Carnegie Mellon University

Intro to Database Systems (15-445/645)

Lecture #20

Database Logging

SPRING 2024 ❯❯ Prof. Jignesh Patel

ADMINISTRIVIA

Project #3 is due Sun April 7, 2024 @ 11:59pm

→ Special OH: Sat April 4, 2024 @ 03:00pm-05:00pm, GHC 5117

HW #5 is due Sat April 20, 2024 @ 11:59pm

Final Exam

→ Thu May 2, 2024 @ 05:30pm-08:30pm

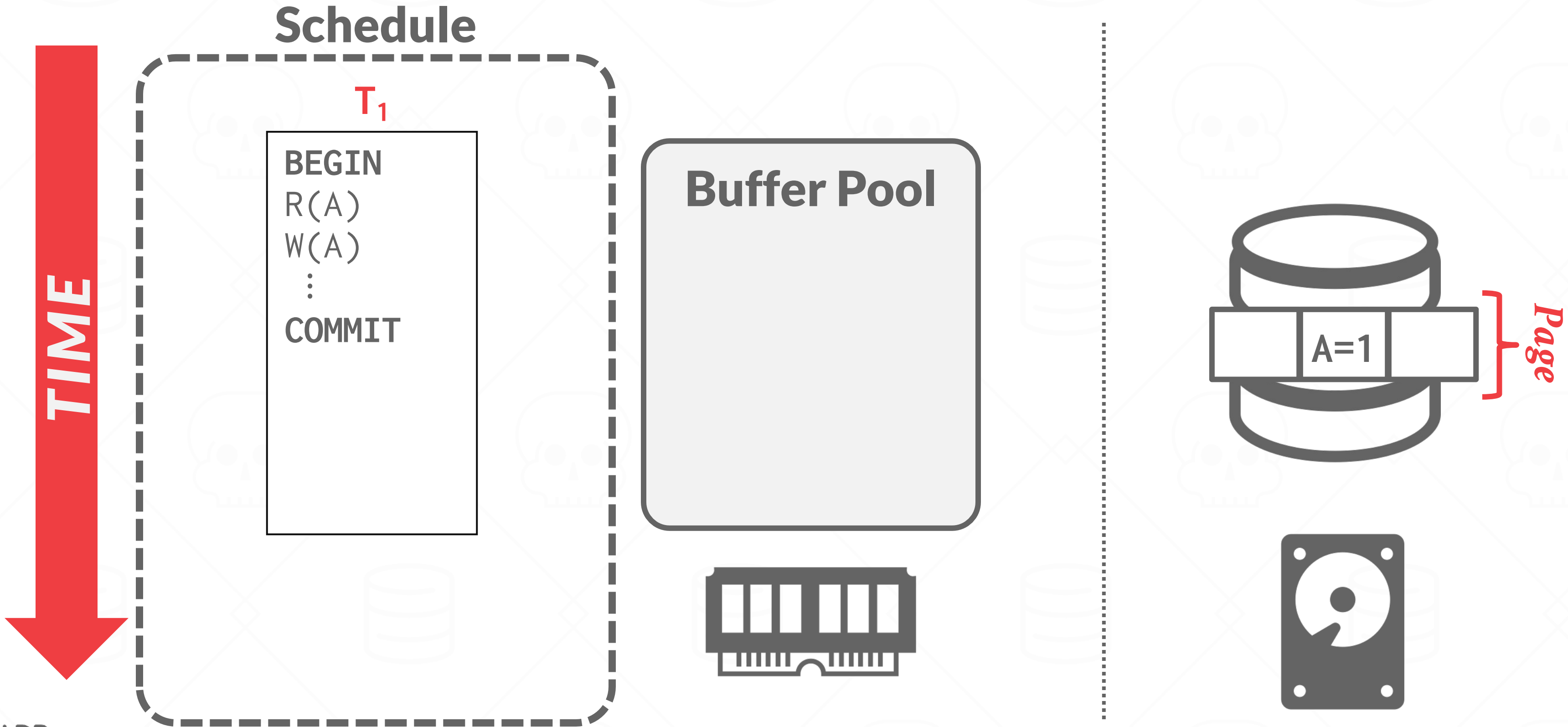
Lectures #23 and #24

→ Recorded lectures and will be posted next week

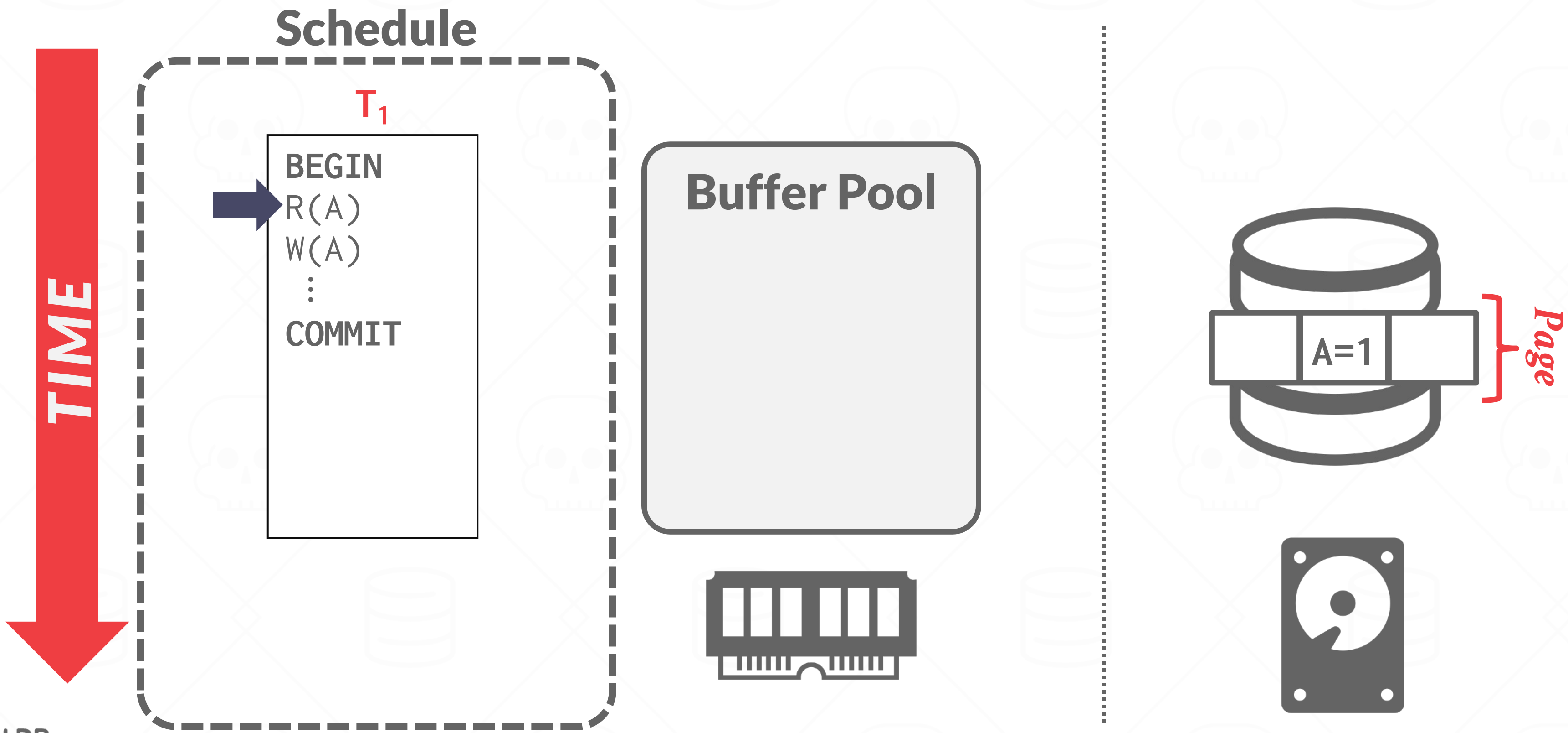
Lecture #26: Guest Speaker from Snowflake

→ Devin Petersohn on "Beyond SQL: Dataframes in the Database"
https://db.cs.cmu.edu/events/spring-2024-beyond-sql-dataframes-in-the-database-devin-petersohn/

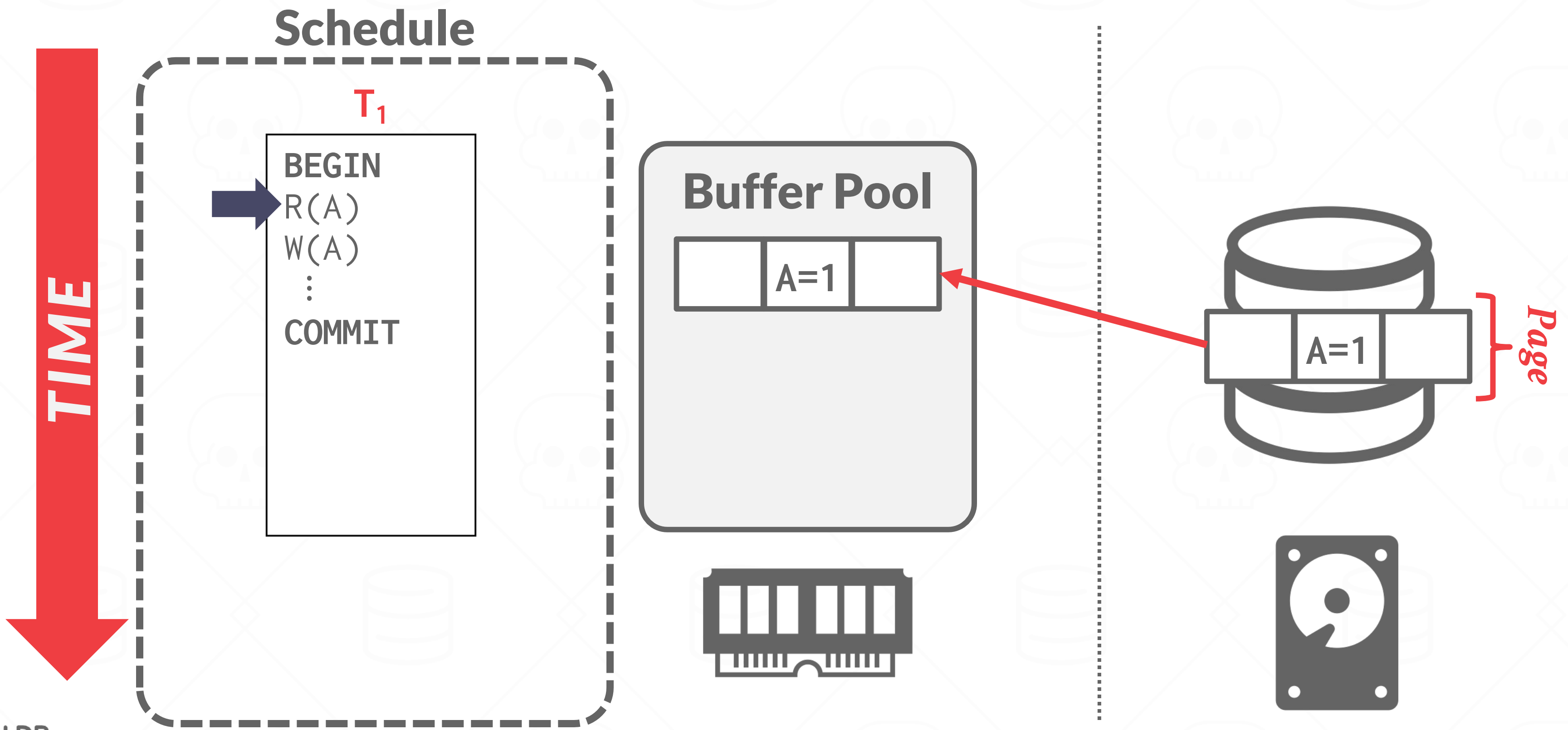
MOTIVATION



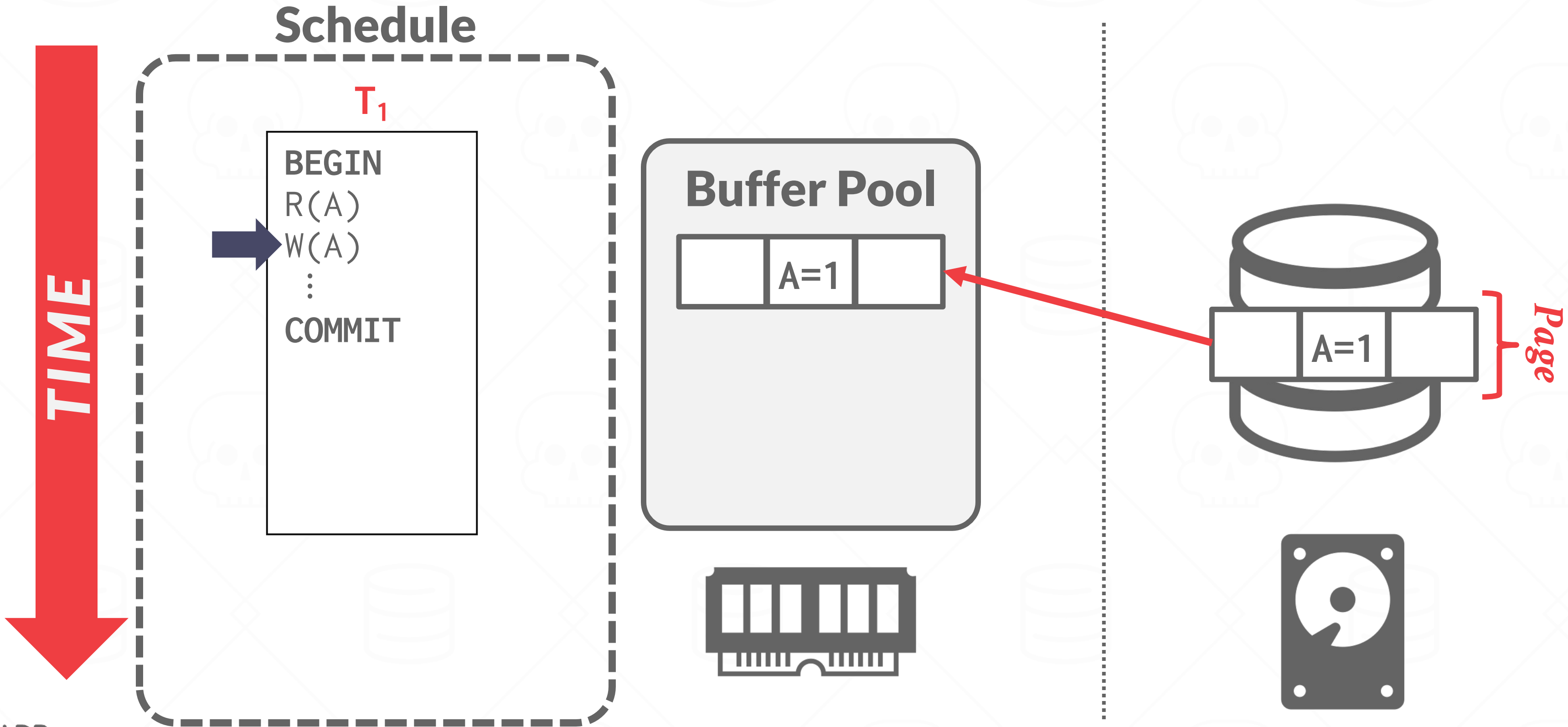
MOTIVATION



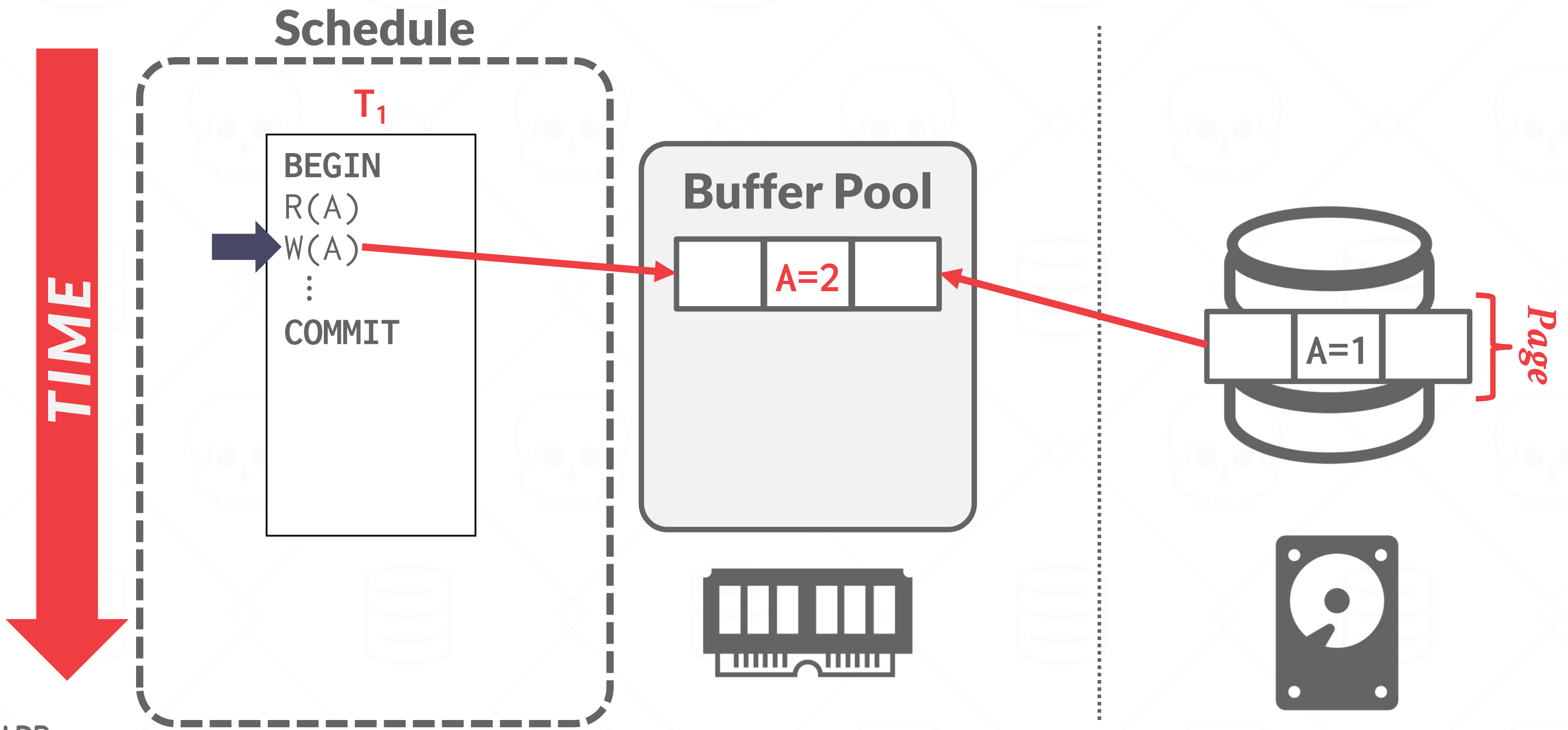
MOTIVATION



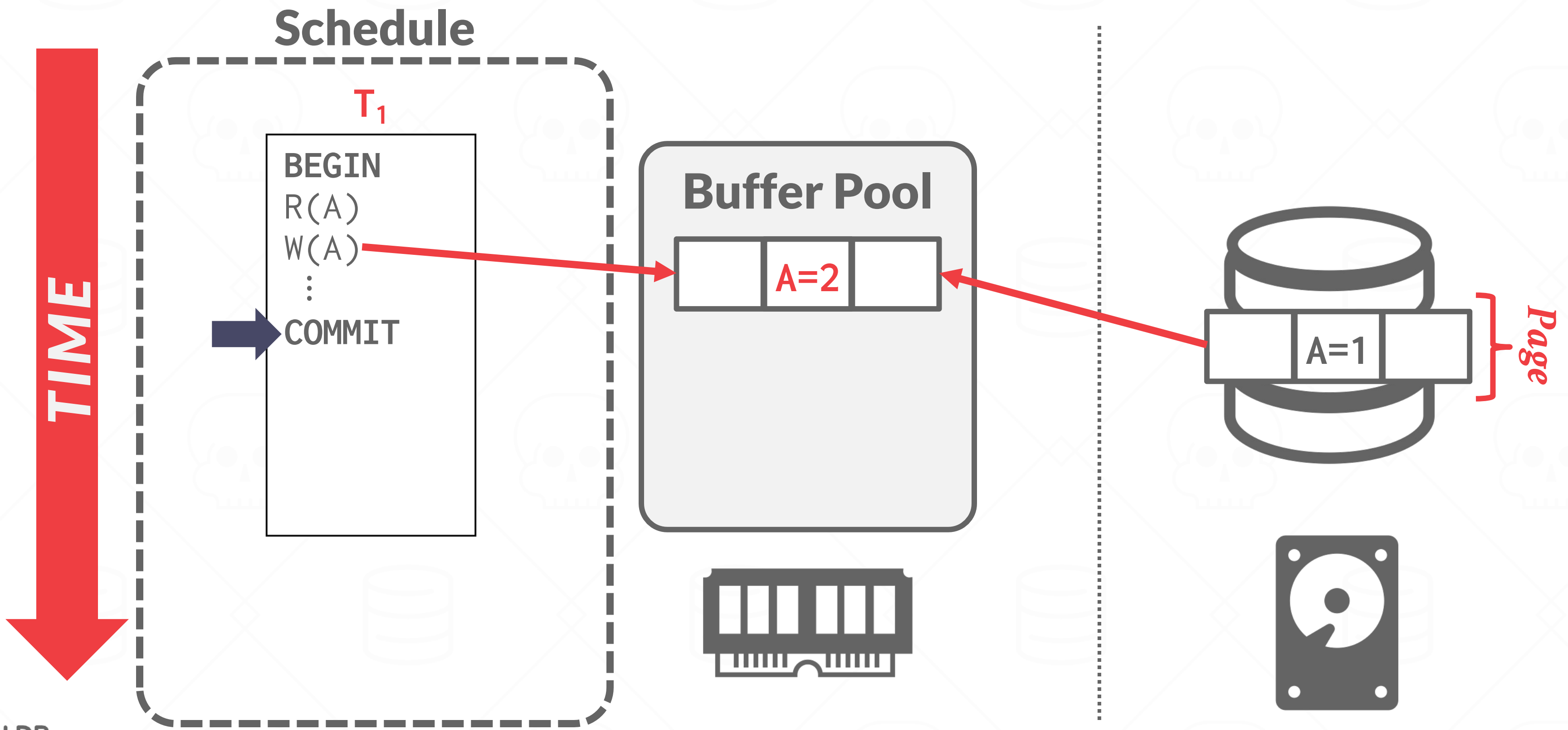
MOTIVATION



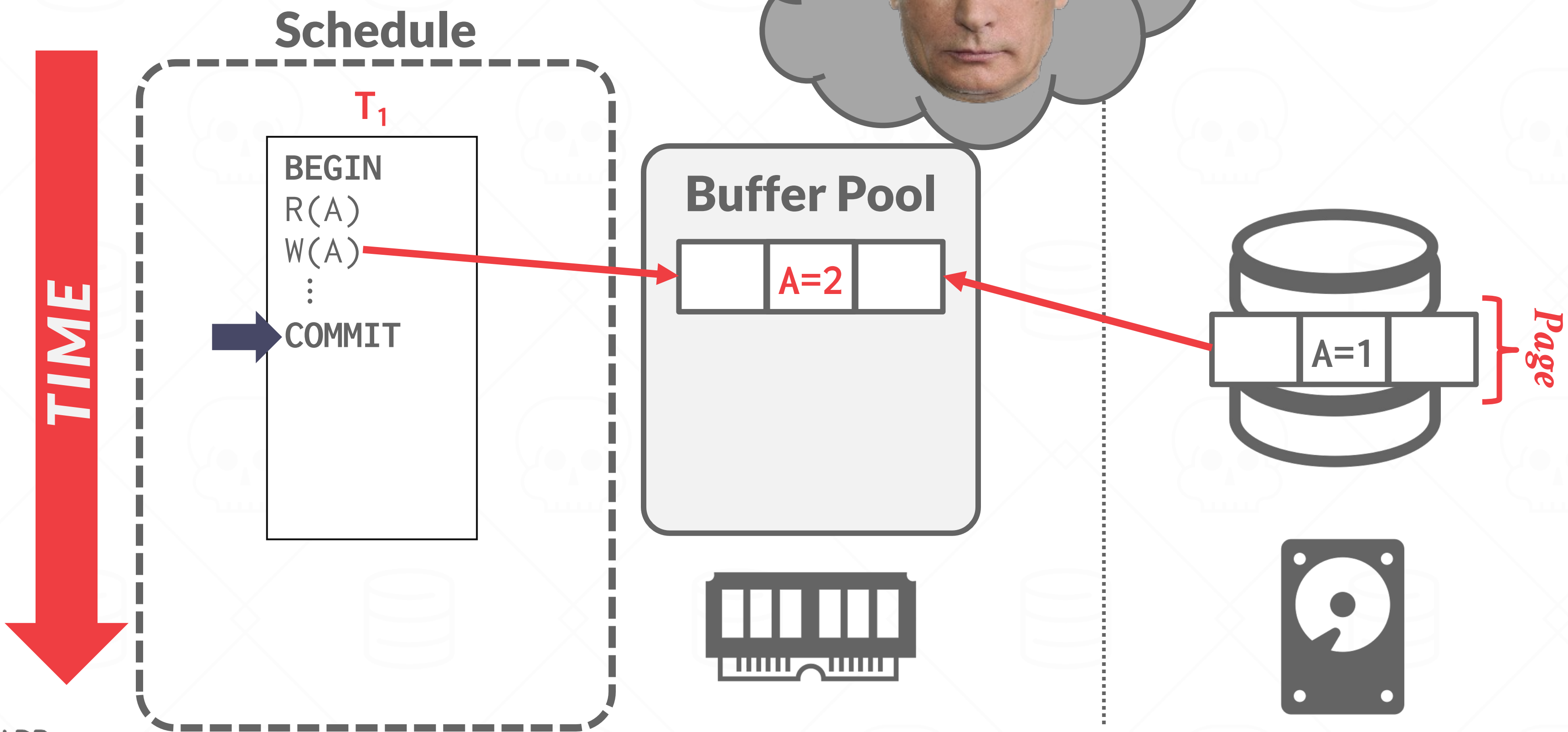
MOTIVATION



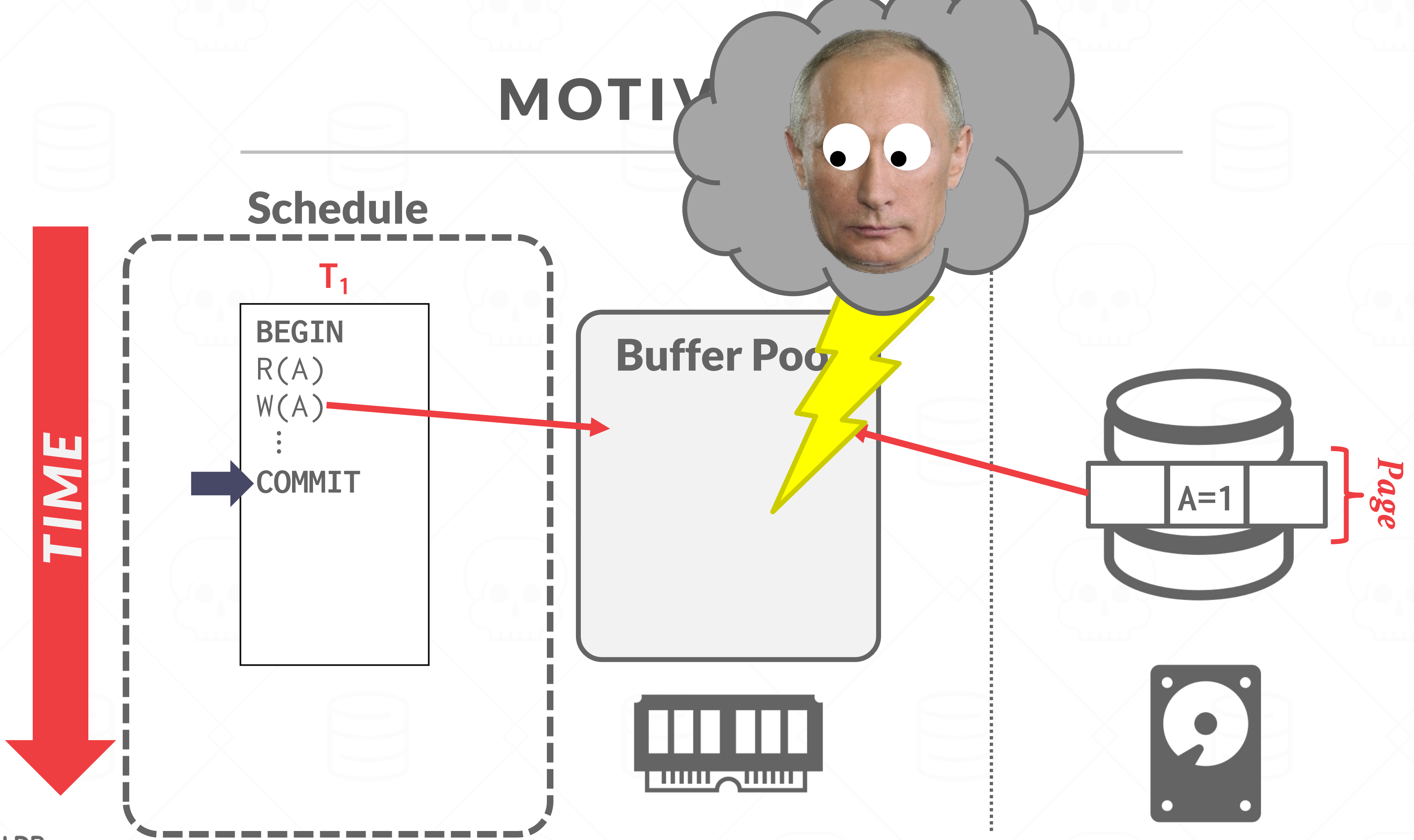
MOTIVATION



MOTIV



MOTIV



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. ***Today***
- →Actions after a failure to recover the database to a state that ensures atomicity, consistency, and durability.

TODAY'S AGENDA

Failure Classification

Buffer Pool Policies

Shadow Paging

Write-Ahead Log

Logging Schemes

Checkpoints

STORAGE TYPES

Volatile Storage:

→ Data does not persist after power loss or program exit.
→ Examples: DRAM, SRAM

Non-volatile Storage:

→ Data persists after power loss and program exit.
→ Examples: HDD, SDD

Stable Storage:

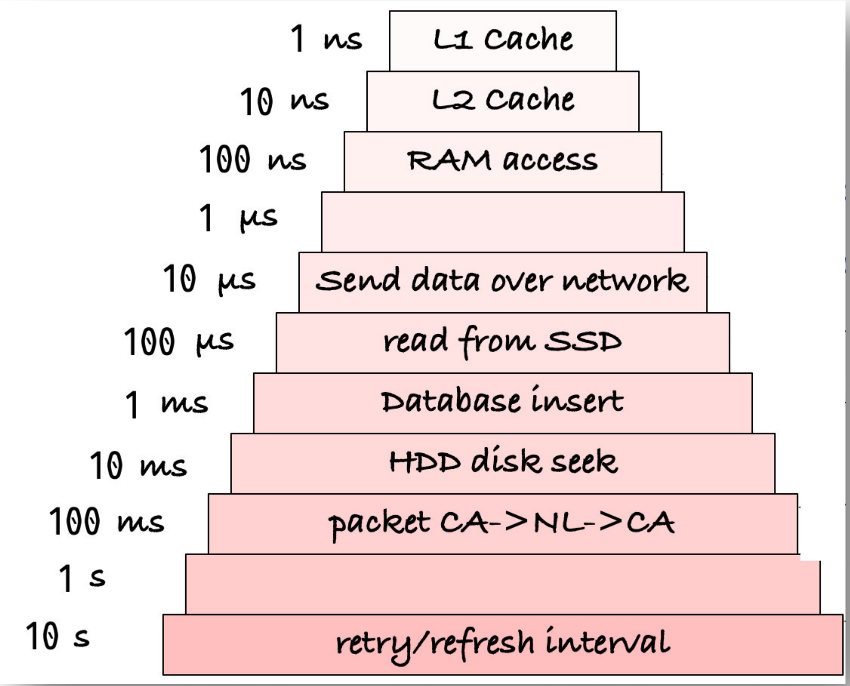
→ A non-existent form of non-volatile storage that survives all possible failures scenarios.

