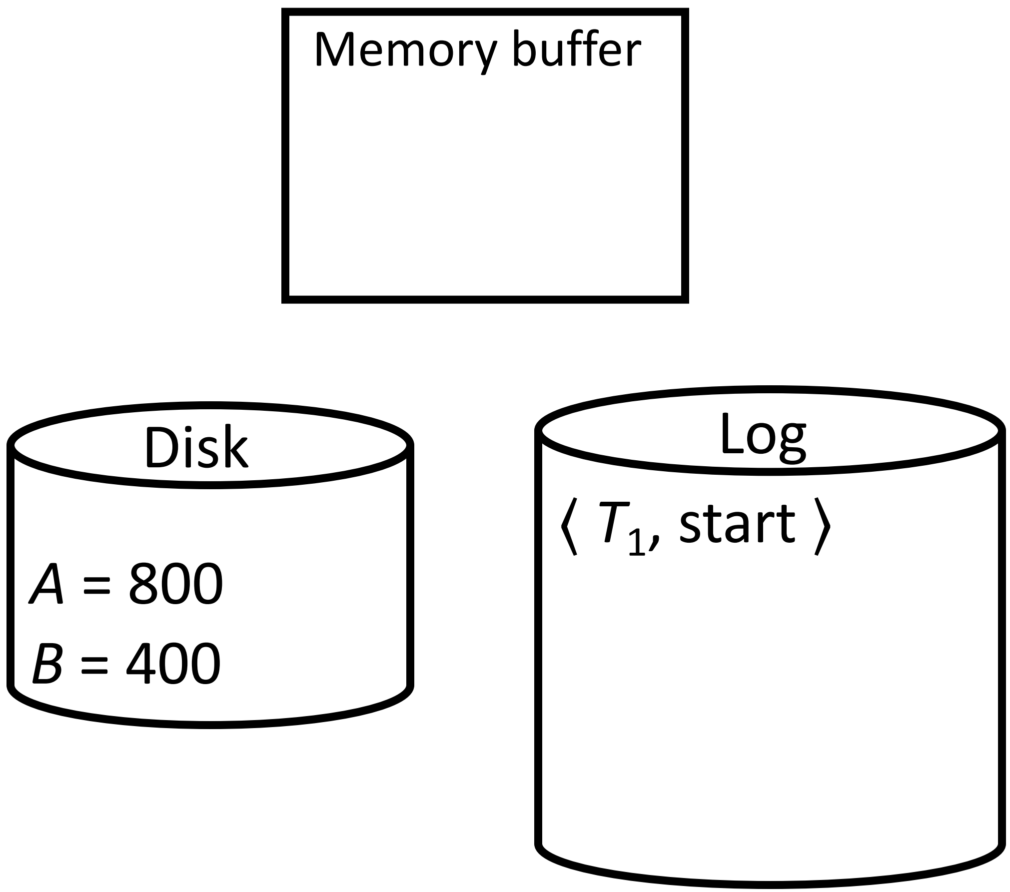
T_1 (balance transfer of \$100 from A to B)



Undo/redo logging example

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

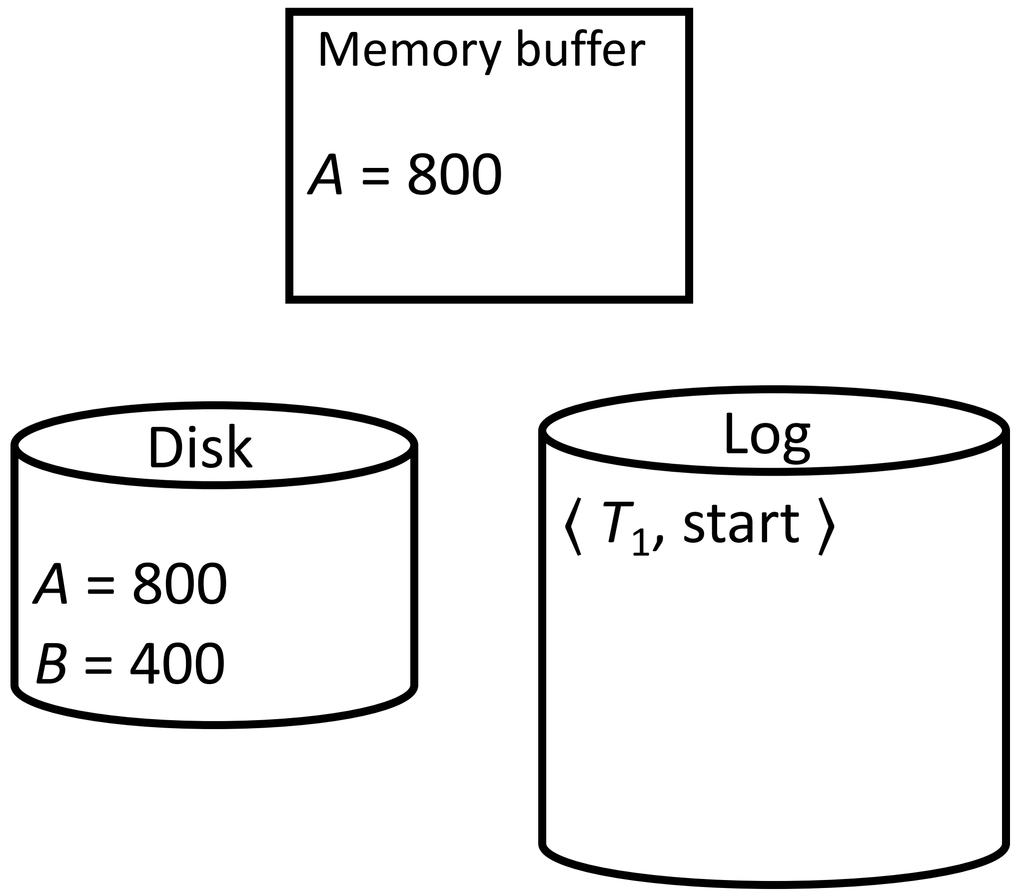
read(A, a); $a = a - 100$;



