THE UNIVERSITY of EDINBURGH

Advanced Databases

Spring 2020

Lecture #15:

Crash Recovery

Milos Nikolic

Review: The ACID Properties

Atomicity: All actions in the txn happen, or none happen

Consistency: If each txn is consistent and the DB starts consistent, then it ends up consistent

Isolation: Execution of one txn is isolated from that of other txns

Durability: If a txn commits, its effects persist

The recovery manager ensures atomicity, DB consistency, and durability

Motivation

Atomicity:

Transactions may abort (“rollback”)

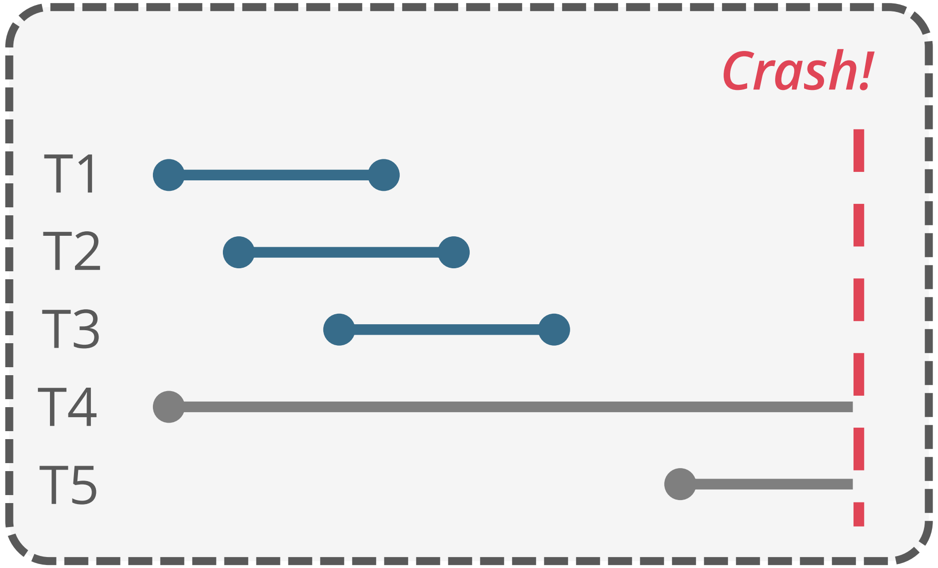
Durability:

What if the DBMS stops running?

Desired behaviour after system restarts:

T1, T2 & T3 should be durable

T4 & T5 should be aborted (effects not seen)



Types of Failures

Logical Errors

Txn cannot complete due to an internal error condition (e.g., integrity constraint violation)

Internal State Errors

DBMS must terminate an active transaction due to an error condition (e.g., deadlock)

Transaction Failures

Software Failures

Problem with the DBMS implementation (e.g., uncaught divide-by-zero exception)

Hardware Failures

The computer hosting the DBMS crashes (e.g., power plug gets pulled)

Fail-stop assumption: Non-volatile storage contents are not corrupted by system crash

System Failures

Non-Repairable Hardware Failure

A head crash or similar disk failure destroys all or part of non-volatile storage

Destruction is assumed to be detectable (e.g., disk controller use checksums to detect failures)

No DBMS can recover from this! Database must be restored from an archived version (replica).

Storage Media Failures

Crash Recovery

Recovery algorithms are techniques to ensure database consistency, transaction atomicity, and durability despite failures

Recovery algorithms have two parts:

- Actions during normal txn processing to ensure that the DBMS can recover from a failure
- Actions after a failure to recover the database to a state that ensures atomicity, consistency, and durability

Observation

The primary storage location of the database is on non-volatile storage, but this is much slower than volatile storage

Use volatile memory for faster access:

- Bring pages into memory, perform writes in memory, write dirty pages back to disk

The DBMS needs to guarantee that:

- The changes of any txn are durable once the DBMS has confirmed that it committed
- No partial changes are durable if the txn aborted

How the DBMS supports this depends on how it manages the buffer pool…

Handling the Buffer Pool

Steal Policy

Whether the DBMS allows an uncommitted txn to overwrite the most recent committed value of an object in non-volatile storage

STEAL: Is allowed

NO-STEAL: Is **not** allowed

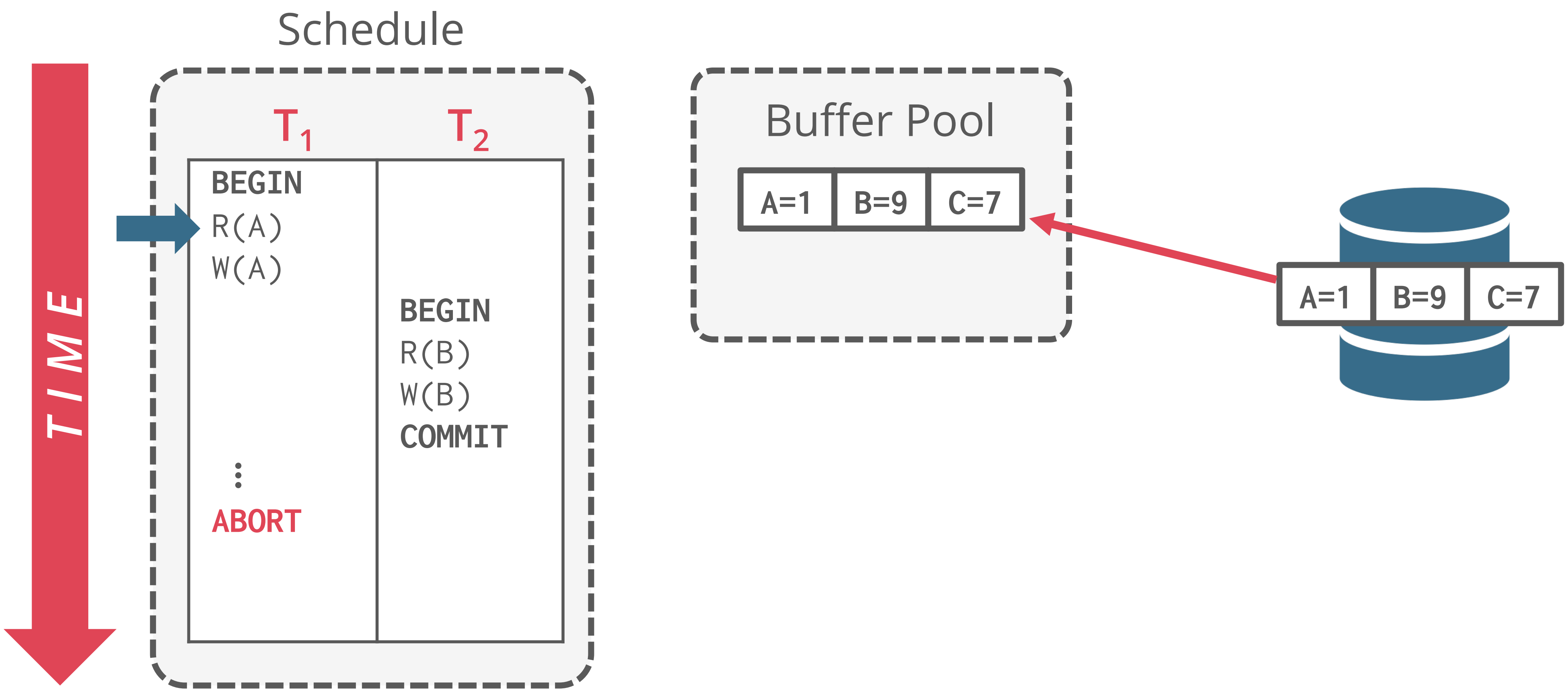
Force Policy

Whether the DBMS requires that all updates made by a txn are reflected on non-volatile storage **before** the txn is allowed to commit