CRASH RECOVERY: INTUITION

Volatile storage where the buffer pool sits, and non-volatile storage below it.

We want high performance, hence want to write to volatile storage.

Allow dirty pages in the buffer pool for performance, with buffer pool replacement policy dictating flush to non-volatile storage.



FAILURE CLASSIFICATION

Type #1 – Transaction Failures

Type #2 – System Failures

Type #3 – Storage Media Failures

TRANSACTION FAILURES

Logical Errors:

→ Transaction cannot complete due to some 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).

SYSTEM FAILURES

Software Failure:

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

Hardware Failure:

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

→ Fail-stop Assumption: Non-volatile storage contents are assumed to not be corrupted by system crash.

STORAGE MEDIA FAILURE

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).

The recovery protocol can't recover from this! Database must be restored from an archived version.

OBSERVATION

The database's primary storage location is on non-volatile storage, but this is slower than volatile storage. Use volatile memory for faster access:

→ First copy target record into memory.
→ Perform the writes in memory.
→ Write dirty records back to disk.

The DBMS needs to ensure the following:

→ The changes for any txn are durable once the DBMS has told somebody that it committed.
→ No partial changes are durable if the txn aborted.

UNDO VS. REDO

Undo: The process of removing the effects of an incomplete or aborted txn.

Redo: The process of re-applying the effects of a committed txn for durability.

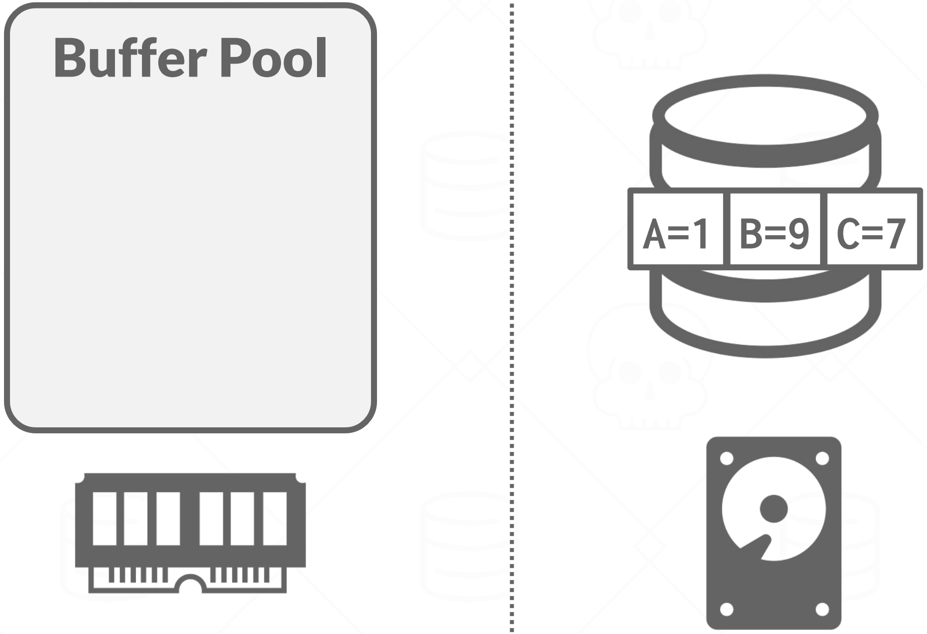
How the DBMS supports this functionality depends on how it manages the buffer pool …

BUFFER POOL

TIME

Schedule

T_1	T_2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(B)
	W(B)
	COMMIT
⋮	
ABORT	

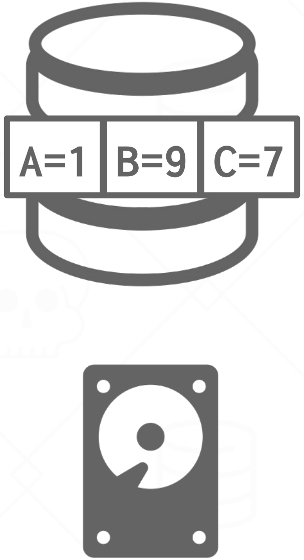


BUFFER POOL

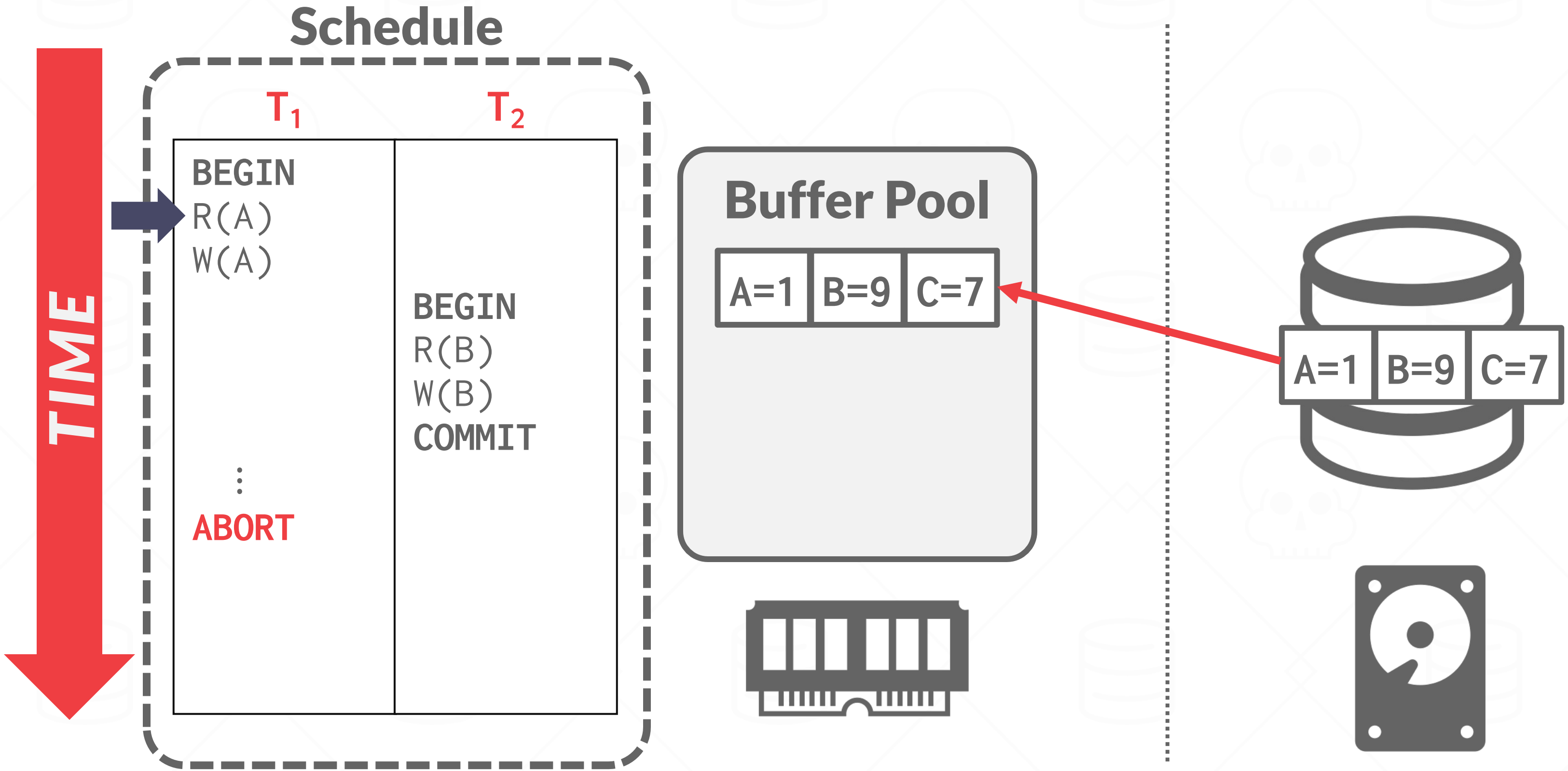
Schedule

TIME

T_1	T_2
BEGIN R(A) W(A)	
	BEGIN R(B) W(B) COMMIT
⋮ ABORT	



BUFFER POOL



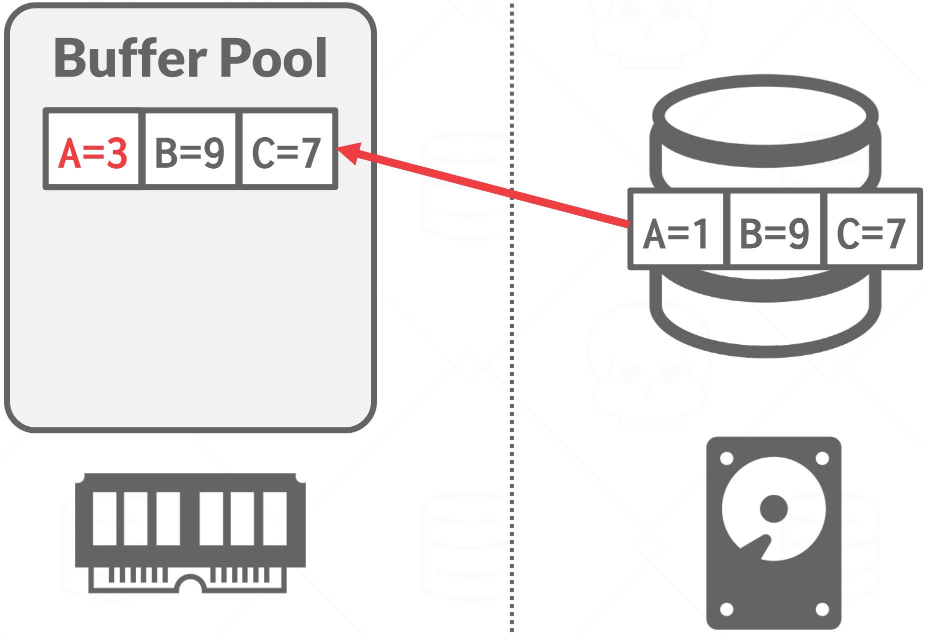
Schedule
TIME
T1
T2
BEGIN
R(A)
W(A)
⋮
ABORT
BEGIN
R(B)
W(B)
COMMIT
Buffer Pool
A=1
B=9
C=7
A=1
B=9
C=7

BUFFER POOL

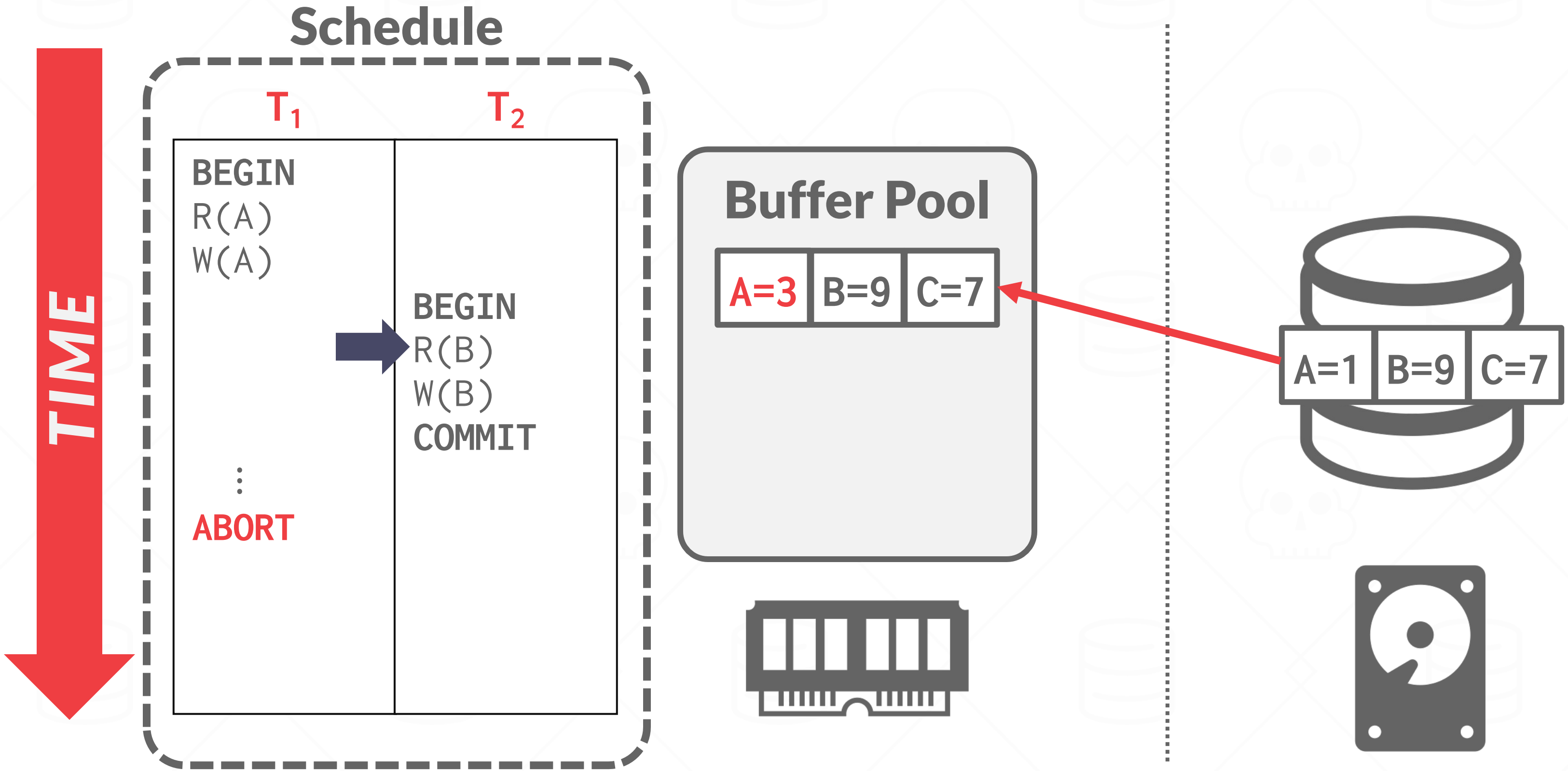
Schedule

TIME

T_1	T_2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(B)
	W(B)
	COMMIT
⋮	
ABORT	

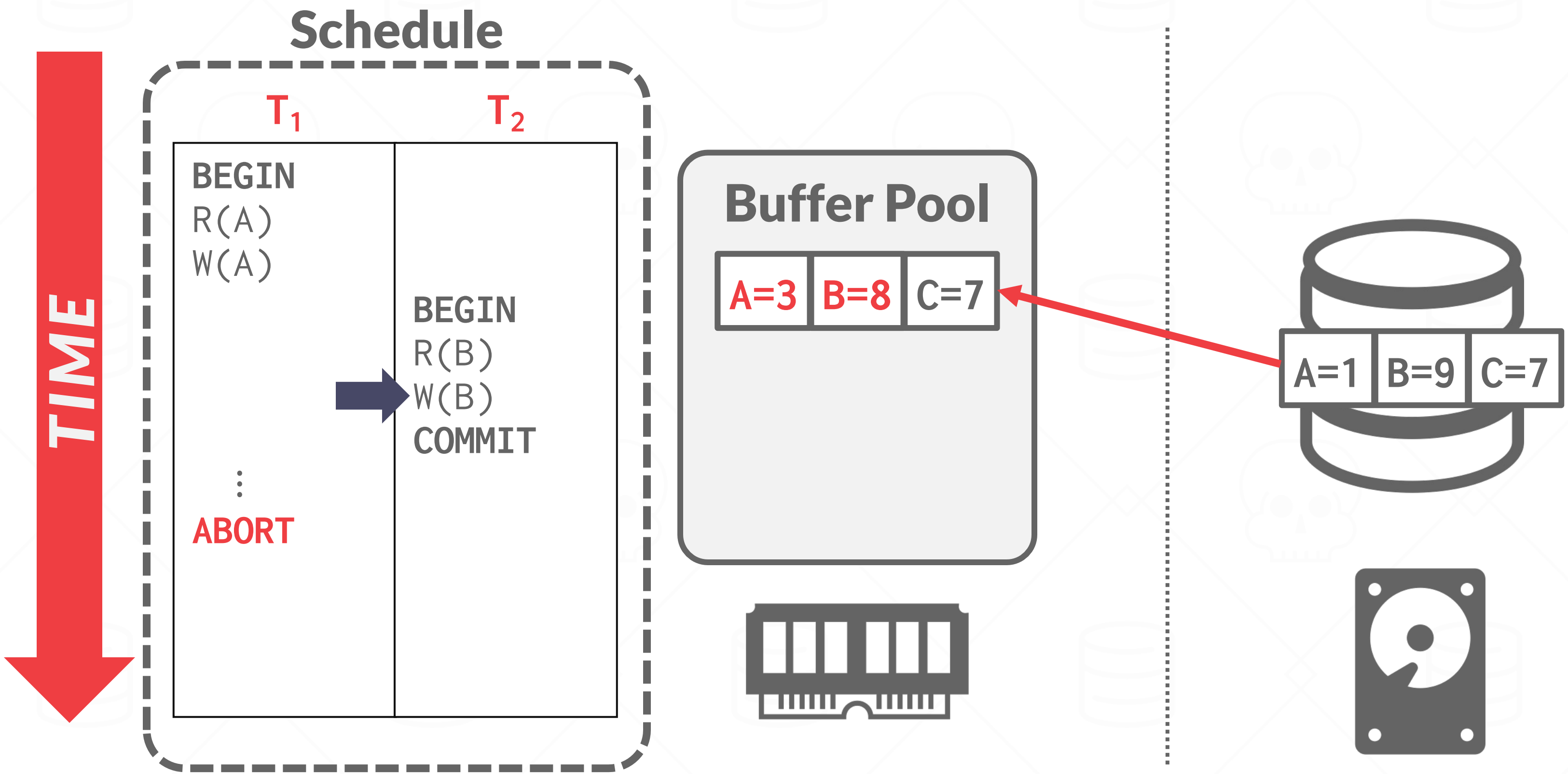


BUFFER POOL

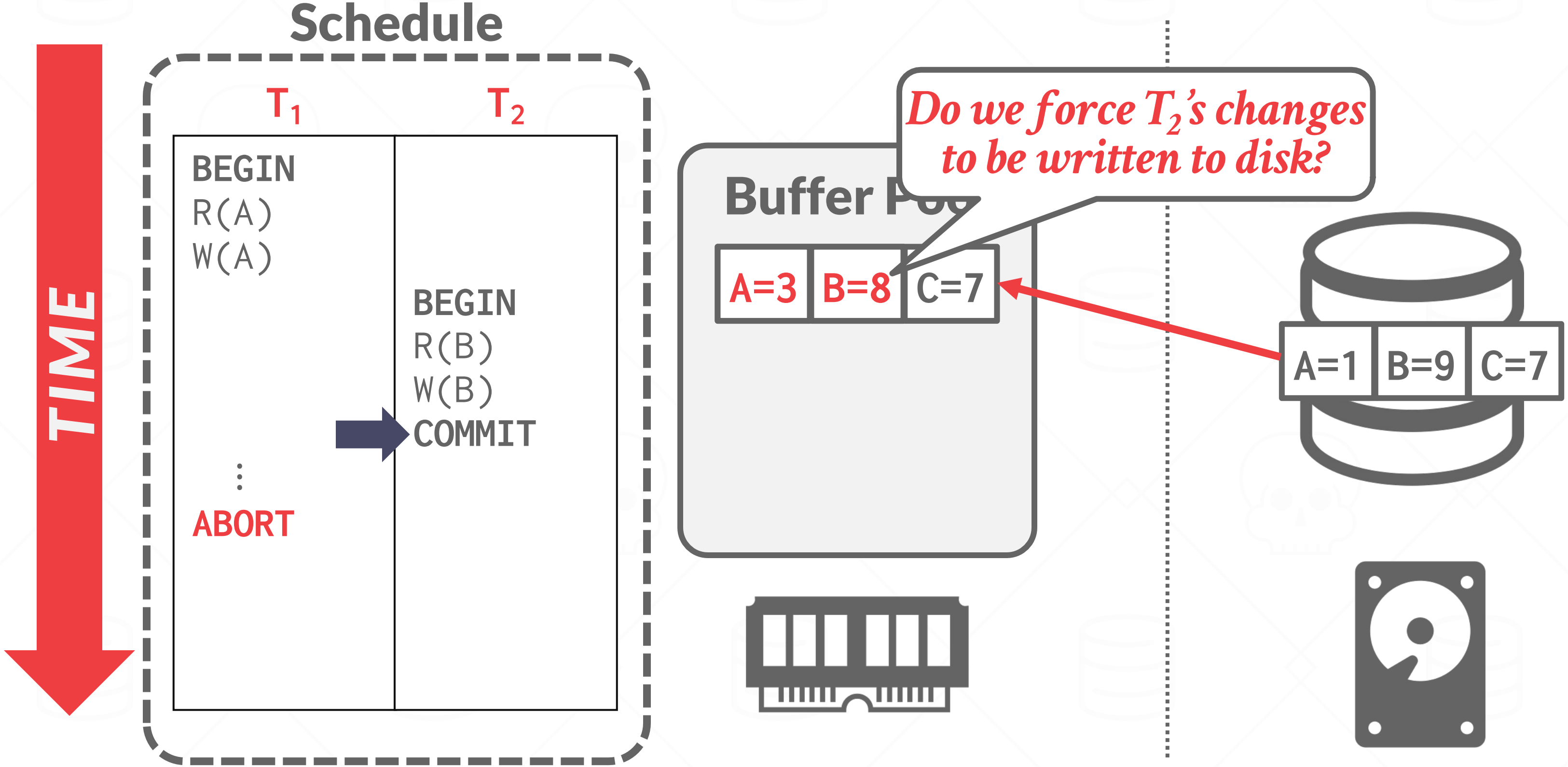


Schedule
TIME
T_1
T_2
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
⋮
ABORT
Buffer Pool
A=3
B=9
C=7
A=1
B=9
C=7

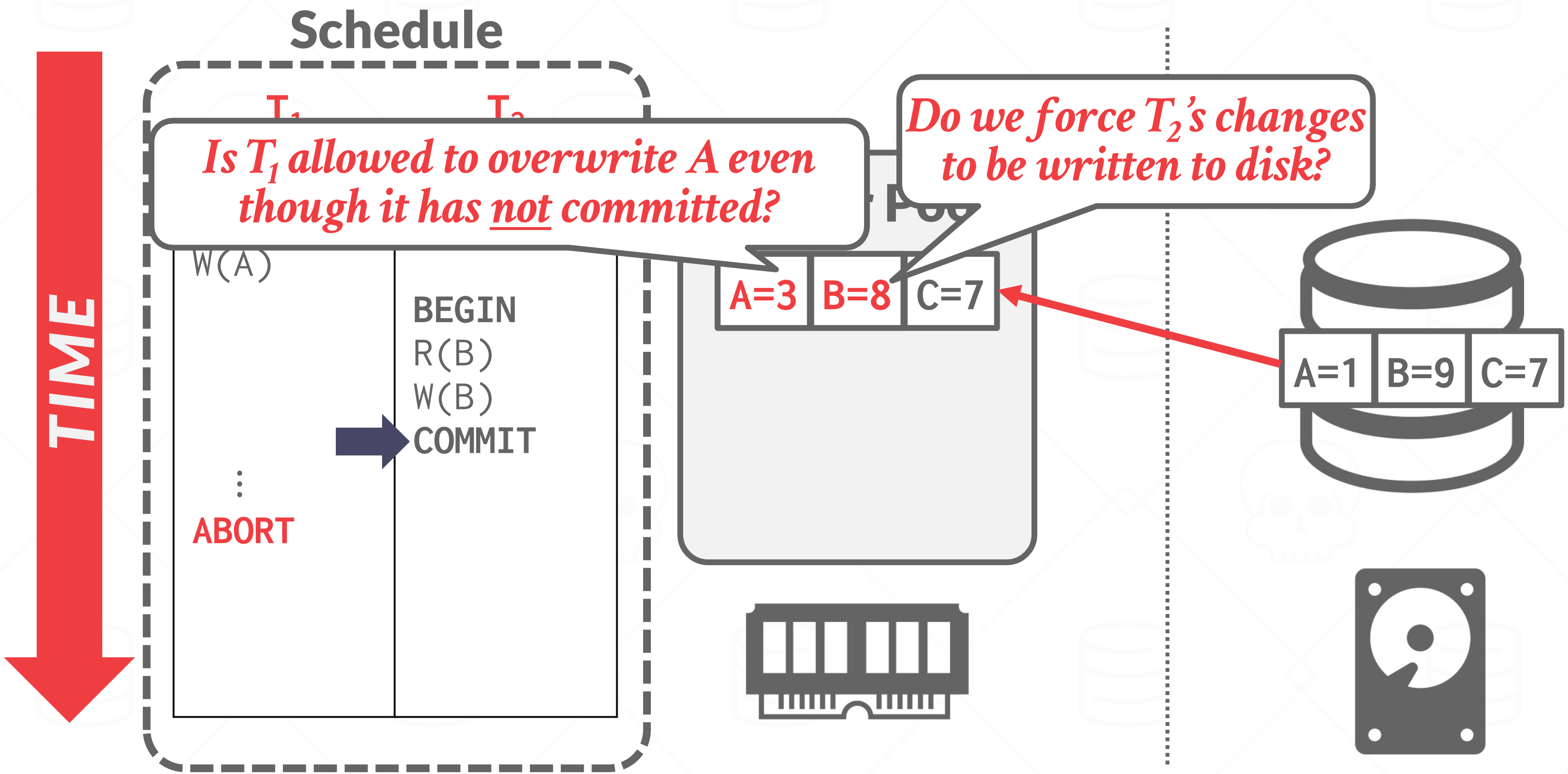
BUFFER POOL


Schedule
TIME
T1
T2
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
⋮
ABORT
Buffer Pool
A=3
B=8
C=7
A=1
B=9
C=7

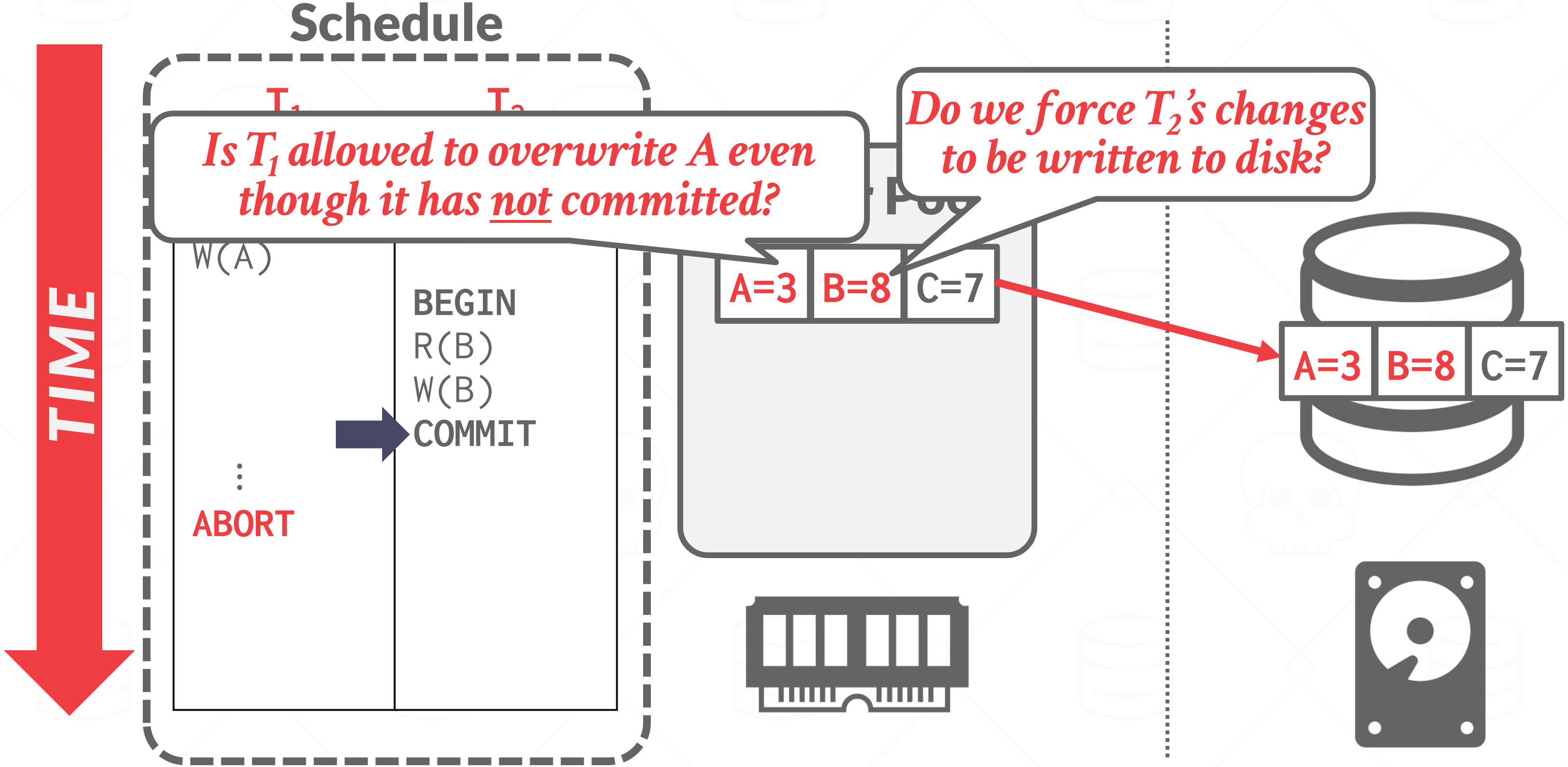
BUFFER POOL


Schedule
TIME
T_1
T_2
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
⋮
ABORT
Buffer Poo
A=3
B=8
C=7
Do we force T_2's changes to be written to disk?
A=1
B=9
C=7