Undo/redo logging example

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

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

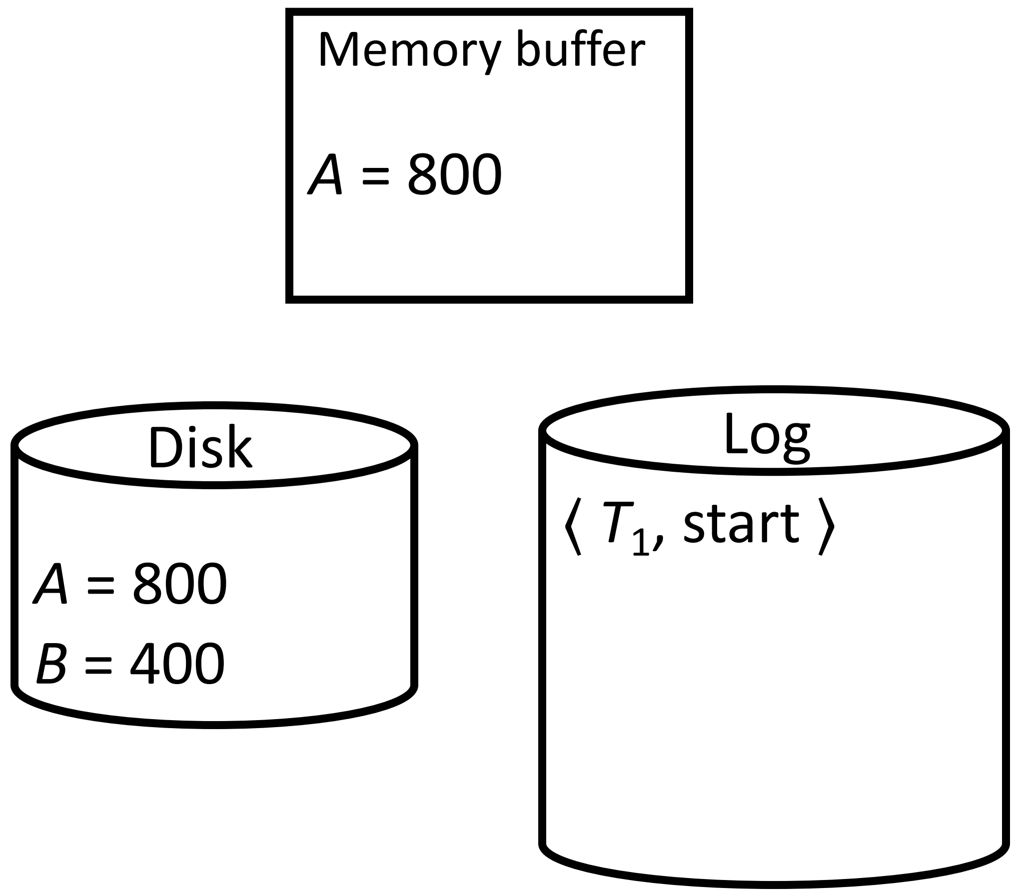


Undo/redo logging example

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

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

write(A, a);