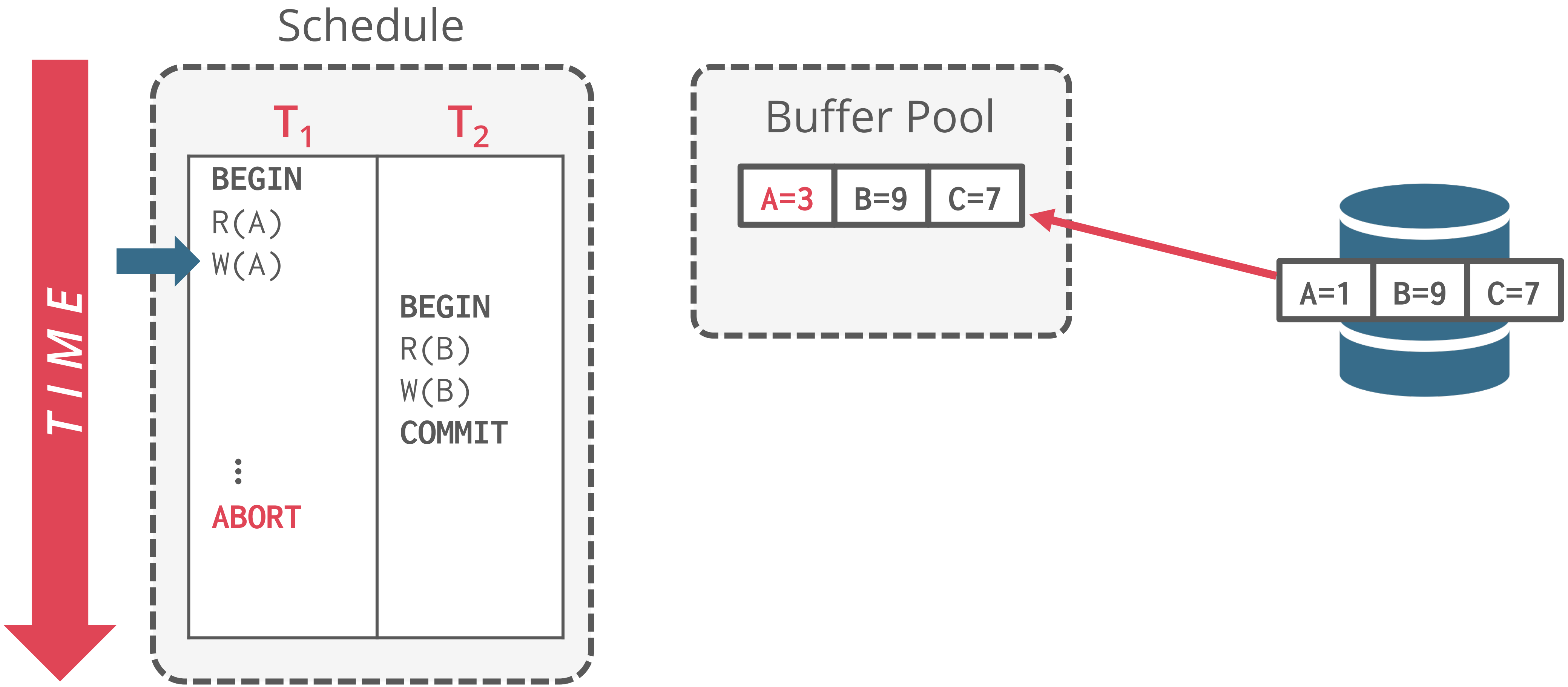
FORCE: Is enforced

NO-FORCE: Is **not** enforced

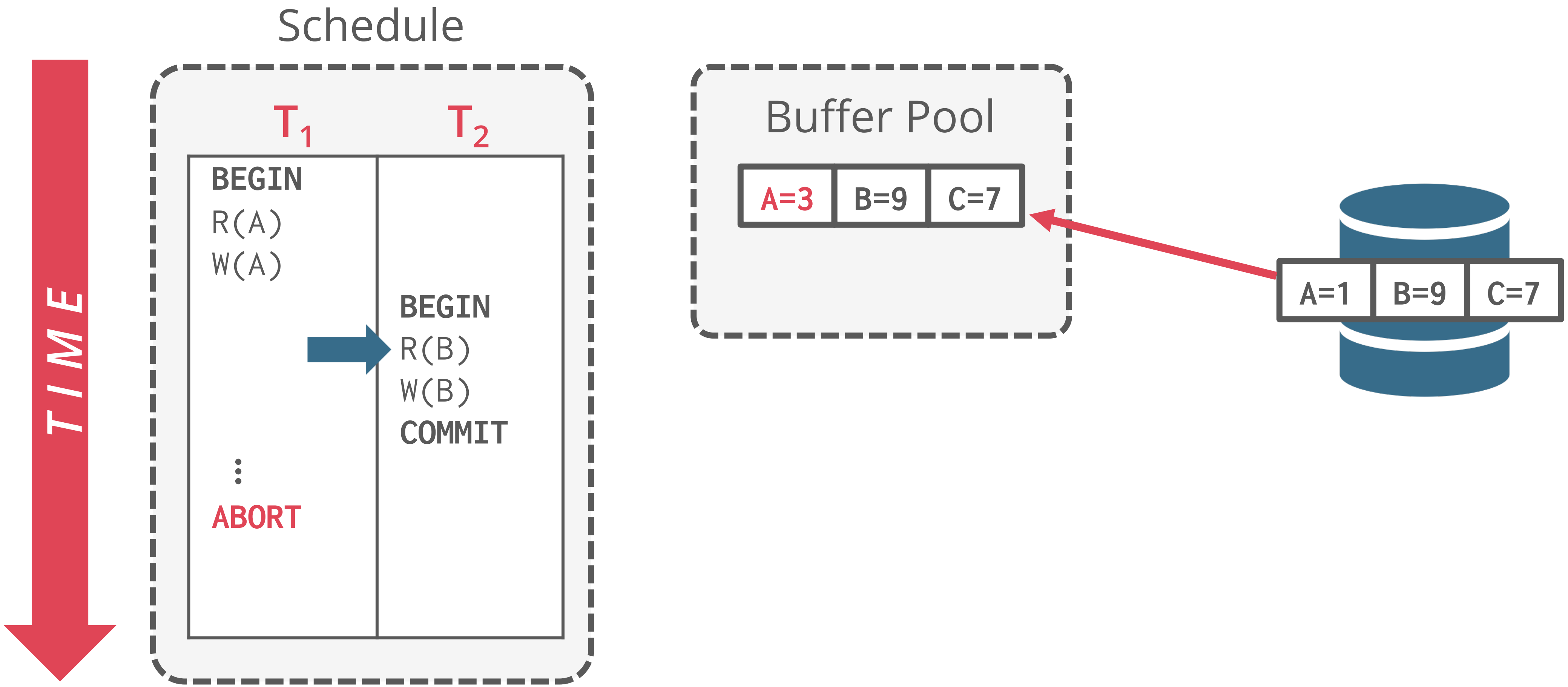
NO-STEAL + FORCE



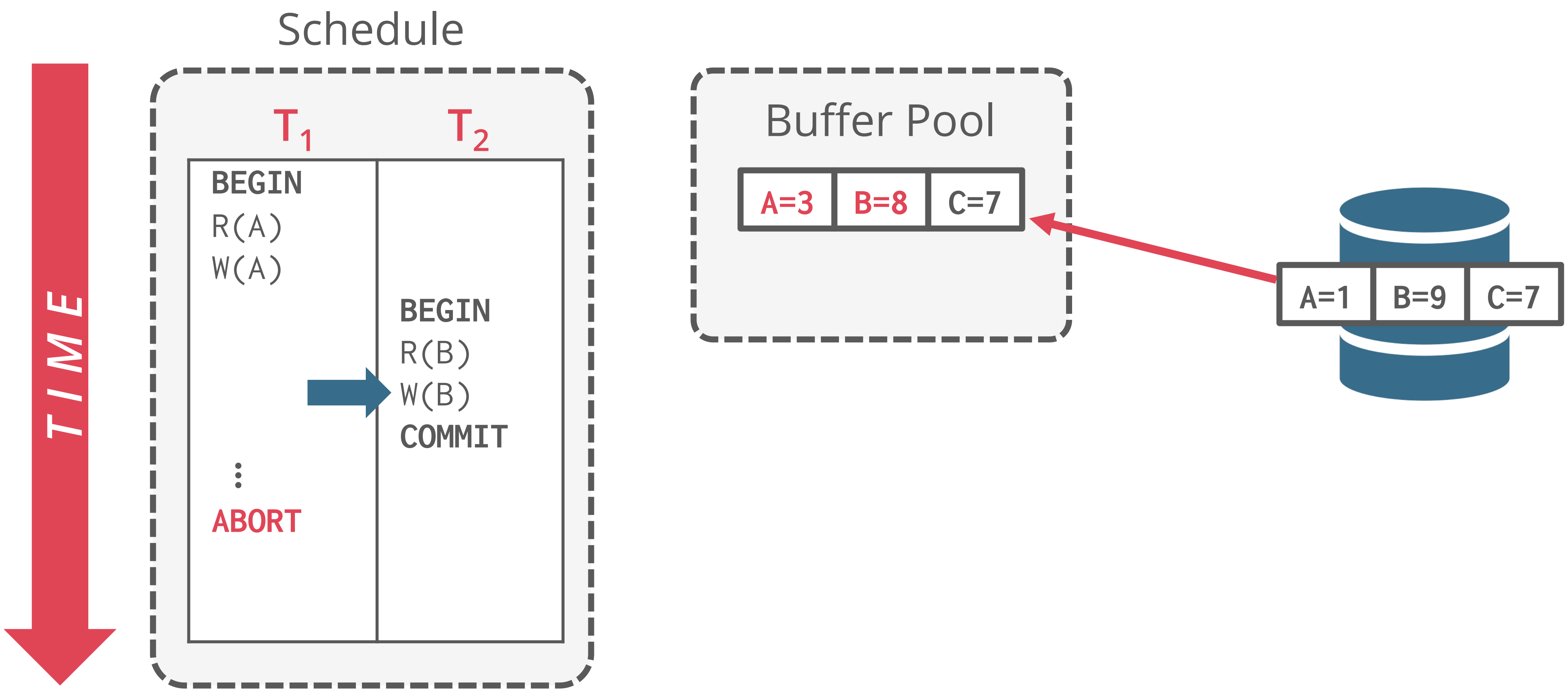
No-Steal + Force



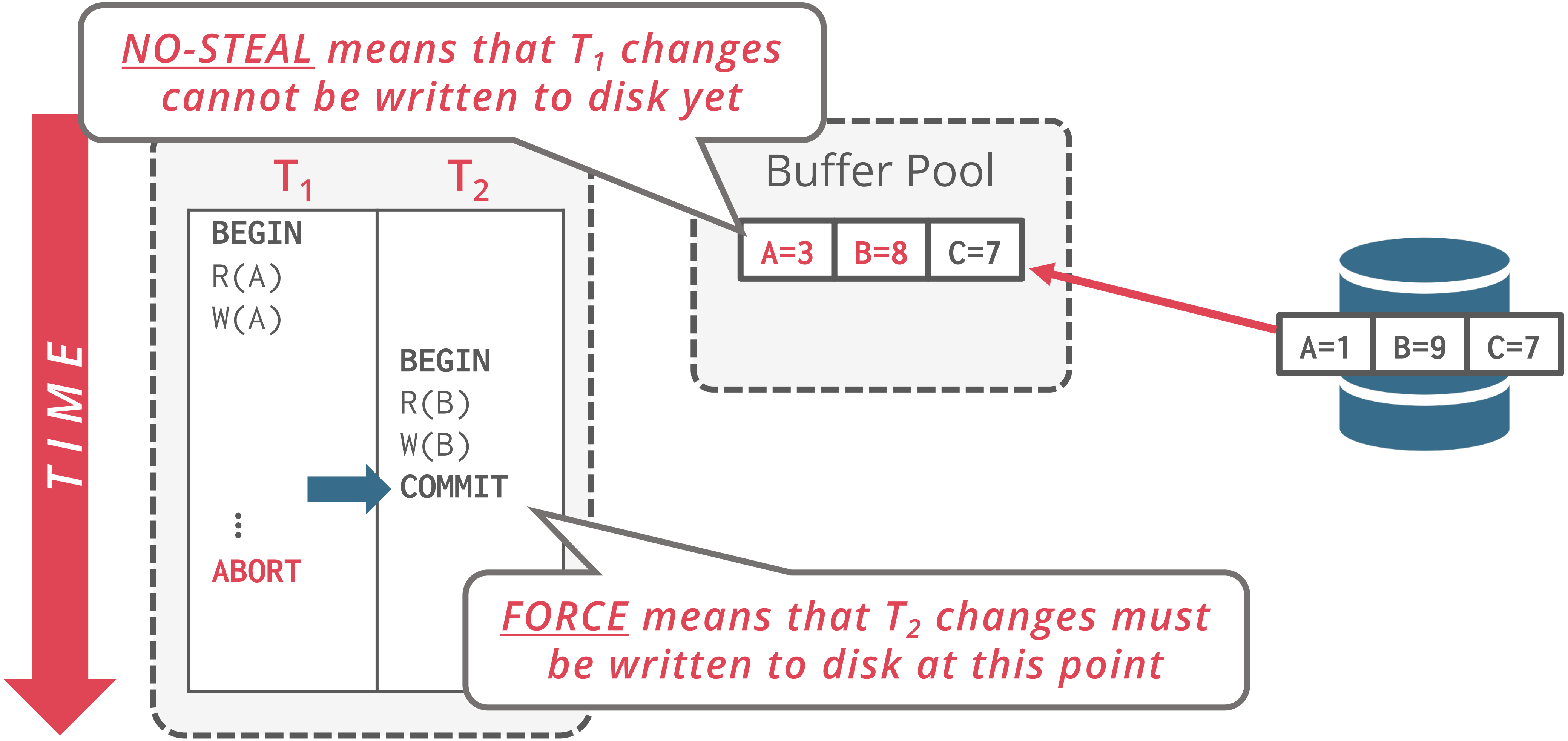
No-Steal + Force



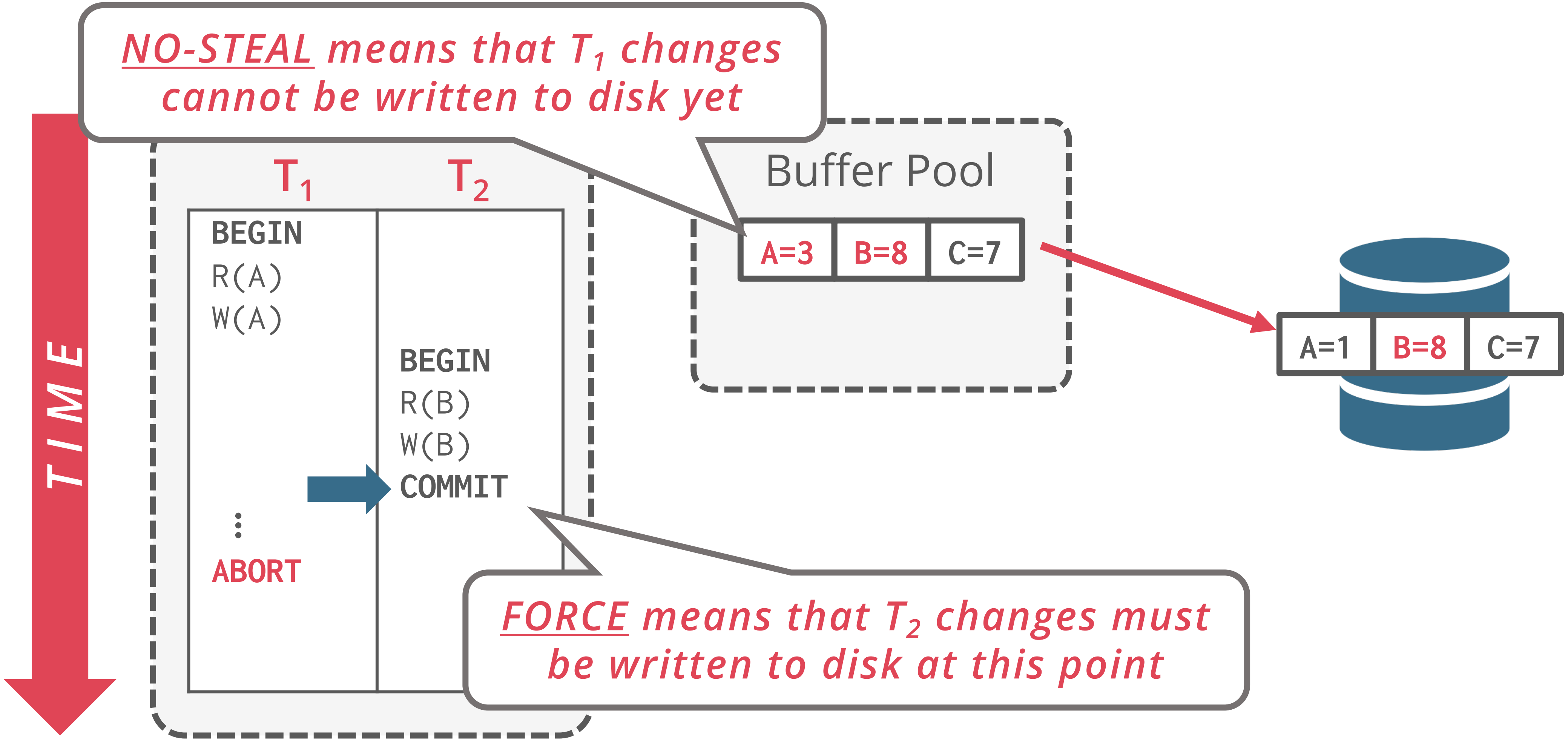
No-Steal + Force



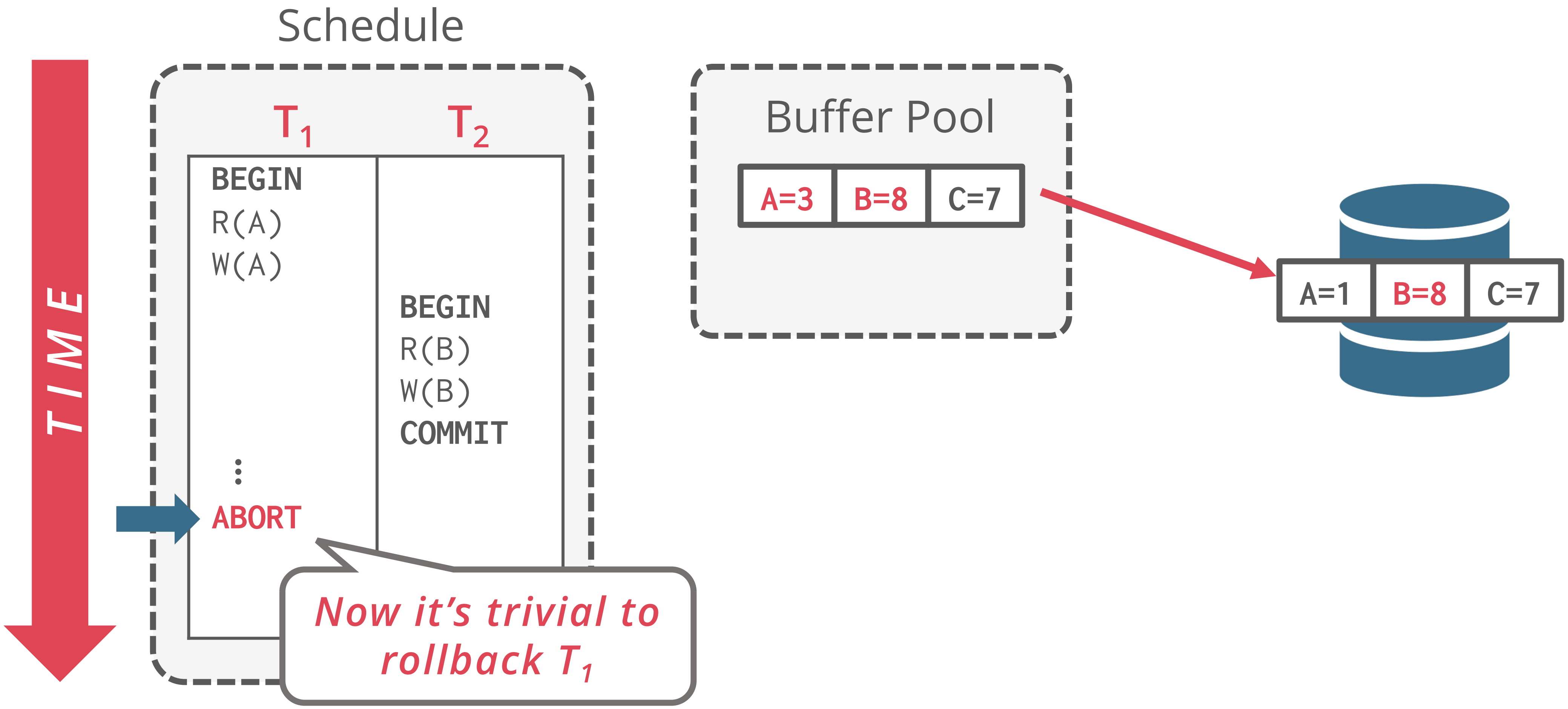
NO-STEAL + FORCE



NO-STEAL + FORCE



No-Steal + Force



No-Steal + Force

This approach is the **easiest to implement**

- **Never have to undo** changes of an aborted txn because the changes were not written to disk
- **Never have to redo** changes of a committed txn because all the changes are guaranteed to be written to disk at commit time

But has **important drawbacks**

- **Poor performance**: flushing non-contiguous pages (random writes) is slow
- **Memory requirements**: NO-STEAL assumes that all pages modified by uncommitted transactions can be accommodated in the buffer pool

More on Steal and Force

STEAL: Why enforcing atomicity is hard?

- **Stealing frame F**: Current page ***P*** in ***F*** is written to disk; some txn holds lock on ***P***
  - What if the txn with the lock on ***P*** aborts?
  - Must remember the old value of ***P*** at steal time to support **UNDO**ing the write to ***P***

NO-FORCE: Why enforcing durability is hard?

- What if the DBMS crashes before a modified page is written to disk?