BUFFER POOL

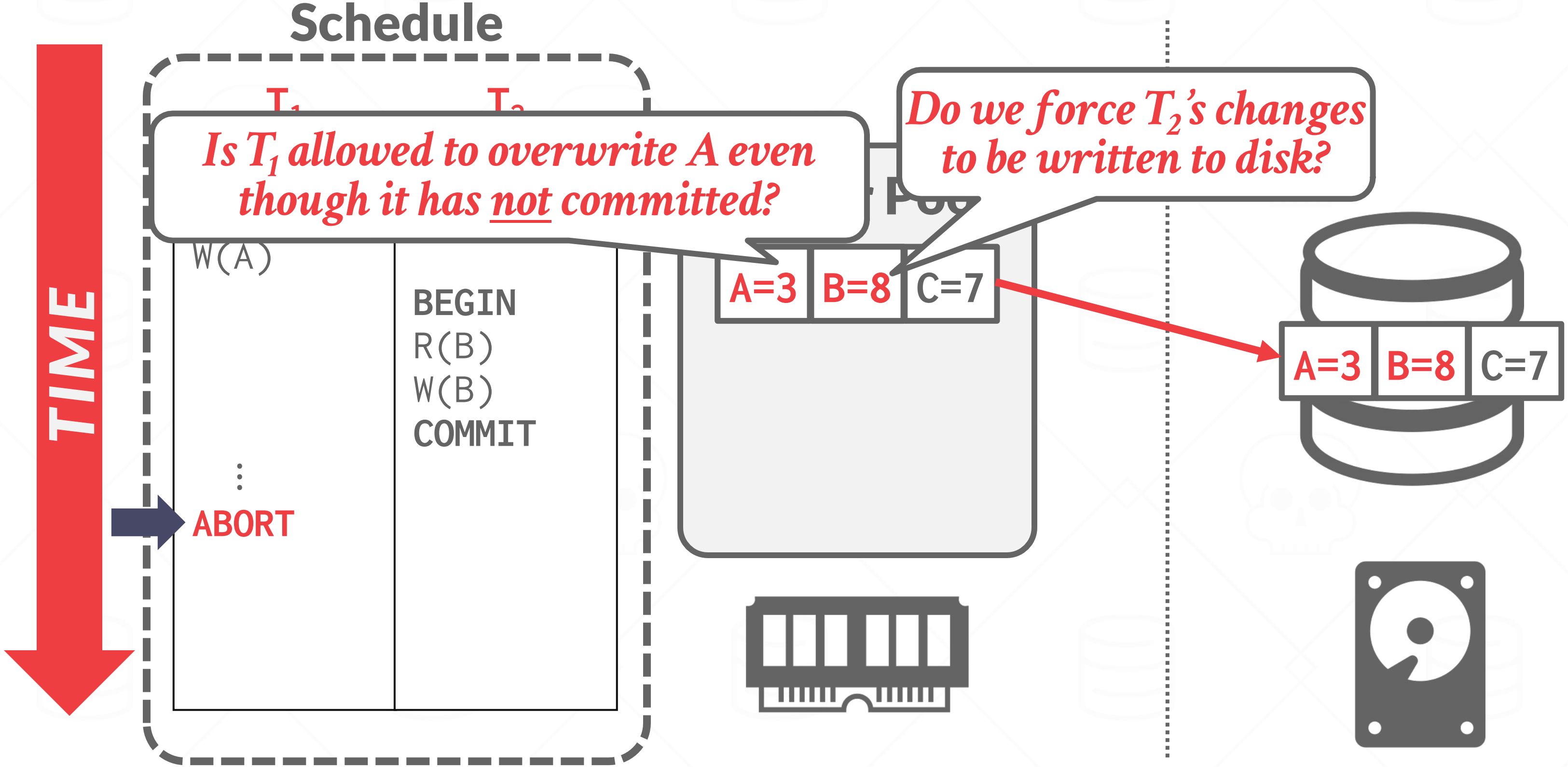

Schedule
TIME
T1
T2
W(A)
BEGIN
R(B)
W(B)
COMMIT
⋮
ABORT
Is T1 allowed to overwrite A even though it has not committed?
Do we force T2's changes to be written to disk?
A=3
B=8
C=7
A=1
B=9
C=7

BUFFER POOL

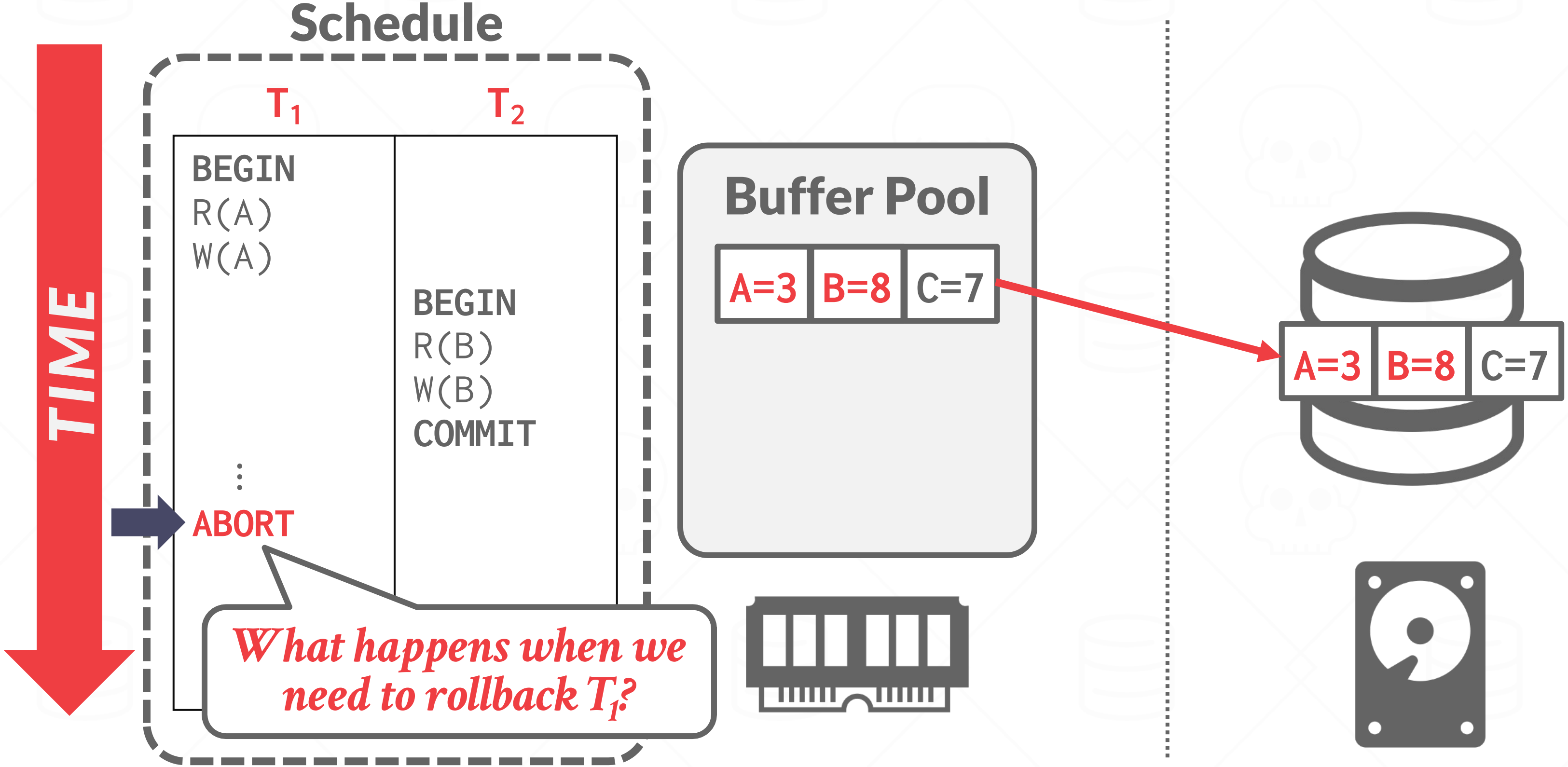


Schedule
T1
T2
Is T1 allowed to overwrite A even though it has not committed?
W(A)
BEGIN
R(B)
W(B)
COMMIT
⋮
ABORT
TIME
Do we force T2's changes to be written to disk?
A=3
B=8
C=7
A=3
B=8
C=7

BUFFER POOL


Schedule
TIME
T1
T2
Is T1 allowed to overwrite A even though it has not committed?
W(A)
BEGIN
R(B)
W(B)
COMMIT
⋮
ABORT
Do we force T2's changes to be written to disk?
A=3
B=8
C=7
A=3
B=8
C=7

BUFFER POOL


Schedule
TIME
T1
T2
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
⋮
ABORT
What happens when we need to rollback T1?
Buffer Pool
A=3
B=8
C=7
A=3
B=8
C=7

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.

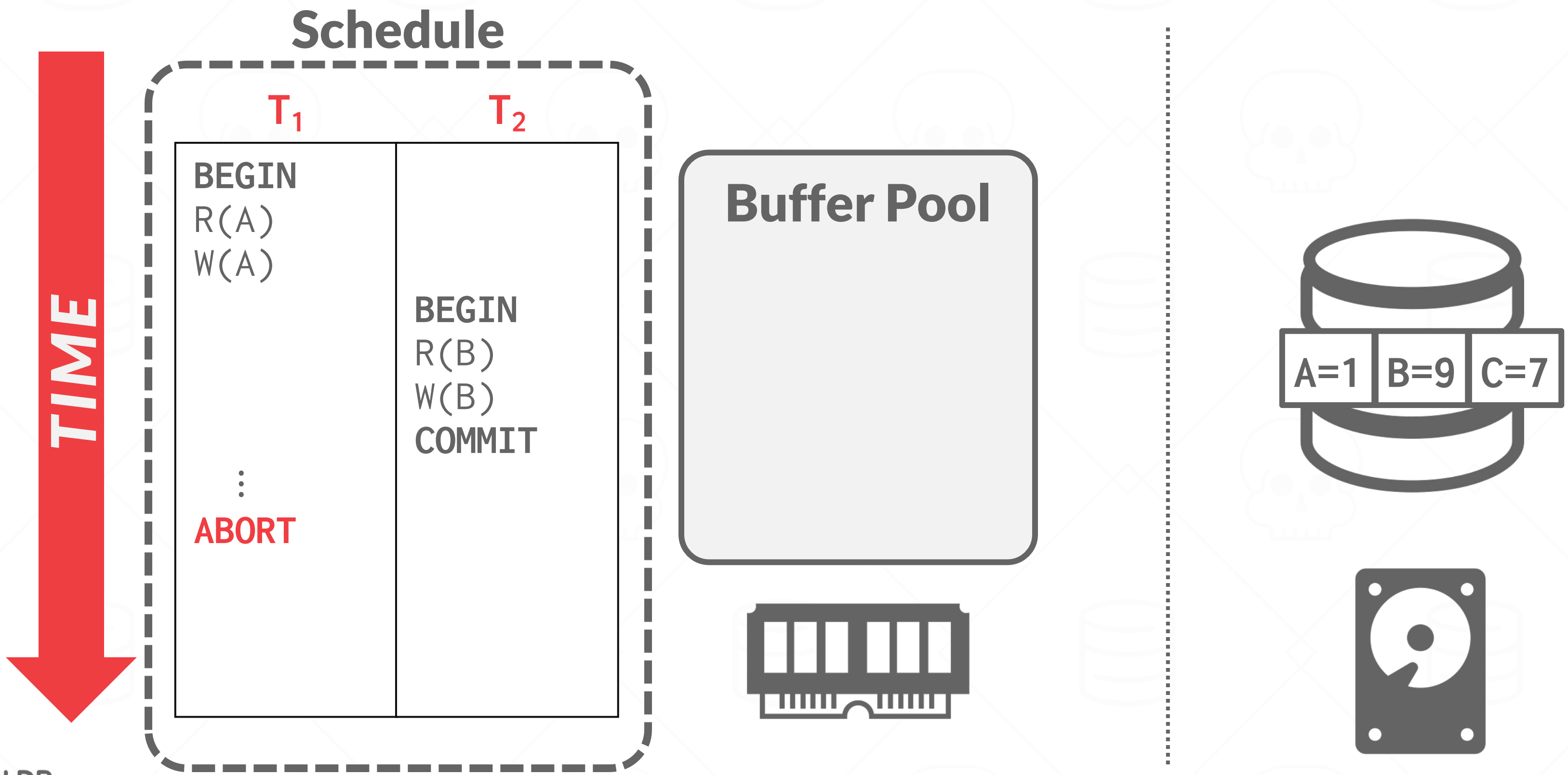
FORCE POLICY

Whether the DBMS requires that all updates made by a txn are reflected on non-volatile storage before the txn can commit.

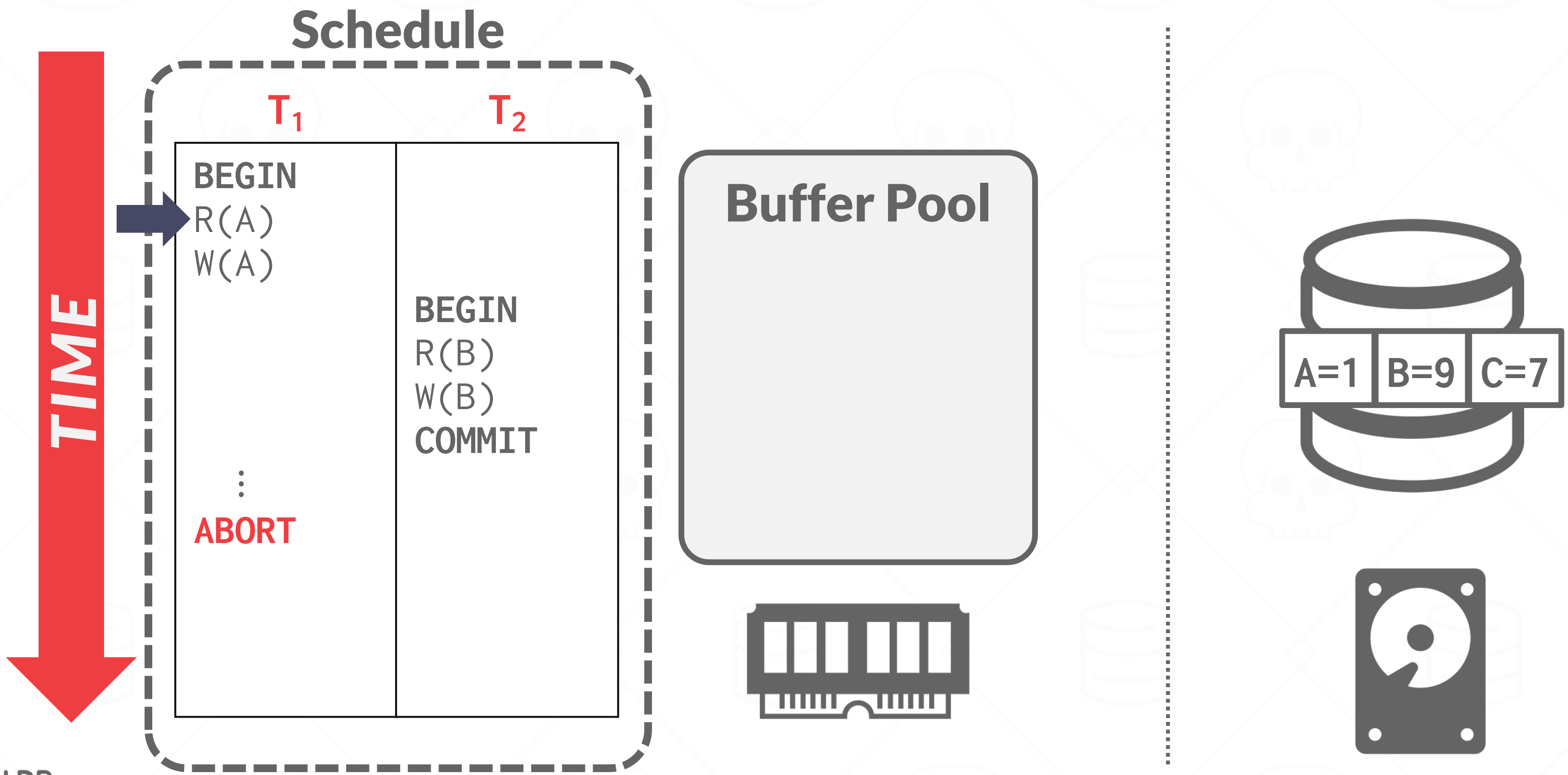
FORCE: Is required.

NO-FORCE: Is not required.

NO-STEAL + FORCE



NO-STEAL + FORCE

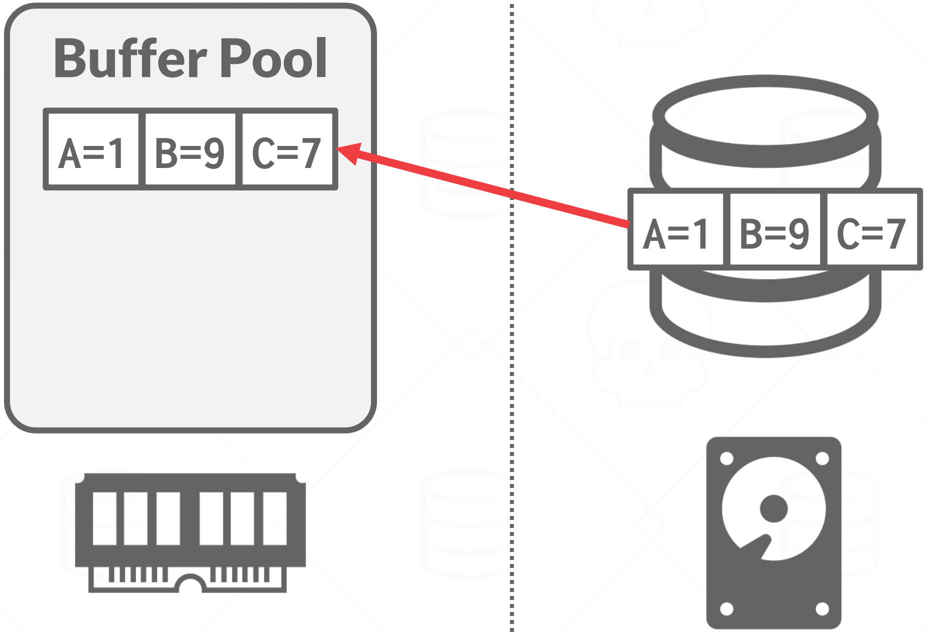


NO-STEAL + FORCE

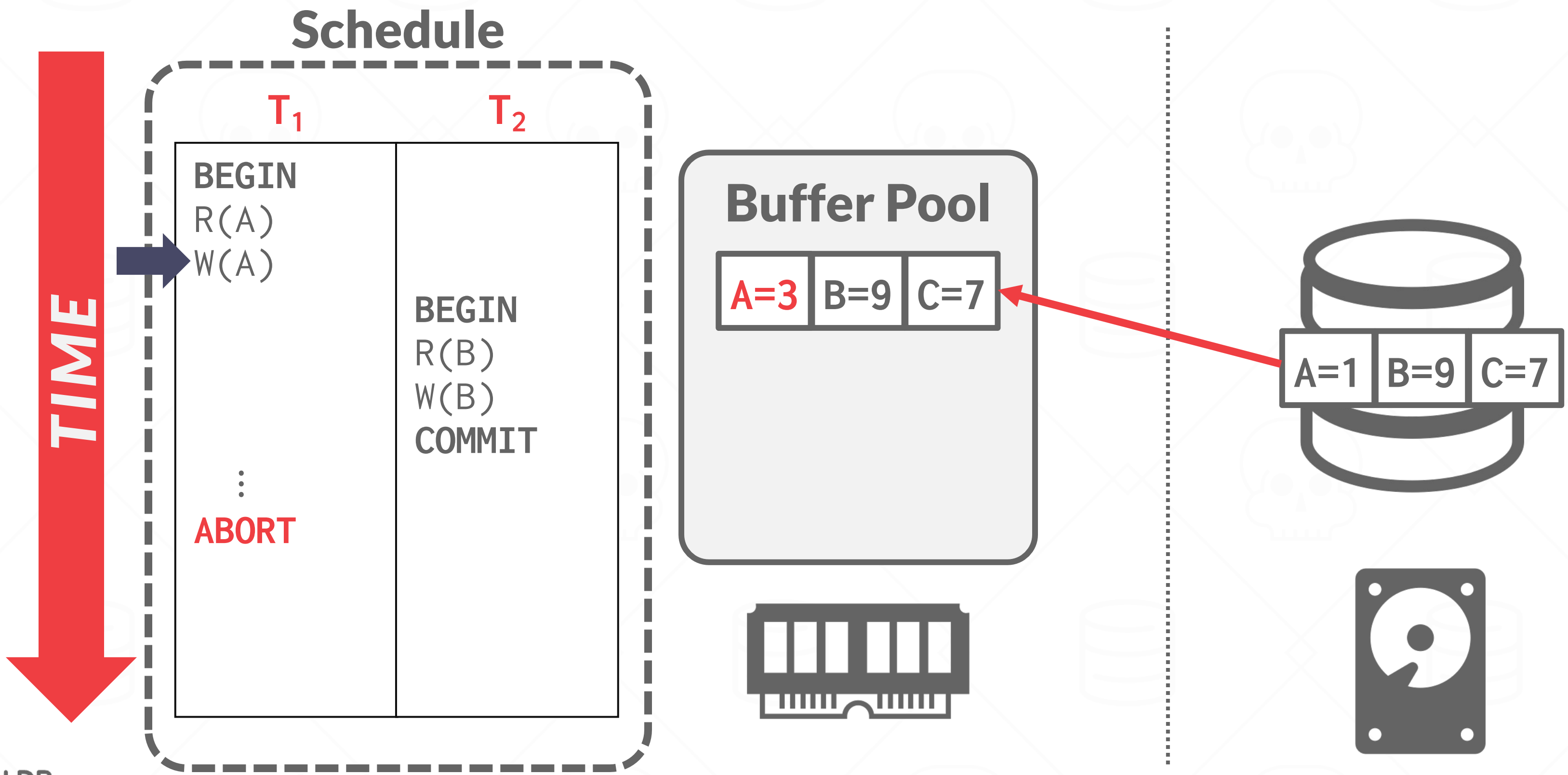
Schedule

TIME

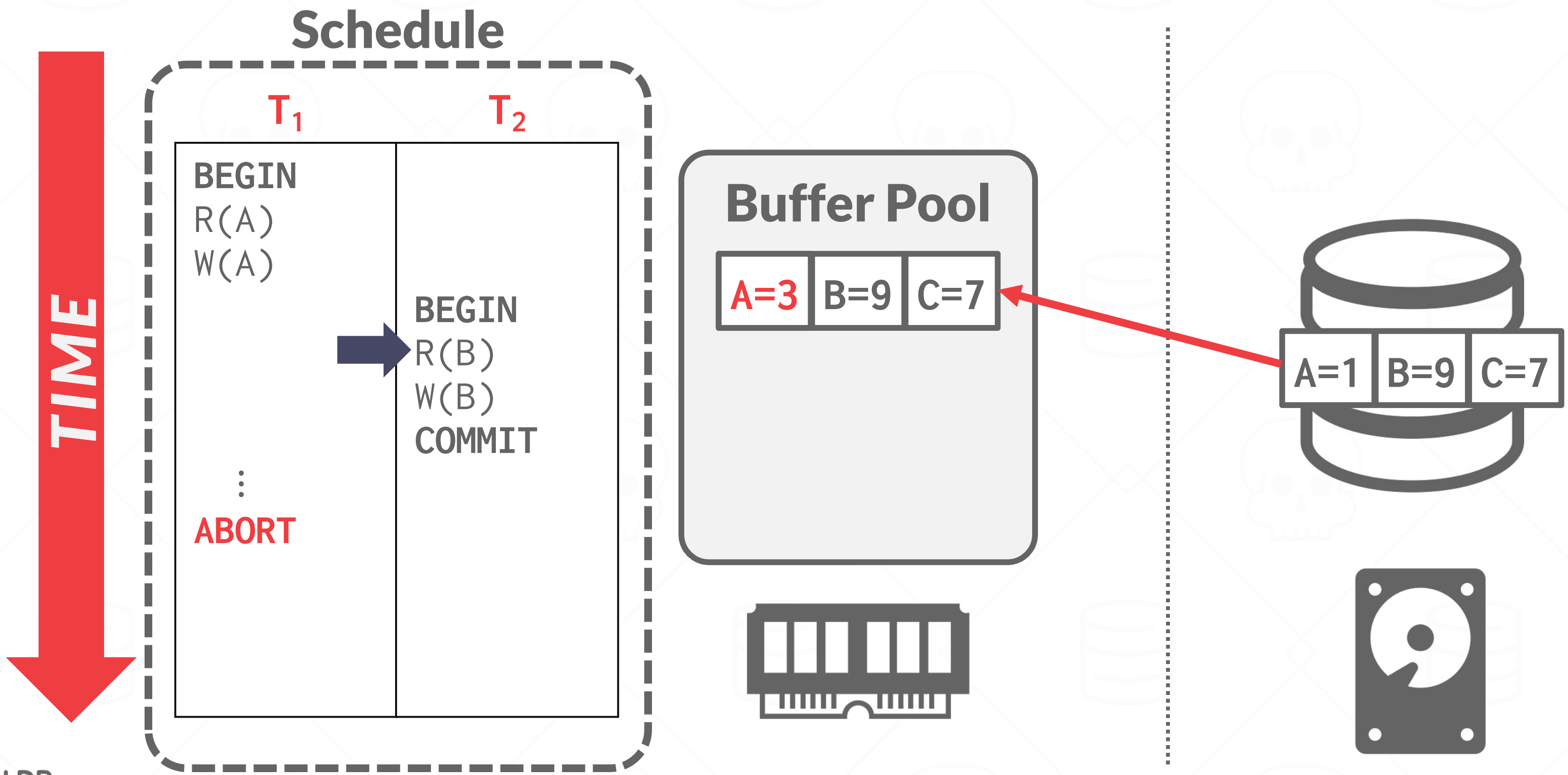
T_1	T_2
BEGIN R(A) W(A)	
	BEGIN R(B) W(B) COMMIT
⋮ ABORT	