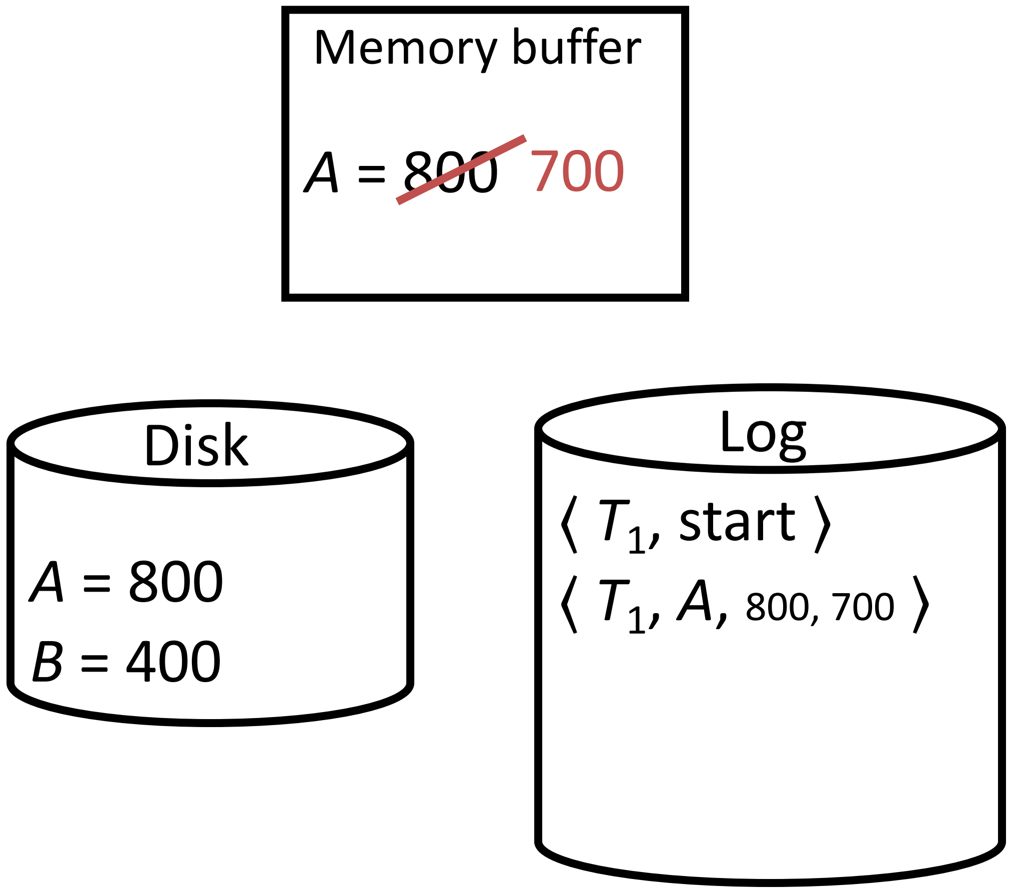


Undo/redo logging example

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

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

write(A, a);



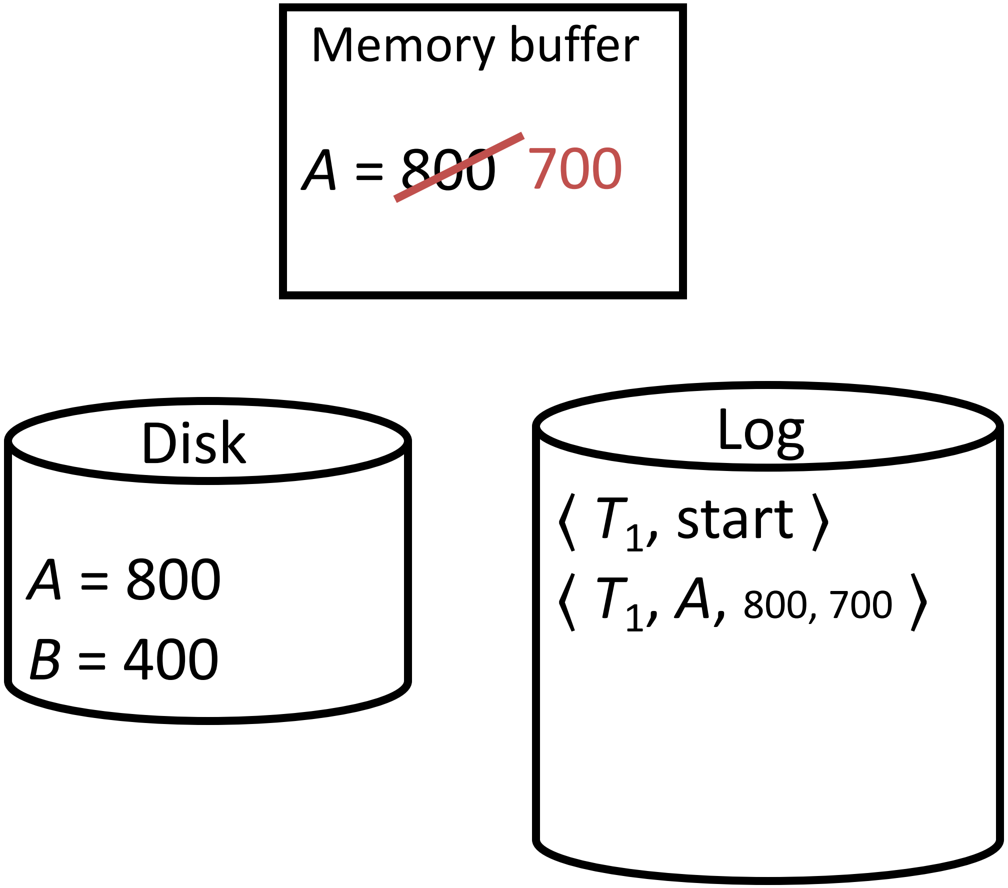
Undo/redo logging example

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;