- Write as little as possible, in a convenient place, at commit time, to support **REDO**ing modifications

BUFFER POOL POLICIES

Runtime Performance

	NO-STEAL	STEAL
NO-FORCE	–	Fastest
FORCE	Slowest	–

Recovery Performance

Undo + Redo

	NO-STEAL	STEAL
NO-FORCE	–	Slowest
FORCE	Fastest	–

No Undo + No Redo

Undo: removing the effects of an incomplete or aborted txn

Redo: re-instating the effects of a committed txn for durability

Almost every DBMS uses STEAL + NO-FORCE

Basic Idea: Logging

Record UNDO and REDO information, for every update, in a log file

- Assume that the log is on stable storage
- Log file is separated from actual data
- Sequential writes to the log
- Minimal info (diff) written to the log, so multiple updates fit in a single log page

Log contains sufficient information to perform the necessary undo and redo actions to restore the database after a crash

WRITE-AHEAD LOGGING (WAL)

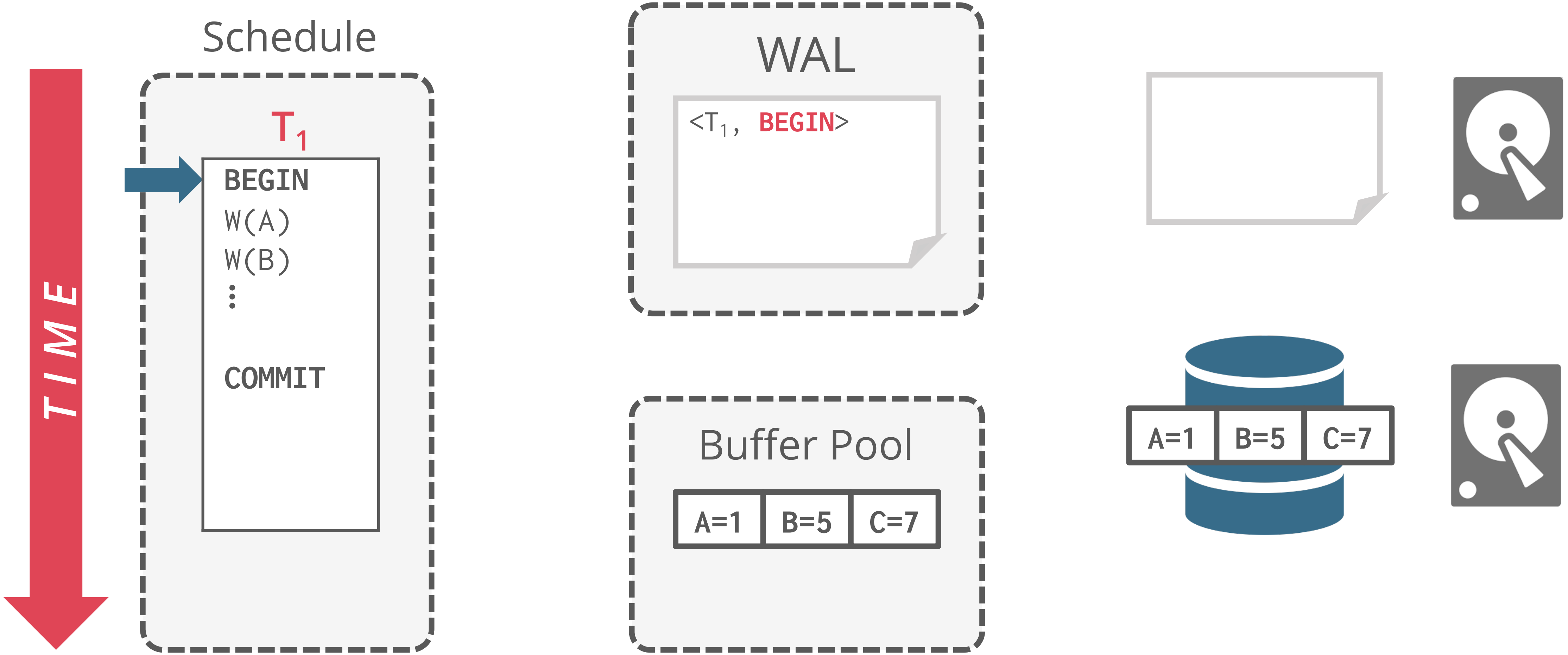
Record the changes made to the database in a log file **before** the change is made

The DBMS stages all of a txn's log records in volatile storage (usually backed by buffer pool)

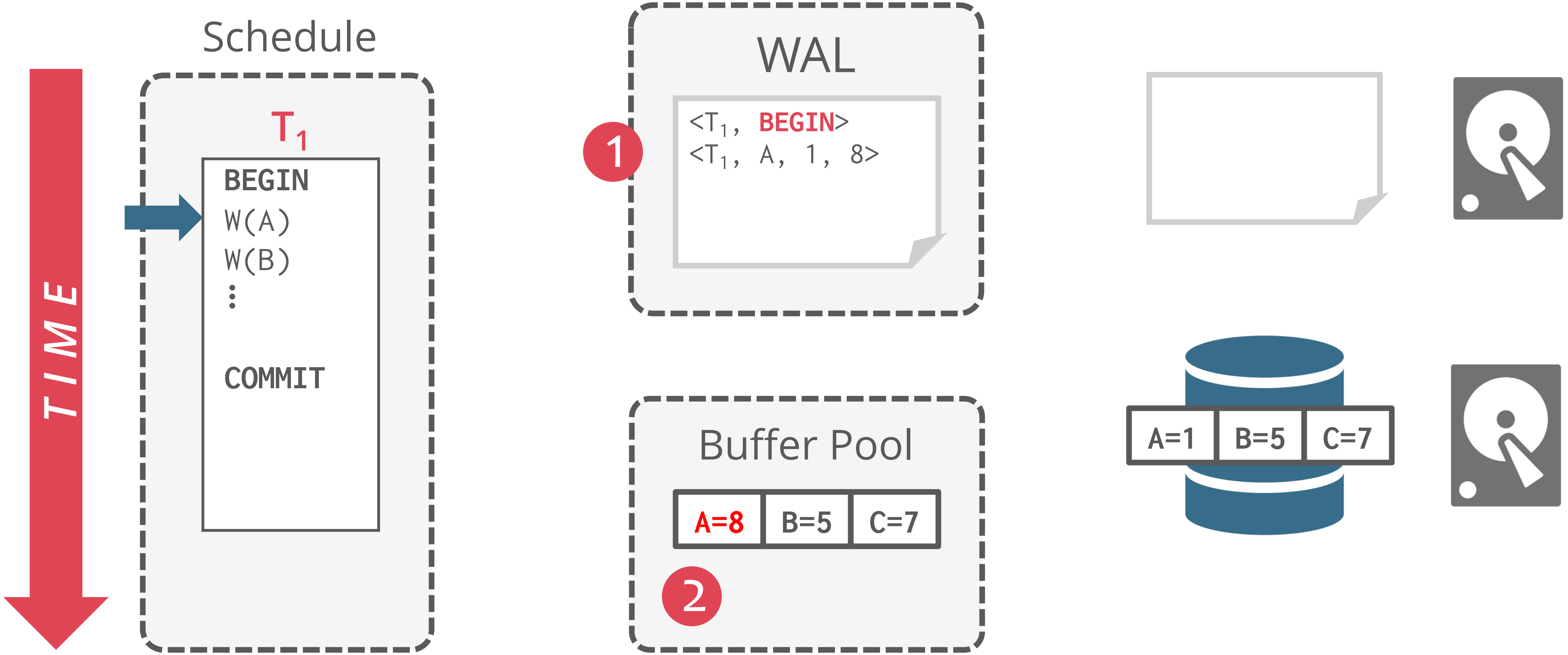
All log records pertaining to an updated page must be written to non-volatile storage **before** the page itself is overwritten in non-volatile storage

A txn is not considered committed until **all** its log records have been written to non-volatile storage