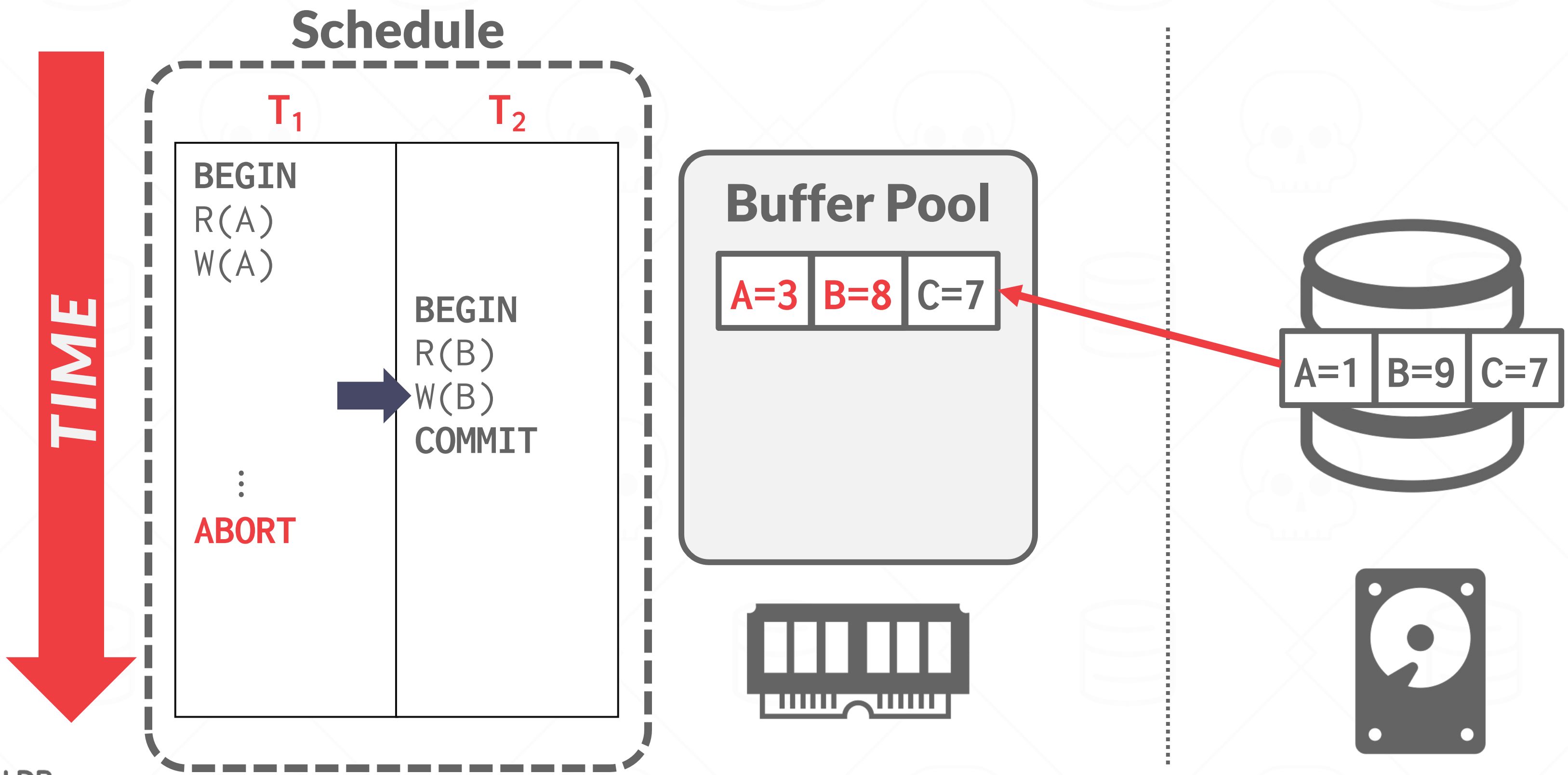
NO-STEAL + FORCE



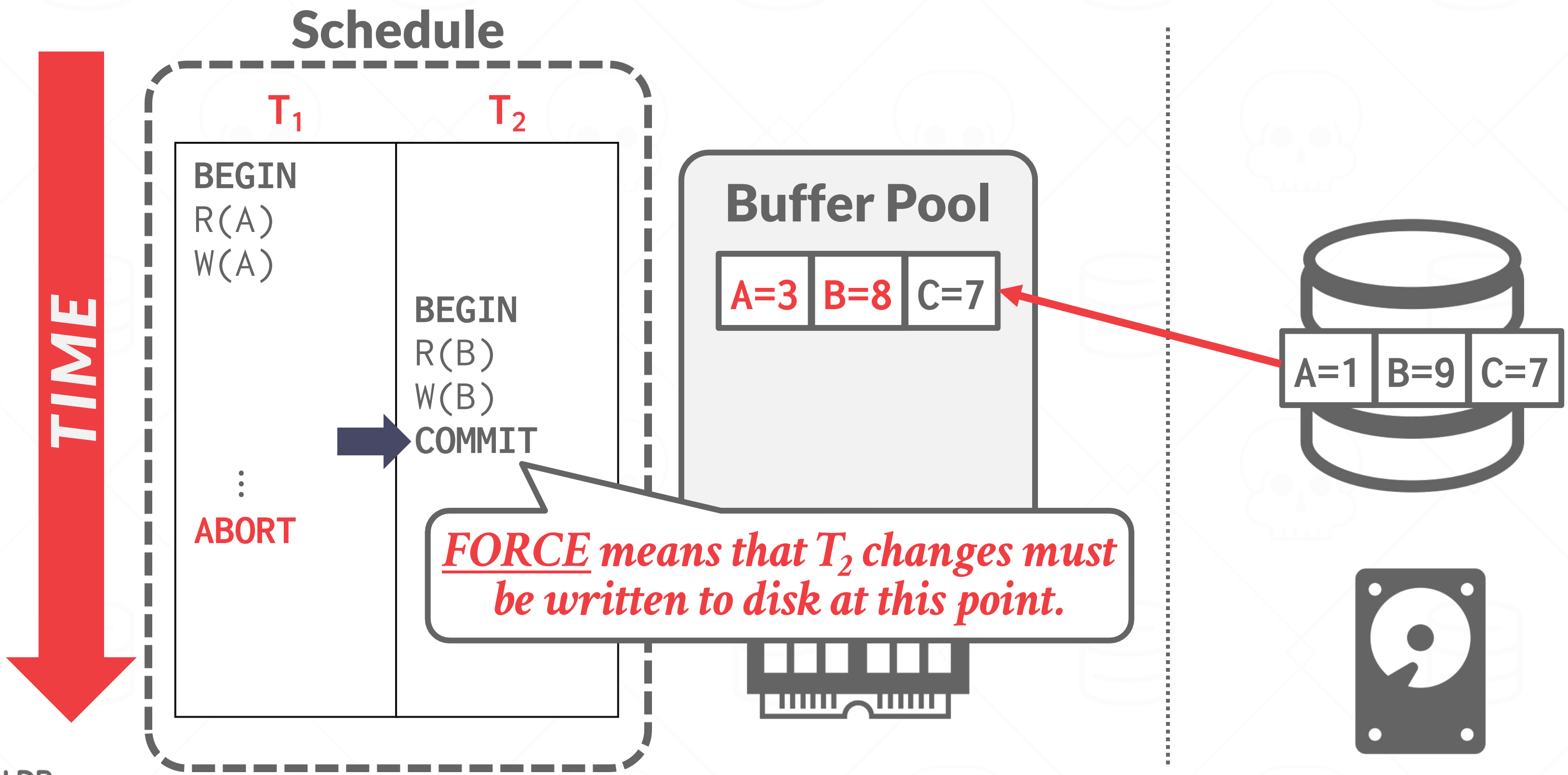
NO-STEAL + FORCE



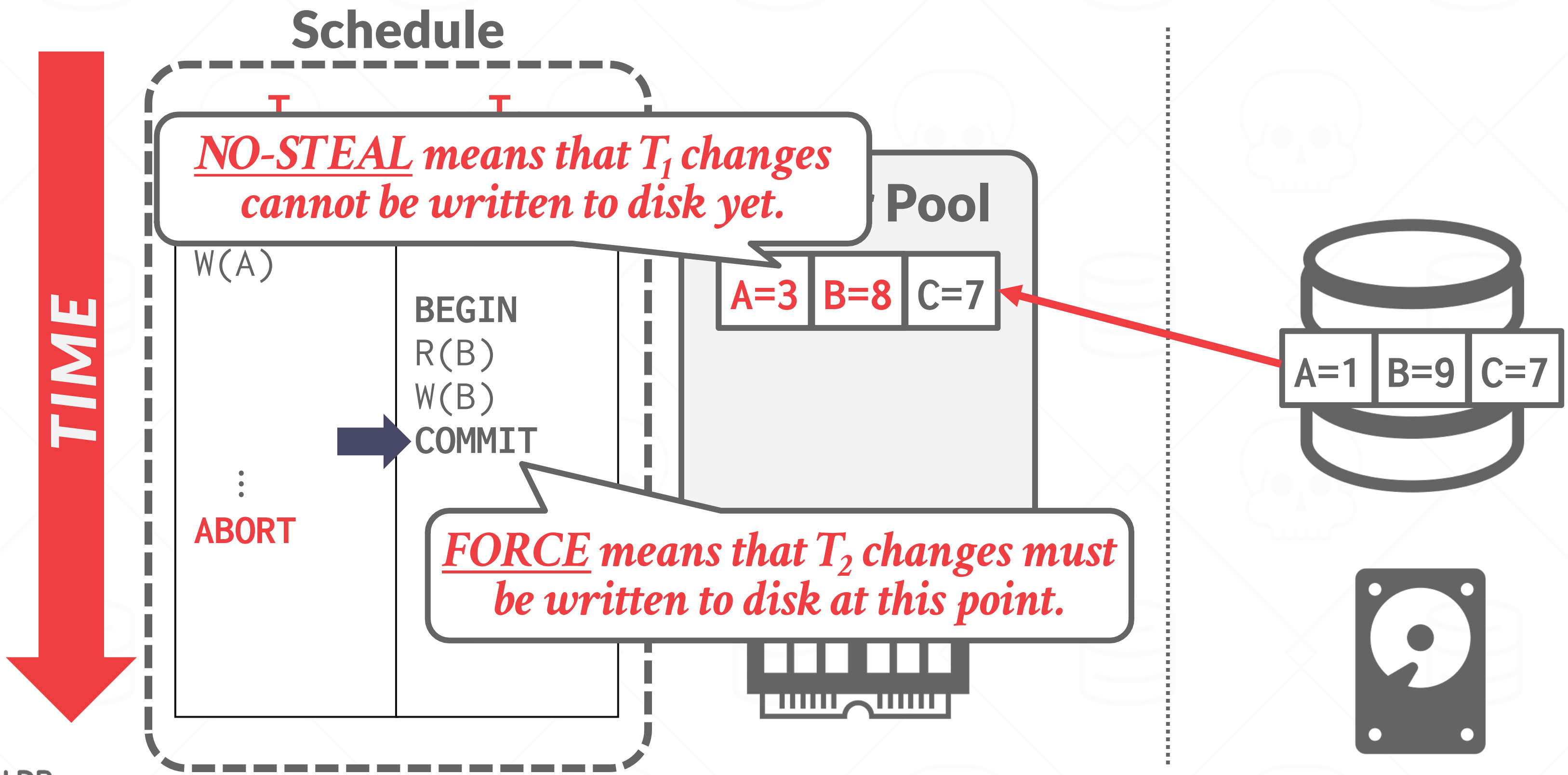
NO-STEAL + FORCE



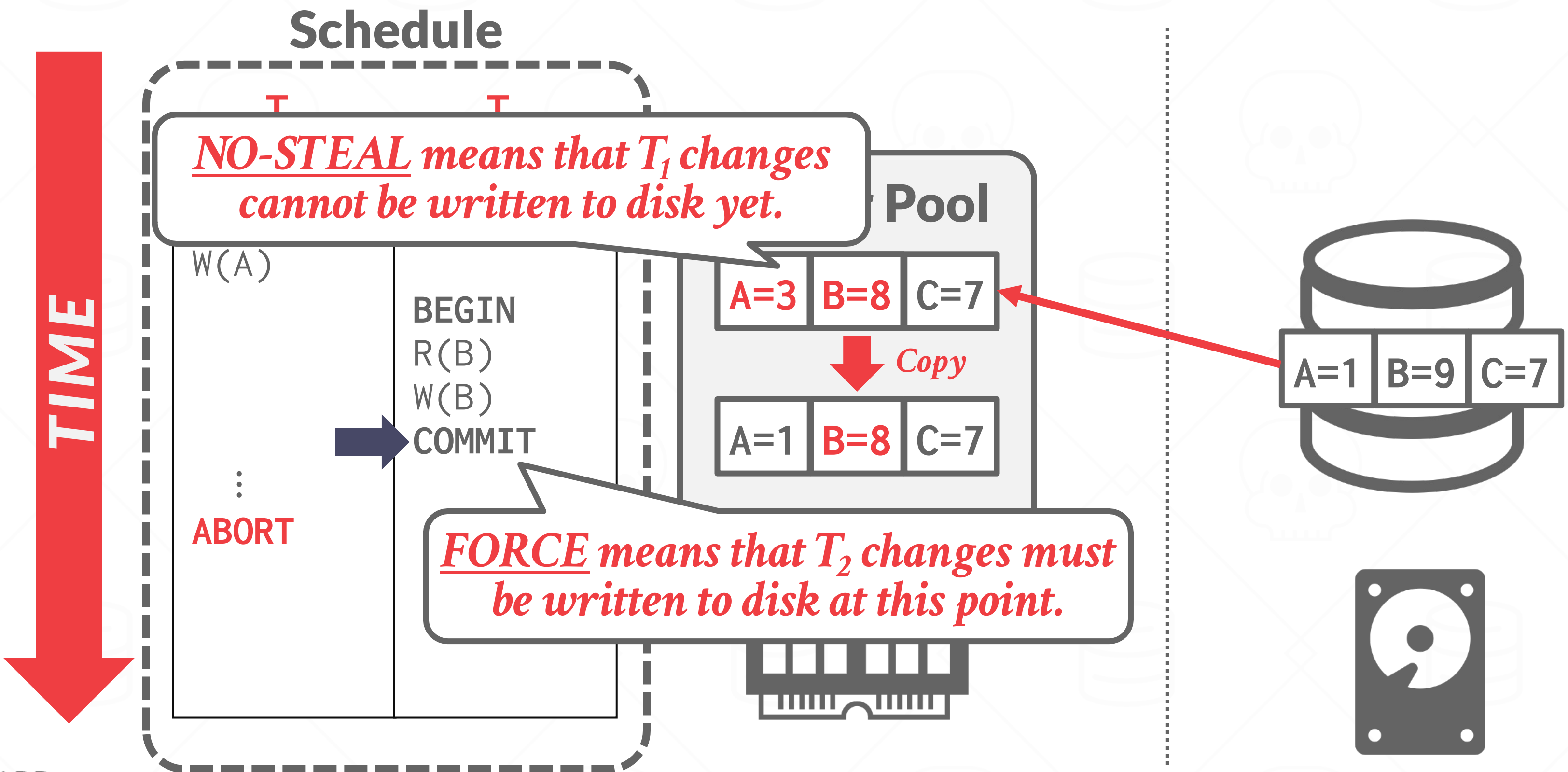
NO-STEAL + FORCE



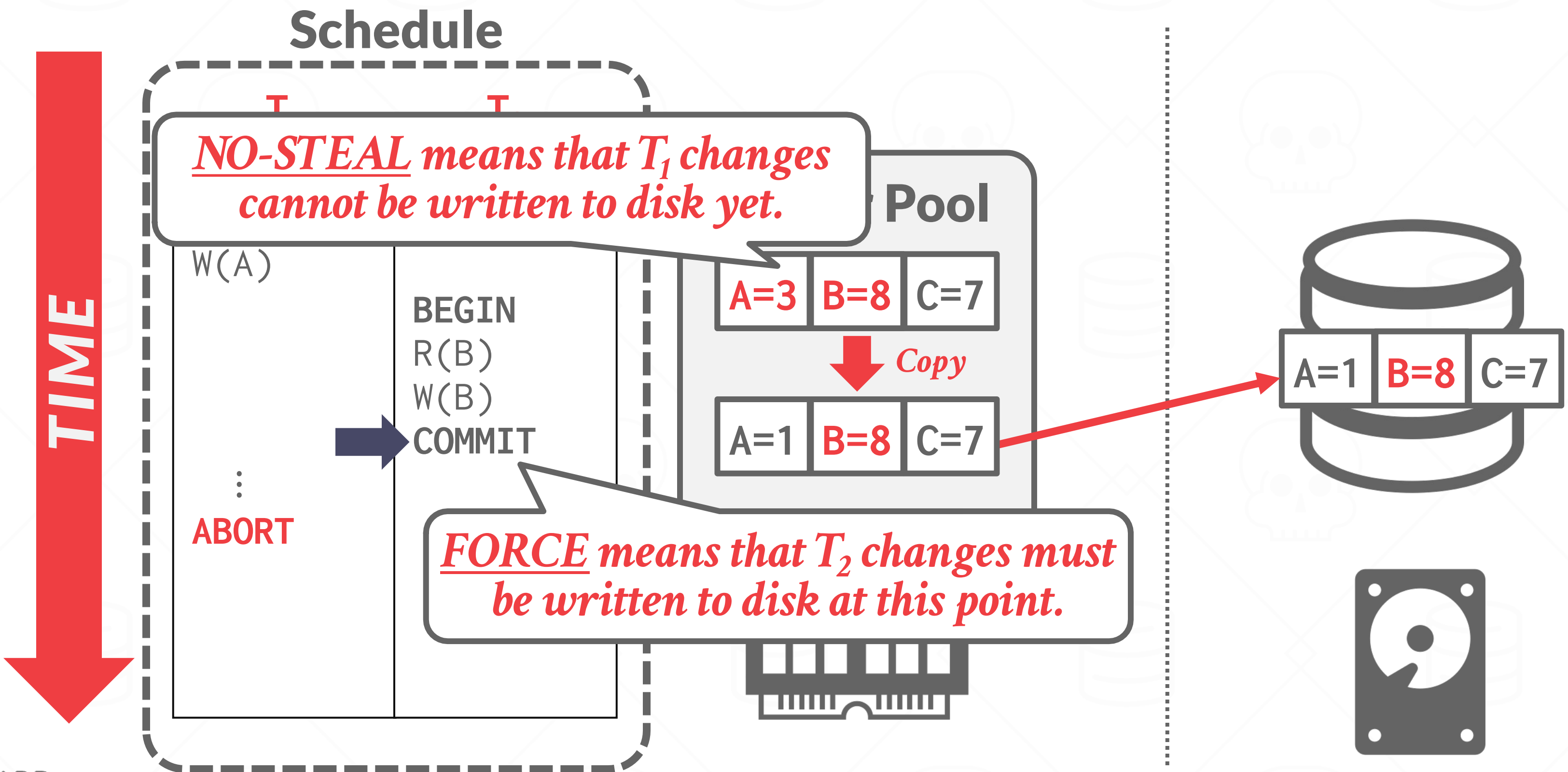
NO-STEAL + FORCE



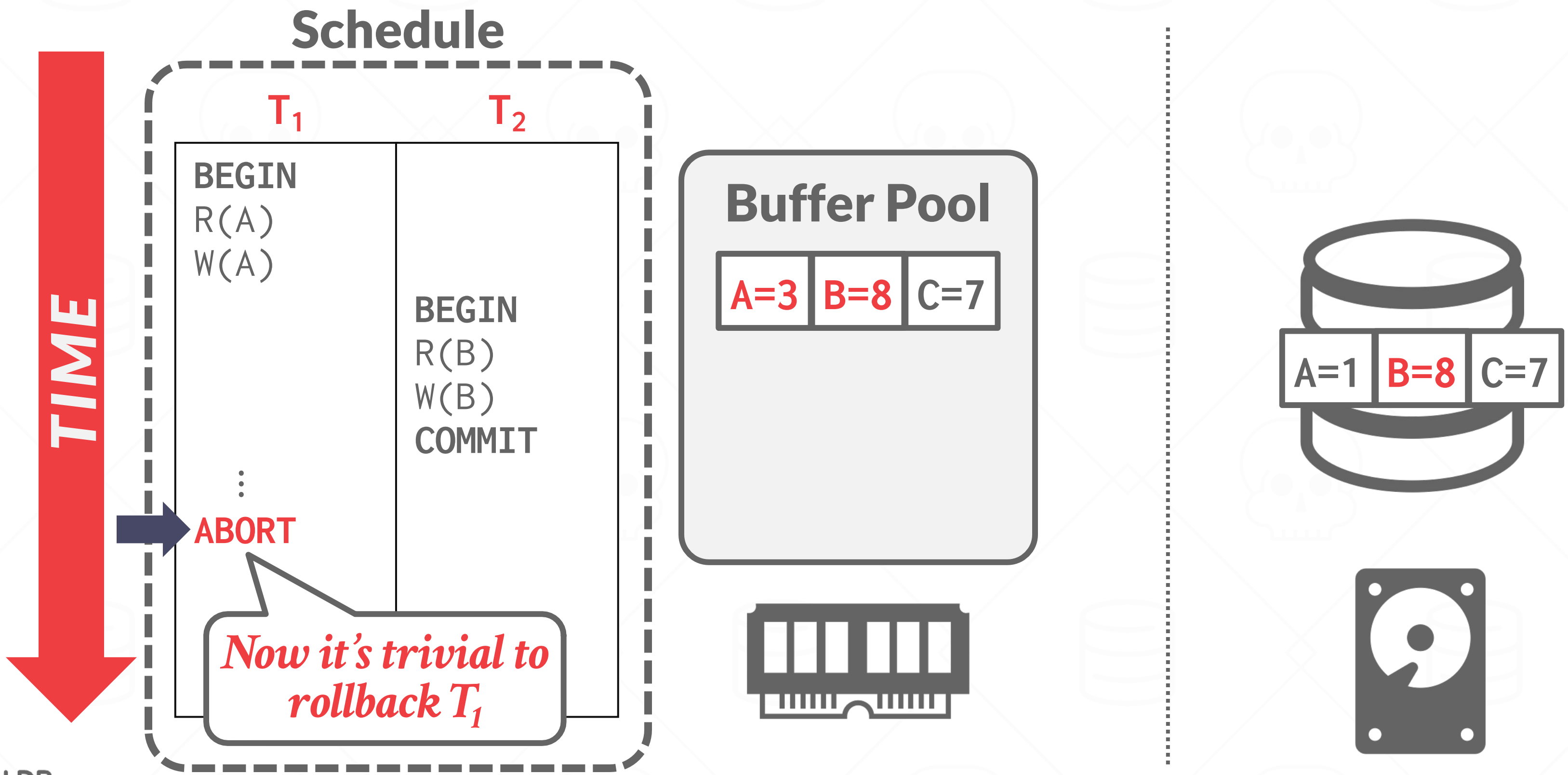
NO-STEAL + FORCE



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 (assuming atomic hardware writes).

Previous example cannot support **write sets** that exceed the amount of physical memory available.

SHADOW PAGING

Instead of copying the entire database, the DBMS copies pages on write to create two versions:

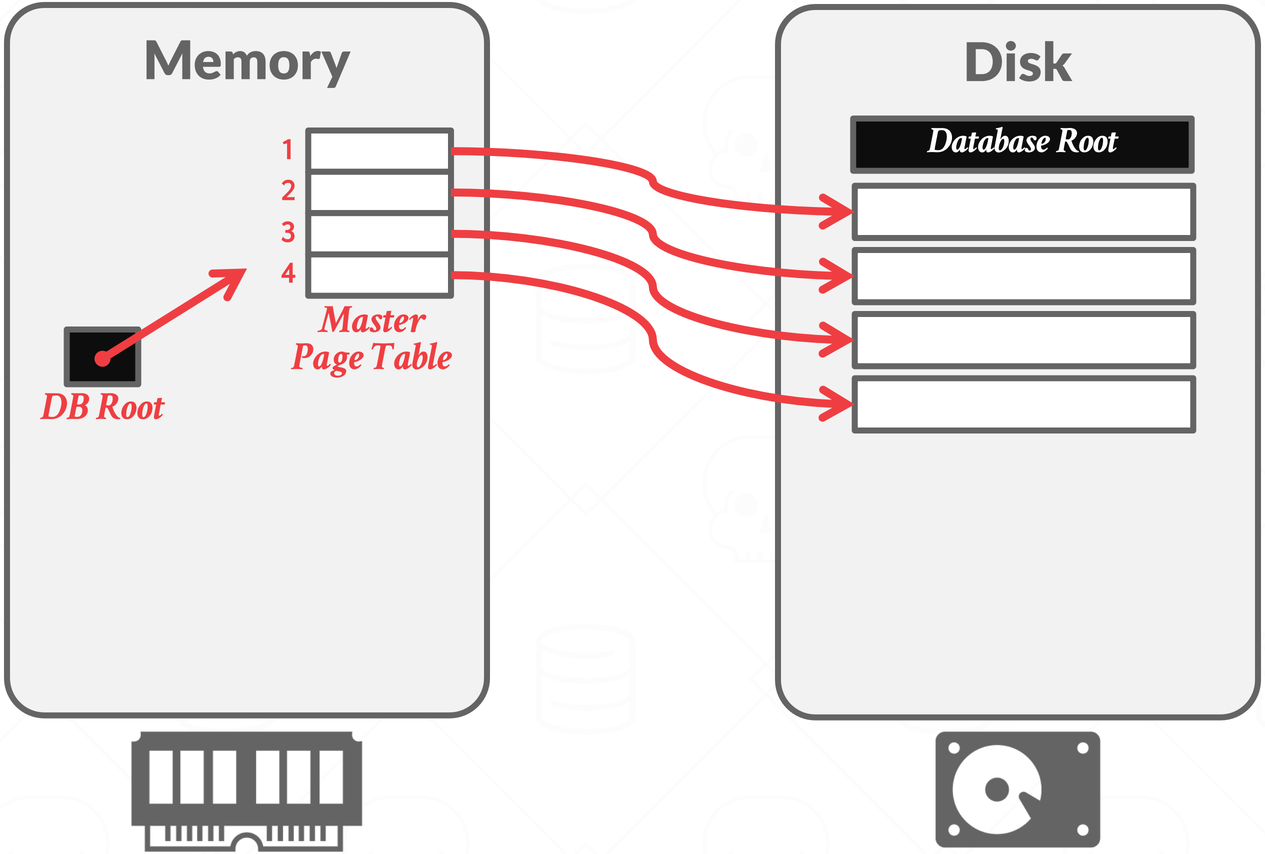
→ Master: Contains only changes from committed txns.

→ Shadow: Temporary database with changes made from uncommitted txns.

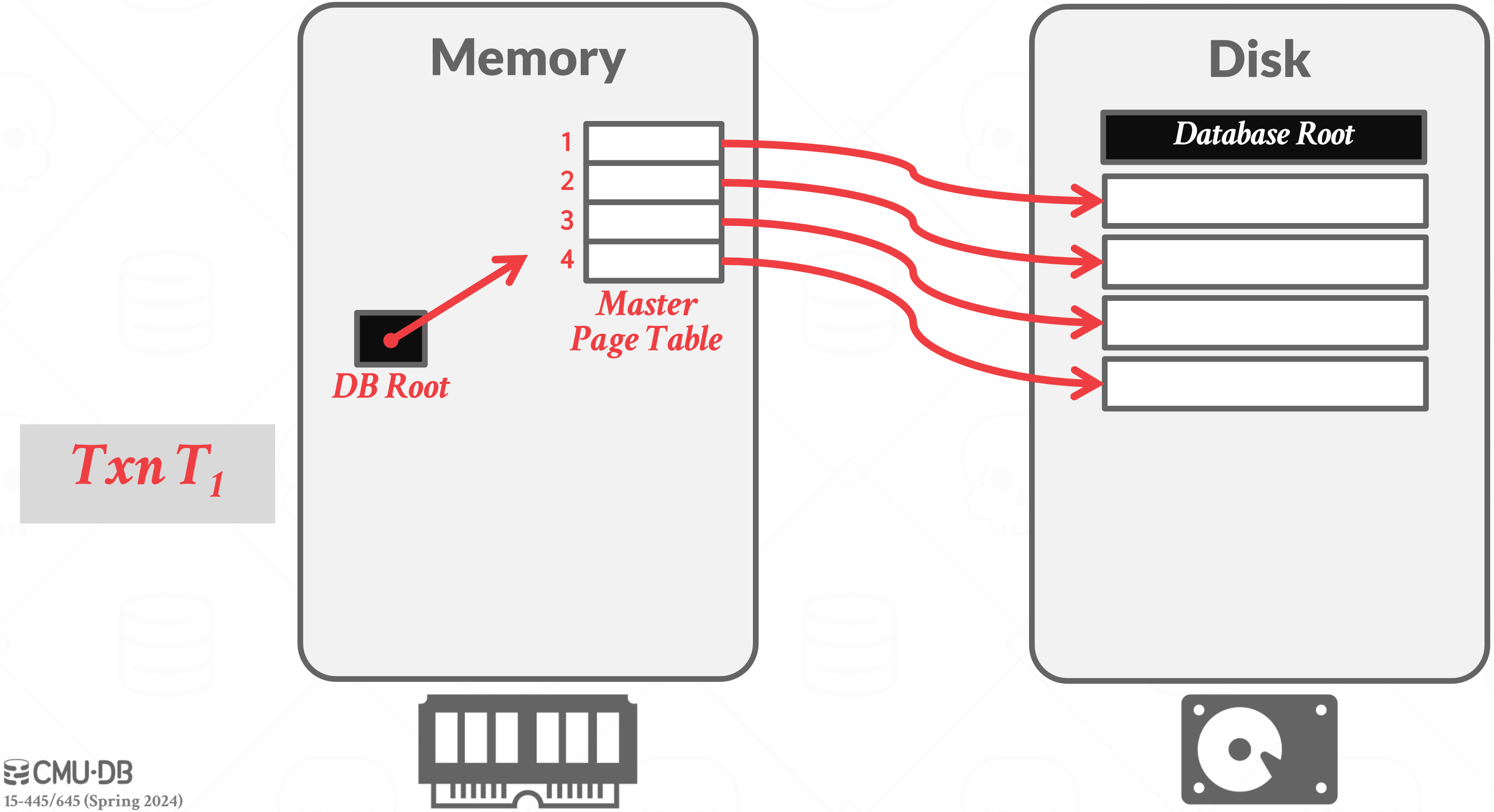
To install updates when a txn commits, overwrite the root so it points to the shadow, thereby swapping the master and shadow.

