Review

- Consider the following sequence of lock requests:

  $l_1(B); l_2(A); l_3(C); l_1(C); l_2(B); l_3(A)$
- Assume that upon start, transactions T_1, T_2, T_3 were assigned timestamps 10, 20, 30, respectively. What is the order in which transactions commit in a wait-die scheme?
- What is the order in which transactions commit in a wound-wait scheme?

Review

$l_1(B); l_2(A); l_3(C); l_1(C); l_2(B); l_3(A)$

- wait-die scheme

  T_3, T_1, T_2

- wound-wait scheme

  T_1, T_2, T_3

Review

- Four transactions T1, T2, T3, T4 used 2PL for concurrency control. Since 2PL ensures conflict-serializability, the schedule (say S) of actions of the four transactions has to be conflict equivalent to some serial schedule. We do not have access to the entire schedule S but only a part of it, which looks as follows:

  S = :::; u_4(A); l_1(B); :::; u_1(B); l3(A); l_2(B); :::; u_3(A); l_2(A); :::

-
- Which of the following schedules are possible serial schedules that are conflict-equivalent to S.

  (a) T_1, T_3, T_2, T_4

  (b) T_1, T_4, T_3, T_2

  (c) T_4, T_1, T_3, T_2 (Correct)

Log-Based Recovery Schemes

If you are going to be in the logging business, one of the things that you have to do is to learn about heavy equipment.

Robert VanNatta,
Logging History of Columbia County

Integrity or consistency constraints

- Predicates data must satisfy, e.g.
  - x is key of relation R
  - $x \rightarrow y$ holds in R
  - Domain(x) = {Red, Blue, Green}
  - no employee should make more than twice the average salary
- Definitions
  - Consistent state: satisfies all constraints
  - Consistent DB: DB in consistent state

Observation:

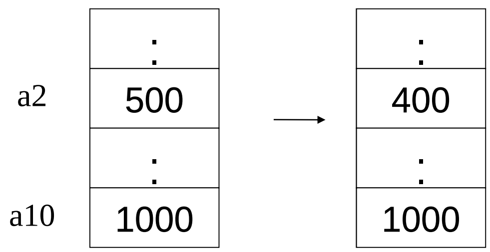
DB cannot always be consistent!

Observation:

DB <u>cannot</u> always be consistent!

<u>Example:</u> Transfer 100 from a2 to a10

a2 ← a2 - 100
a10 ← a10 + 100

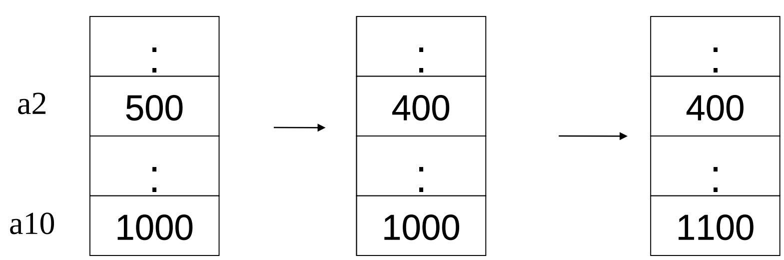


Observation:

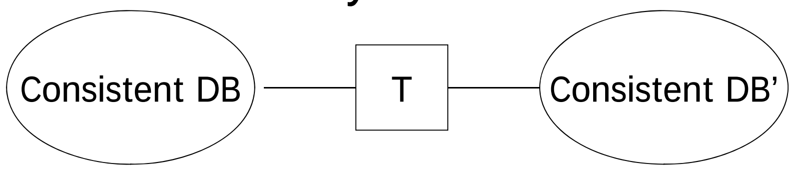
DB <u>cannot</u> always be consistent!

<u>Example:</u> Transfer 100 from a2 to a10

a2 ← a2 - 100
a10 ← a10 + 100



Transaction: collection of actions that preserve consistency



If T starts with consistent state + T executes in isolation (and absence of errors)

⇒ T leaves consistent state

Reasons for crashes

- Transaction failures
  - Logical errors, deadlocks
- System crash
  - Power failures, operating system bugs etc
- Disk failure
  - Head crashes
    - ***STABLE STORAGE*****: *Data never lost. Can approximate by using RAID and maintaining geographically distant copies of the data***

Review: The ACID properties

- **A**tomicity – All actions in a transaction are carried out, or none are, i.e., no incomplete transactions
- **C**onsistency – Each transaction preserves DB consistency
  - User's responsibility, e.g., Funds transfer between bank accounts
- **I**solation – A transaction isolated or protected from the effects of other transactions
- **D**urability – When a transaction commits, its effects persist

Review: The ACID properties

- **A**tomicity – All actions in a transaction are carried out, or none are, i.e., no incomplete transactions
- **C**onsistency – Each transaction preserves DB consistency
  - User's responsibility, e.g., Funds transfer between bank accounts
- **I**solation – A transaction isolated or protected from the effects of other transactions
- **D**urability – When a transaction commits, its effects persist

- **Question: which ones do the Recovery Manager help with?**

Atomicity & Durability

Actions of Transaction:

- Read
- Write
- Commit
- Abort

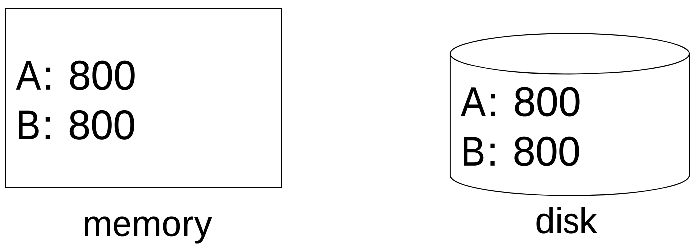
Key problem: Unfinished transaction

Example Transfer fund from A to B

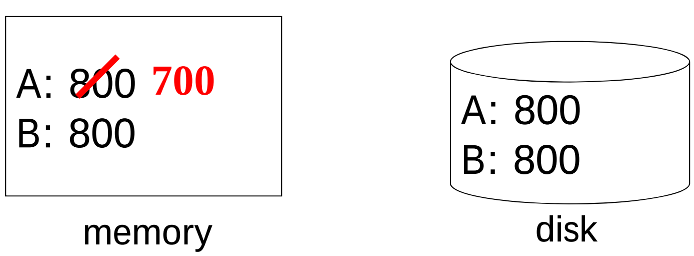
T_1: $A \leftarrow A - 100$

$B \leftarrow B + 100$

T1: Read (A);
A ← A-100
Write (A);
Read (B);
B ← B+100
Write (B);

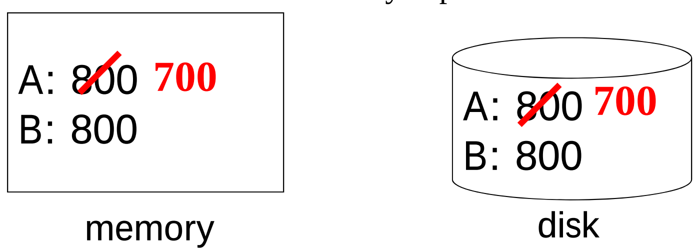


T1: Read (A);
A ← A-100
Write (A);
Read (B);
B ← B+100
Write (B);



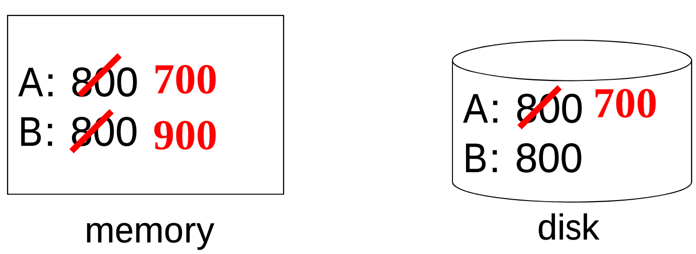
T1: Read (A);
A ← A-100
Write (A);
Read (B);
B ← B+100
Write (B);

Updated A value is written to disk.
This may be triggered "ANYTIME"
by explicit command or DBMS or OS.



CS5208 – Crash Recovery 17

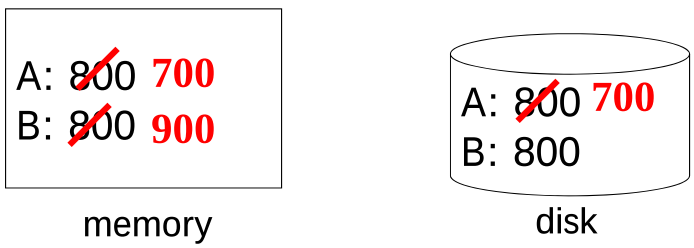
T1: Read (A);
A ← A-100
Write (A);
Read (B);
B ← B+100
Write (B);



CS5208 – Crash Recovery 18

T1: Read (A);
A ← A-100
Write (A);
Read (B);
B ← B+100
Write (B);

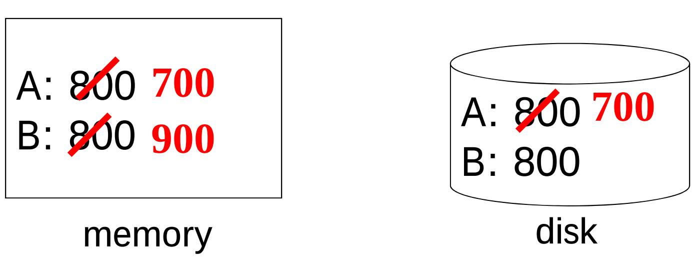
failure!



T1: Read (A);
A ← A-100
Write (A);
Read (B);
B ← B+100
Write (B);

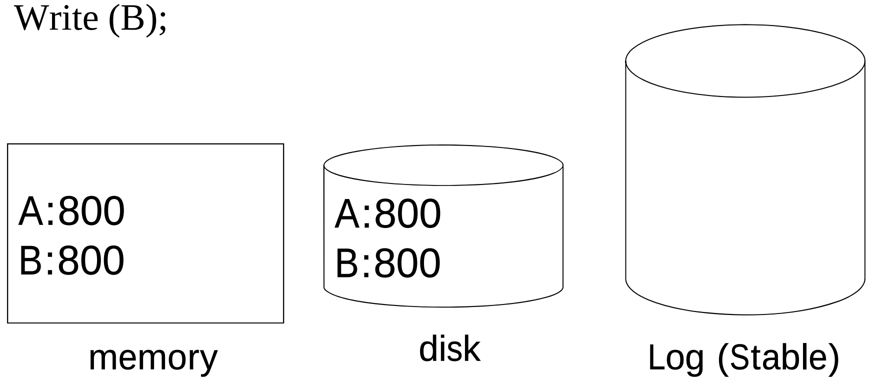
Need <u>atomicity:</u> execute all actions of a transaction or none at all

failure!