WAL – Example

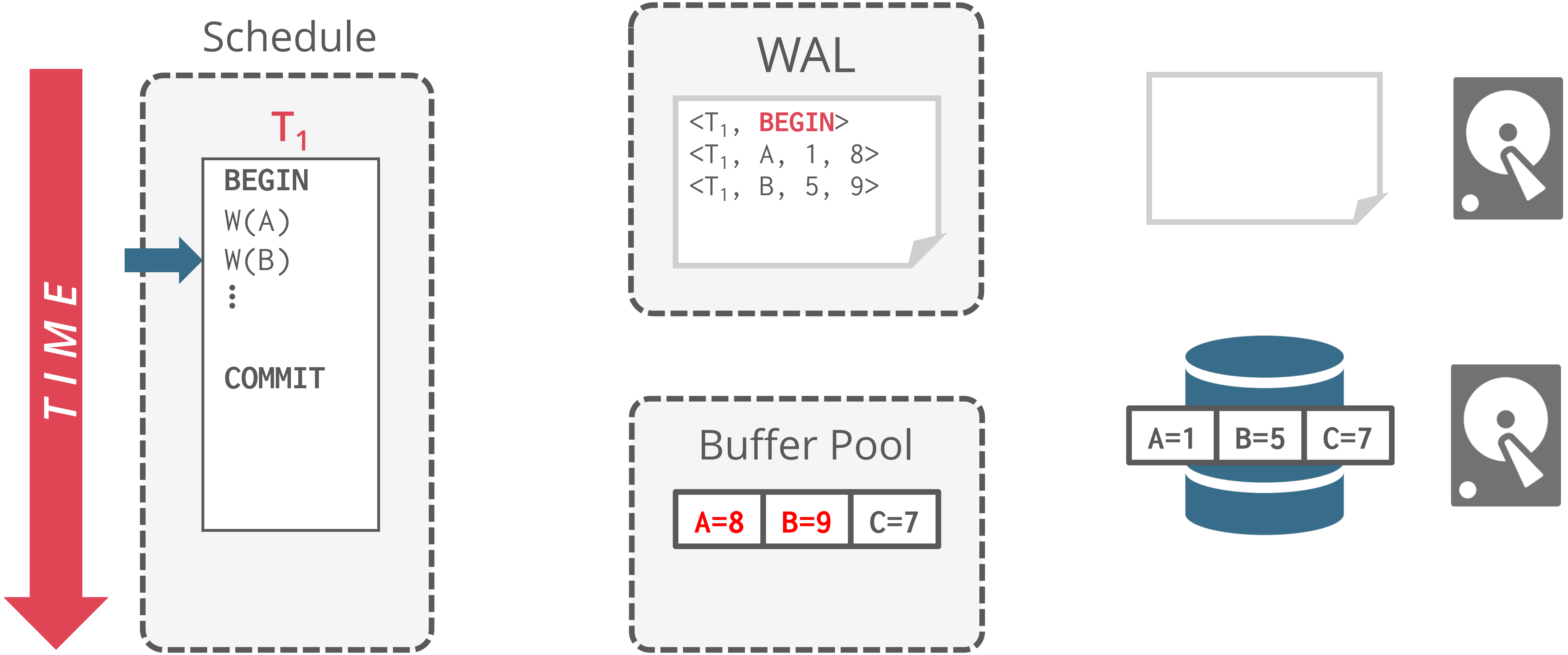

TIME
Schedule
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL
<T_1, BEGIN>
Buffer Pool
A=1
B=5
C=7
A=1
B=5
C=7

WAL – EXAMPLE

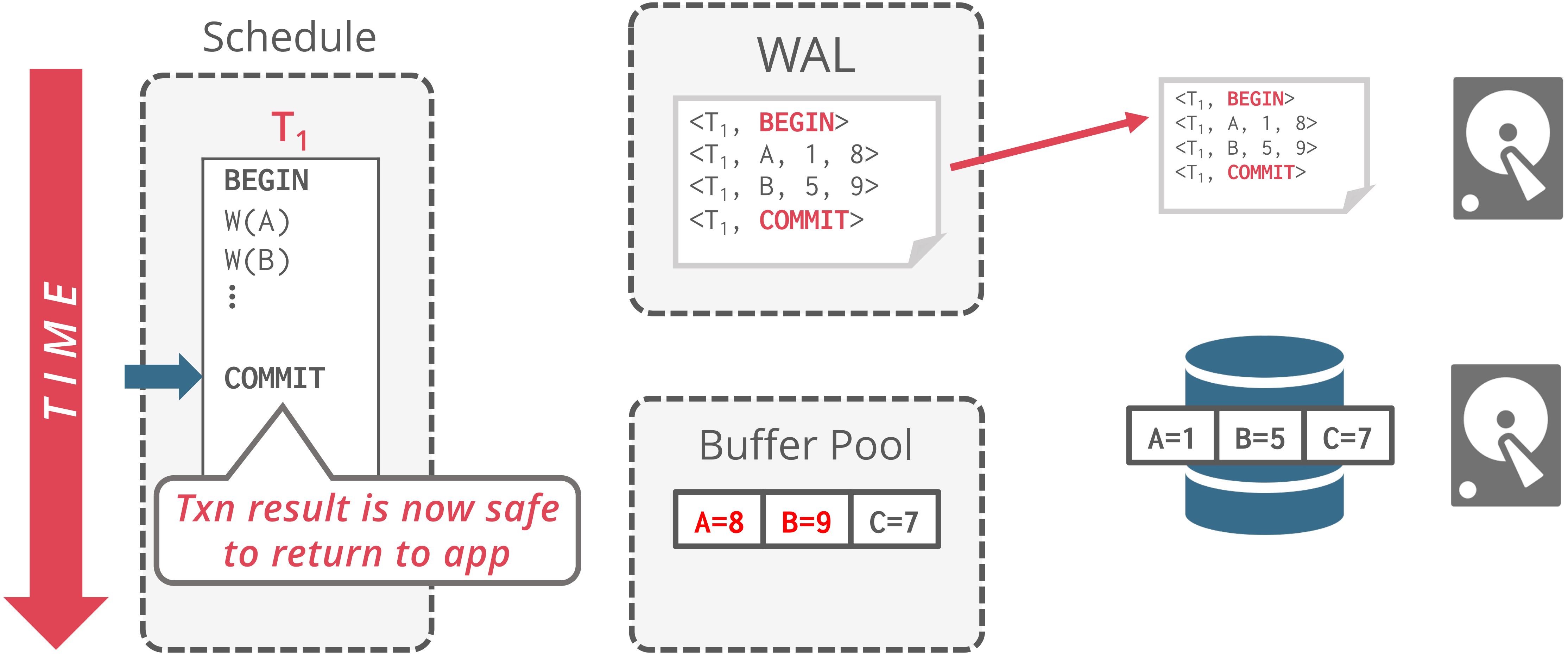


TIME
Schedule
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL
1
<T_1, BEGIN>
<T_1, A, 1, 8>
Buffer Pool
A=8
B=5
C=7
2
A=1
B=5
C=7

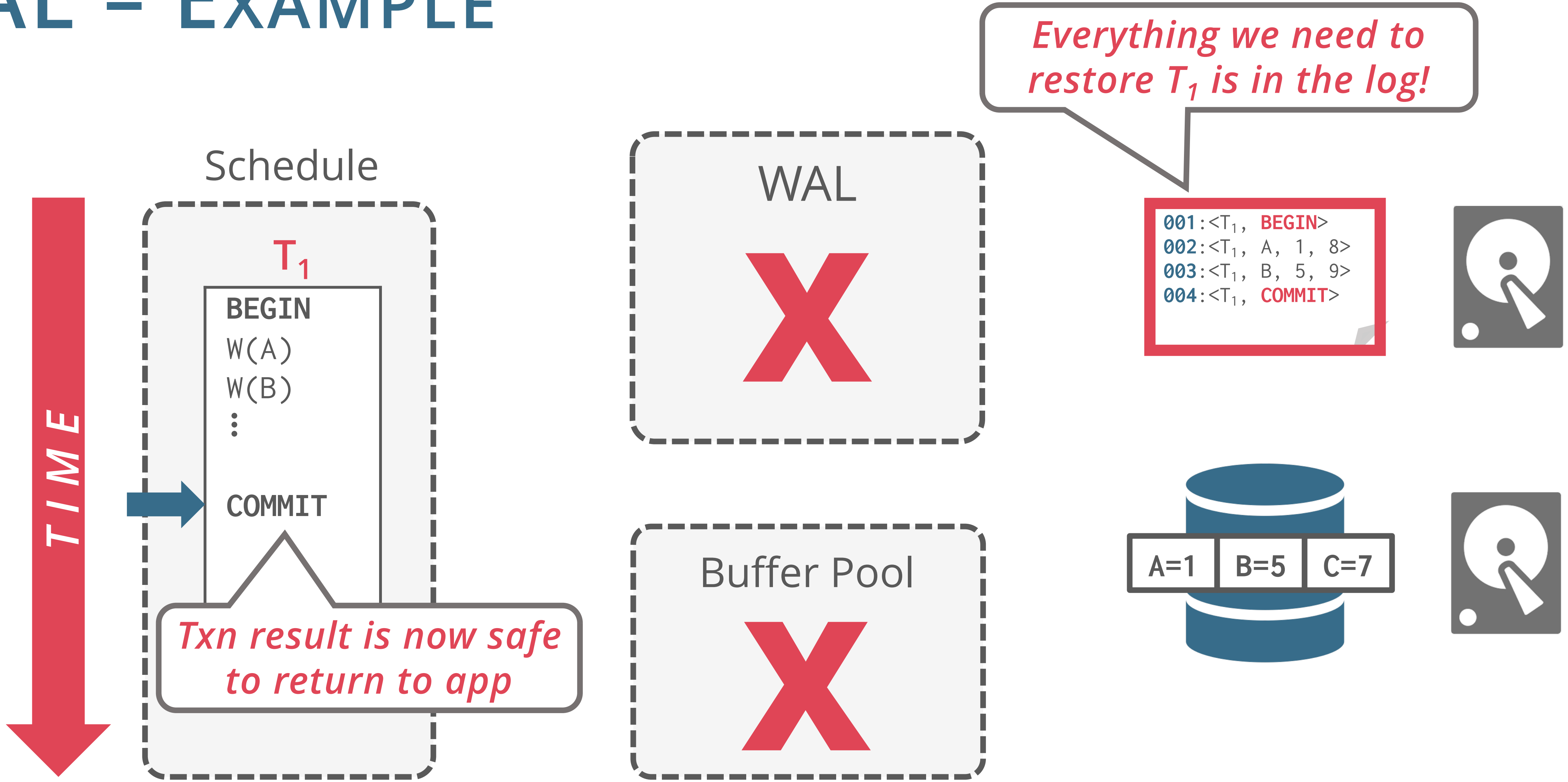
WAL – EXAMPLE


TIME
Schedule
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL
<T_1, BEGIN>
<T_1, A, 1, 8>
<T_1, B, 5, 9>
Buffer Pool
A=8
B=9
C=7
A=1
B=5
C=7

WAL – Example


TIME
Schedule
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
Txn result is now safe to return to app
WAL
<T_1, BEGIN>
<T_1, A, 1, 8>
<T_1, B, 5, 9>
<T_1, COMMIT>
Buffer Pool
A=8
B=9
C=7
<T_1, BEGIN>
<T_1, A, 1, 8>
<T_1, B, 5, 9>
<T_1, COMMIT>
A=1
B=5
C=7

WAL – EXAMPLE


Everything we need to restore T_1 is in the log!
Schedule
TIME
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
Txn result is now safe to return to app
WAL
Buffer Pool
001:<T_1, BEGIN>
002:<T_1, A, 1, 8>
003:<T_1, B, 5, 9>
004:<T_1, COMMIT>
A=1
B=5
C=7

ARIES

Recovery algorithm developed at IBM Research in early 1990s

Write-Ahead Logging

- Any change is recorded in log on stable storage before the change is written to disk
- Must use **STEAL** + **NO-FORCE** buffer pool policies

Repeating History During Redo

- On restart, retrace actions and restore database to exact state before crash

Logging Changes During Undo

- Log also undo actions to ensure action is not repeated in the event of repeated failures

Log Records

Each log record has a unique **Log Sequence Number (LSN)**

- LSNs are always increasing

Log record fields:

LSN	*Transaction ID*	*Type*

Log Record Types:

- **UPDATE** – consists of Object ID + Before Value (UNDO) + After Value (REDO)
- **BEGIN** / **COMMIT** / **ABORT**
- **TXN-END** – signifies the end of commit or abort
- **CLR** (**Compensation Log Record**) – for UNDO actions

Log Sequence Numbers

Name	Where	Definition
flushedLSN	Memory	Last LSN in log on disk
pageLSN	$page_x$	Newest update to $page_x$
recLSN	$page_x$	Oldest update to $page_x$ since it was last flushed
lastLSN	T_i	Latest record of txn T_i
MasterRecord	Disk	LSN of latest checkpoint