Buffer Pool Policy: **NO-STEAL** + **FORCE**

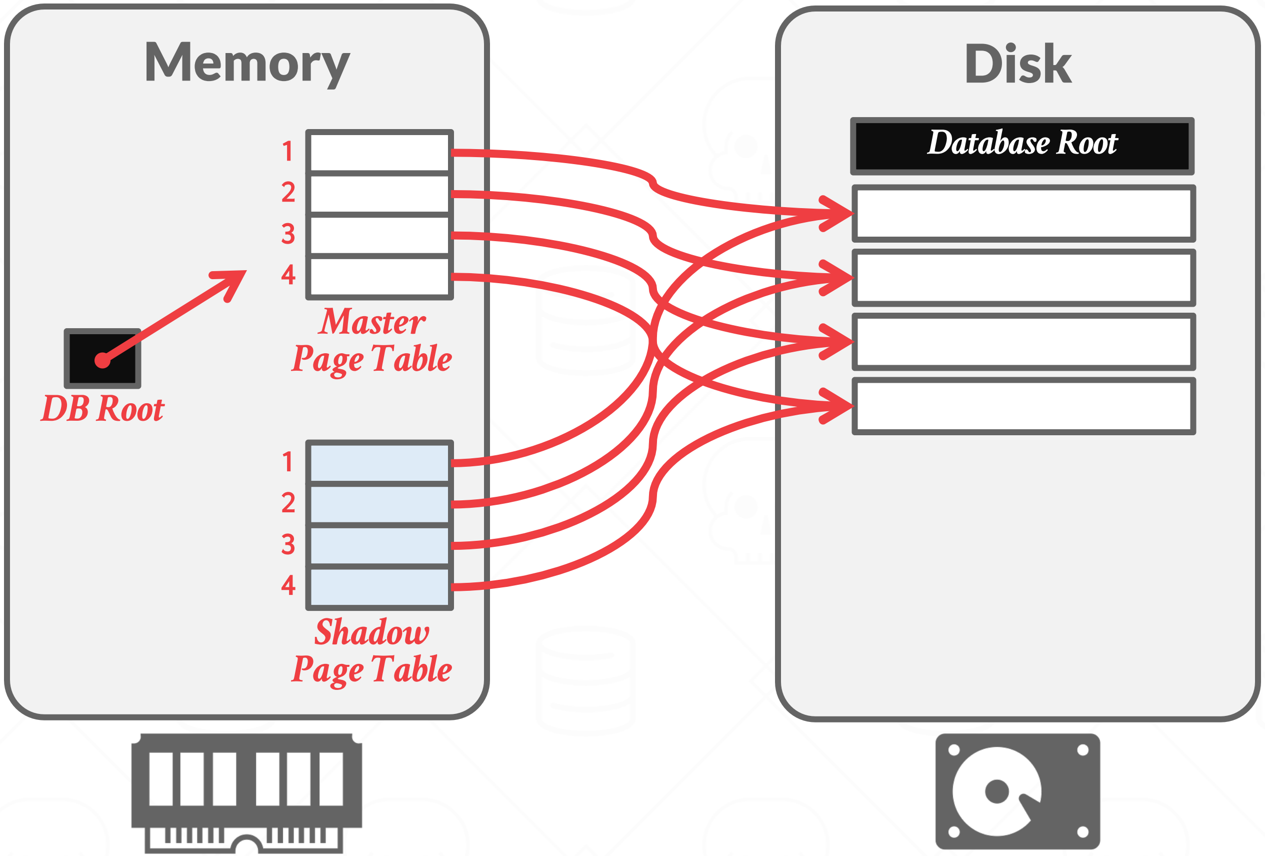
SHADOW PAGING – EXAMPLE



SHADOW PAGING – EXAMPLE

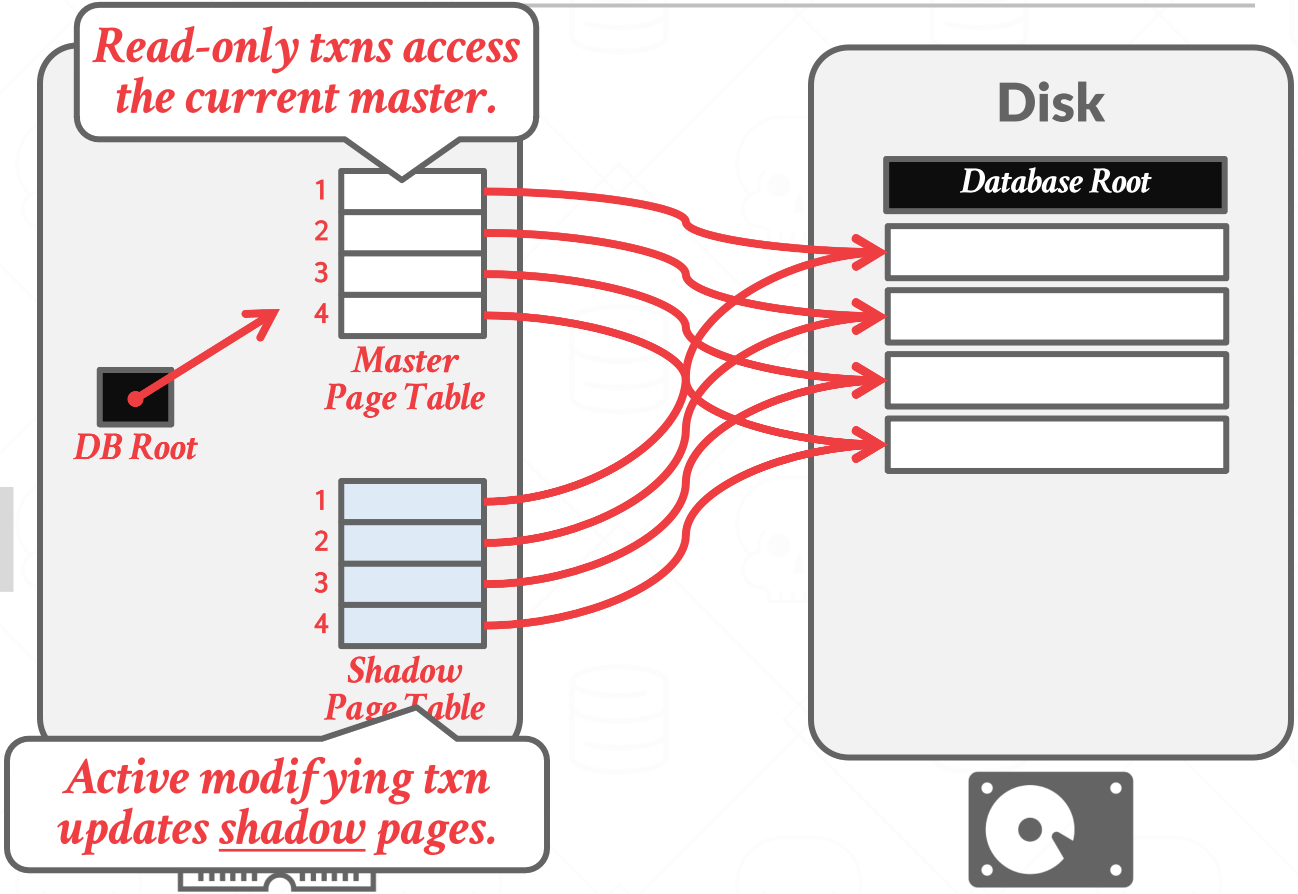


SHADOW PAGING – EXAMPLE



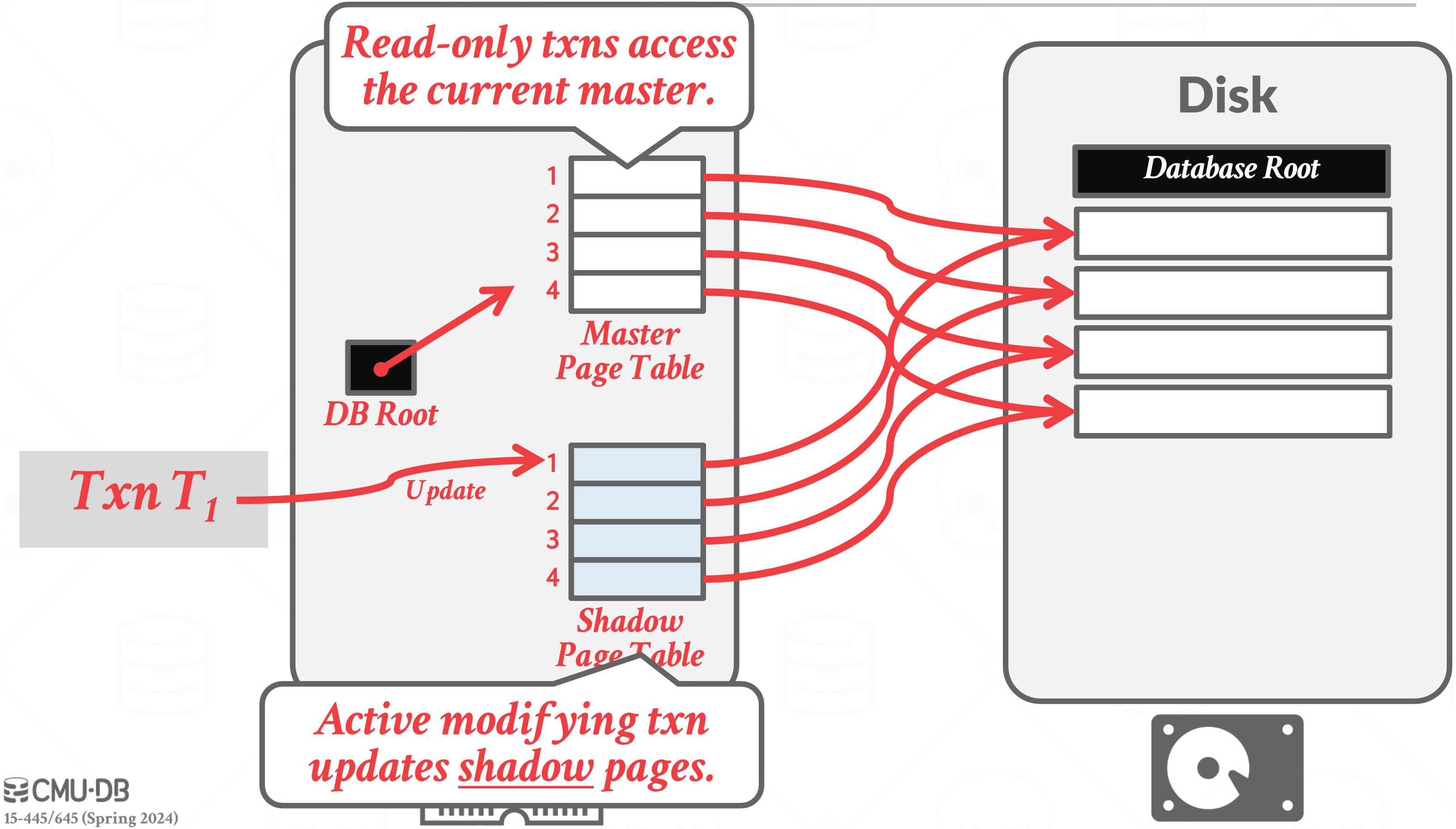
Txn T_1

SHADOW PAGING – EXAMPLE

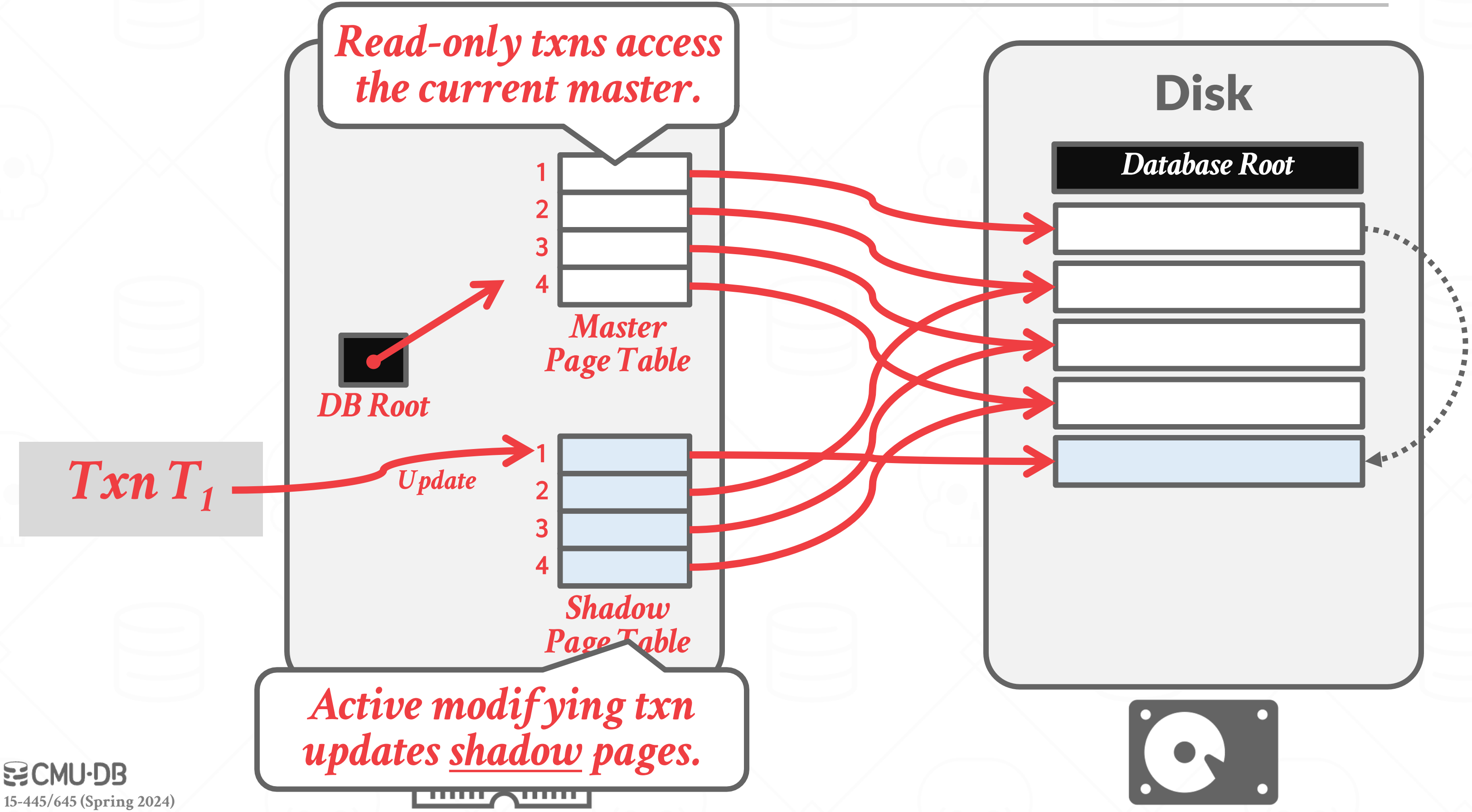


Txn T_1

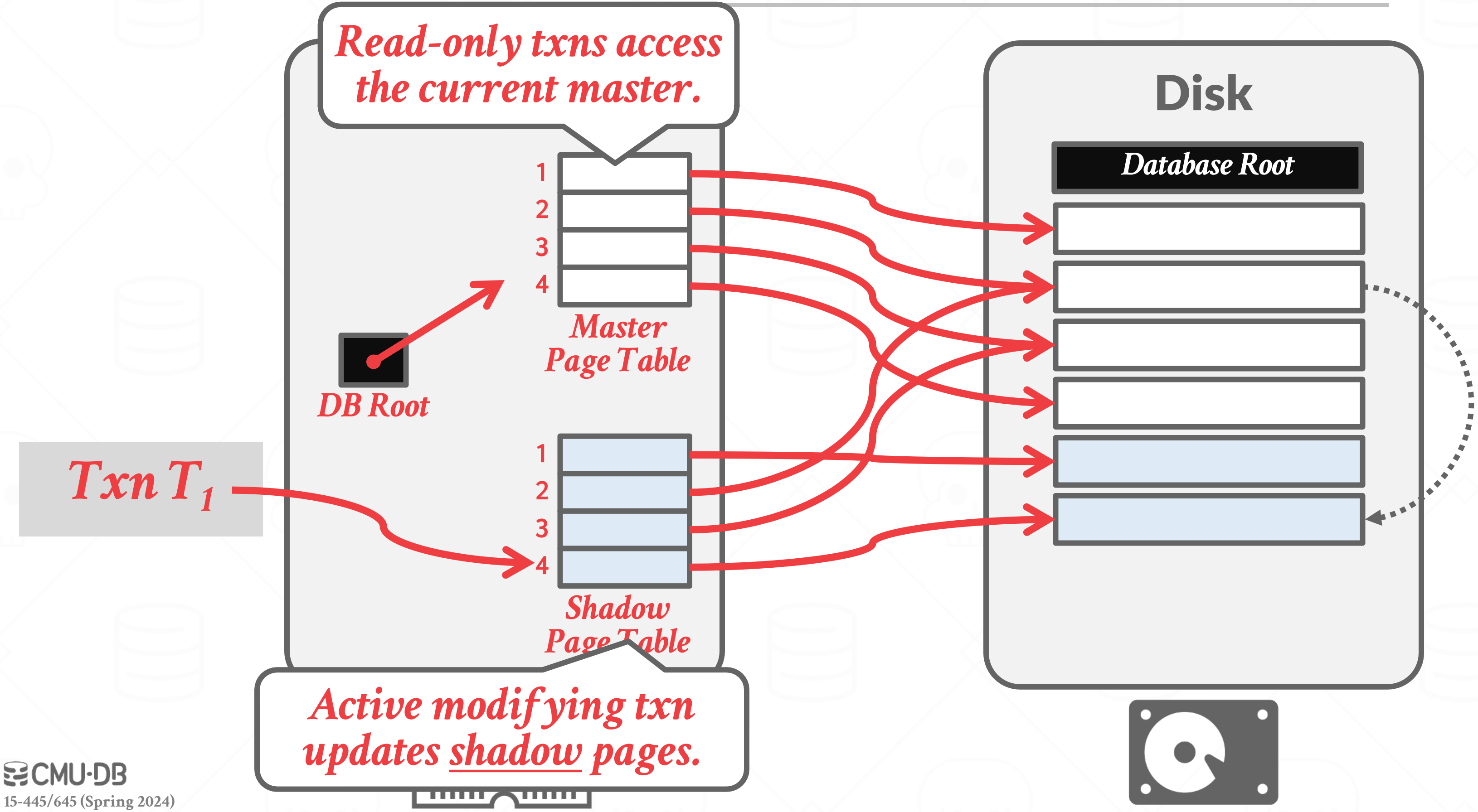
SHADOW PAGING – EXAMPLE



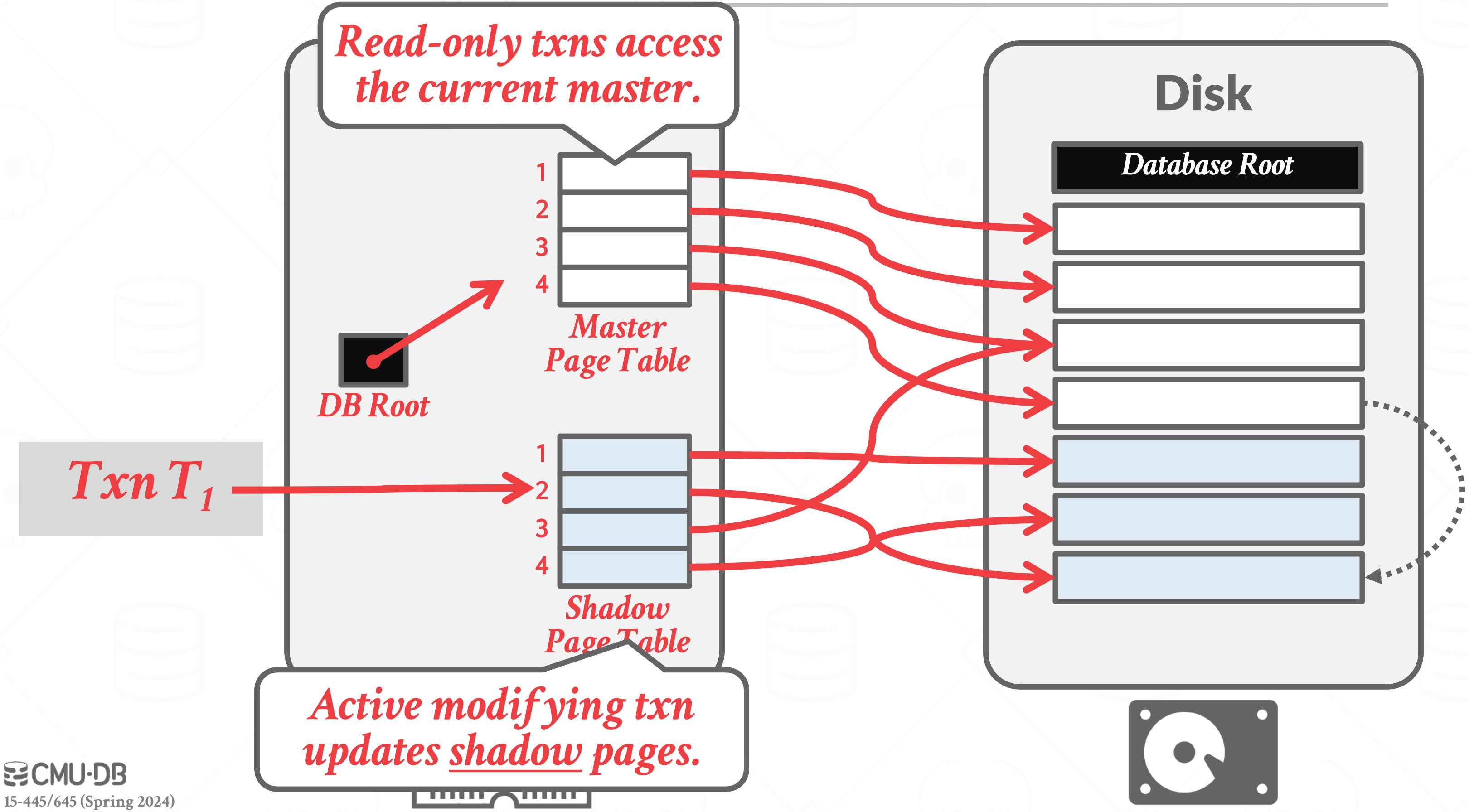
SHADOW PAGING – EXAMPLE



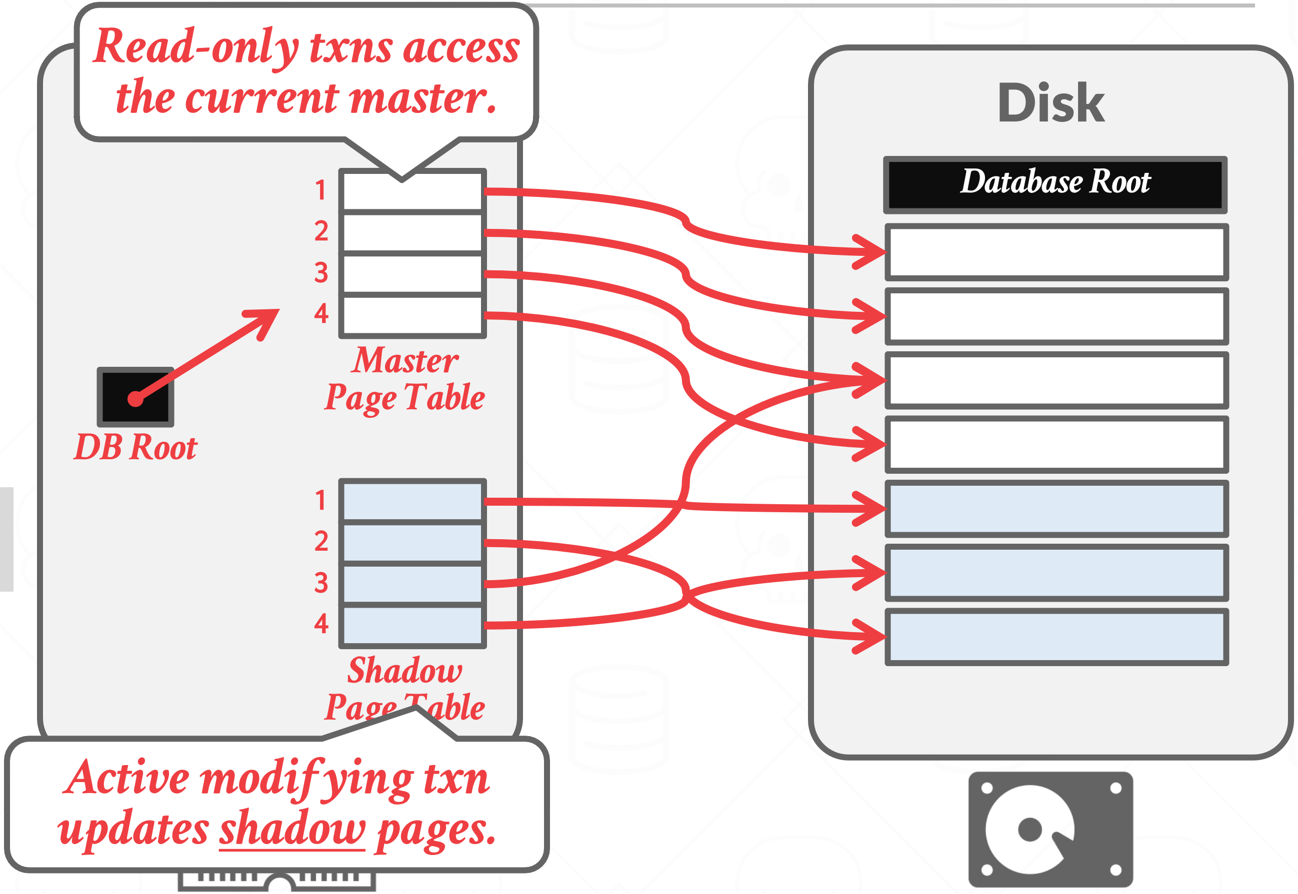
SHADOW PAGING – EXAMPLE



SHADOW PAGING – EXAMPLE



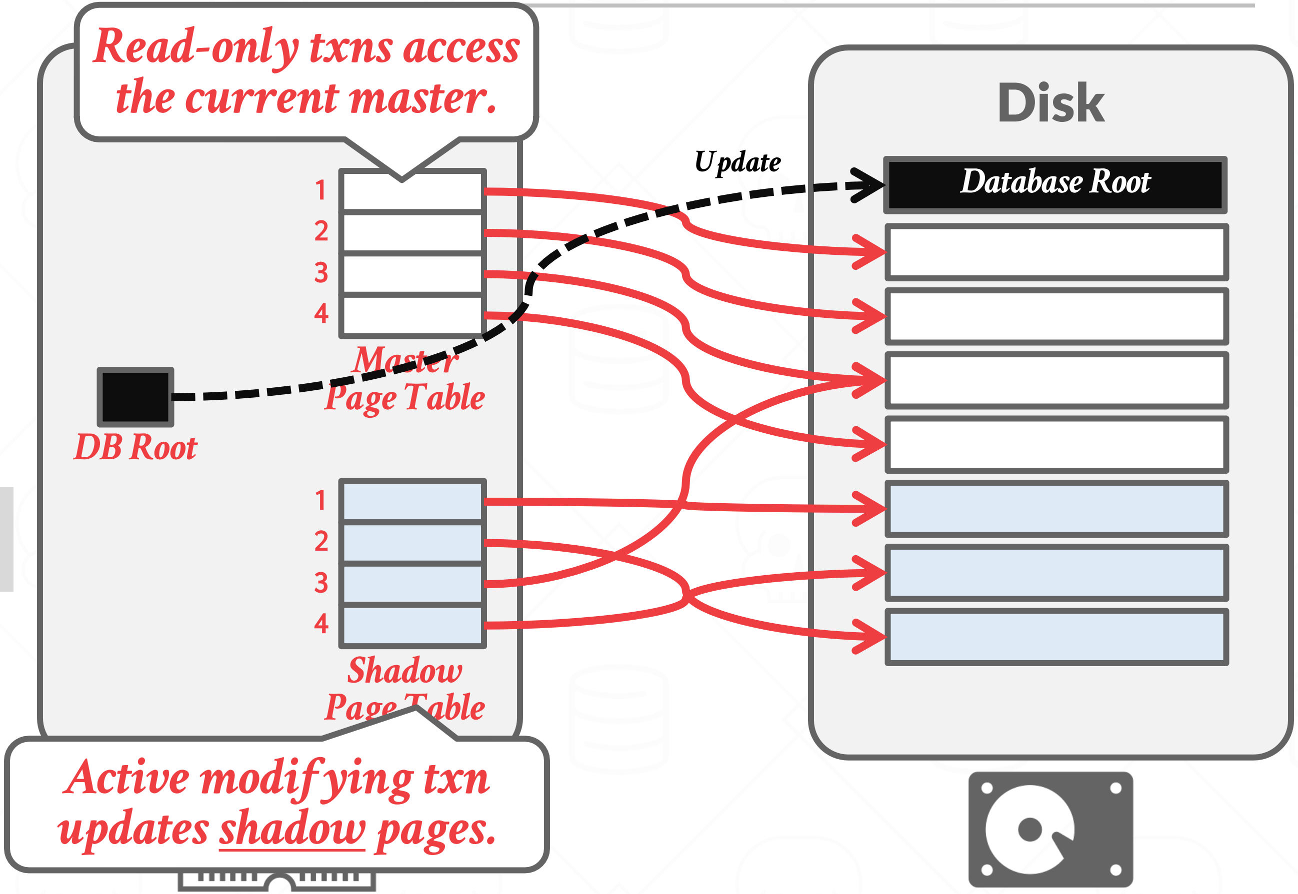
SHADOW PAGING – EXAMPLE



Txn T_1

COMMIT

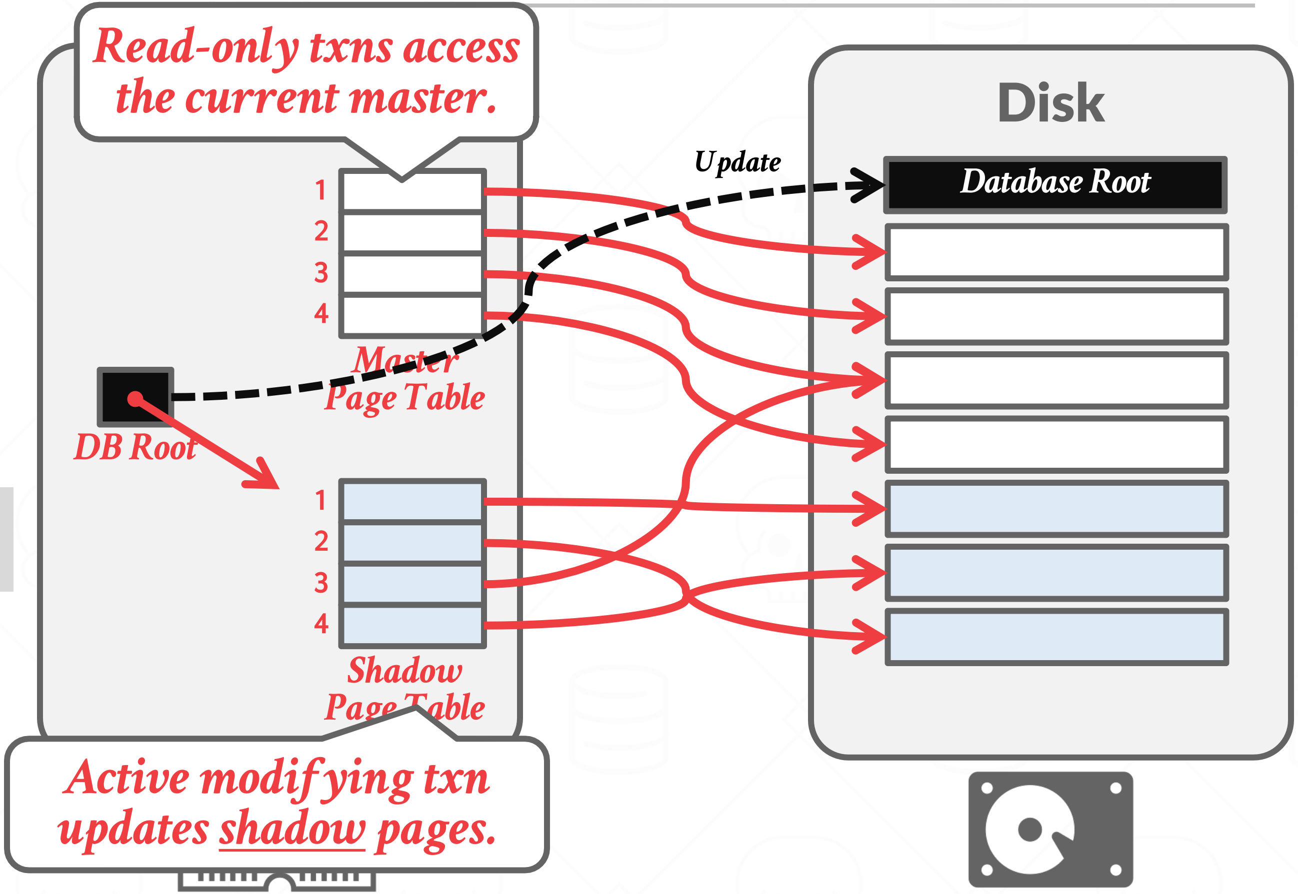
SHADOW PAGING – EXAMPLE



Txn T_1

COMMIT

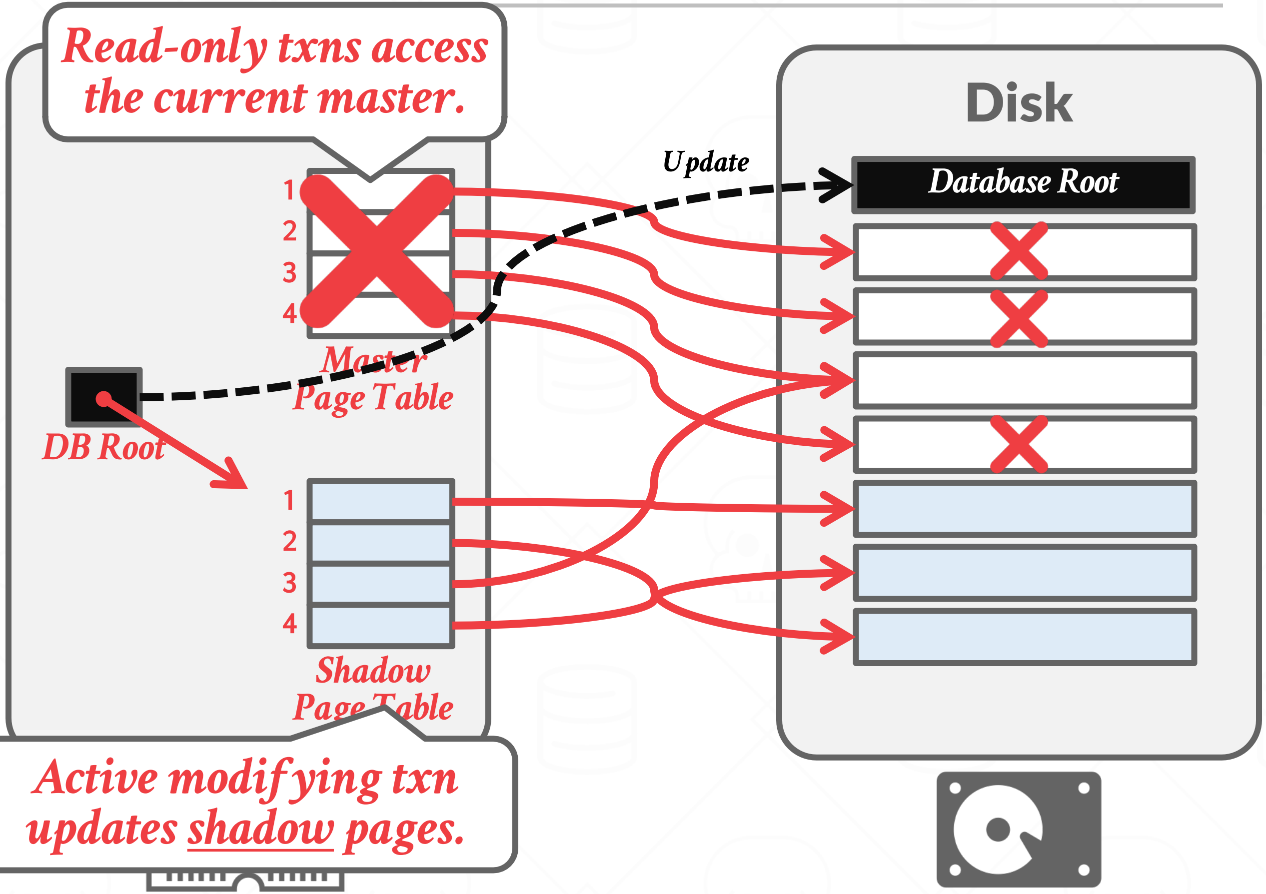
SHADOW PAGING – EXAMPLE



Txn T_1

COMMIT

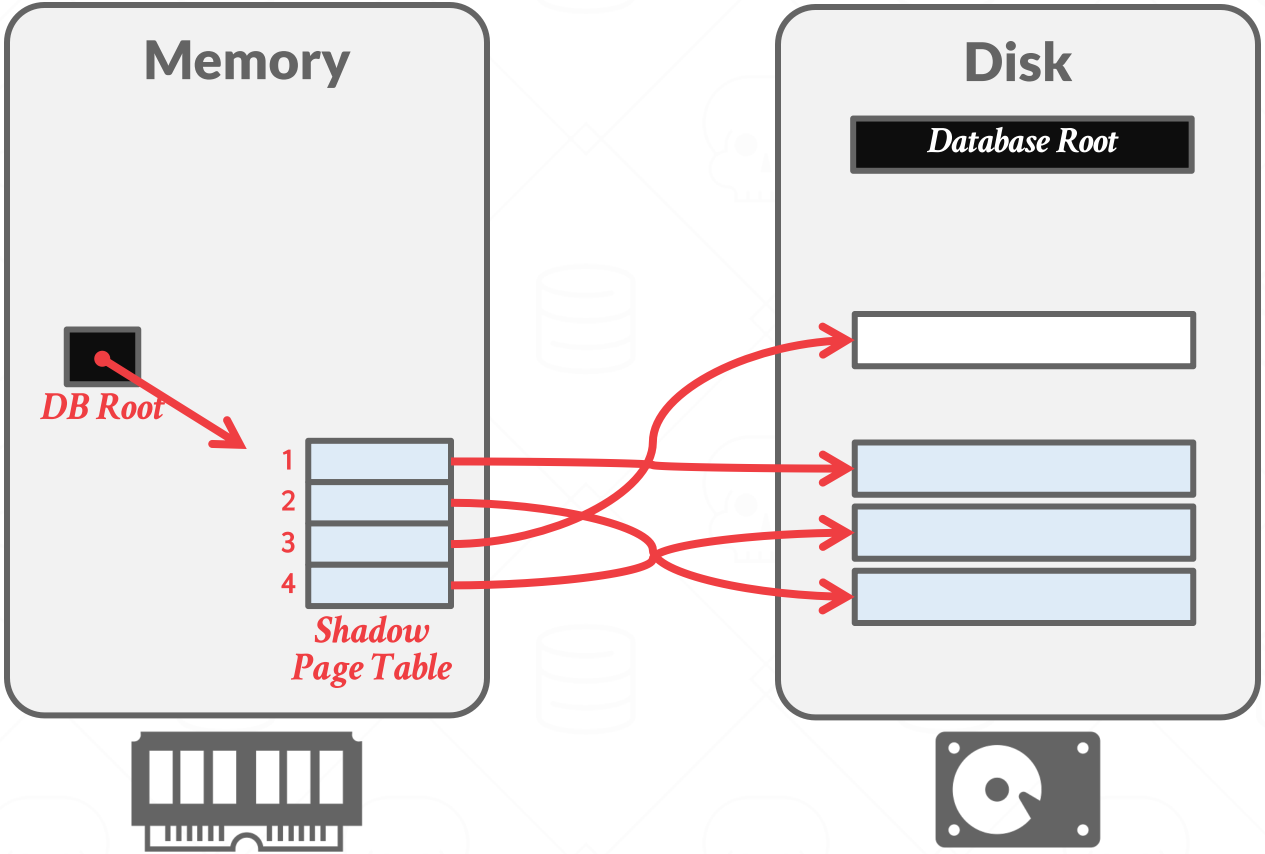
SHADOW PAGING – EXAMPLE



Txn T_1

COMMIT

SHADOW PAGING – EXAMPLE



Txn T_1

COMMIT

SHADOW PAGING – UNDO/REDO

Supporting rollbacks and recovery is easy.