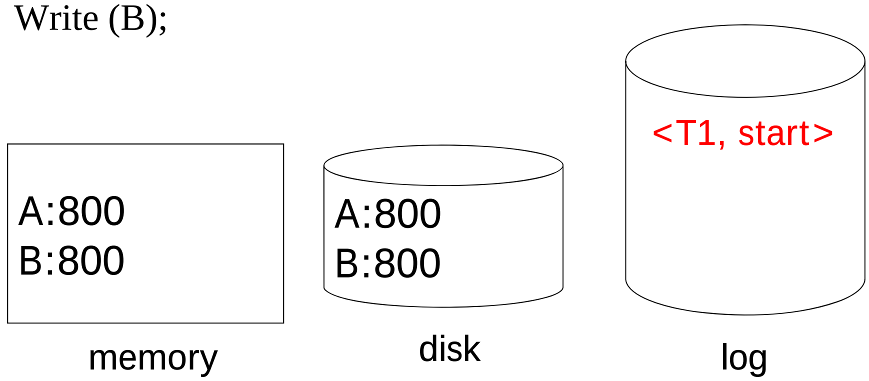
One Solution: Undo logging (Immediate modification)

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



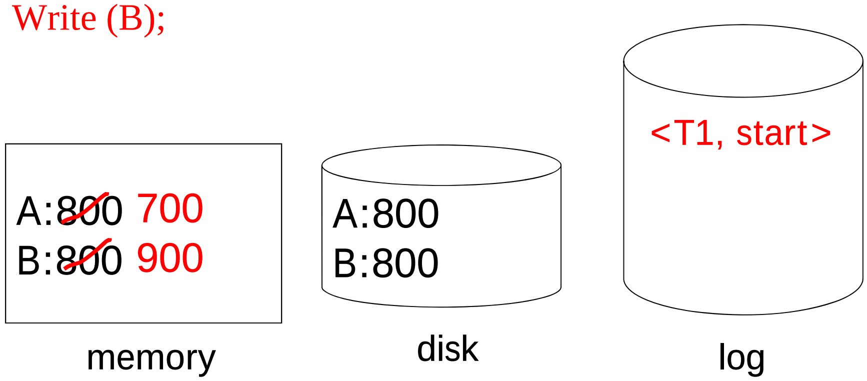
One Solution: Undo logging (Immediate modification)

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



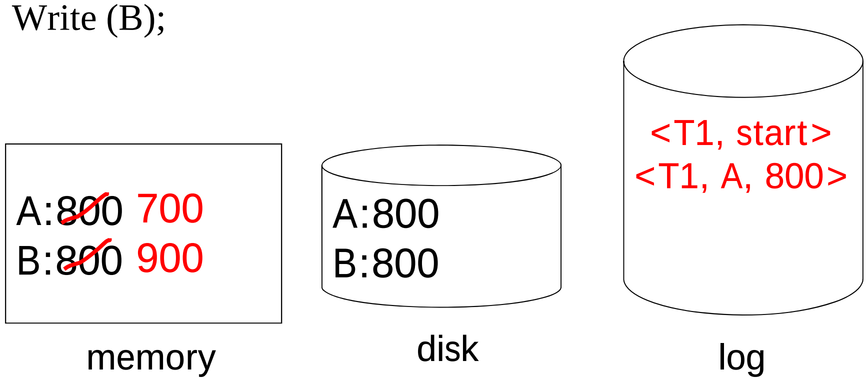
One Solution: Undo logging (Immediate modification)

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



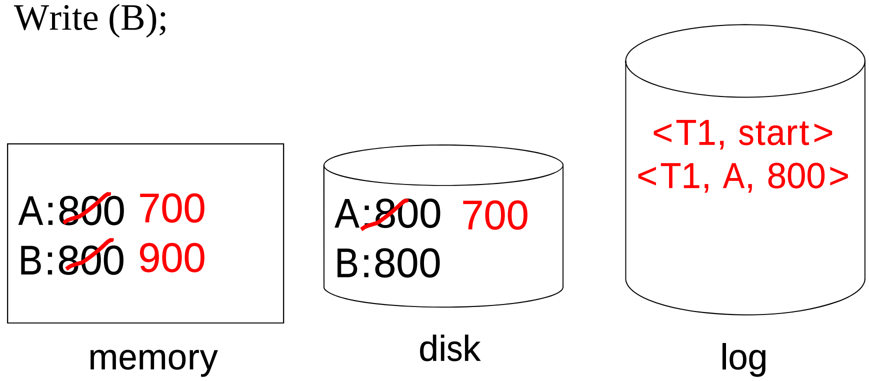
One Solution: Undo logging (Immediate modification)

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



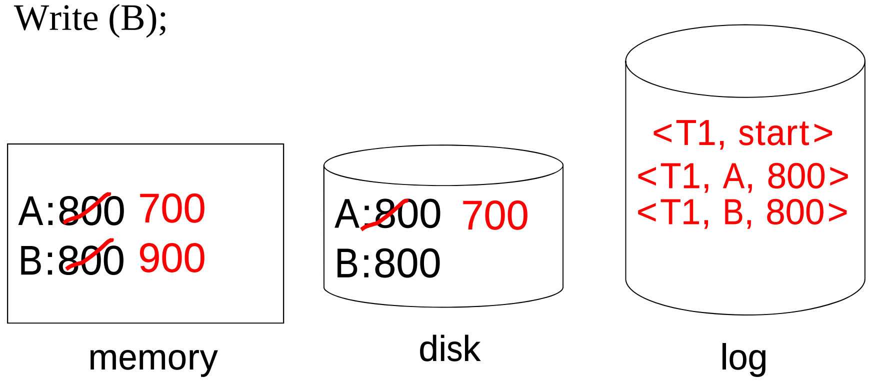
One Solution: Undo logging (Immediate modification)

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



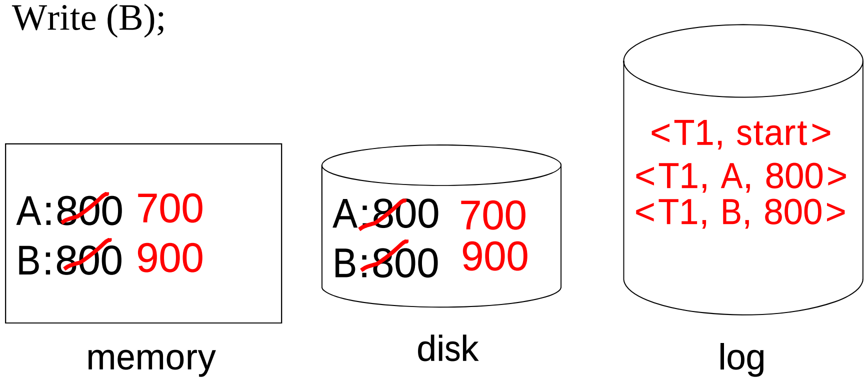
One Solution: Undo logging (Immediate modification)