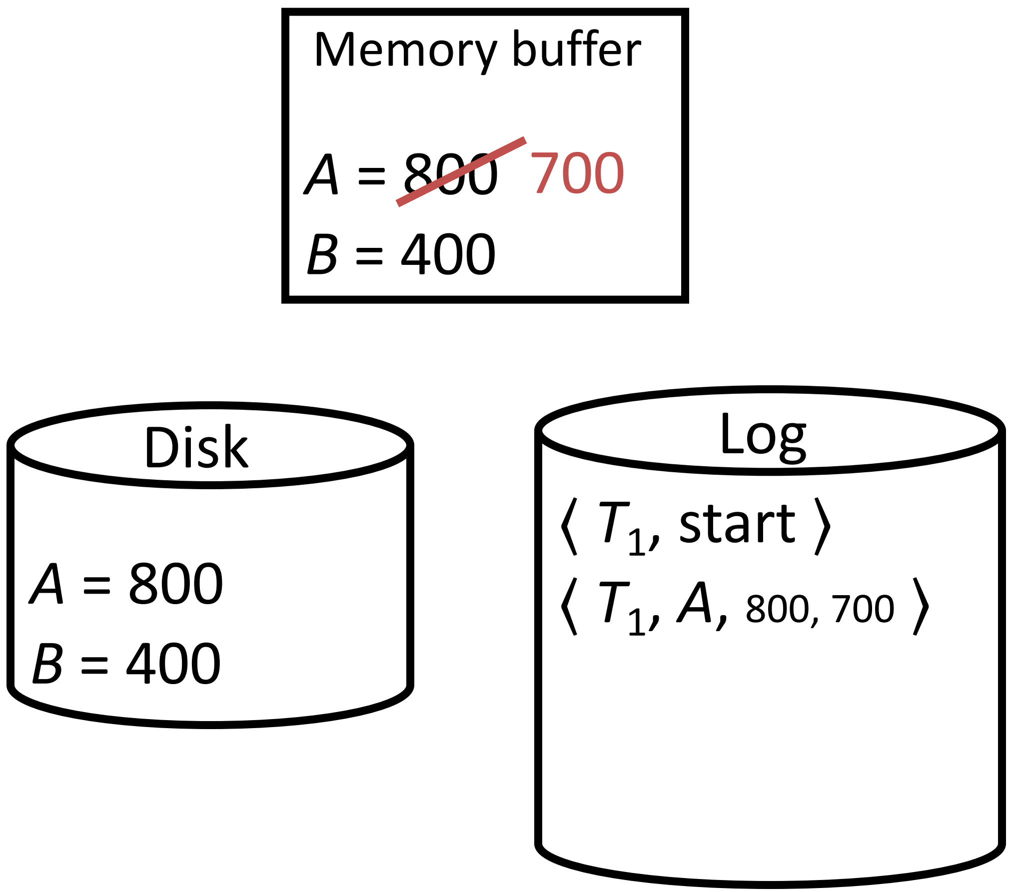
Undo/redo logging example

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

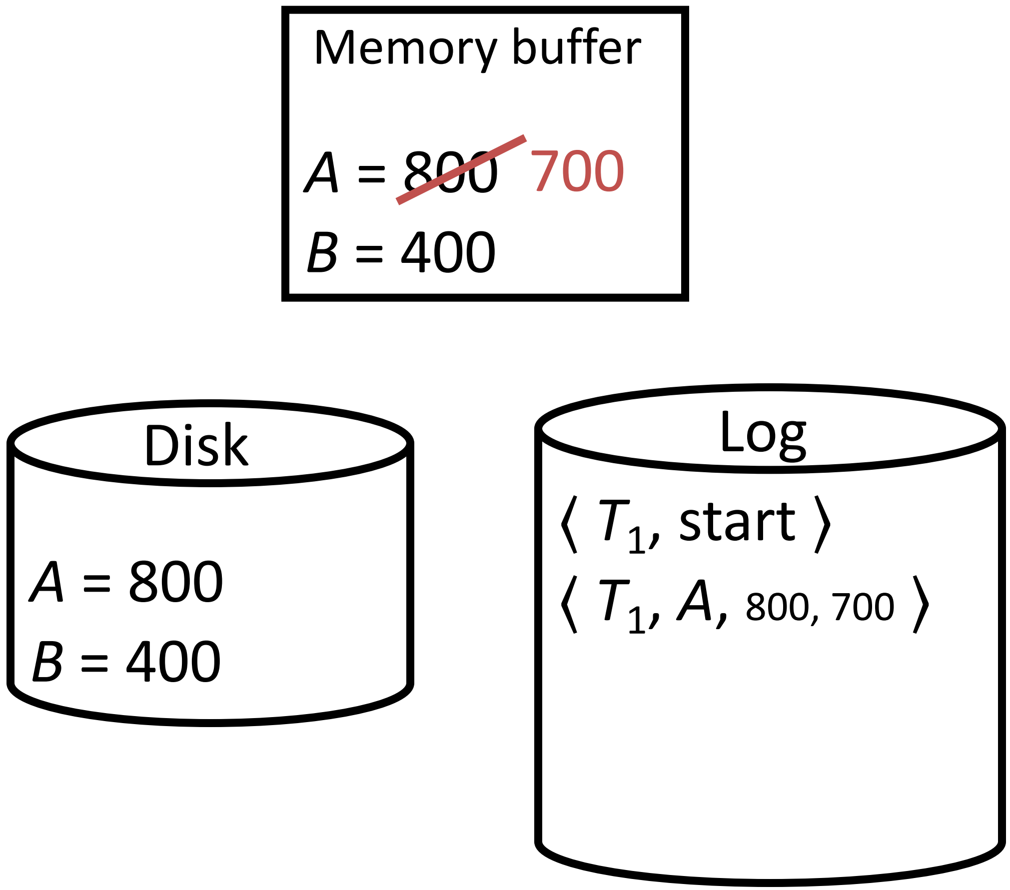
T_1 (balance transfer of \$100 from *A* to *B*)