Undo: Remove the shadow pages. Leave the master and the DB root pointer alone.

Redo: Not needed at all.

SHADOW PAGING – DISADVANTAGES

Copying the entire page table is expensive:

→ Use a page table structured like a B+tree (LMDB).

→ No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes.

Commit overhead is high:

→ Flush every updated page, page table, and root.

→ Data gets fragmented (bad for sequential scans).

→ Need garbage collection.

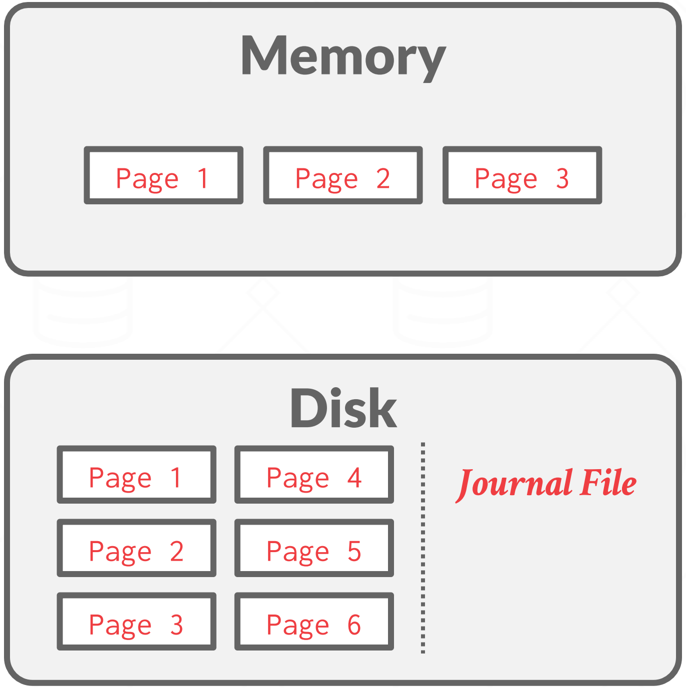
→ Only supports one writer txn at a time or txns in a batch.

SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "rollback mode".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

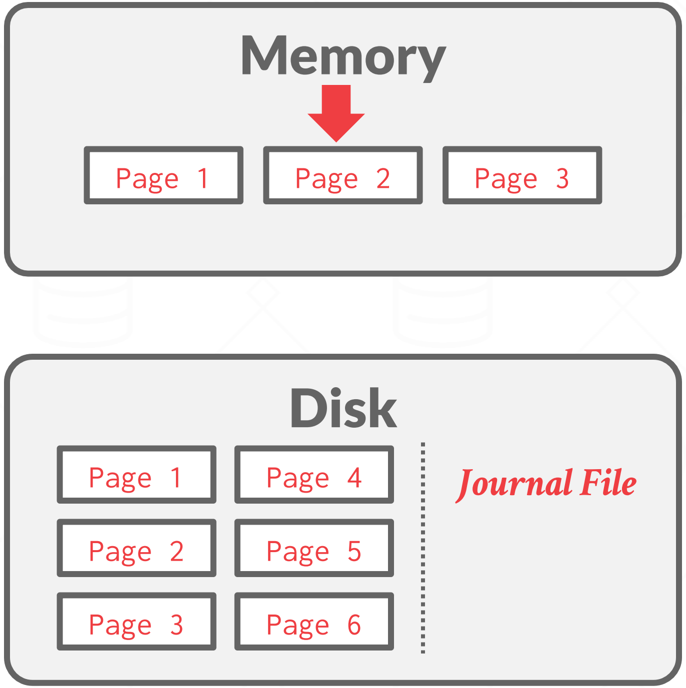


SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "rollback mode".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

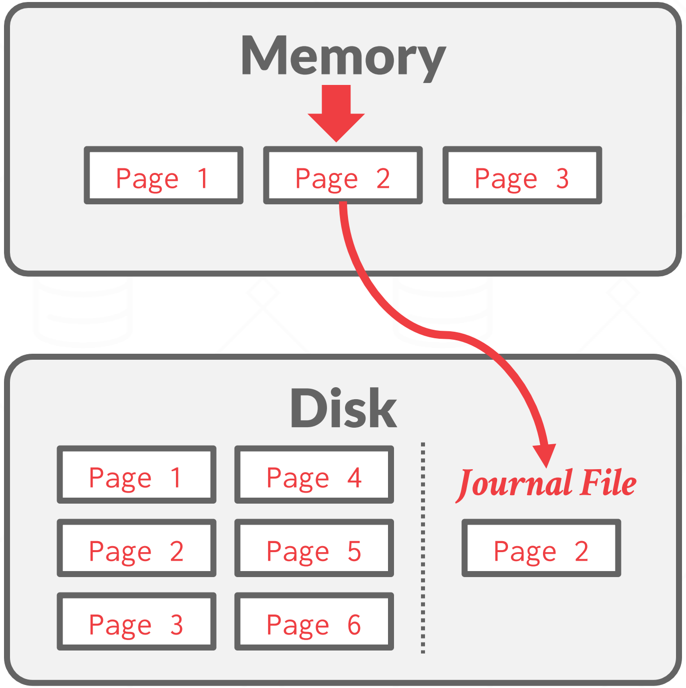


SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "rollback mode".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

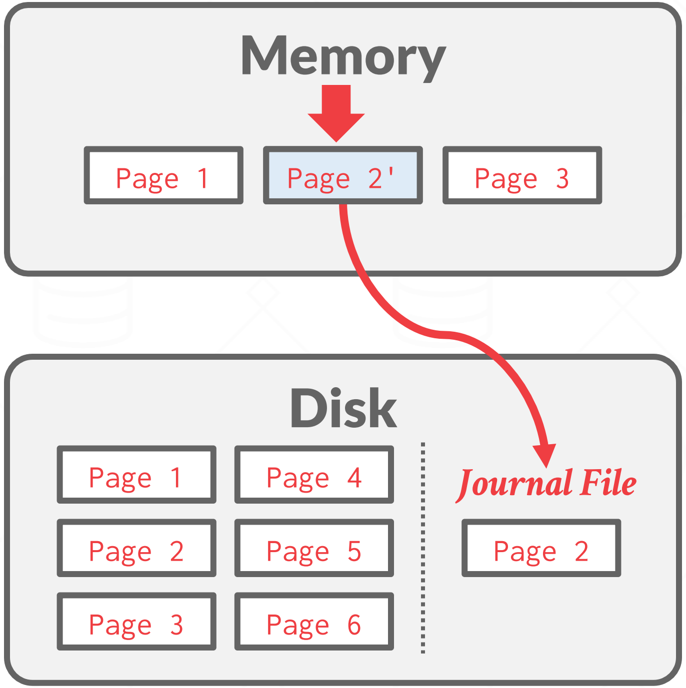


SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "rollback mode".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

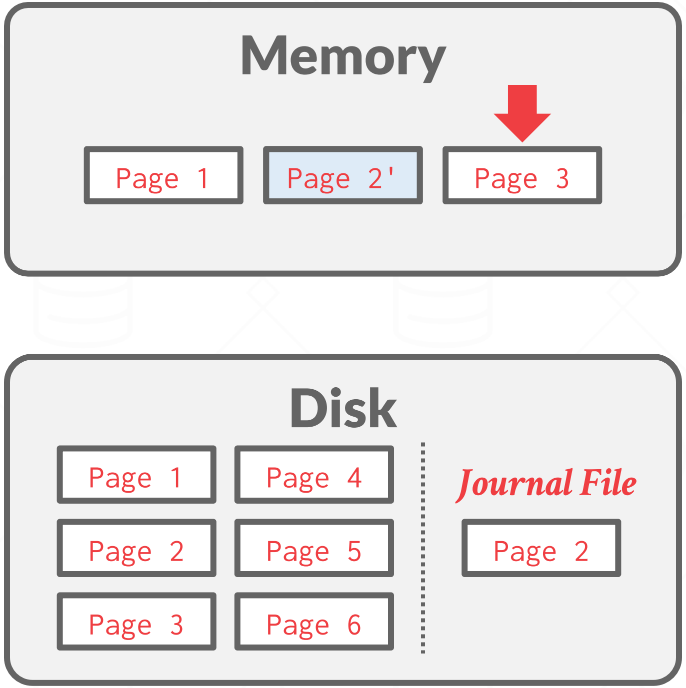


SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "rollback mode".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

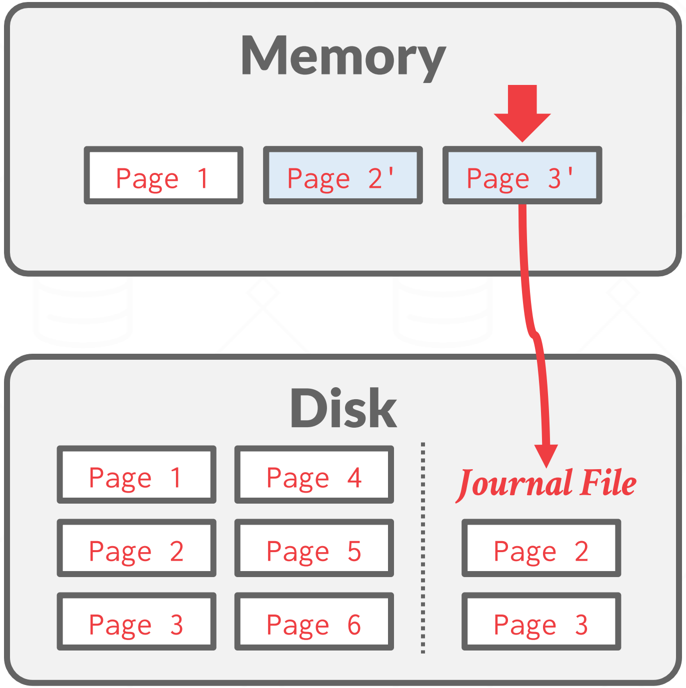


SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "rollback mode".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

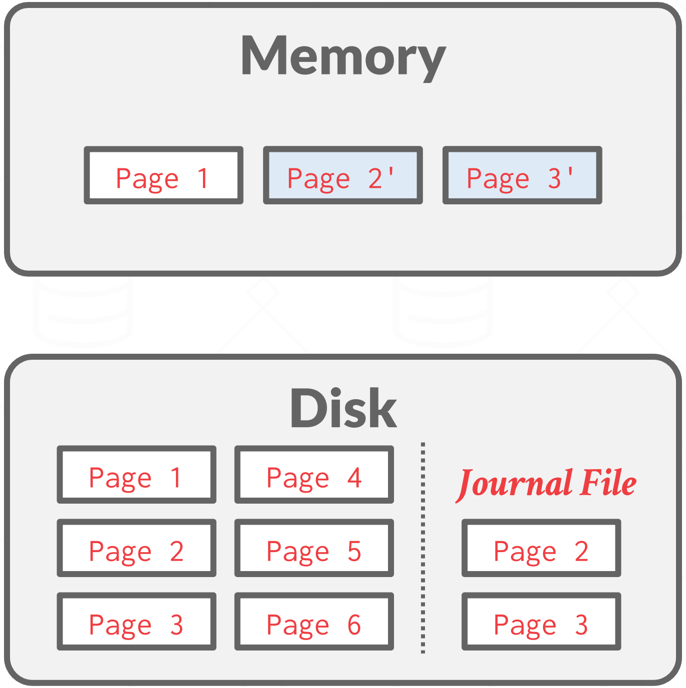


SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "rollback mode".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

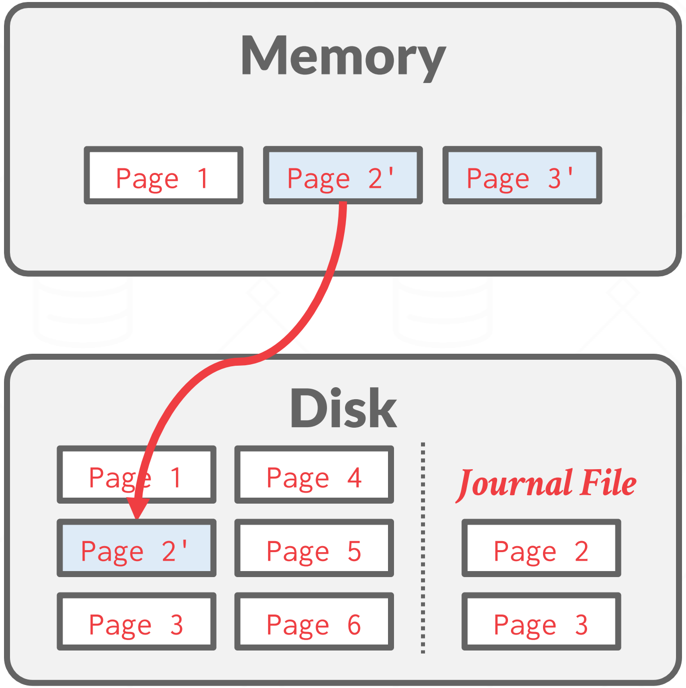


SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "rollback mode".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

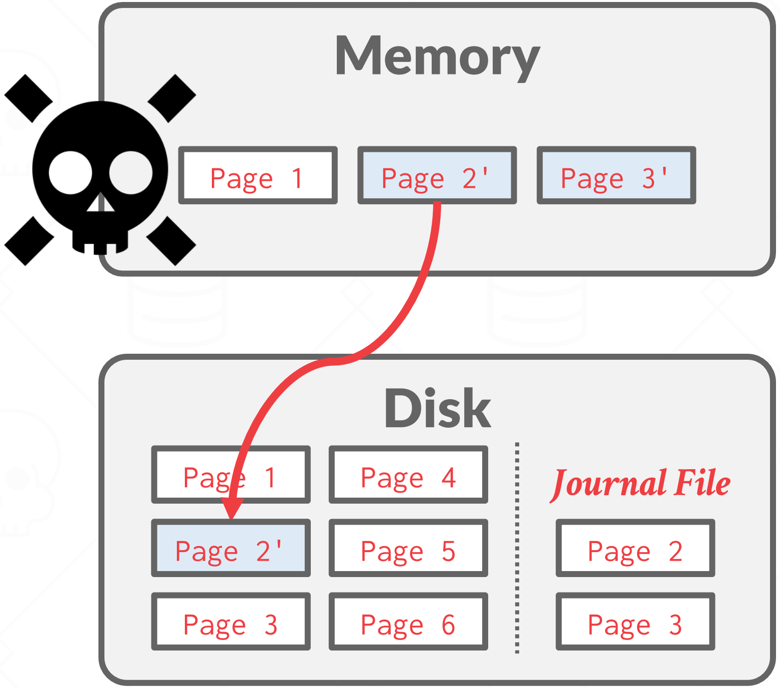


SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "rollback mode".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

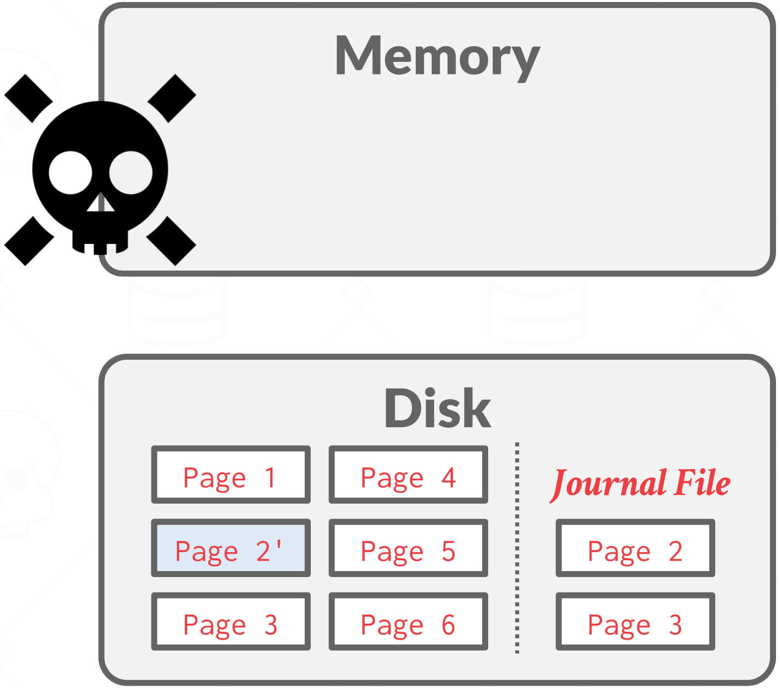


SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "[rollback mode](https://www.sqlite.org/lockingv3.html#rollback)".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

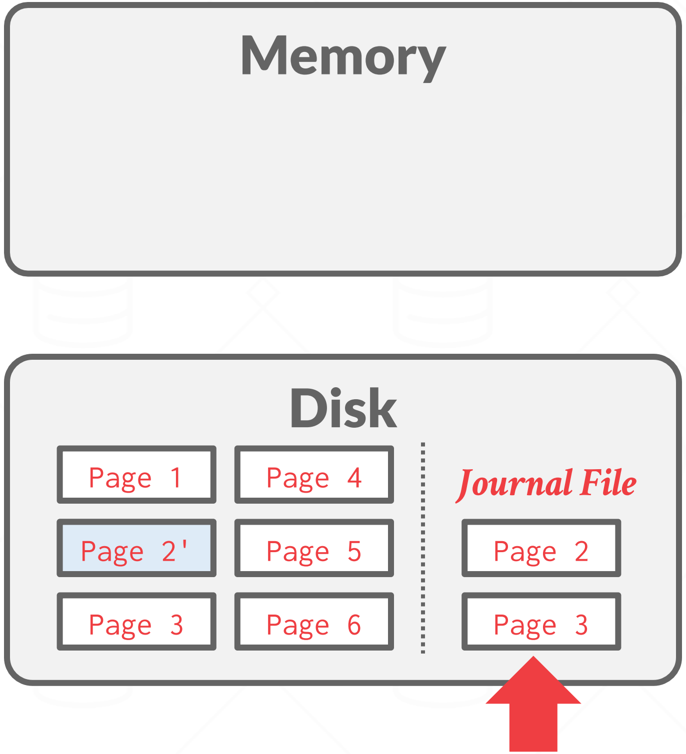


SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "rollback mode".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

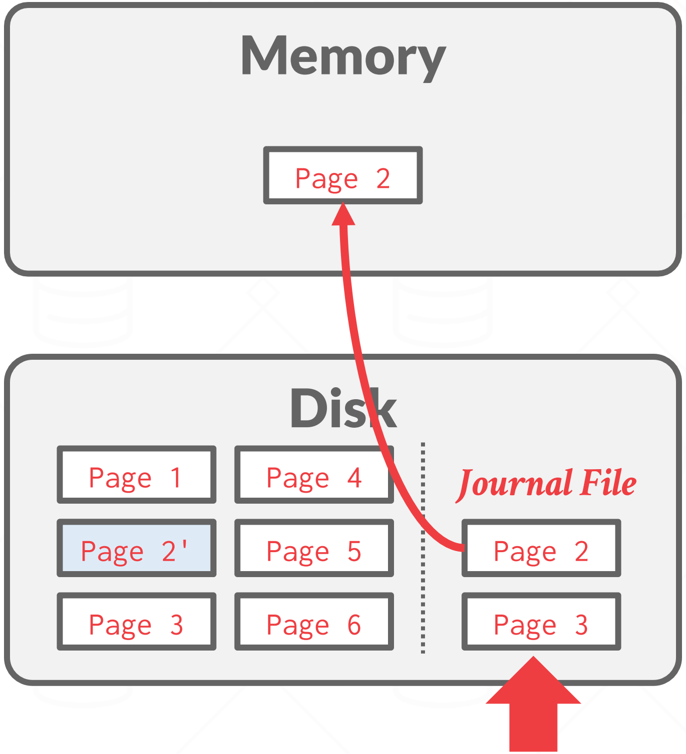


SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "rollback mode".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

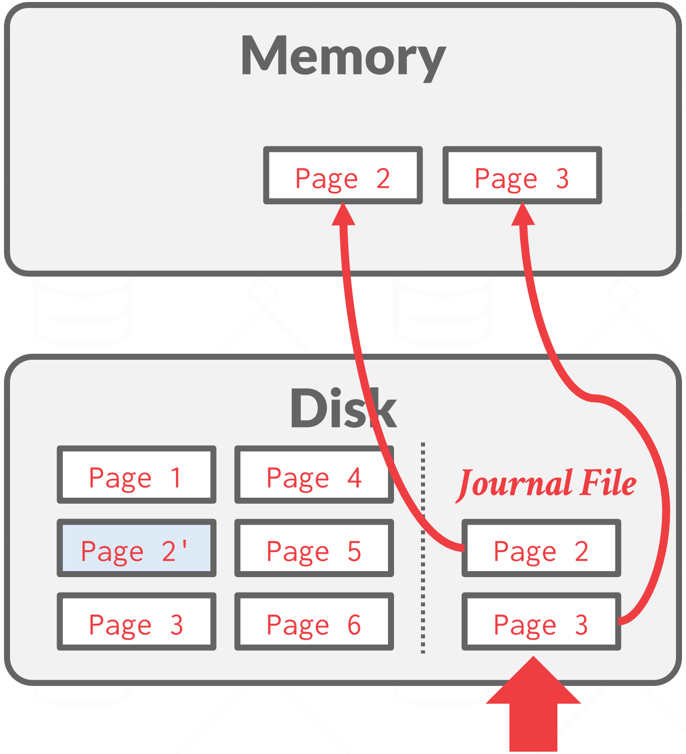


SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "rollback mode".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

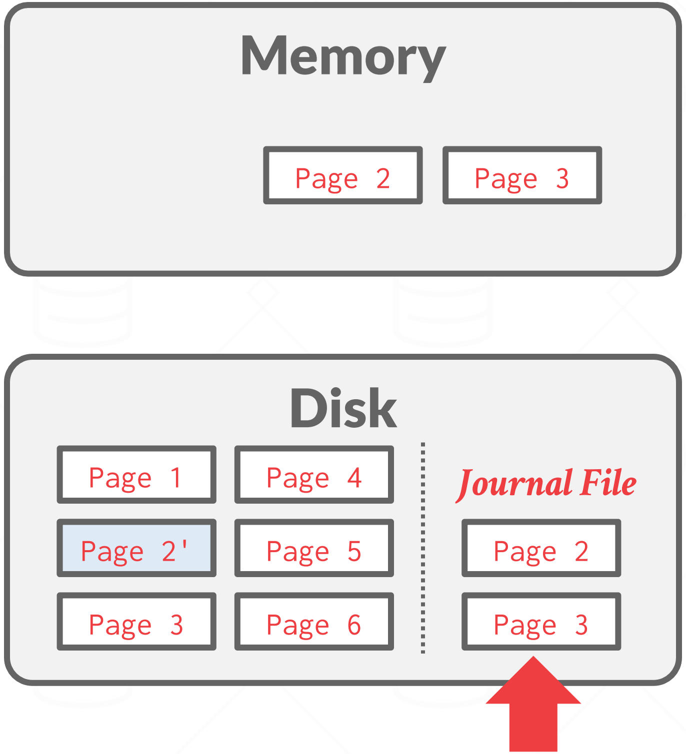


SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "rollback mode".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

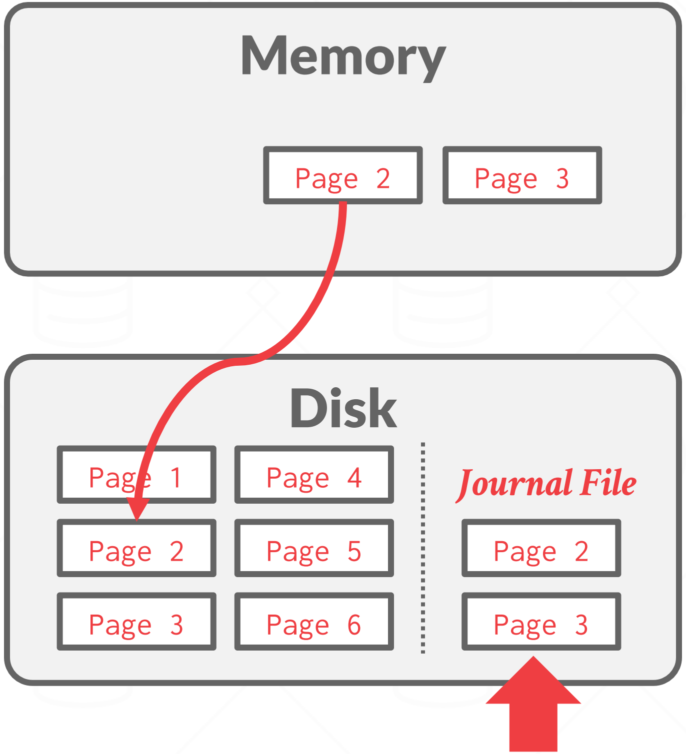


SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "rollback mode".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

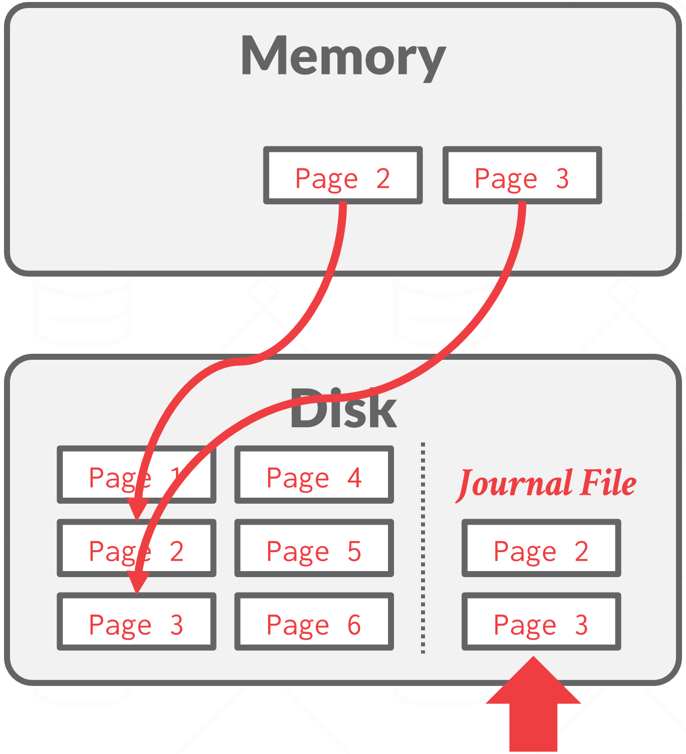


SQLITE (PRE-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting master version.

→ Called "rollback mode".

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



OBSERVATION

Shadowing page requires the DBMS to perform writes to random non-contiguous pages on disk.

We need a way for the DBMS convert random writes into sequential writes.

WRITE-AHEAD LOG

Maintain a log file separate from data files that contains the changes that txns make to database.

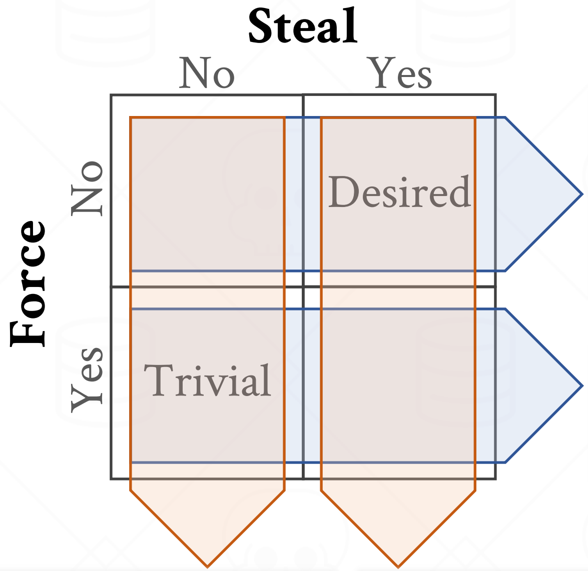
→ Assume that the log is on stable storage.

→ Log contains enough information to perform the necessary undo and redo actions to restore the database.

DBMS must write to disk the log file records that correspond to changes made to a database object before it can flush that object to disk.

Buffer Pool Policy: **STEAL** + **NO-FORCE**

BUFFER POOL AND WAL



No-Force

Concern: Crash before a page is flushed to disk. Durability?

Solution: Force a summary/log @ commit. Use to **REDO**.

Force (on every update, flush the updated page to disk)

Poor response time, but enforces durability of committed txns.

No-Steal

Low throughtput, but works for aborted txns.

Steal (flush an unpinned dirty page even if the updating txn. is active)

Concern: A stolen+flushed page was modified by an uncommitted txn. T. If T aborts, how is atomicity enforced?

Solution: Remember old value (logs). Use to **UNDO**.

WAL PROTOCOL

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

All log records pertaining to an updated page are written to non-volatile storage before the page itself is over-written in non-volatile storage.

A txn is not considered committed until all its log records have been written to stable storage.

WAL PROTOCOL

Write a **<BEGIN>** record to the log for each txn to mark its starting point.

When a txn finishes, the DBMS will:

→ Write a **<COMMIT>** record on the log.

→ Make sure that all log records are flushed before it returns an acknowledgement to application.