Writing Log Records

Each data page contains a **pageLSN**

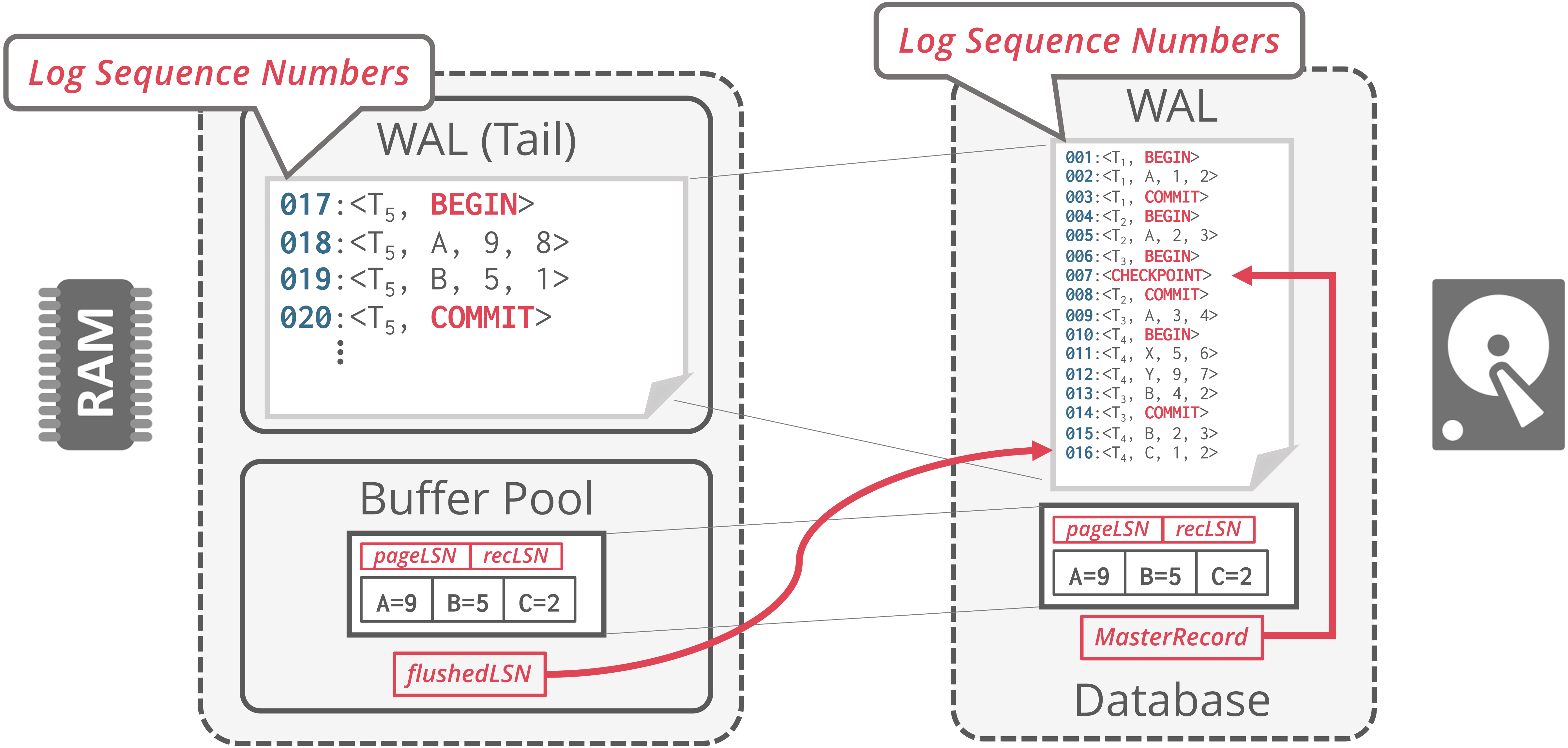
- The ***LSN*** of the most recent log record for an update to that page
- Update the **pageLSN** every time a txn modifies a record in the page

System keeps track of **flushedLSN**

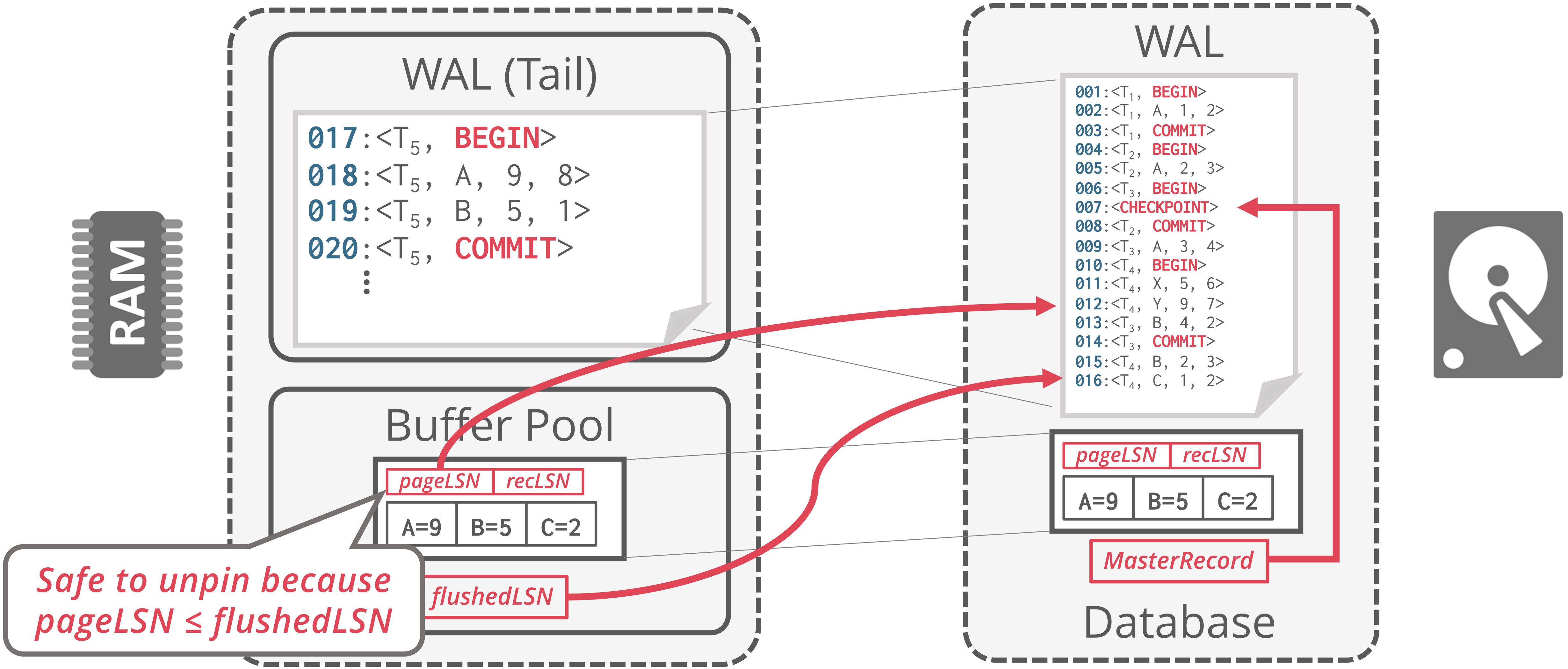
- The max ***LSN*** flushed so far
- Update the **flushedLSN** in memory every time DBMS writes out the WAL buffer to disk

Before a page **X** can be written to disk, we must flush log at least to the point where $\textbf{pageLSN}_X \le \textbf{flushedLSN}$

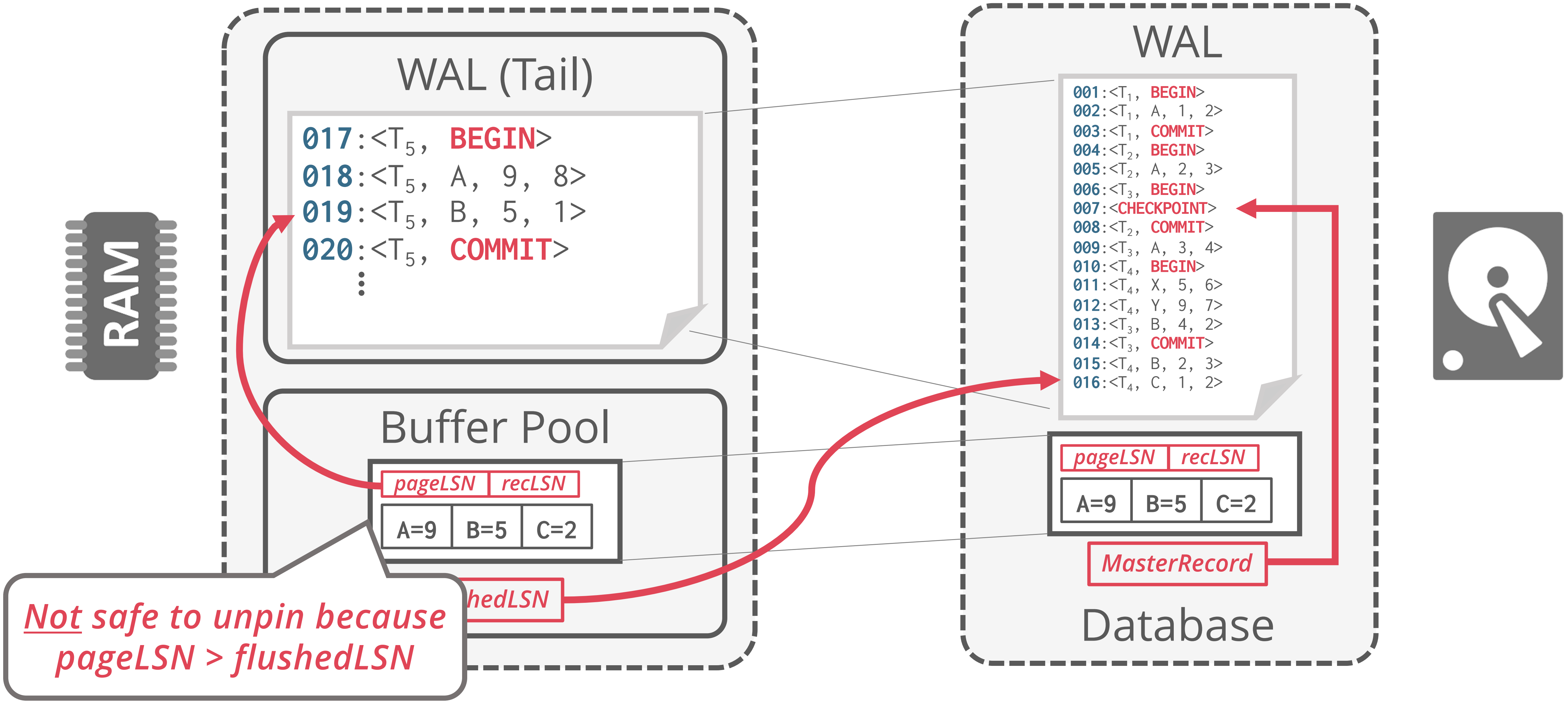
Writing Log Records



Writing Log Records



Writing Log Records



Normal Execution

Each txn invokes a sequence of reads and writes, followed by commit or abort

Assumptions in this lecture:

- All log records fit within a single page
- Disk writes are atomic
- Single-versioned tuples with Strict 2PL
- STEAL + NO-FORCE buffer management with WAL

Transaction Commit

Write COMMIT record to log

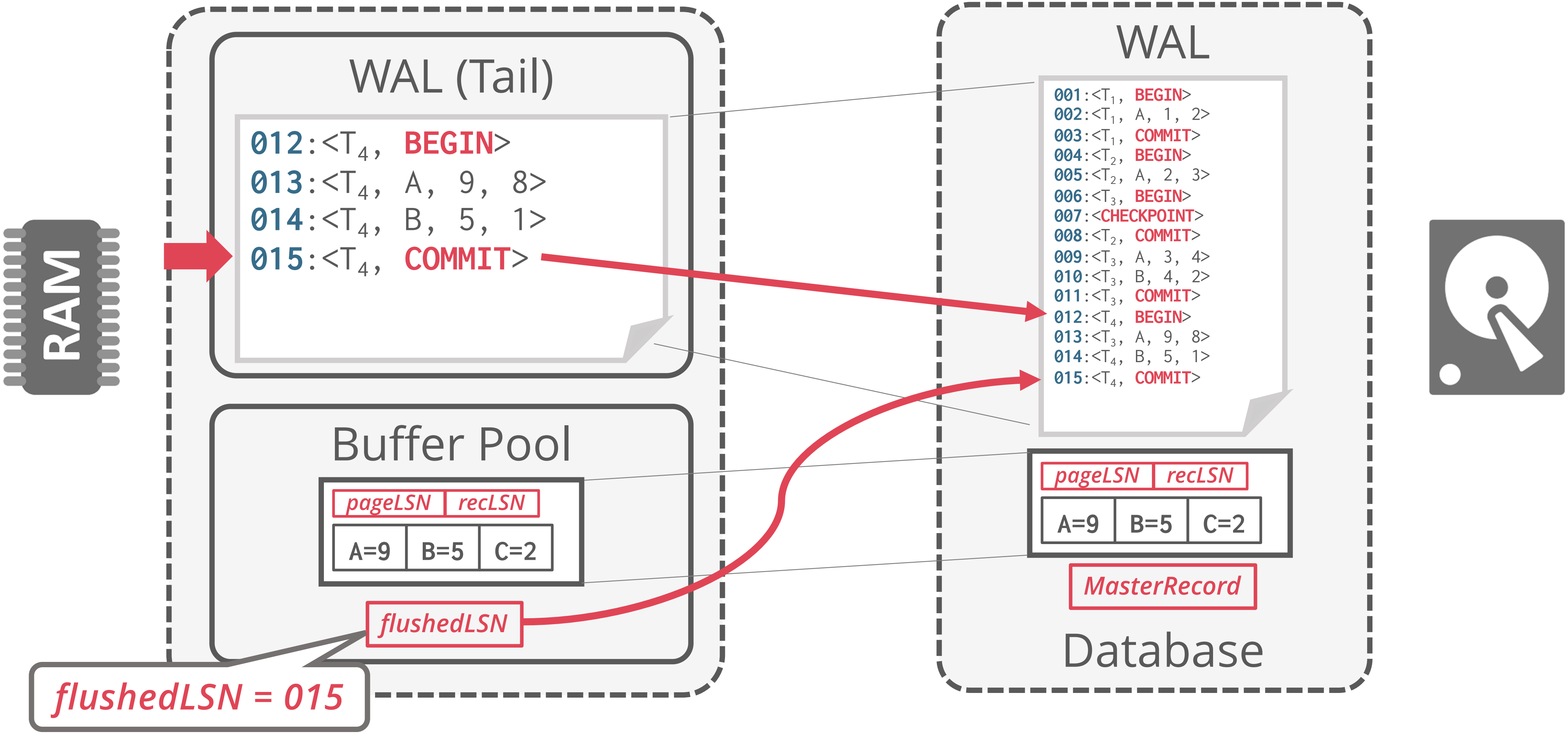
All log records up to txn's COMMIT record are flushed to disk