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



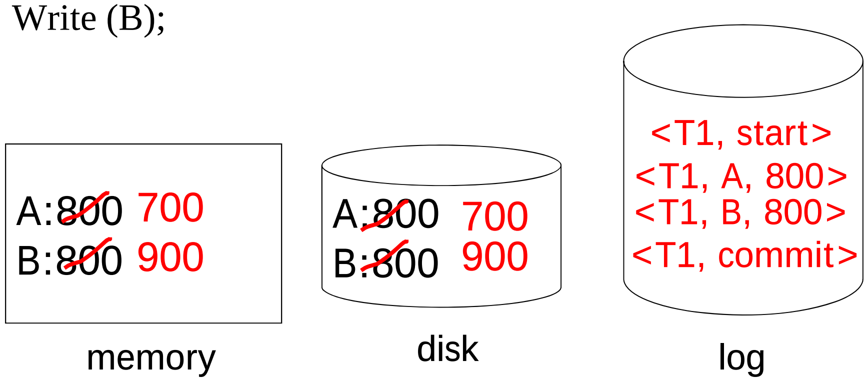
One Solution: Undo logging (Immediate modification)

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



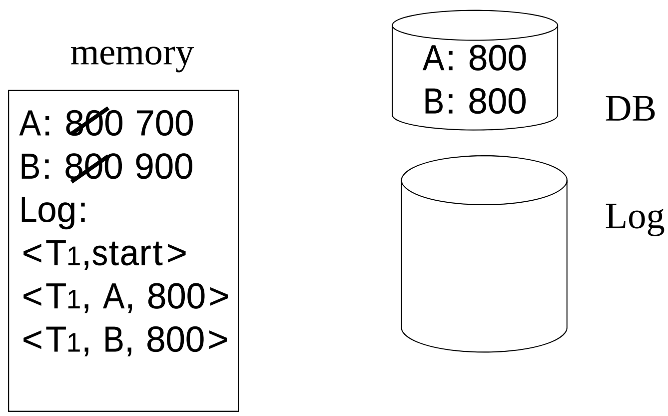
One Solution: Undo logging (Immediate modification)

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



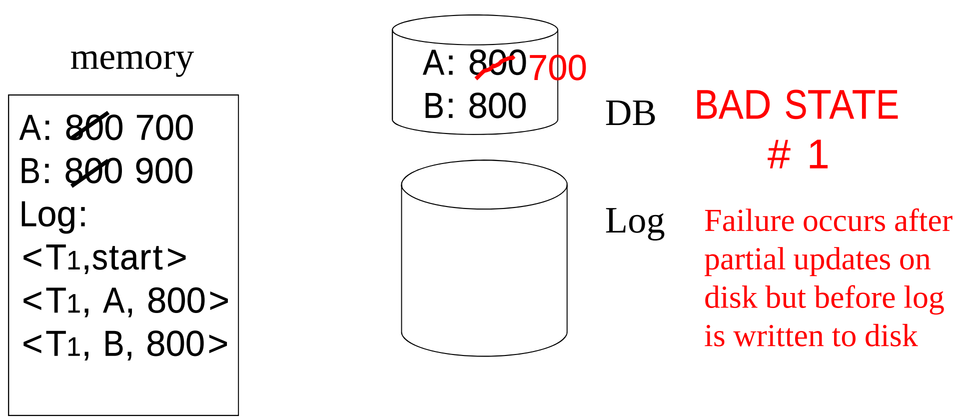
Complication

- Log is first written in memory



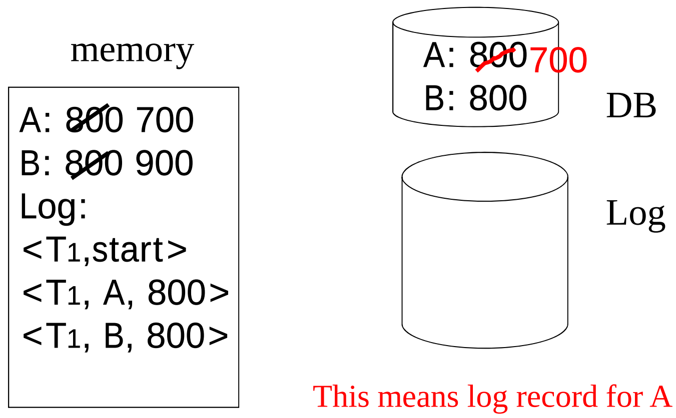
Complication

- Log is first written in memory



Complication

- Log is first written in memory



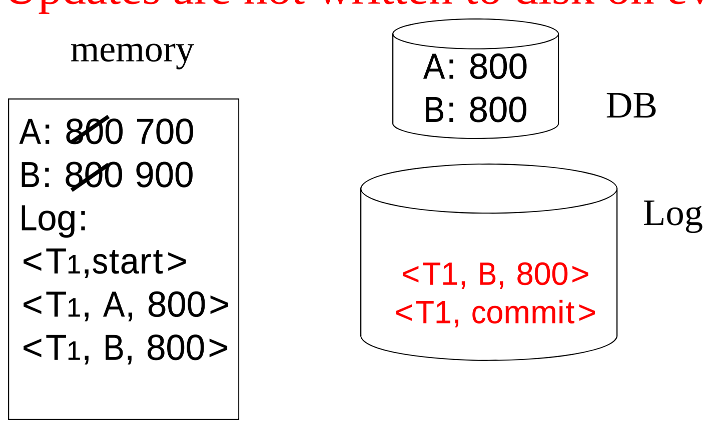
BAD STATE
1

Failure occurs after partial updates on disk but before log is written to disk

This means log record for A must be on log disk before A can be updated on data disk (DB)