WAL PROTOCOL

Each log entry contains information about the change to a single object:

→ Transaction Id

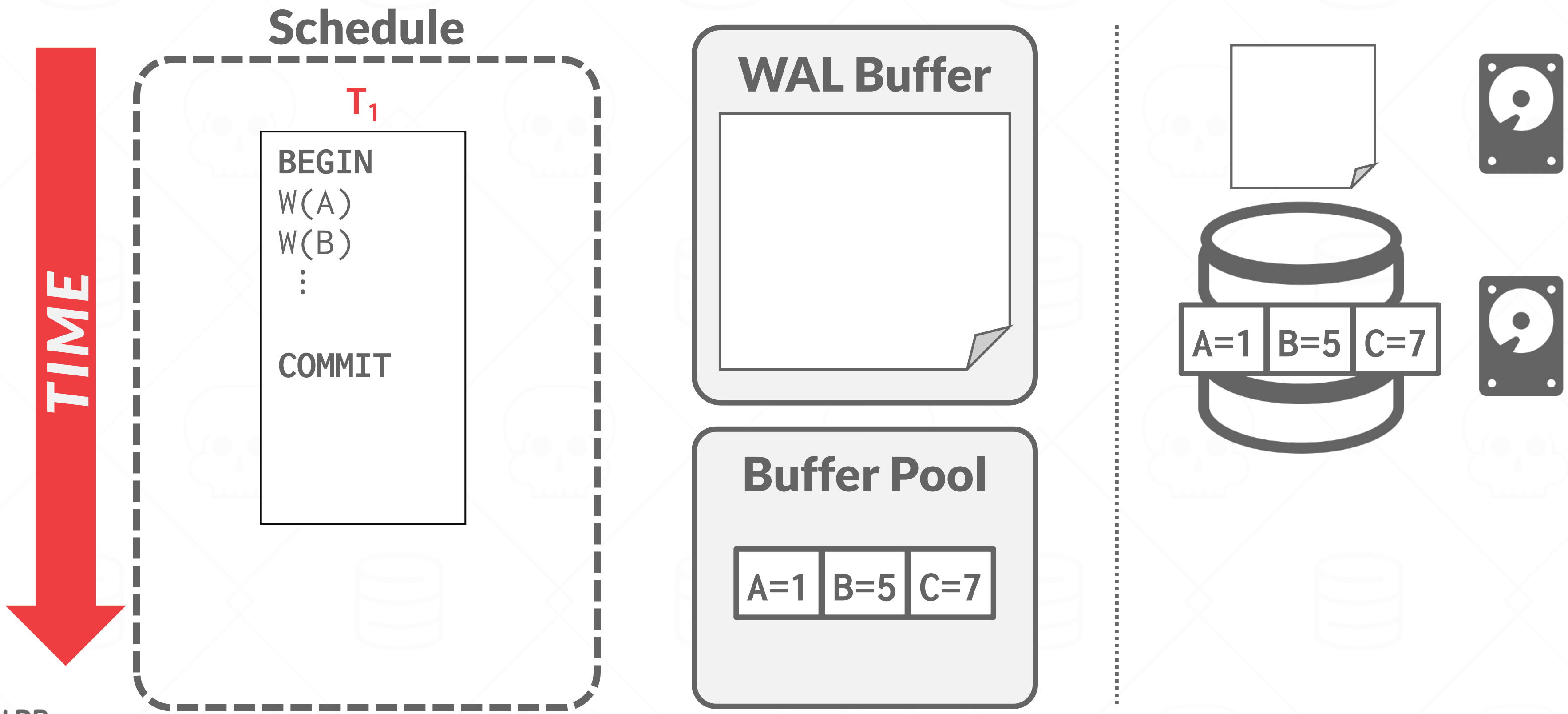
→ Object Id

→ Before Value (**UNDO**)

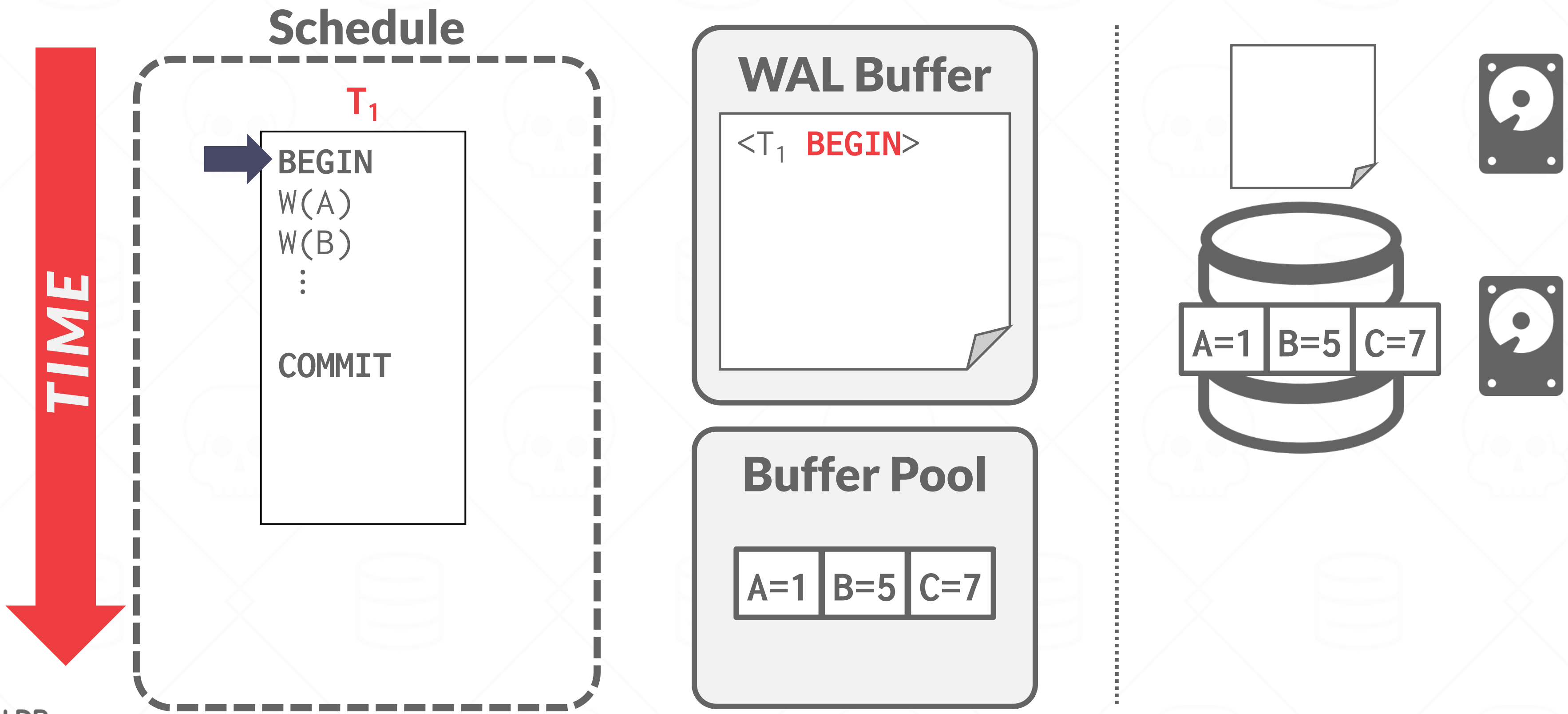
→ After Value (**REDO**)

Not necessary if using append-only MVCC

WAL – EXAMPLE

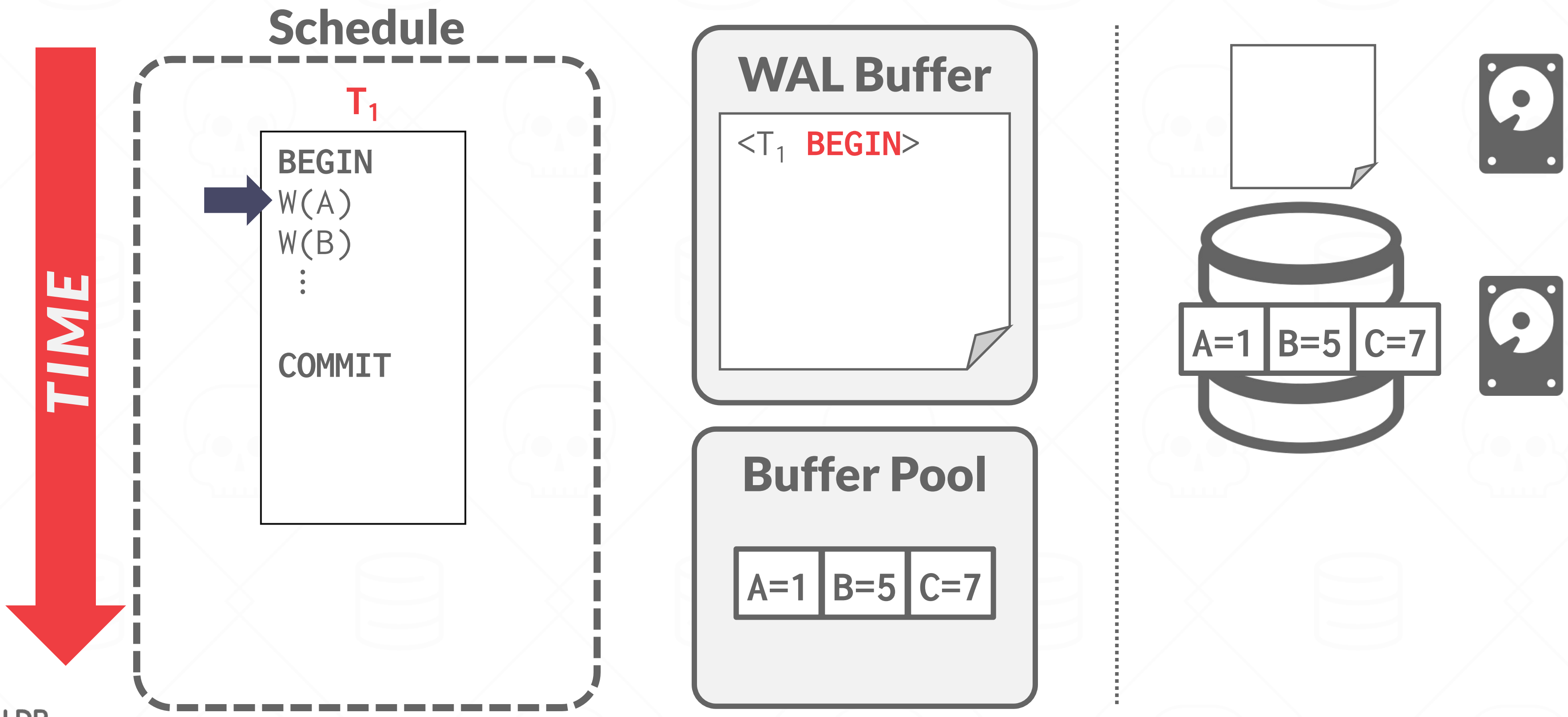

TIME
Schedule
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL Buffer
Buffer Pool
A=1
B=5
C=7
A=1
B=5
C=7

WAL – EXAMPLE

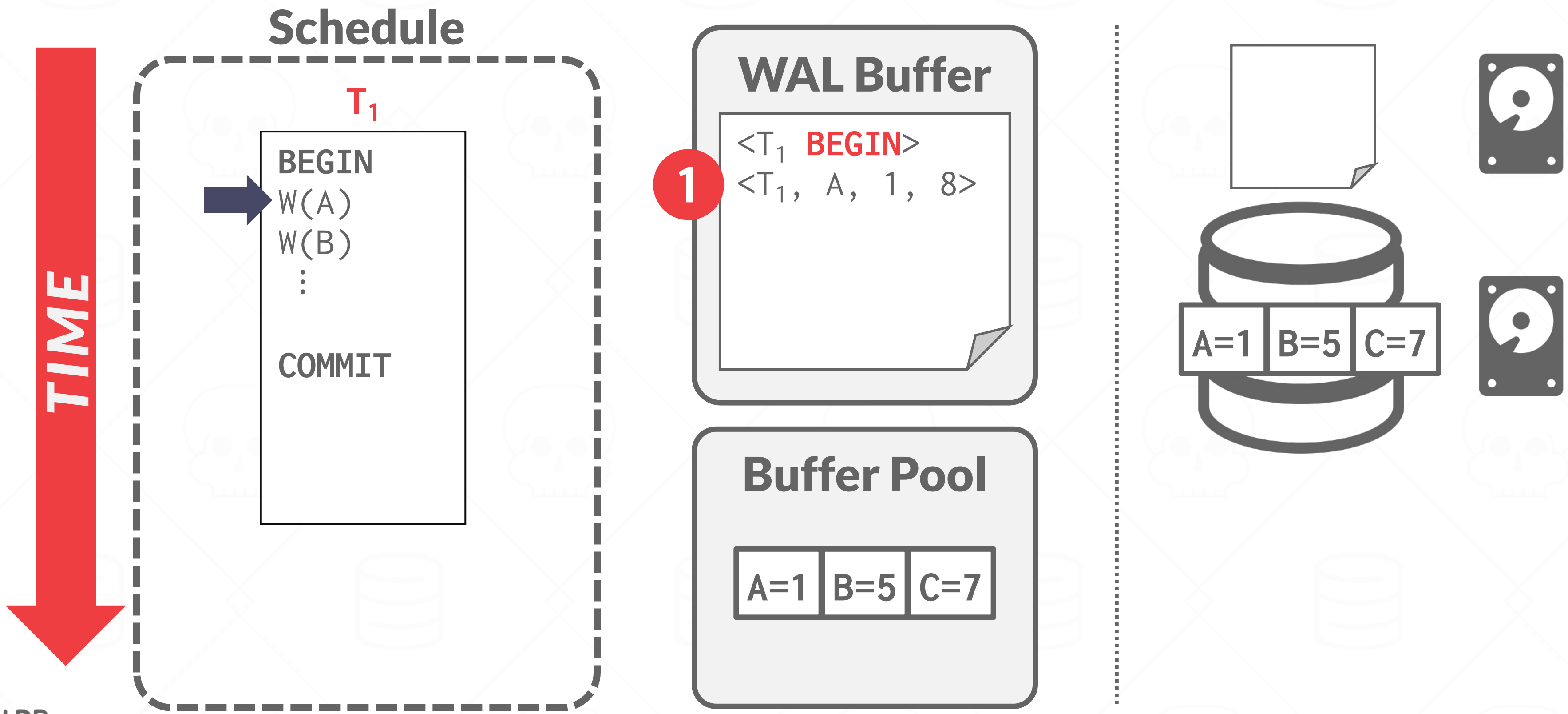


Schedule
TIME
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL Buffer
<T_1 BEGIN>
Buffer Pool
A=1
B=5
C=7
A=1
B=5
C=7

WAL – EXAMPLE

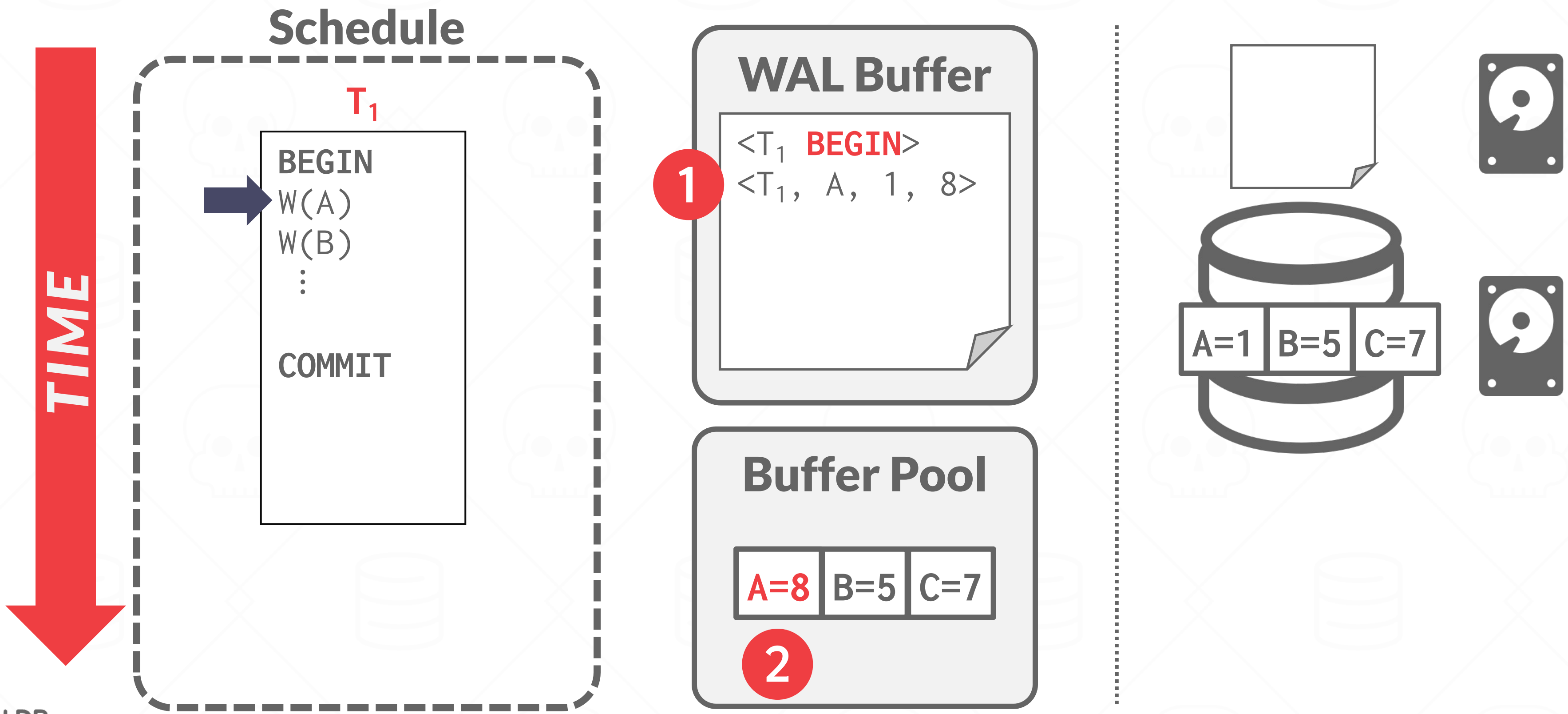

TIME
Schedule
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL Buffer
<T_1 BEGIN>
Buffer Pool
A=1
B=5
C=7
A=1
B=5
C=7

WAL – EXAMPLE

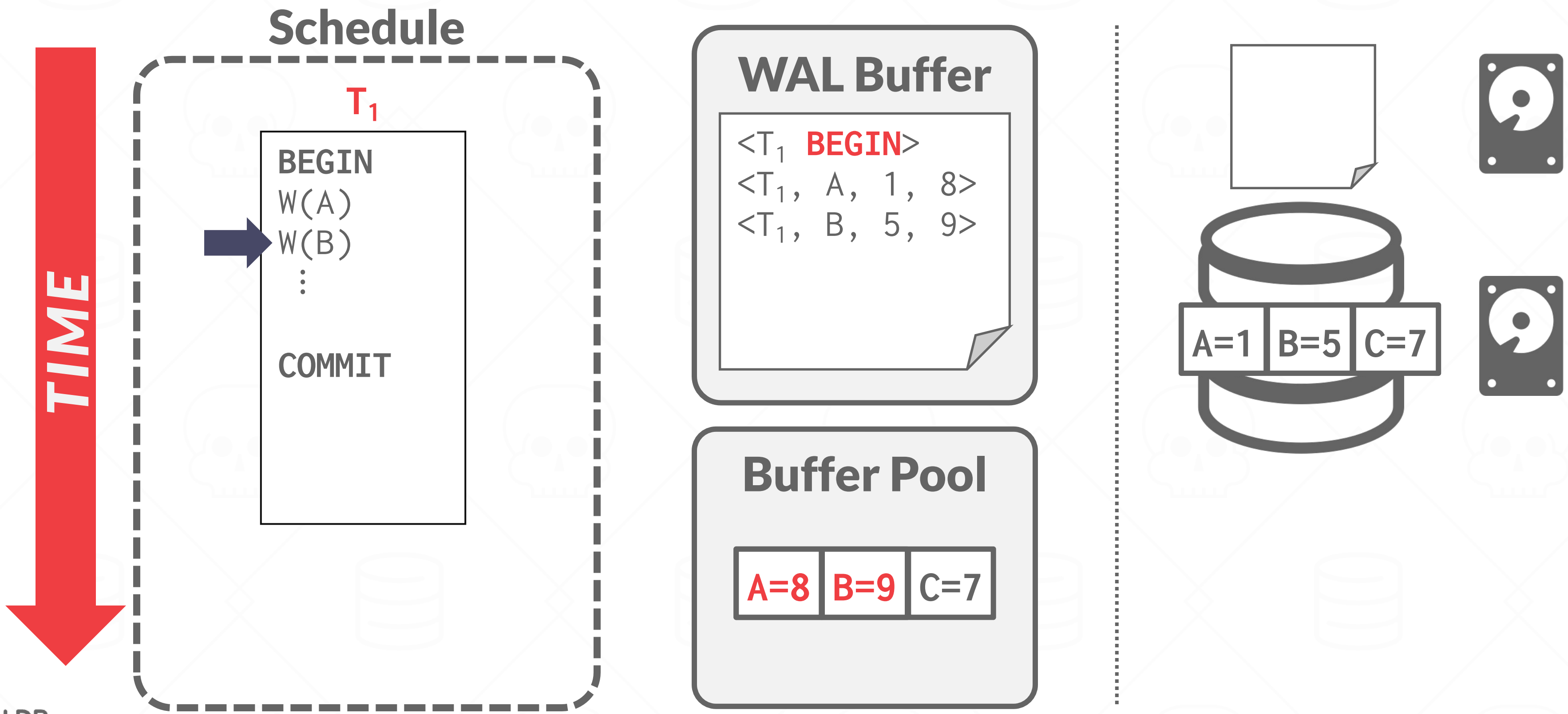


TIME
Schedule
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL Buffer
<T_1 BEGIN>
<T_1, A, 1, 8>
1
Buffer Pool
A=1 B=5 C=7
A=1 B=5 C=7

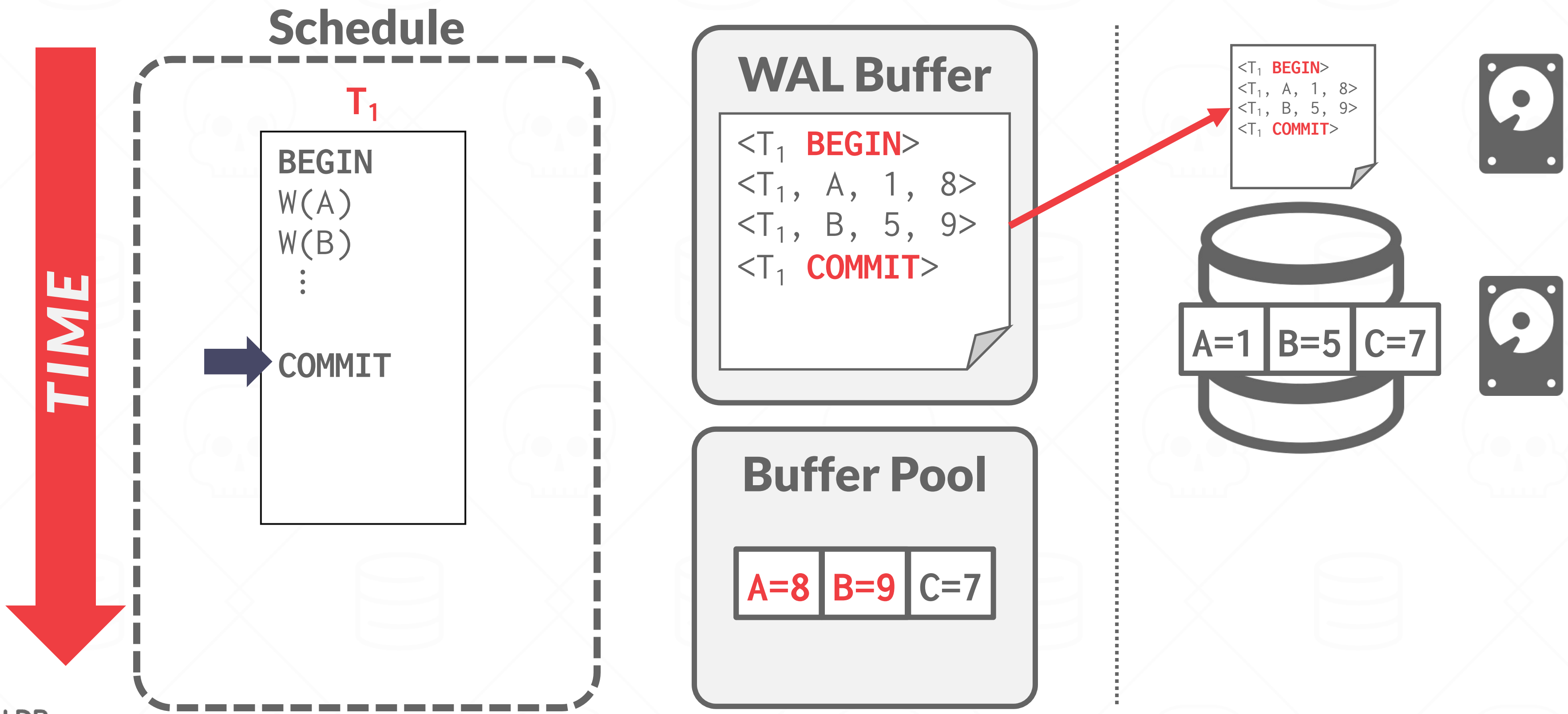
WAL – EXAMPLE


TIME
Schedule
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL Buffer
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

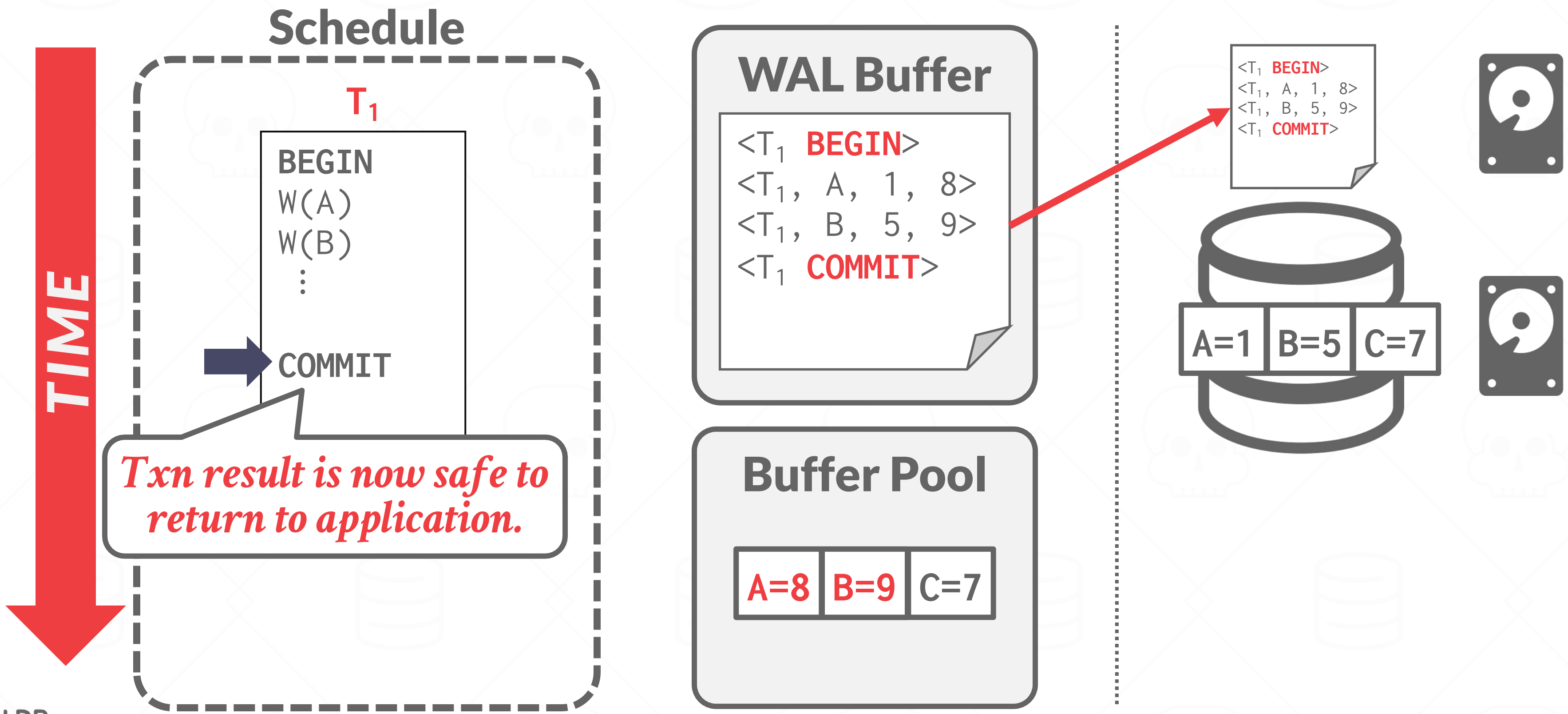

TIME
Schedule
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL Buffer
<T_1 BEGIN>
<T_1, A, 1, 8>
<T_1, B, 5, 9>
Buffer Pool
A=8
B=9
C=7
A=1
B=5
C=7

WAL – EXAMPLE



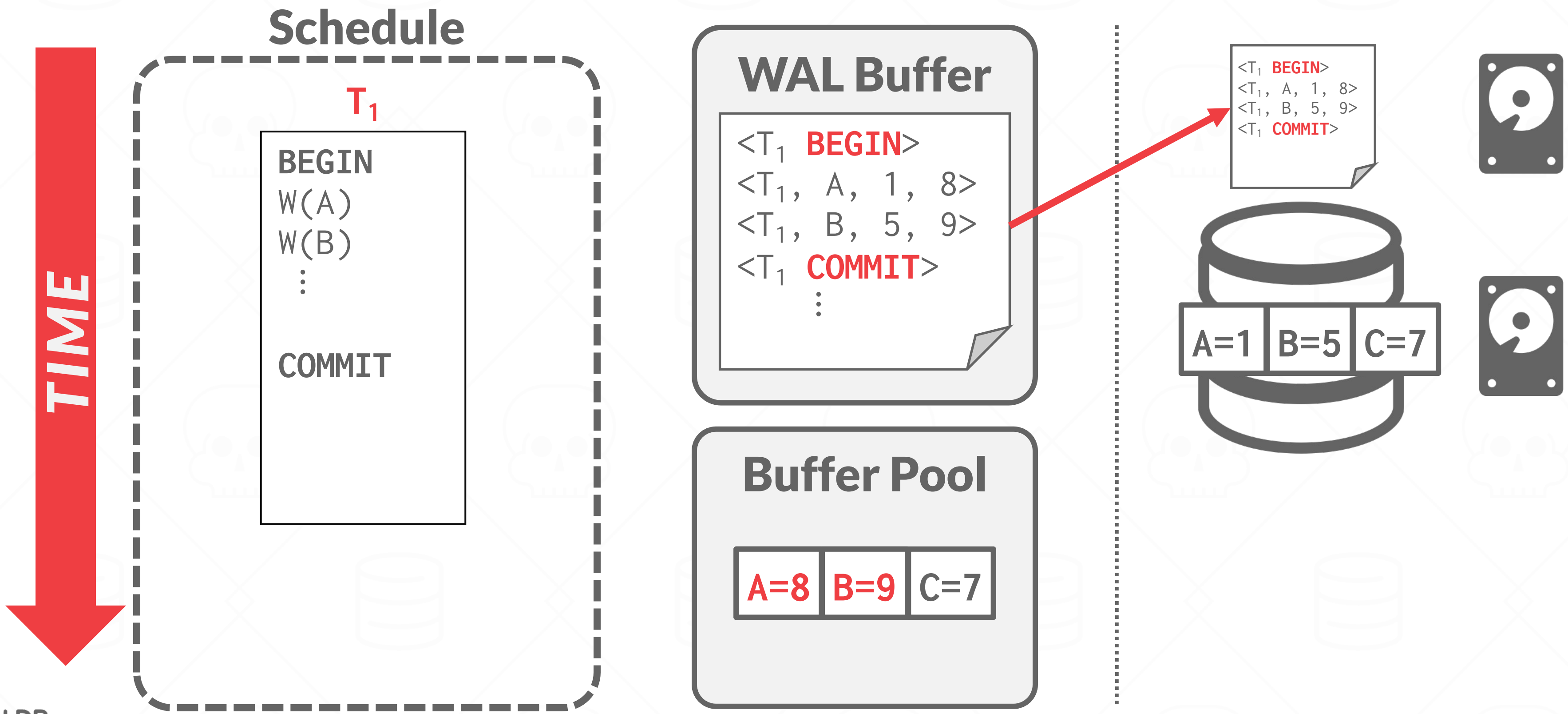
Schedule
TIME
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL Buffer
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
<T1 COMMIT>
Buffer Pool
A=8
B=9
C=7
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
<T1 COMMIT>
A=1
B=5
C=7

WAL – EXAMPLE



TIME
Schedule
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
Txn result is now safe to return to application.
WAL Buffer
<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


TIME
Schedule
T1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL Buffer
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
<T1 COMMIT>
⋮
Buffer Pool
A=8 B=9 C=7
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
<T1 COMMIT>
A=1 B=5 C=7

WAL – EXAMPLE

TIME

Schedule

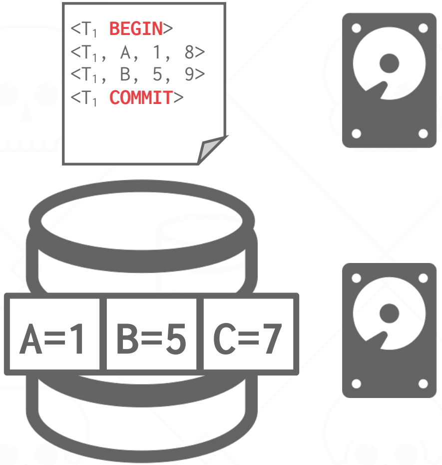
T_1

```
BEGIN
W(A)
W(B)
  ⋮

COMMIT
```

WAL Buffer

Buffer Pool



WAL – EXA

Everything we need to restore T_1 is in the log!