- Note that log flushes are sequential, synchronous writes to disk
- Many log records per log page

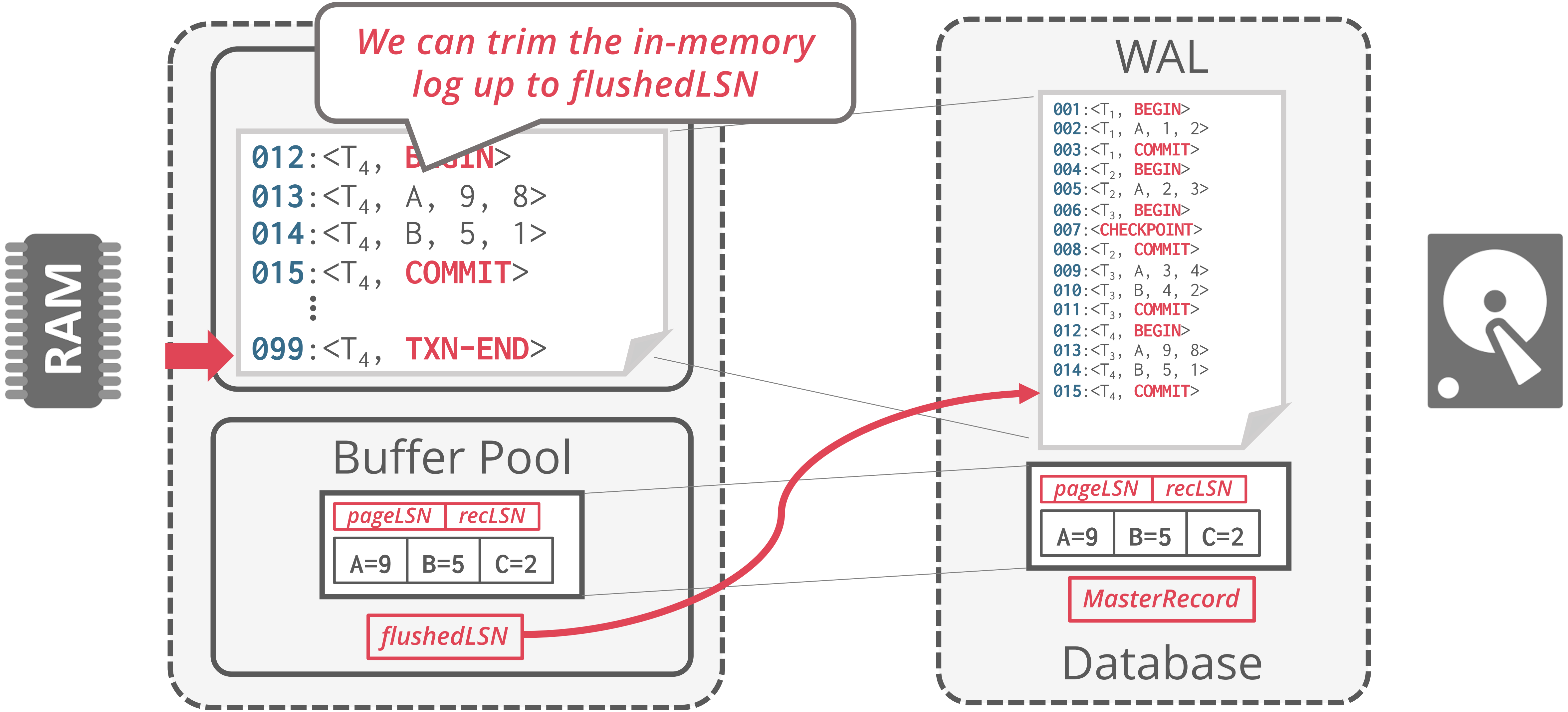
When the commit succeeds, write a special TXN-END record to log

- This does **not** need to be flushed immediately

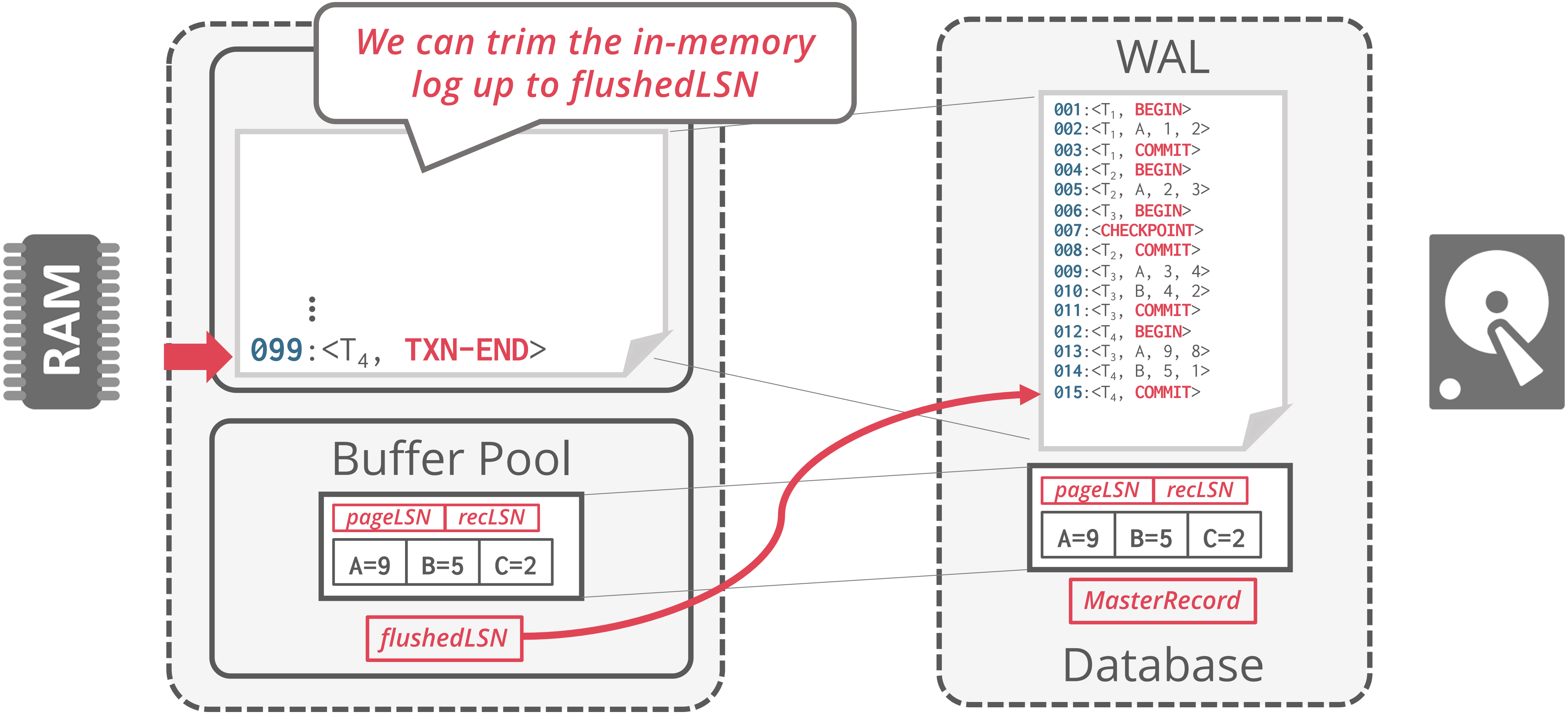
TRANSACTION COMMIT



Transaction Commit



Transaction Commit



Transaction Abort

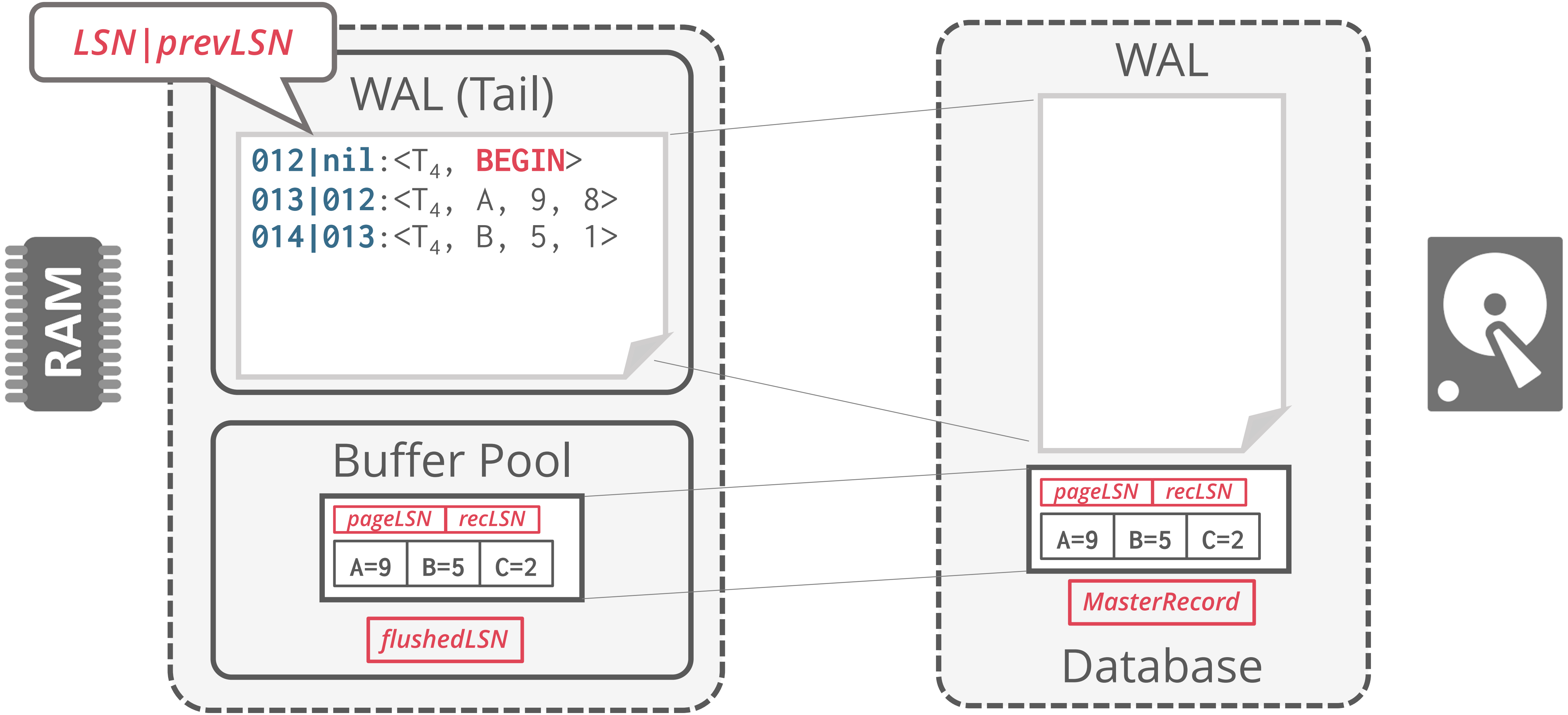
We need to add another field to our log records

- **prevLSN**: The previous ***LSN*** for the txn
- Maintains a linked list for each txn that makes it easy to walk through its records