Complication

- Log is first written in memory
- Updates are not written to disk on every action

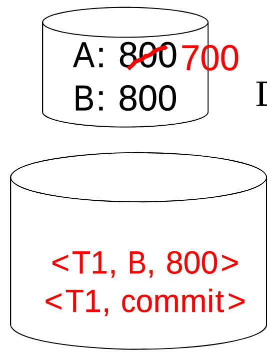


Complication

- Log is first written in memory
- Updates are not written to disk on every action

memory

A: 800 700 (800 crossed out)
B: 800 900 (800 crossed out)
Log:
<T1,start>
<T1, A, 800>
<T1, B, 800>
<T1,commit>



DB

Log

BAD STATE # 2

All logs are on disk (including commit log) but only partial updates on disk.

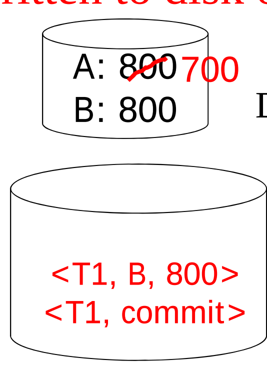
CS5208 – Crash Recovery 33

Complication

- Log is first written in memory
- Updates are not written to disk on every action

memory

A: 800 700 (800 crossed out)
B: 800 900 (800 crossed out)
Log:
<T1,start>
<T1, A, 800>
<T1, B, 800>



DB

Log

BAD STATE # 2

All logs are on disk (including commit log) but only partial updates on disk.

Before commit log is written to Log, all updates must be on disk (DB)

CS5208 – Crash Recovery

Undo logging rules

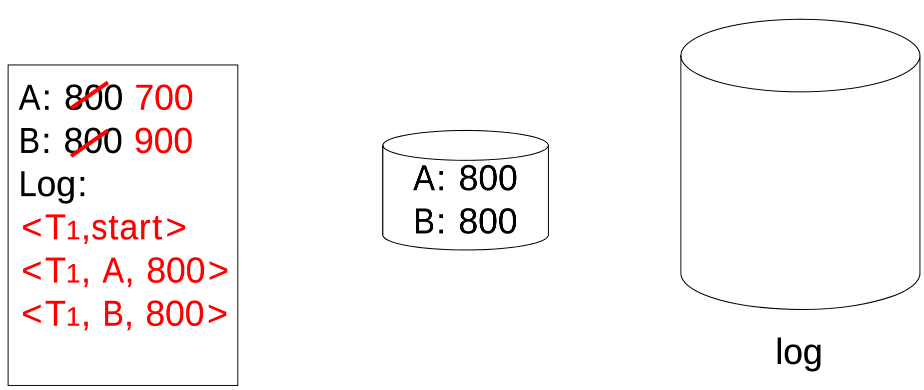
(1) For every action generate undo log record (containing old value)

(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

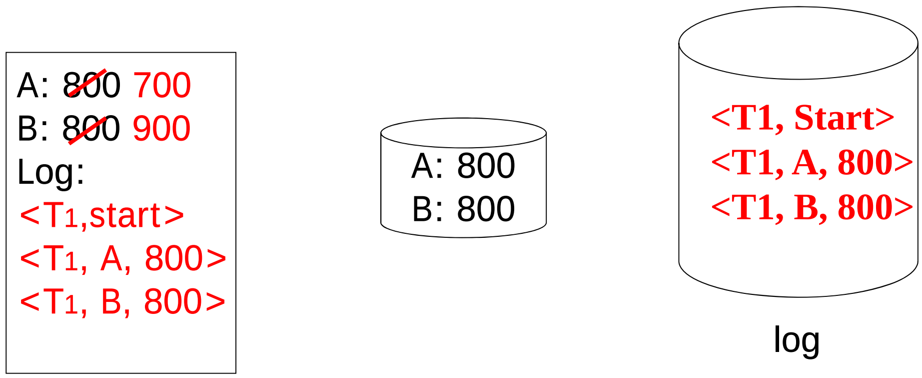
Undo Logging

T1: Read (A); $A \leftarrow A\text{-}100$
Write (A);
Read (B); $B \leftarrow B\text{+}100$
Write (B);