read(*A*, *a*); *a* = *a* – 100;

write(*A*, *a*);

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

write(*B*, *b*);



Undo/redo logging example

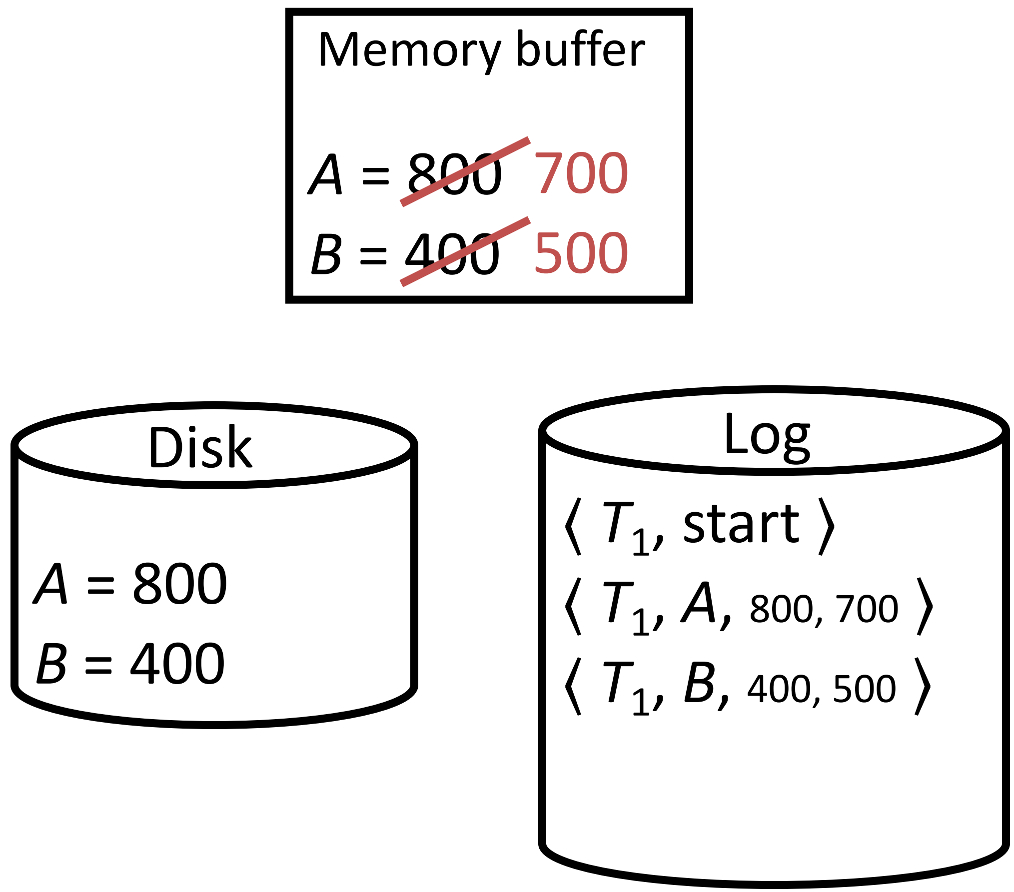
T_1 (balance transfer of \$100 from A to B)

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

write(A, a);

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

write(B, b);