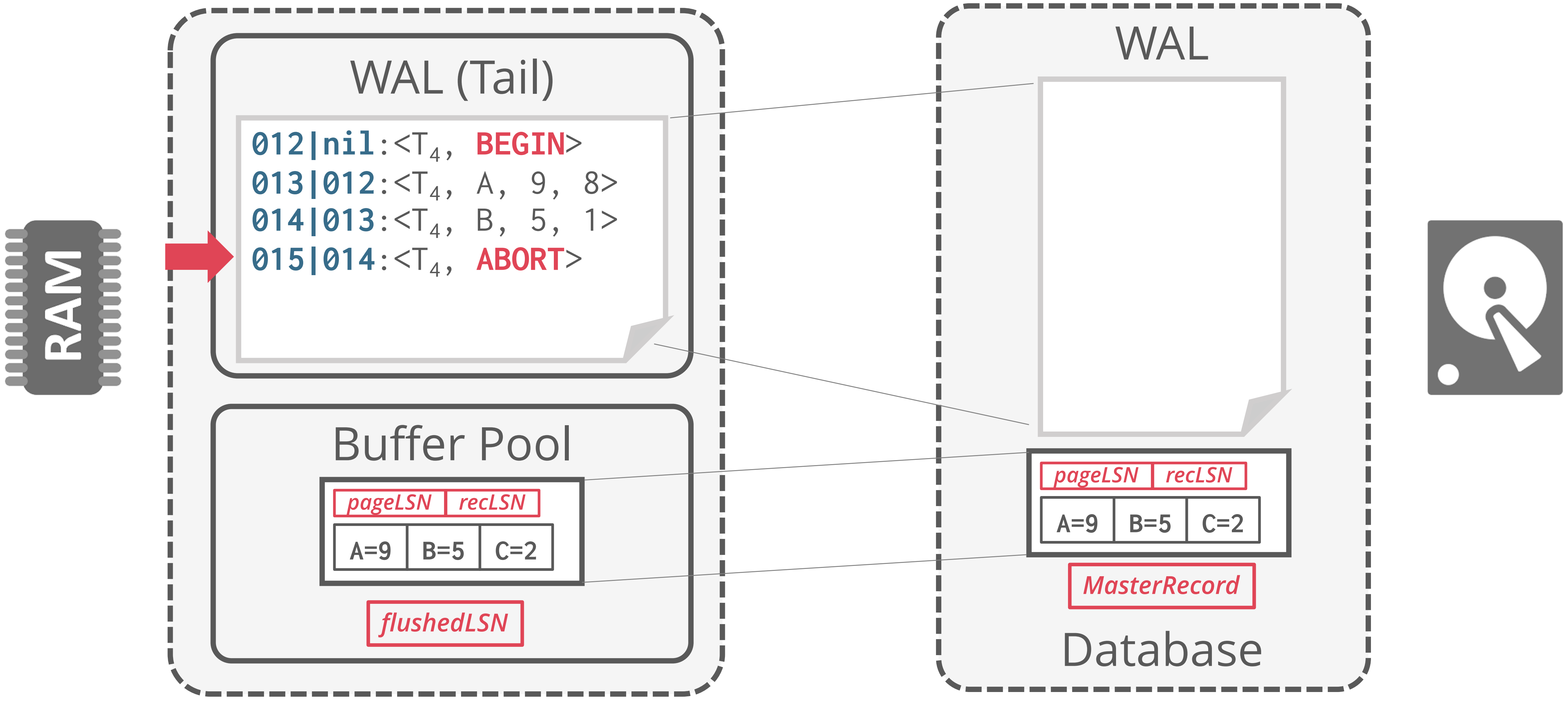
We want to "play back" the log in reverse order, **UNDO**ing updates

- Get **lastLSN** of txn stored in the active txn table (more details later)
- Can follow chain of log records backward via the **prevLSN** field
- Before starting UNDO, write an **ABORT** log record
  - ***For recovering from crash during UNDO!***

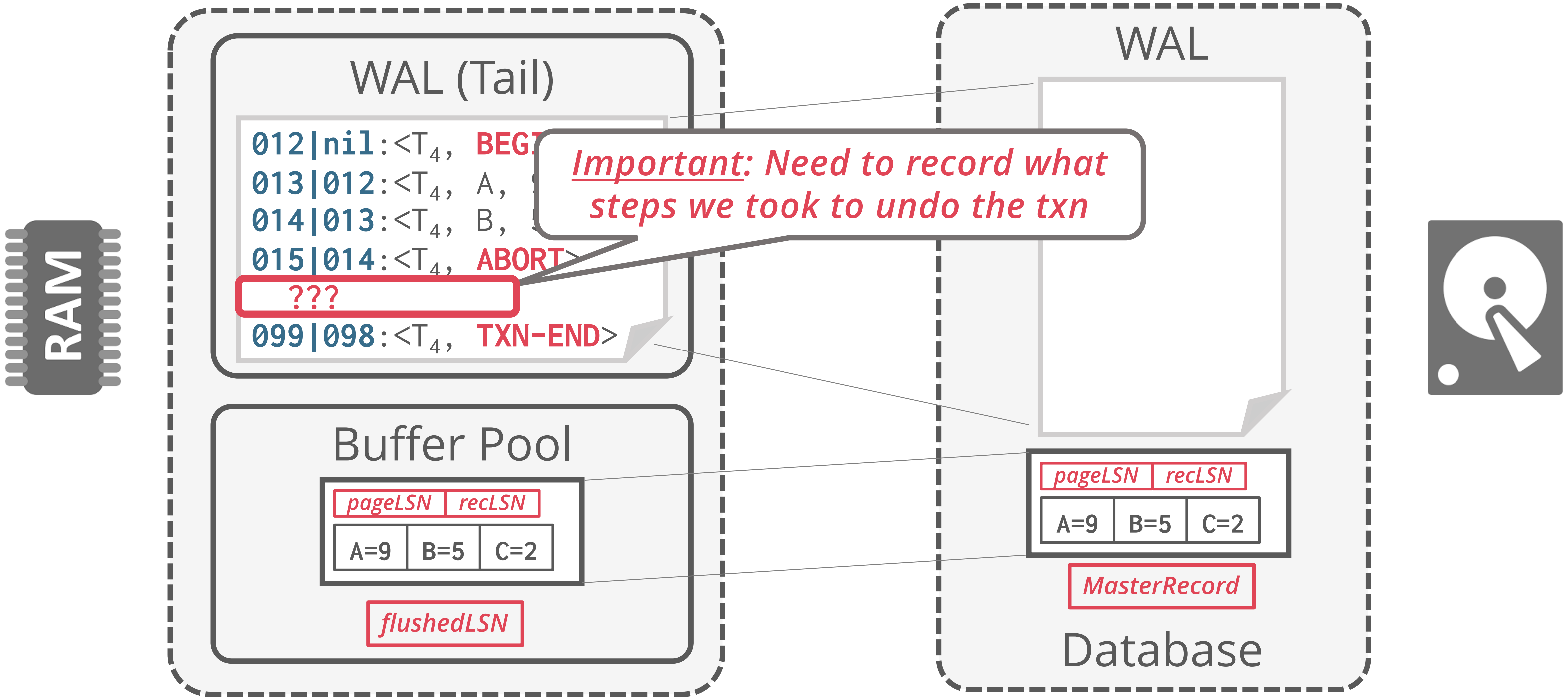
TRANSACTION ABORT



TRANSACTION ABORT



Transaction Abort



Compensation Log Records

A **CLR** describes the actions taken to undo the actions of a previous update record

It has all the fields of an update log record plus the **undoNext** pointer (the next-to-be-undone LSN)

CLRs are added to log like any other record

Transaction Abort – CLR Example

LSN	prevLSN	TxnID	Type	Object	Before	After	UndoNext
001	nil	T_1	BEGIN	–	–	–	–
002	001	T_1	UPDATE	A	30	40	–
⋮							
011	002	T_1	ABORT	–	–	–	–

Transaction Abort – CLR Example

TIME

LSN	prevLSN	TxnID	Type	Object	Before	After	UndoNext
001	nil	T_1	BEGIN	–	–	–	–
002	001	T_1	UPDATE	A	30	40	–
⋮							
011	002	T_1	ABORT	–	–	–	–
⋮							
026	011	T_1	CLR-002	A	40	30	001

Transaction Abort – CLR Example

TIME

LSN	prevLSN	TxnID	Type	Object	Before	After	UndoNext
001	nil	T_1	BEGIN	–	–	–	–
002	001	T_1	UPDATE	A	30	40	–
⋮							
011	002	T_1	ABORT	–	–	–	–
⋮							
026	011	T_1	CLR-002	A	40	30	001

TRANSACTION ABORT – CLR EXAMPLE

TIME

LSN	prevLSN	TxnID	Type	Object	Before	After	UndoNext
001	nil	T_1	BEGIN	–	–	–	–
002	001	T_1	UPDATE	A	30	40	–
⋮							
011	002	T_1	ABORT	–	–	–	–
⋮							
026	011	T_1	CLR-002	A	40	30	001

The LSN of the next log record to be undone

Transaction Abort – CLR Example

TIME

LSN	prevLSN	TxnID	Type	Object	Before	After	UndoNext
001	nil	T_1	BEGIN	–	–	–	–
002	001	T_1	UPDATE	A	30	40	–
⋮							
011	002	T_1	ABORT	–	–	–	–
⋮							
026	011	T_1	CLR-002	A	40	30	001
027	026	T_1	TXN-END	–	–	–	nil

Abort Algorithm

First write an **ABORT** record to log for the txn

Then play back updates in reverse order. For each update record:

- Write a **CLR** entry
- Restore old value

At end, write a **TXN-END** log record

Notice: CLRs **never** need to be undone

- but they might be redone when repeating history: guarantees atomicity!

Other Log-Related State

Active Transaction Table (ATT)

One entry per currently active txn. Entry removed when txn commits or aborts

- **txnId**: unique txn identifier
- **status**: the current “mode” of the txn (running / committed / undo)
- **lastLSN**: most recent LSN created by txn

Dirty Page Table (DPT)

Record which pages in the buffer pool contain changes from uncommitted txns

One entry per dirty page in the buffer pool

- **recLSN**: the LSN of the log record that <u>first</u> caused the page to be dirty

CHECKPOINTING

Periodically, the DBMS creates a **checkpoint** to minimize the time taken to recover in the event of a system crash

"Fuzzy checkpointing" writes to log:

- CHECKPOINT-BEGIN record: Indicates when checkpoint began
- CHECKPOINT-END record: Contains current txn table and dirty page table
- The recorded state is accurate as of the time of the CHECKPOINT-BEGIN record
- No attempt to force dirty pages to disk! (too expensive)

Store LSN of checkpoint record in a safe place (MasterRecord)

ARIES – Recovery Phases

Phase #1 – Analysis

Read WAL from last checkpoint to identify dirty pages in the buffer pool and active txns at the time of the crash

Phase #2 – Redo

Repeat **all** actions starting from an appropriate point in the log (even txns that will abort)

Phase #3 – Undo

Reverse the actions of txns that did not commit before the crash

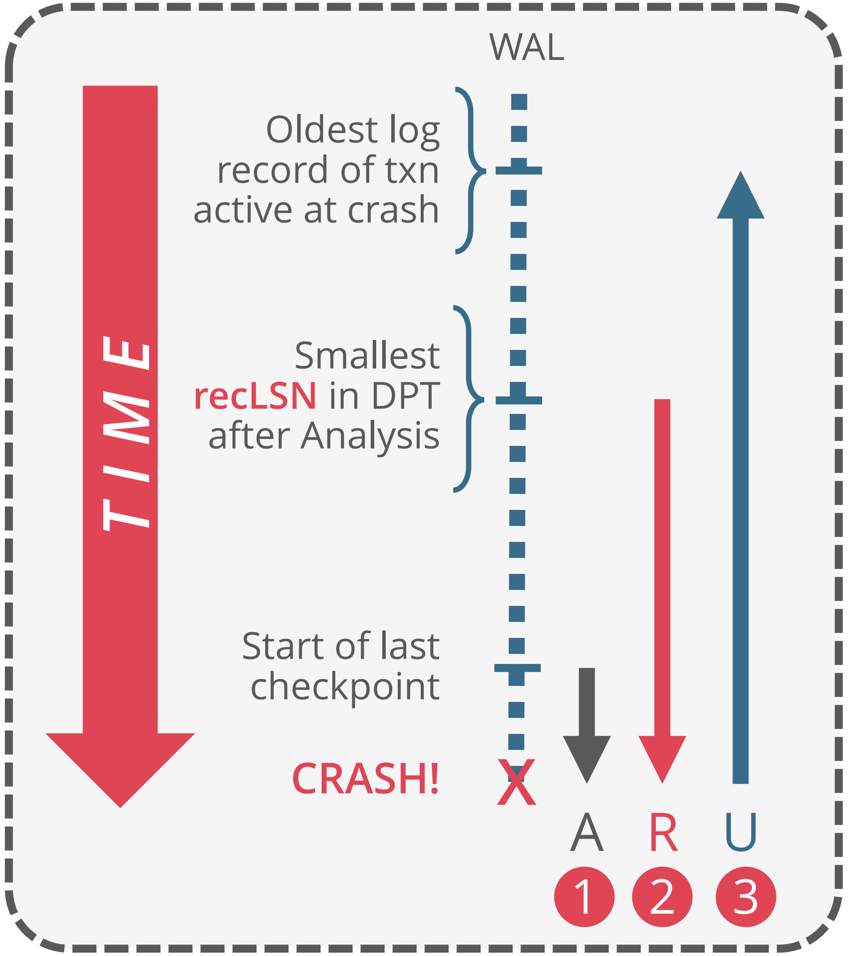
ARIES – OVERVIEW

Start from last **BEGIN-CHECKPOINT** found via **MasterRecord**

Analysis: Figure out which txns committed or failed since checkpoint

Redo: Repeat <u>all</u> actions

Undo: Reverse effects of failed txns



Analysis Phase

Scan log forward from last successful checkpoint

If you find a **TXN-END** record, remove its corresponding txn from **ATT**

All other records:

- Add txn to **ATT** with status **UNDO** if not already there
- Set **lastLSN** to the **LSN** of the current log record
- On commit, change status to **COMMIT**

For **UPDATE** records:

- If page **X** not in **DPT**, add **X** to **DPT** and set its **recLSN**=**LSN**

ANALYSIS PHASE

At the end of the analysis phase:

- **ATT** tells the DBMS which txns were active at time of crash
- **DPT** tells the DBMS which dirty pages might not have made it to disk

Redo Phase

Goal: repeat history to reconstruct state at the moment of the crash

- Reapply all updates (even aborted txns!) and redo **CLRs**

Scan forward from the log record containing smallest **recLSN** in **DPT**

For each update log record or ***CLR*** with a ***LSN***, redo the action unless:

- Affected page is not in the **DPT**, or
- Affected page is in **DPT** but its **recLSN** is greater than the record's ***LSN***, or
- Affected **pageLSN** (on disk) ≥ the record's ***LSN***

Redo Phase

To redo an action

- Reapply logged action
- Set **pageLSN** to log record's ***LSN***
- No additional logging, no forced flushes!