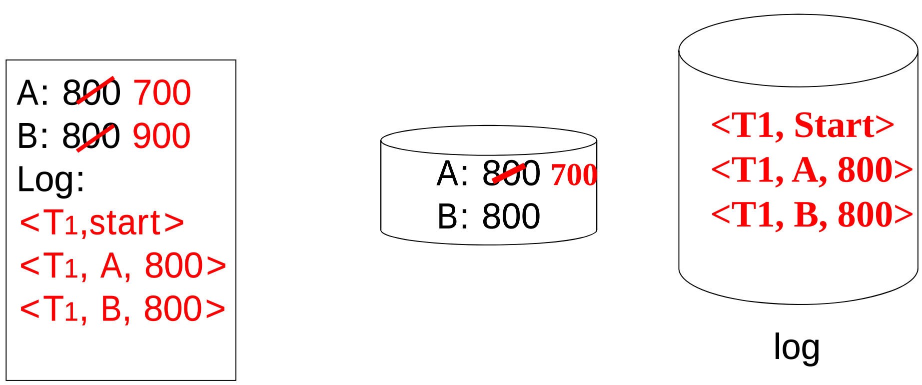
Undo Logging

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



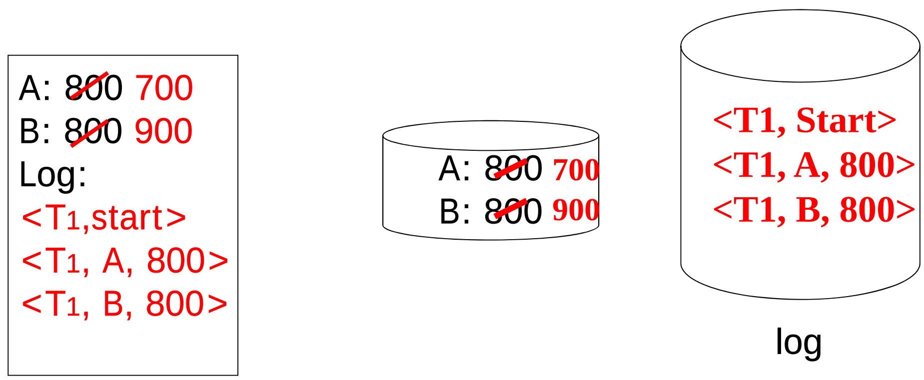
Undo Logging

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



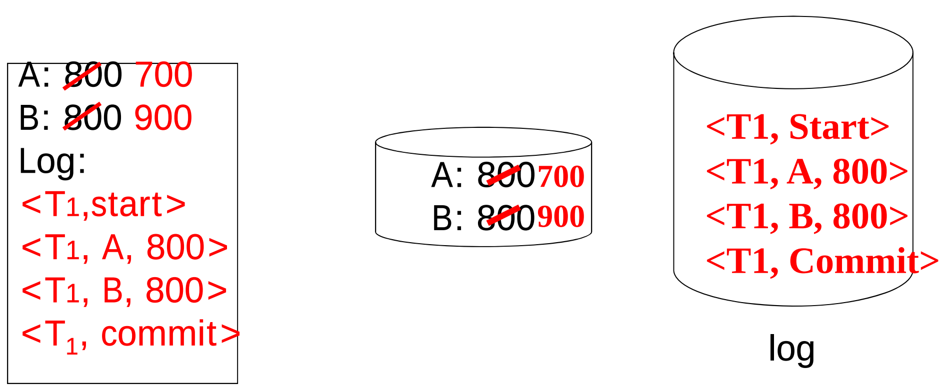
Undo Logging

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



Undo Logging

T1: Read (A); A ← A-100
Write (A);
Read (B); B ← B+100
Write (B);



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
  - X ← v
  - Update disk

(3) For each Ti ∈ S do

- write <Ti, abort> to log

What if failure during recovery?

No problem! Undo is idempotent

Redo logging (deferred modification)

- In UNDO logging, we remember the "old" values.
- How about remembering the "new" (updated) values instead?
- What does this mean?

Redo logging (deferred modification)

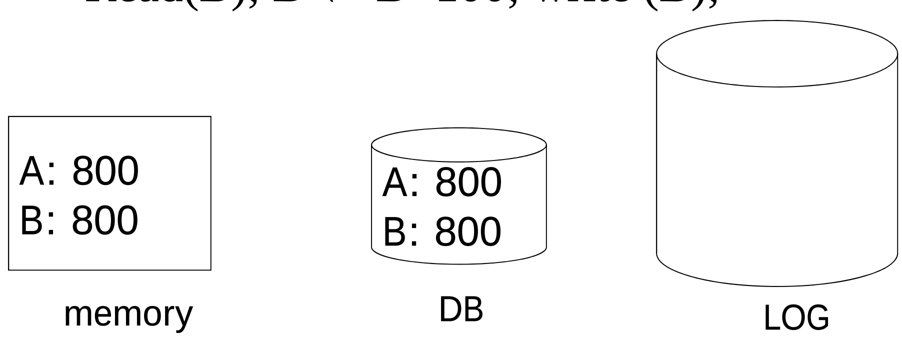
- In UNDO logging, we remember the "old" value.
- How about remembering the "new" (updated) values instead?
- What does this mean?
  - NO updates must be written to disk until a transaction commits! So?

Redo logging (deferred modification)

- In UNDO logging, we remember the “old” value.
- How about remembering the “new” (updated) values instead?
- What does this mean?
  - NO updates must be written to disk until a transaction commits!
  - All updates have to be buffered in memory!

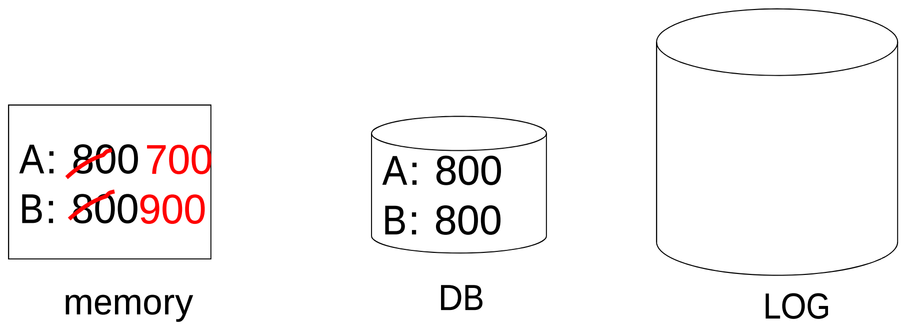
Redo logging (deferred modification)

T1: Read(A); A ← A-100; write (A);
Read(B); B ← B+100; write (B);



Redo logging (deferred modification)