Undo/redo logging example

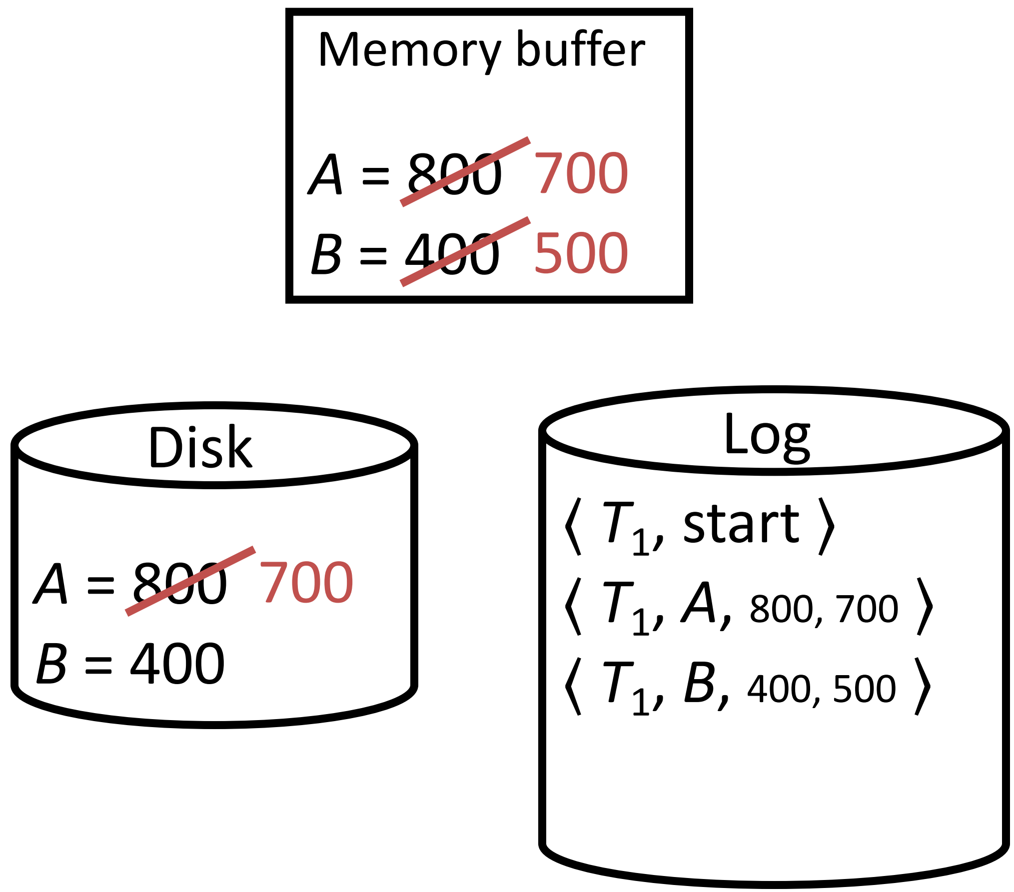
T_1 (balance transfer of \$100 from A to B)

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); $a = a - 100$;

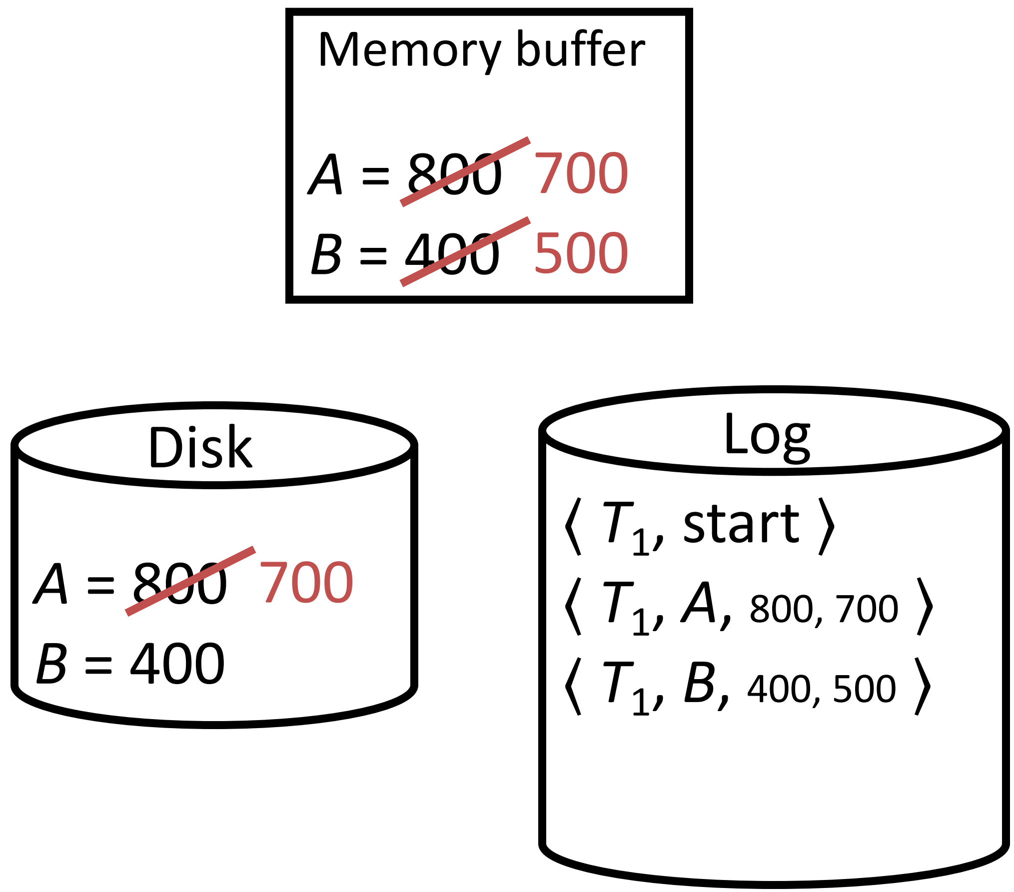
write(A, a);