At the end of the Redo Phase, write **TXN-END** log records for all txns with status **COMMIT** and remove them from the **ATT**

UNDO PHASE

Undo all txns that were active at the time of crash and therefore will never commit

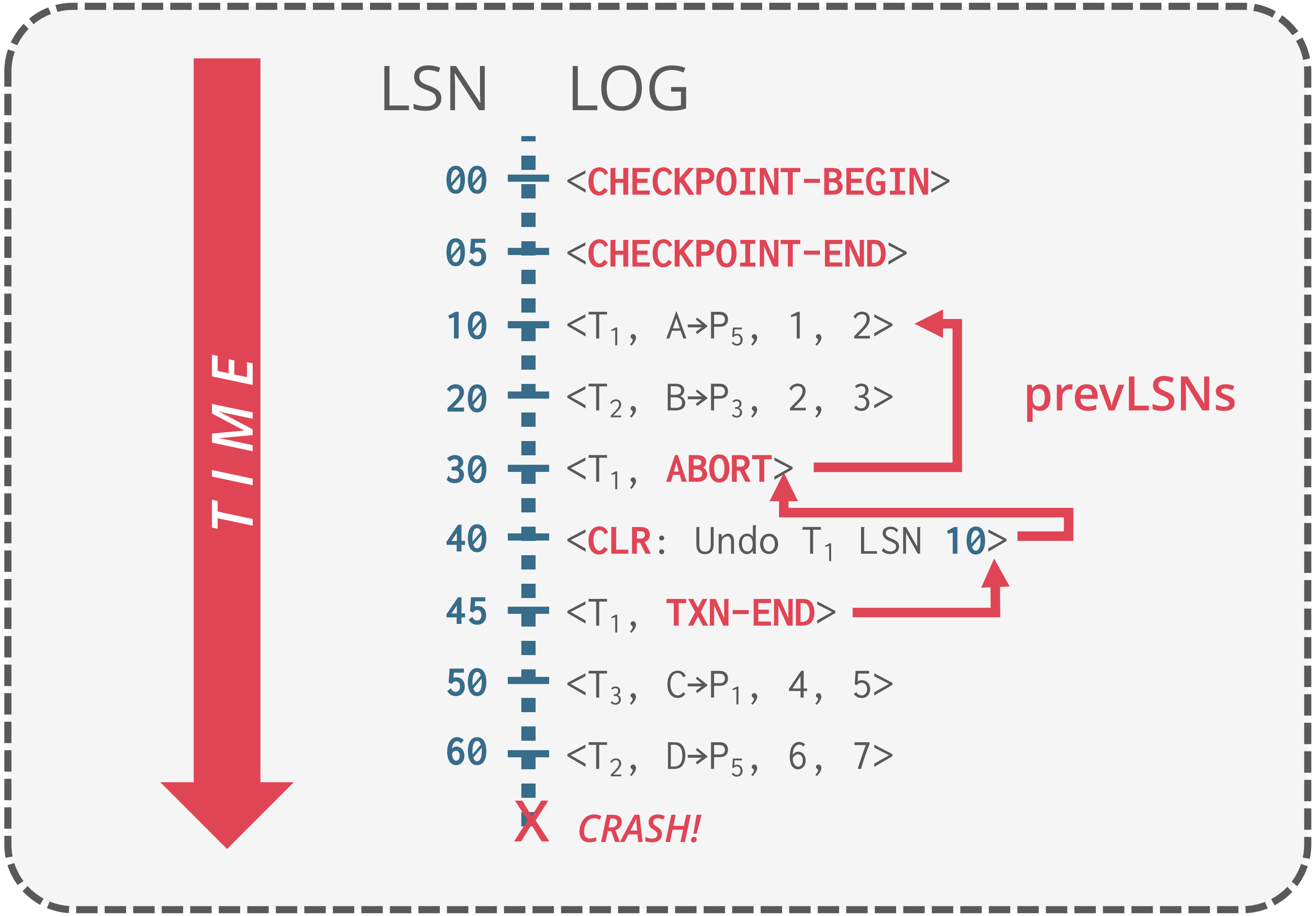
- These are all txns with UNDO status in the **ATT** after the Analysis Phase

Process them in reverse ***LSN*** order using lastLSN to speed up traversal

Write a **CLR** for every modification

Aborting a txn is a special case of Undo Phase applied to only one txn

FULL EXAMPLE



Full Example

RAM

ATT

TxnID	Status	lastLSN
T_2	U	60
T_3	U	50
–	–	–

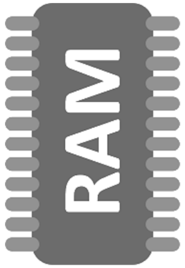
DPT

PageID	recLSN
P_1	50
P_3	20
P_5	10

flushedLSN

LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1, ABORT>
40,45	<CLR: Undo T_1 LSN 10>, <T_1, TXN-END>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
X	*CRASH! RESTART!*

FULL EXAMPLE



ATT

TxnID	Status	lastLSN
T_2	U	60
T_3	U	50
–	–	–

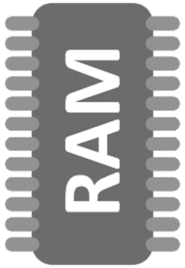
DPT

PageID	recLSN
P_1	50
P_3	20
P_5	10

flushedLSN

LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1, ABORT>
40,45	<CLR: Undo T_1 LSN 10>, <T_1, TXN-END>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
X	*CRASH! RESTART!*
70	<CLR: Undo T_2 LSN 60, UndoNext 20>

FULL EXAMPLE



ATT

TxnID	Status	lastLSN
T_2	U	60
T_3	U	50
–	–	–

DPT

PageID	recLSN
P_1	50
P_3	20
P_5	10

flushedLSN

LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1, ABORT>
40,45	<CLR: Undo T_1 LSN 10>, <T_1, TXN-END>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
X	*CRASH! RESTART!*
70	<CLR: Undo T_2 LSN 60, UndoNext 20>
80,85	<CLR: Undo T_3 LSN 50> <T_3, TXN-END>

Flush dirty pages + WAL to disk!

Full Example

RAM

ATT

TxnID	Status	lastLSN
T_2	U	60
T_3	U	50
–	–	–

DPT

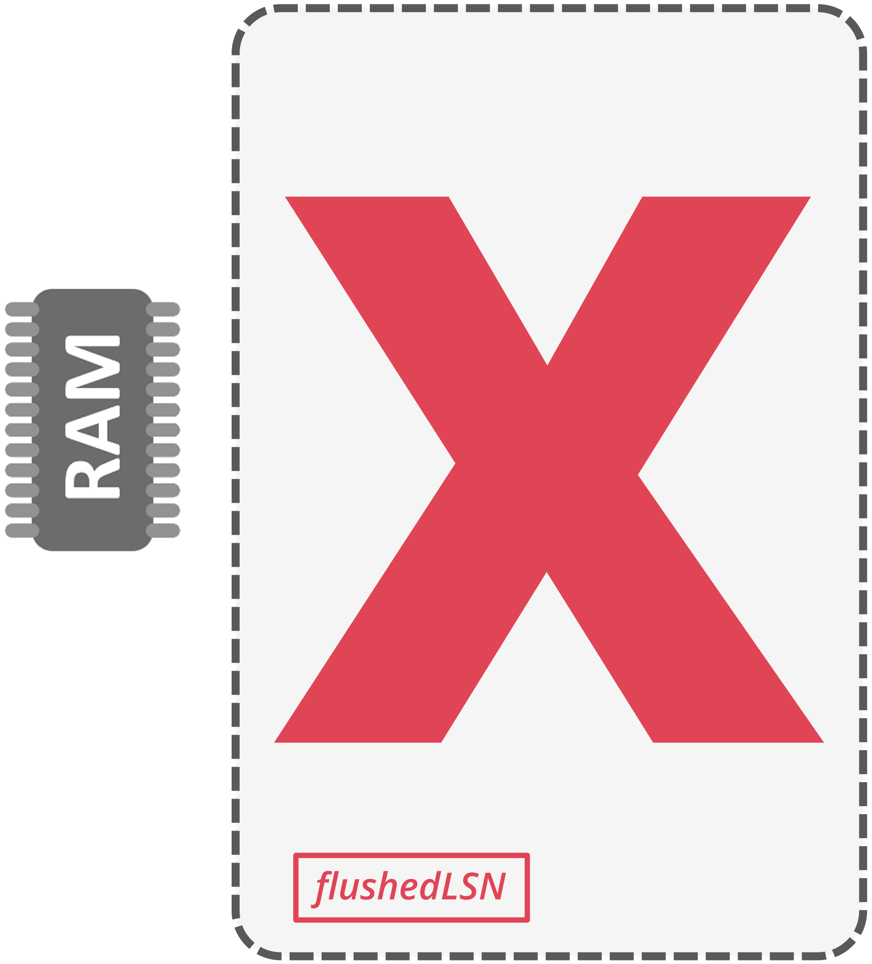
PageID	recLSN
P_1	50
P_3	20
P_5	10

flushedLSN

LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1, ABORT>
40,45	<CLR: Undo T_1 LSN 10>, <T_1, TXN-END>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
X	*CRASH! RESTART!*
70	<CLR: Undo T_2 LSN 60, UndoNext 20>
80,85	<CLR: Undo T_3 LSN 50> <T_3, TXN-END>
X	*CRASH! RESTART!*

Flush dirty pages + WAL to disk!

Full Example



LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1, ABORT>
40,45	<CLR: Undo T_1 LSN 10>, <T_1, TXN-END>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
X	*CRASH! RESTART!*
70	<CLR: Undo T_2 LSN 60, UndoNext 20>
80,85	<CLR: Undo T_3 LSN 50> <T_3, TXN-END>
X	*CRASH! RESTART!*

Full Example

RAM

ATT

TxnID	Status	lastLSN
T_2	U	70
–	–	–
–	–	–

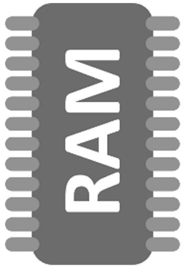
DPT

PageID	recLSN
P_1	50
P_3	20
P_5	10

flushedLSN

LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1, ABORT>
40,45	<CLR: Undo T_1 LSN 10>, <T_1, TXN-END>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
	CRASH! RESTART!
70	<CLR: Undo T_2 LSN 60, UndoNext 20>
80,85	<CLR: Undo T_3 LSN 50> <T_3, TXN-END>
	CRASH! RESTART!

Full Example



ATT

TxnID	Status	lastLSN
T_2	U	70
–	–	–
–	–	–

DPT

PageID	recLSN
P_1	50
P_3	20
P_5	10

flushedLSN

LSN	LOG
00,05	<CHECKPOINT-BEGIN>, <CHECKPOINT-END>
10	<T_1, A→P_5, 1, 2>
20	<T_2, B→P_3, 2, 3>
30	<T_1, ABORT>
40,45	<CLR: Undo T_1 LSN 10>, <T_1, TXN-END>
50	<T_3, C→P_1, 4, 5>
60	<T_2, D→P_5, 6, 7>
X	*CRASH! RESTART!*
70	<CLR: Undo T_2 LSN 60, UndoNext 20>
80,85	<CLR: Undo T_3 LSN 50> <T_3, TXN-END>
X	*CRASH! RESTART!*
90,95	<CLR: Undo T_2 LSN 20> <T_2, TXN-END>

Additional Crash Issues

What does the DBMS do if it crashes during recovery in Analysis Phase?

- Nothing. Just run recovery again

What does the DBMS do if it crashes during recovery in Redo Phase?

- Again nothing. Redo everything again

Conclusion: ARIES

Write-Ahead Logging with STEAL + NO-FORCE

Any change is recorded in log on stable storage before the database change is written to disk

Fuzzy checkpointing (snapshot of dirty pageIDs and currently active txns)

Repeating History During Redo

On restart, redo everything since the earliest dirty page

Undo txns that never commit

Write **CLRs** when undoing to survive failures during restarts

ARIES

Recovery algorithm developed at IBM Research in early 1990s

Write-Ahead Logging

- Any change is recorded in log on stable storage before the change is written to disk
- Must use **STEAL** + **NO-FORCE** buffer pool policies

Repeating History During Redo

- On restart, retrace actions and restore database to exact state before crash

Logging Changes During Undo

- Log also undo actions to ensure action is not repeated in the event of repeated failures