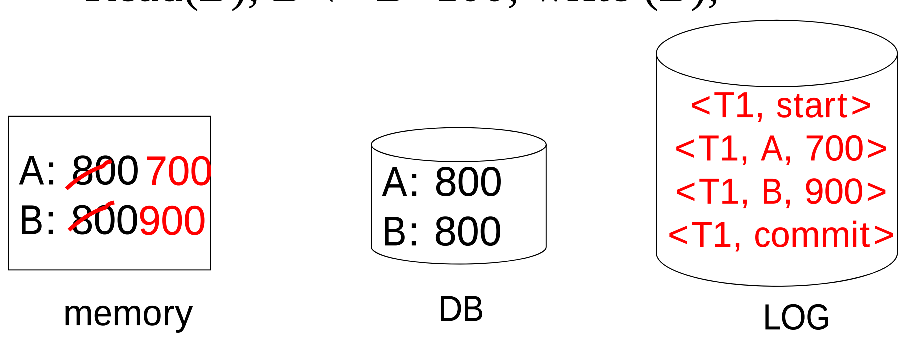
T1: Read(A); A ← A-100; write (A);
Read(B); B ← B+100; write (B);



CS5208 – Crash Recovery 47

Redo logging (deferred modification)

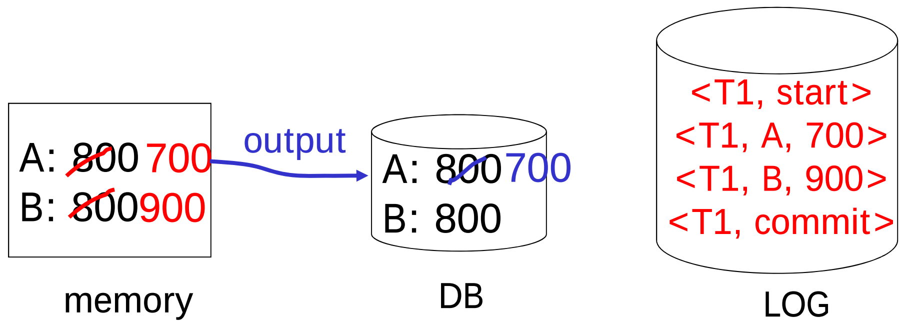
T1: Read(A); A ← A-100; write (A);
Read(B); B ← B+100; write (B);



CS5208 – Crash Recovery 48

Redo logging (deferred modification)

T1: Read(A); A ← A-100; write (A);
Read(B); B ← B+100; write (B);



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

Recovery rules: Redo logging

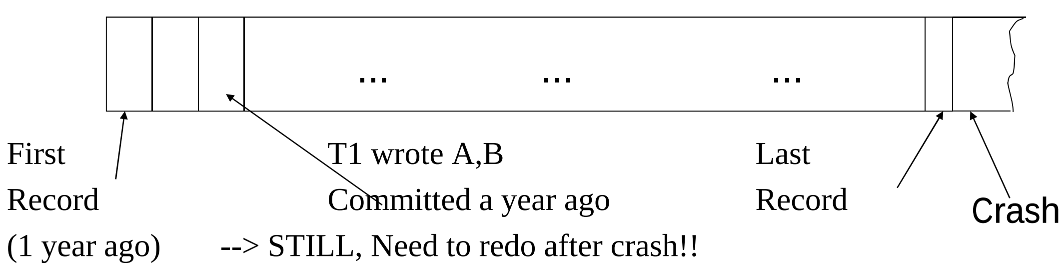
(1) Let S = set of transactions with
<Ti, commit> in log

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

- if Ti $\in$ S then { X $\leftarrow$ v ; Update X on disk }

Recovery is very, very SLOW !

Redo log:



What about UNDO scheme?

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)

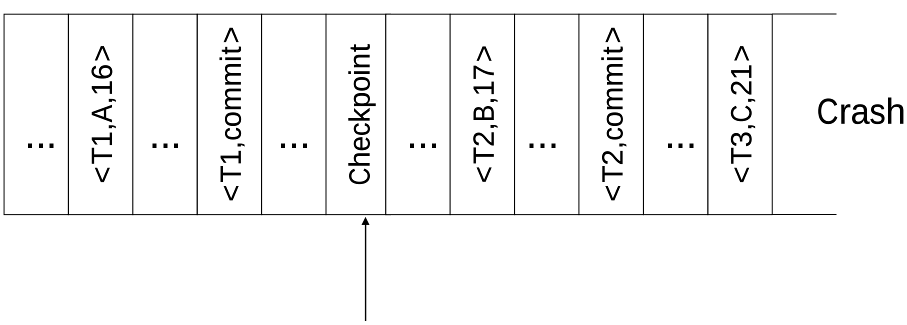
(4) Flush all buffers to disk (DB)

(5) Write "checkpoint" record on disk (log)

(6) Resume transaction processing

Example: what to do at recovery?

Redo log (disk):



No need to examine log records before the most recent Checkpoint

Key drawbacks:

- *Undo logging:* increase the number of disk I/Os
- *Redo logging:* need to keep all modified blocks in memory until commit

Solution: undo/redo logging!

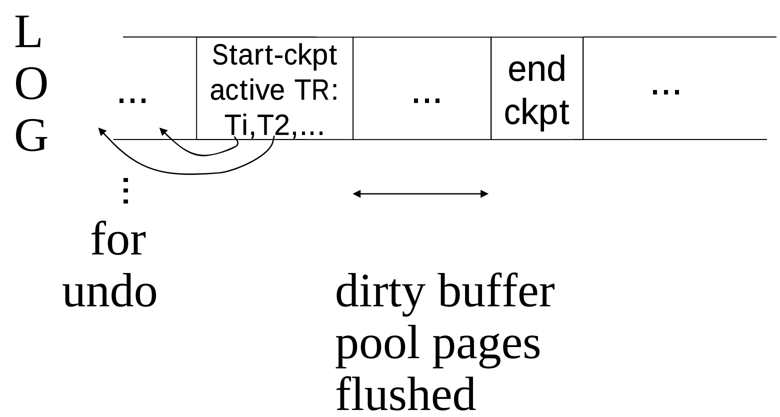
Update $\Rightarrow$ <Ti, Xid, New X val, Old X val>
page X

Rules:

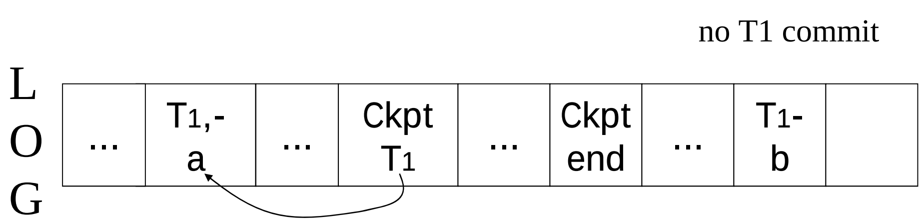
1) Page X can be flushed before or after Ti commit
2) Log record flushed before corresponding updated page (WAL)
3) Flush at commit (log only)

This is adopted in IBM DB2 – known as the Aries Recovery Manager

Non-quiesce checkpoint (for Undo/Redo logging)

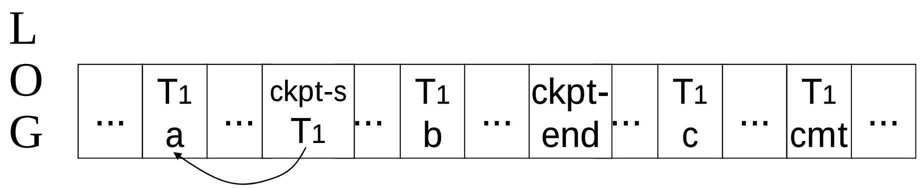


Examples: what to do at recovery time?



- Undo T1 (undo a,b)

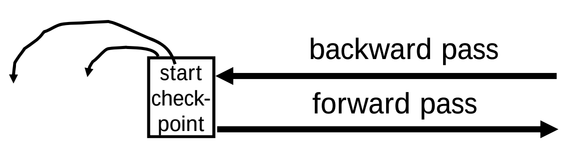
Example



☒ Redo T1: (redo b,c)

Recovery process:

- Backwards pass (end of log ➲ latest checkpoint start)
  - construct set S of committed transactions
  - undo actions of transactions not in S
- Undo transactions that are in checkpoint active list
- Forward pass (latest checkpoint start ➲ end of log)
  - redo actions of S transactions



Undo/Redo Log (Example)

<START S>
<S, A, 60, 61>
<COMMIT S>
<START T>
<T, A, 61, 62>
<START U>
<U, B, 20, 21>
<T, C, 30, 31>
<START V>
<U, D, 40, 41>
<V, F, 70, 71>
<COMMIT U>
<T, E, 50, 51>
<COMMIT T>
<V, B, 21, 22>
<COMMIT V>

Suppose we begin a nonquiescent checkpoint immediately after <T, A, 61, 62>, where could the <END CKPT> record be potentially written in the log?

Undo/Redo Log (Example)

<START S>
<S, A, 60, 61>
<COMMIT S>
<START T>
<T, A, 61, 62>
<START U>
<U, B, 20, 21>
<T, C, 30, 31>
<START V>
<U, D, 40, 41>
<V, F, 70, 71>
<COMMIT U>
<T, E, 50, 51>
<COMMIT T>
<V, B, 21, 22>
<COMMIT V>

Suppose we begin a nonquiescent checkpoint immediately after <T, A, 61, 62>, where could the <END CKPT> record be potentially written in the log?

ANYWHERE after <T, A, 61, 62> since the writing of dirty blocks can be performed independent of whatever actions the transactions are performing in the interim

Undo/Redo Log (Example)

<START S>
<S, A, 60, 61>
<COMMIT S>
<START T>
<T, A, 61, 62>
<START U>
<U, B, 20, 21>
<T, C, 30, 31>
<START V>
<U, D, 40, 41>
<V, F, 70, 71>
<COMMIT U>
<T, E, 50, 51>
<COMMIT T>
<V, B, 21, 22>
<COMMIT V>

Suppose we begin a nonquiescent checkpoint immediately after <T, A, 61, 62>. Suppose further there is a crash at *any* possible point after <T, A, 61, 62>, how far back in the log we must look to find all possible incomplete transactions if

- <END CKPT> was written prior to the crash
- <END CKPT> was not written

CS5208 – Crash Recovery 63

Undo/Redo Log (Example)

<START S>
<S, A, 60, 61>
<COMMIT S>
<START T>
<T, A, 61, 62>
<START U>
<U, B, 20, 21>
<T, C, 30, 31>
<START V>
<U, D, 40, 41>
<V, F, 70, 71>
<COMMIT U>
<T, E, 50, 51>
<COMMIT T>
<V, B, 21, 22>
<COMMIT V>

The only active transaction when the checkpoint began was *T*.

- <END CKPT> was written prior to the crash
  - We need only go back as far as the start of *T*.

Start CKPT

End CKPT

Crash

CS5208 – Crash Recovery 64

Undo/Redo Log (Example)

<START S>
<S, A, 60, 61>
<COMMIT S>
<START T>
<T, A, 61, 62>
<START U>
<U, B, 20, 21>
<T, C, 30, 31>
<START V>
<U, D, 40, 41>
<V, F, 70, 71>
<COMMIT U>
<T, E, 50, 51>
<COMMIT T>
<V, B, 21, 22>
<COMMIT V>

The only active transaction when the checkpoint began was *T*.

- <END CKPT> was written prior to the crash
  - We need only go back as far as the start of *T*.
- <END CKPT> was not written
  - Any transaction that was active when the *previous* checkpoint ended may have written some but not all of its data to disk.
  - So, go to the previous checkpoint
  - In this case, the only other transaction that could qualify is *S*, so we must look back to the beginning of *S*, i.e., to the beginning of the log in this simple example.

Summary

- Consistency of data
- One source of problems: failures
  - Logging
  - Redundancy
- Another source of problems: data sharing
  - Concurrency Control