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)

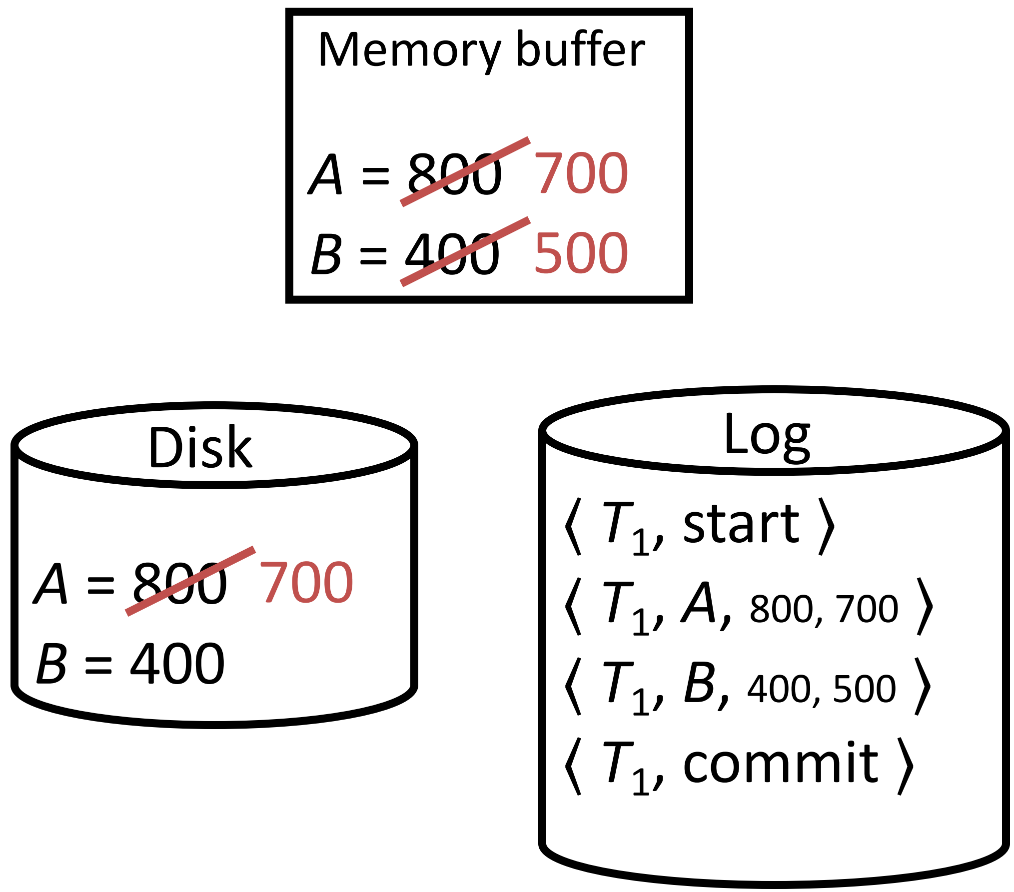
read(A, a); $a = a - 100$;

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*)

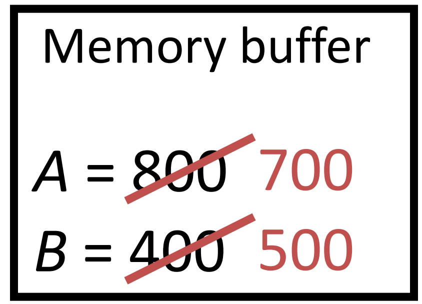
read(*A*, *a*); *a* = *a* – 100;

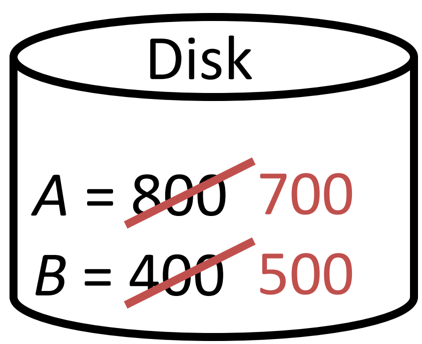
write(*A*, *a*);

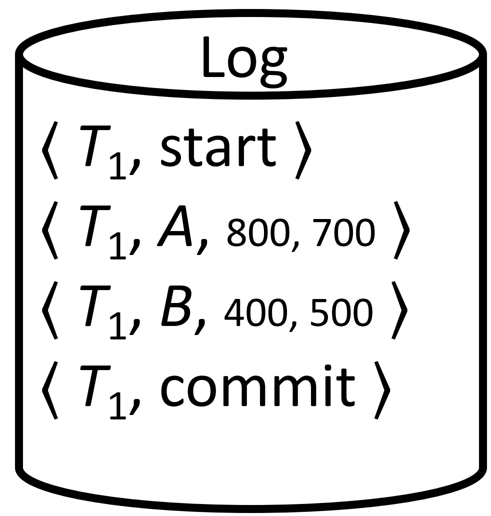
read(*B*, *b*); *b* = *b* + 100;

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;

Memory buffer

A = ~~800~~ 700

B = ~~400~~ 500

Disk

A = ~~800~~ 700

B = ~~400~~ 500

Log

⟨ T_1, start ⟩

⟨ T_1, A, 800, 700 ⟩

⟨ T_1, B, 400, 500 ⟩

⟨ T_1, 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

http://www.saintlouischeckpoints.com/wp-content/uploads/2013/08/dui20checkpoint200220172011.jpg

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 backword 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

Read yourself after seeing the examples next

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 uncomitted 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⟩

Read yourself after seeing the examples next

- Initially, let *U* be *S*
- Scan forward from that start-checkpoint to end of the log
  - For a log record ⟨ *T*, start ⟩, add *T* to *U*
  - For a log record ⟨ *T*, commit | abort ⟩, remove *T* from *U*
  - For a log record ⟨ *T*, *X*, *old*, *new* ⟩, issue write(*X*, *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

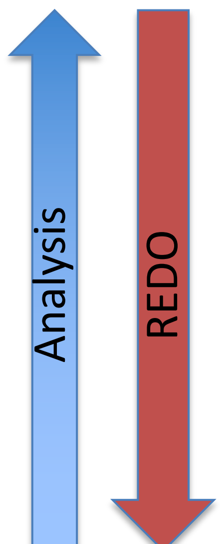
- Scan log backward
  - Undo the effects of transactions in *U*
  - That is, for each log record ⟨ *T*, *X*, *old*, *new* ⟩ where *T* is in *U*, issue write(*X*, *old*), and log this operation too (part of the "repeating-history" paradigm)
  - Log ⟨ *T*, abort ⟩ when all effects of *T* have been undone

Read yourself after seeing the examples next

☞ 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>



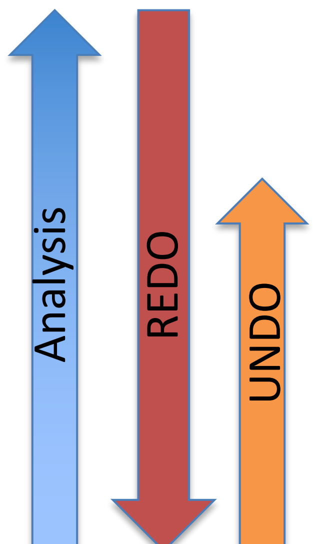
CRASH

- 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 = 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>

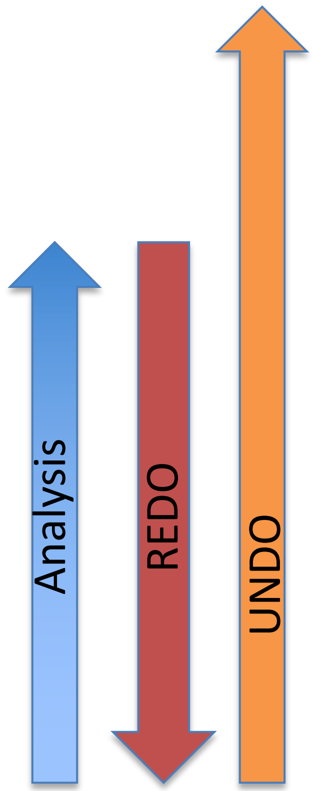


- T1 has committed and writes are already on disk
- After analysis, U = S = {T2}
- REDO all actions
- Write C = 15 (T2)
- UPDATE U to {T2, T3}
- Write D = 20 (T3)
- <COMMIT T2> found: U= {T3}
  - not necessary to set B to 10 (before END CKPT – already on disk)
- UNDO actions of T3 until its start
- Write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



- 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

Developed at IBM, now used in many DBMS, an actual implementation

Optional slide

A Glimpse at ARIES Data Structures

(Details will be covered if we have time)

Dirty page table

pageID	recoveryLSN
P500	101
P600	102
P505	**104**

Transaction table

transID	lastLSN	status
T_{1000}	**104**	**Running**
T_{2000}	103	Running

Log

LSN	prevLSN	tID	pID	Log entry	Type	undoNextLSN
101	-	T1000	P500	Write A “abc” -> “def”	Update	-
102	-	T_{2000}	P600	Write B “hij” -> “klm”	Update	-
103	102	T_{2000}	P500	Write D “mnp” -> “qrs”	Update	-
104	101	T_{1000}	P505	Write C “tuv” -> “wxy”	Update	-

Buffer Pool

P500 PageLSN= 103 A = def D = qrs	**P600** PageLSN= 102 B = klm
P505 **PageLSN= 104**	**P700** PageLSN= - E = pq

Disk

P500 PageLSN= - A = abc D = mnp	**P600** PageLSN= - B = hij
P505 PageLSN= - C = tuv	**P700** PageLSN= - E = pq