Schedule

TIME

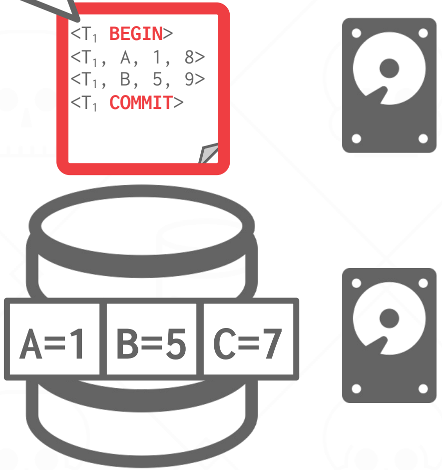
T_1

```
BEGIN
W(A)
W(B)
 ⋮

COMMIT
```

WAL Buffer

Buffer Pool



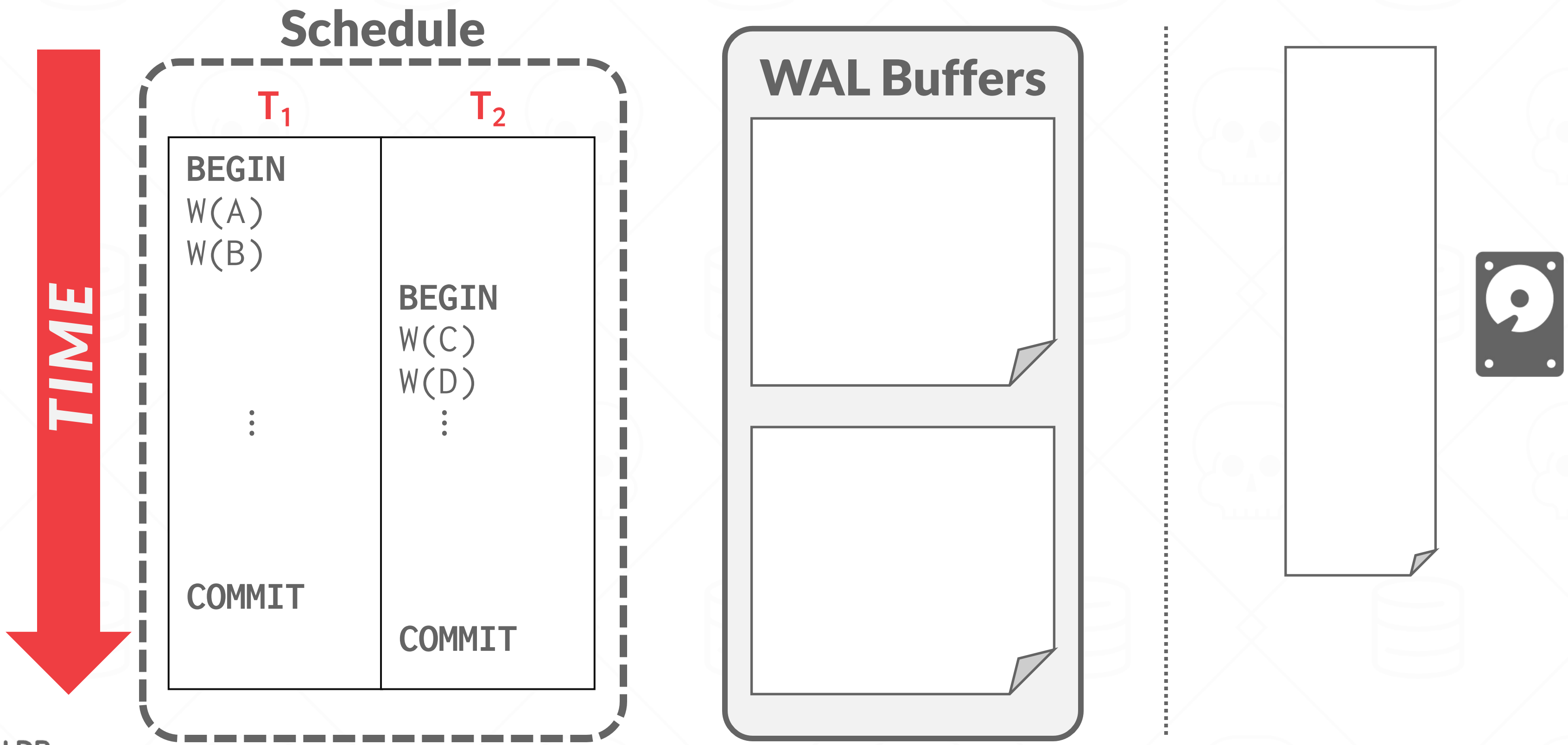
WAL – IMPLEMENTATION

Flushing the log buffer to disk every time a txn commits will become a bottleneck.

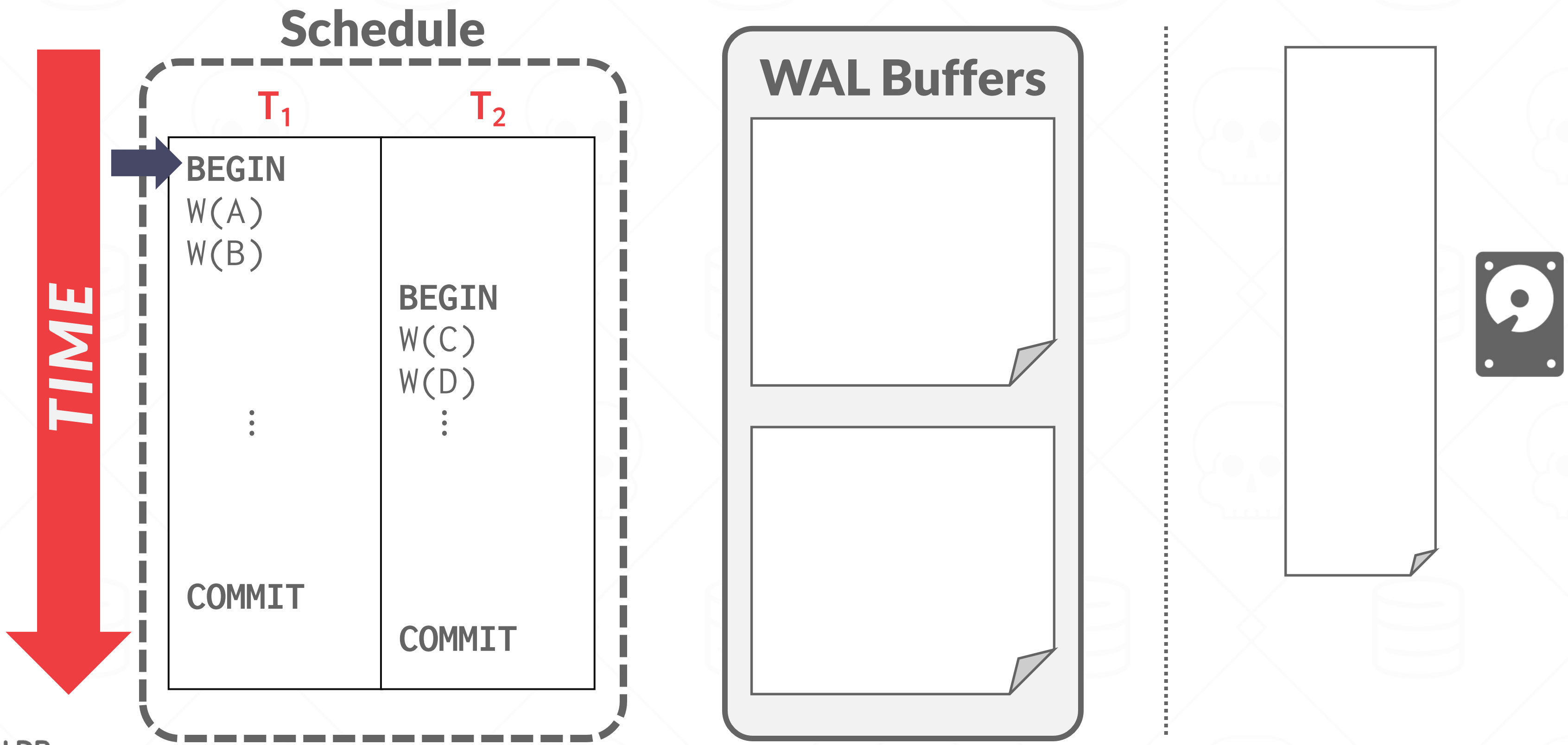
The DBMS can use the group commit optimization to batch multiple log flushes together to amortize overhead.

→ When the buffer is full, flush it to disk.
→ Or if there is a timeout (e.g., 5 ms).

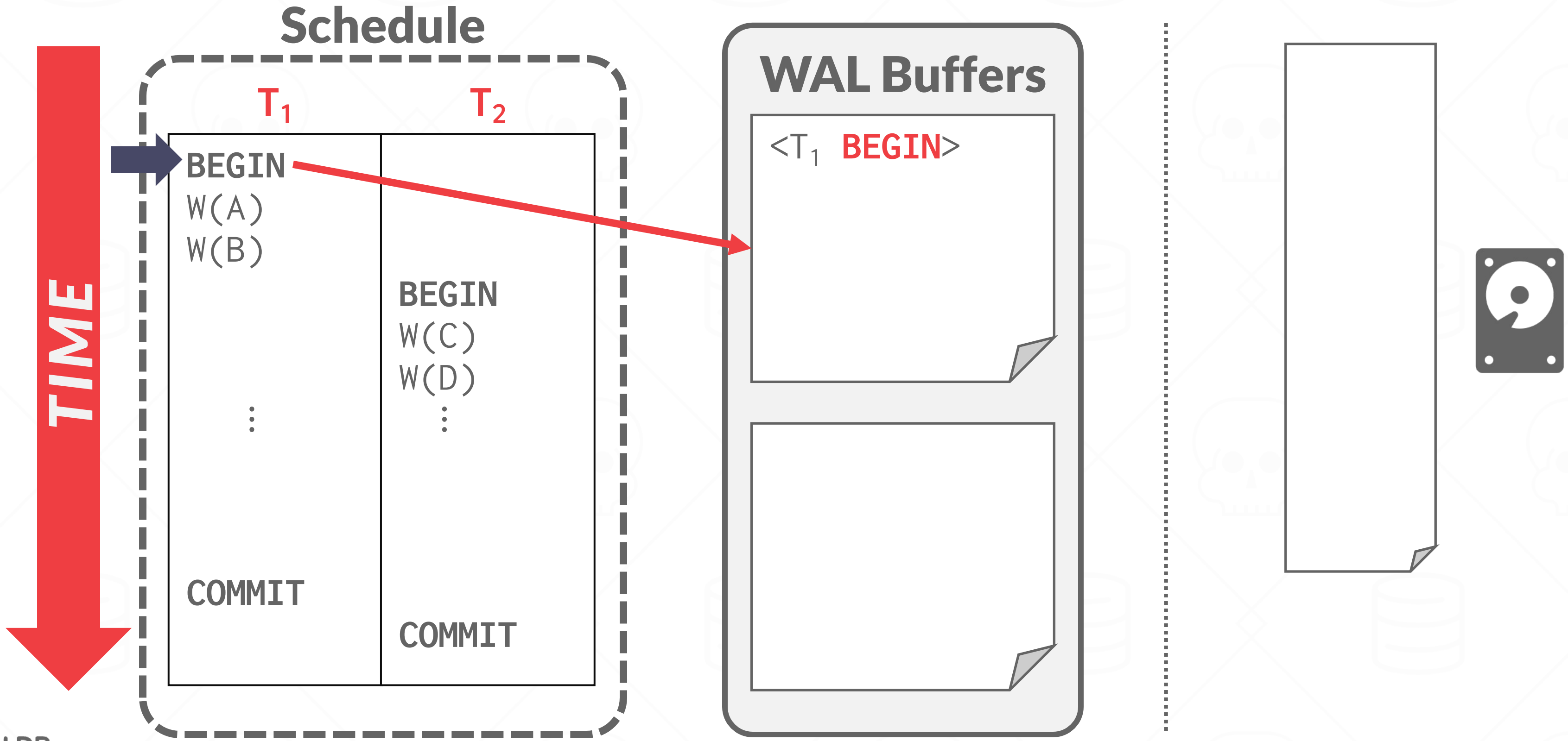
WAL – GROUP COMMIT



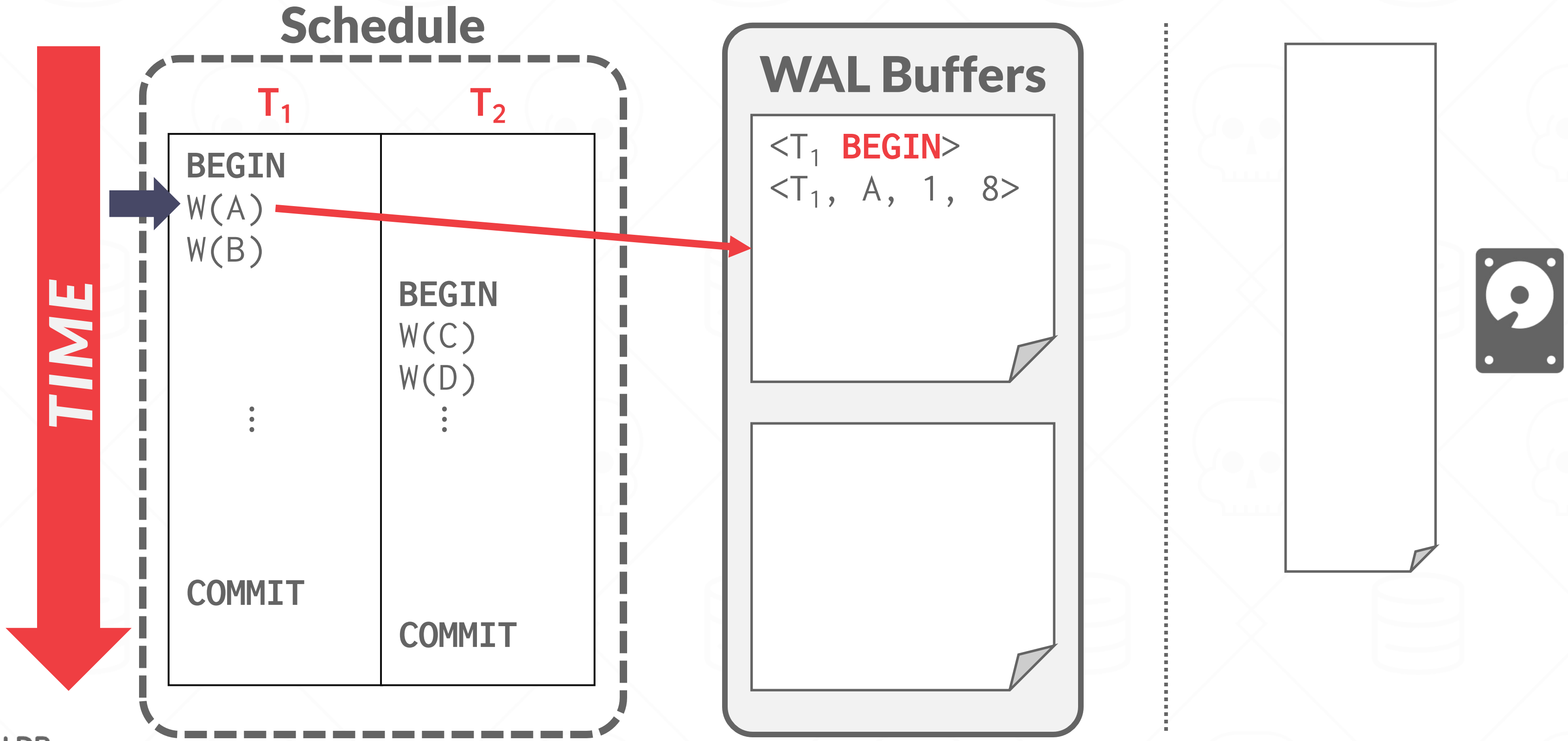
WAL – GROUP COMMIT



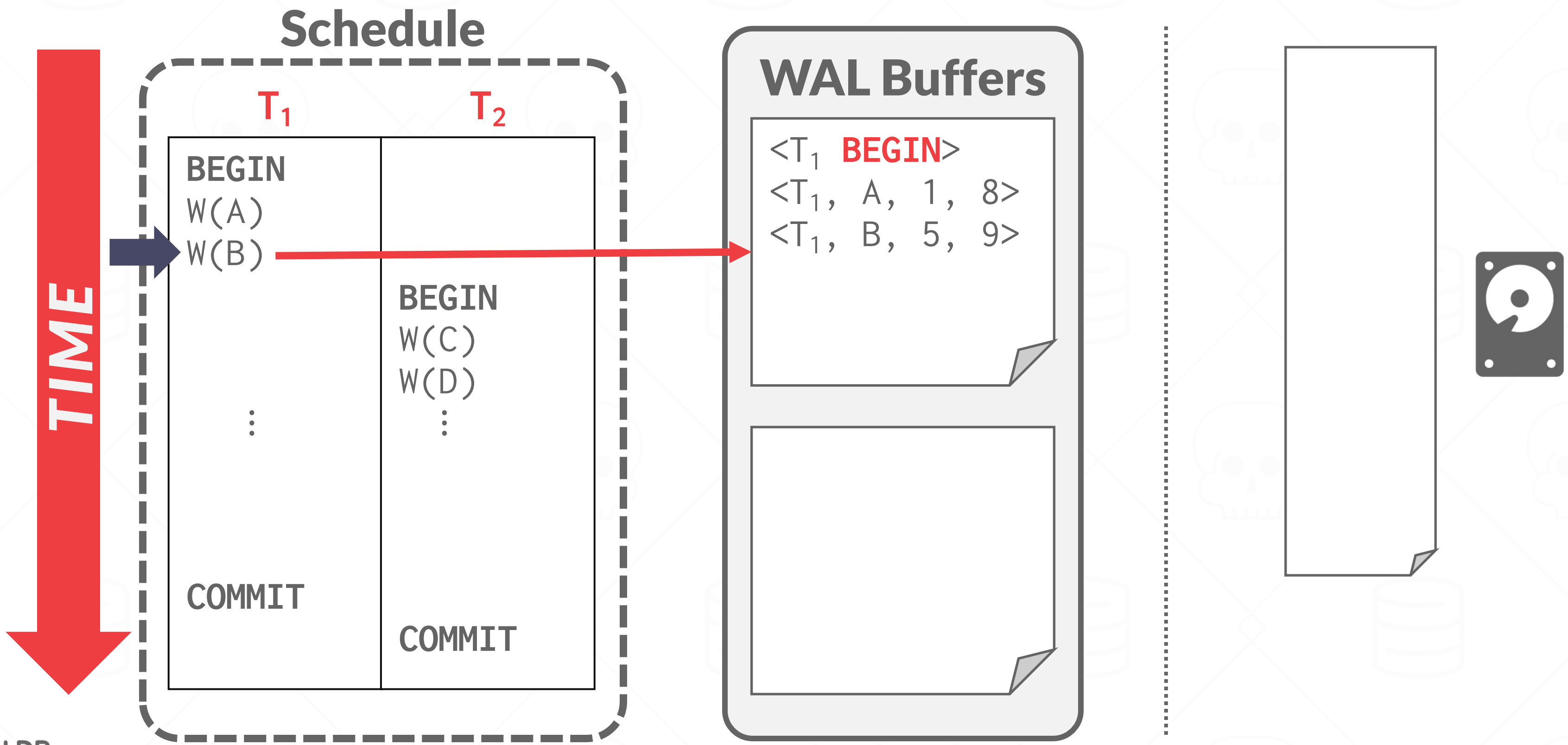
WAL – GROUP COMMIT



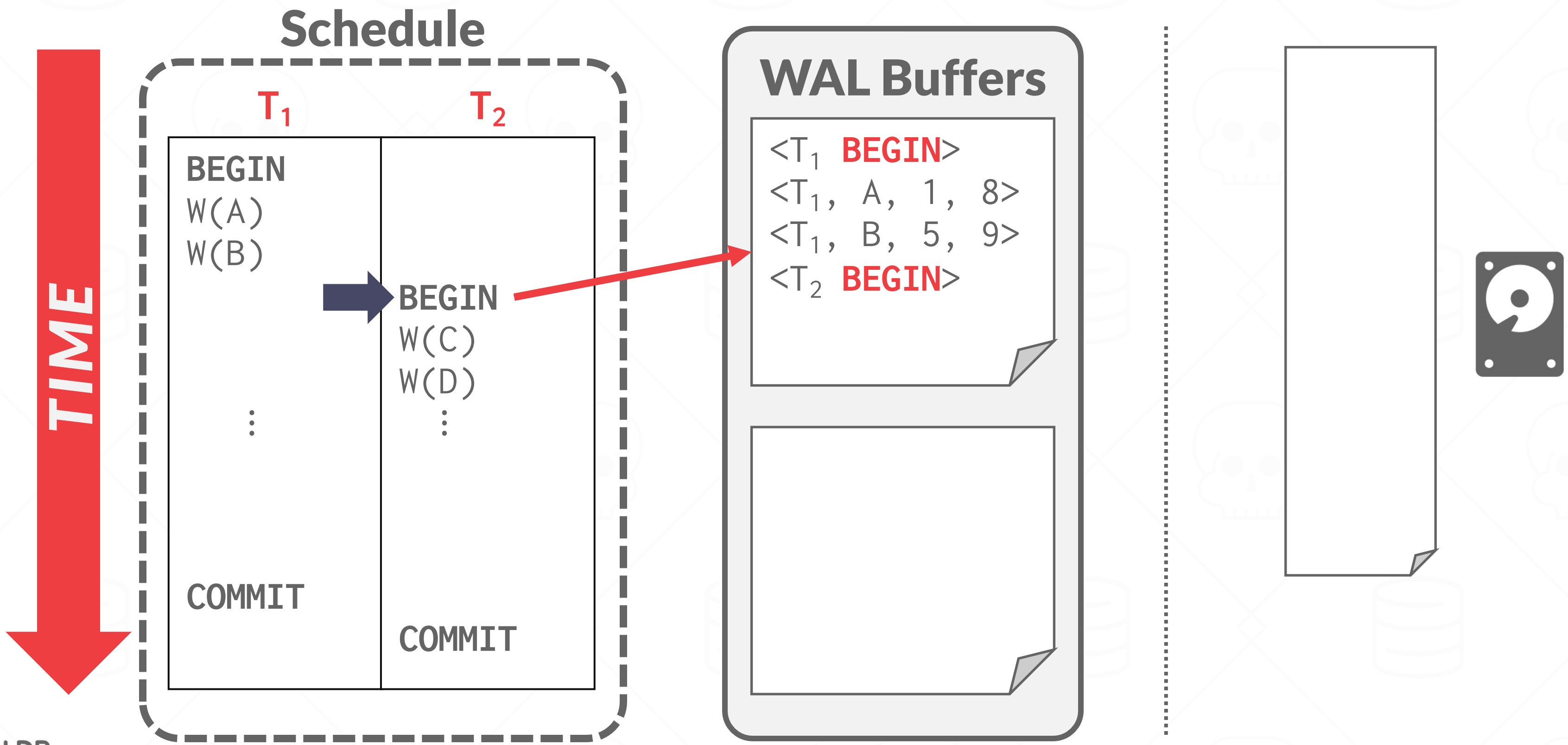
WAL – GROUP COMMIT



WAL – GROUP COMMIT



WAL – GROUP COMMIT



WAL – GROUP COMMIT

TIME

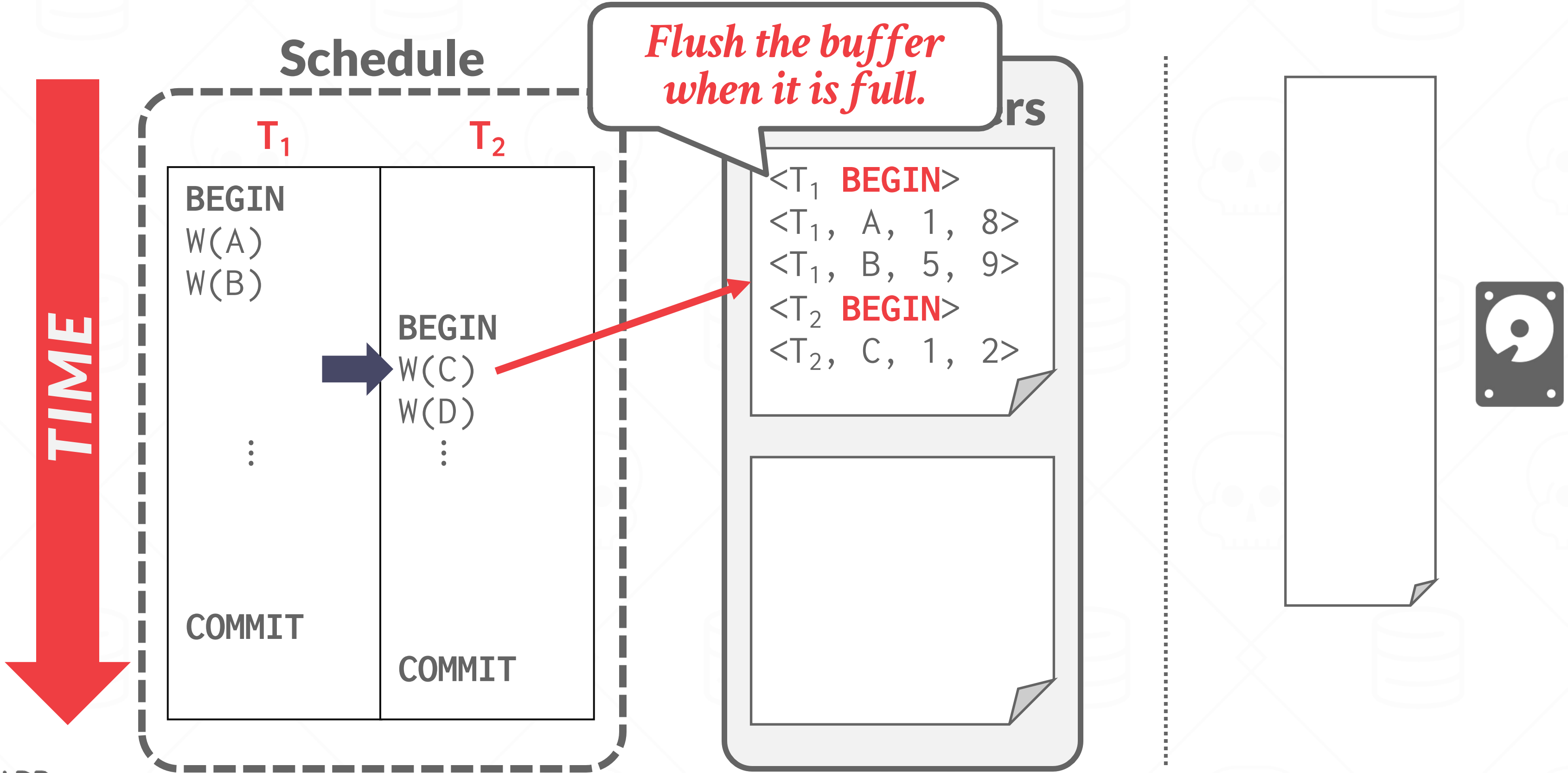
Schedule

T_1	T_2
BEGIN	
W(A)	
W(B)	
	BEGIN
	W(C)
	W(D)
⋮	⋮
COMMIT	
	COMMIT

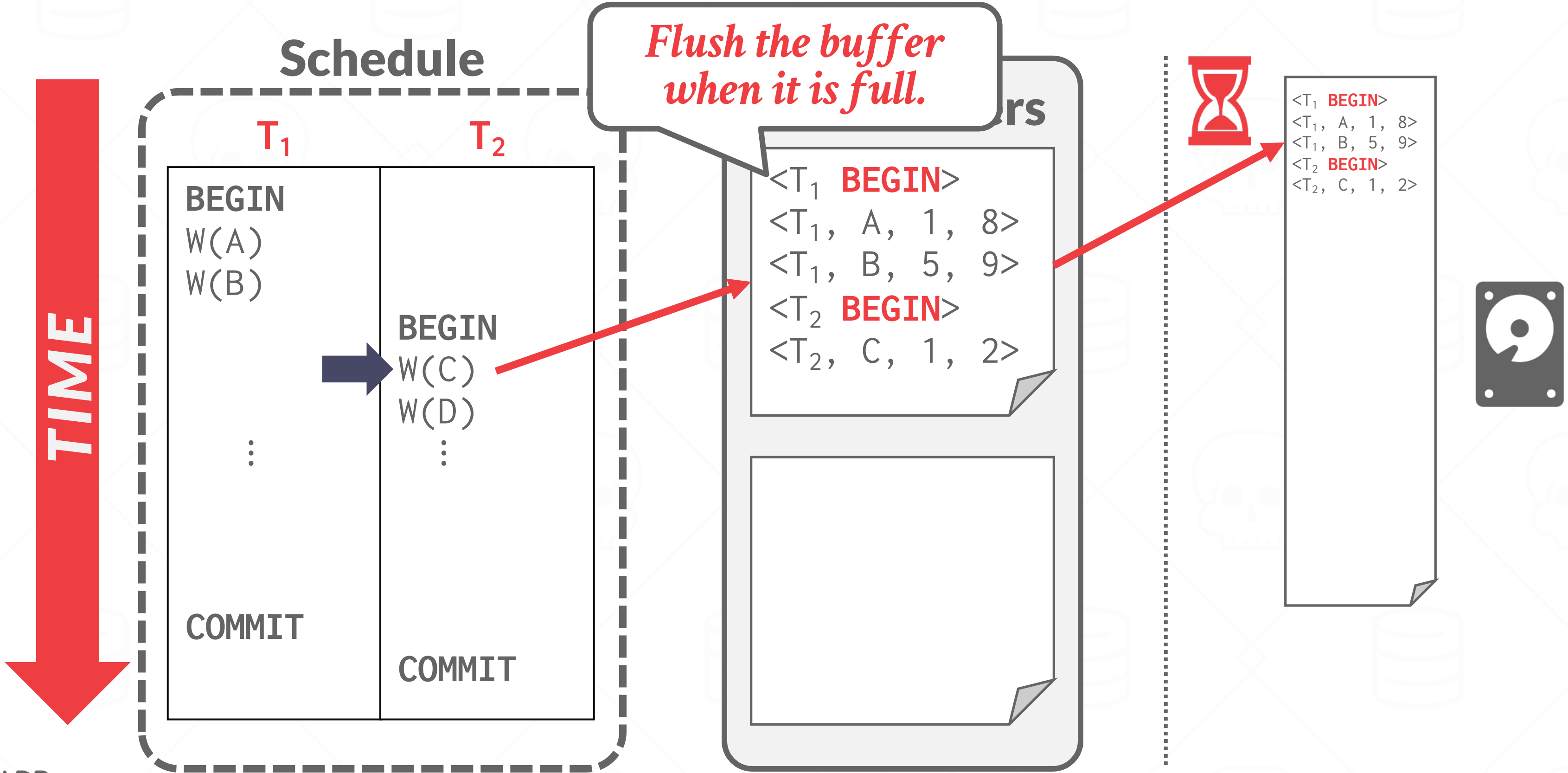
WAL Buffers

<T_1 BEGIN>
<T_1, A, 1, 8>
<T_1, B, 5, 9>
<T_2 BEGIN>
<T_2, C, 1, 2>

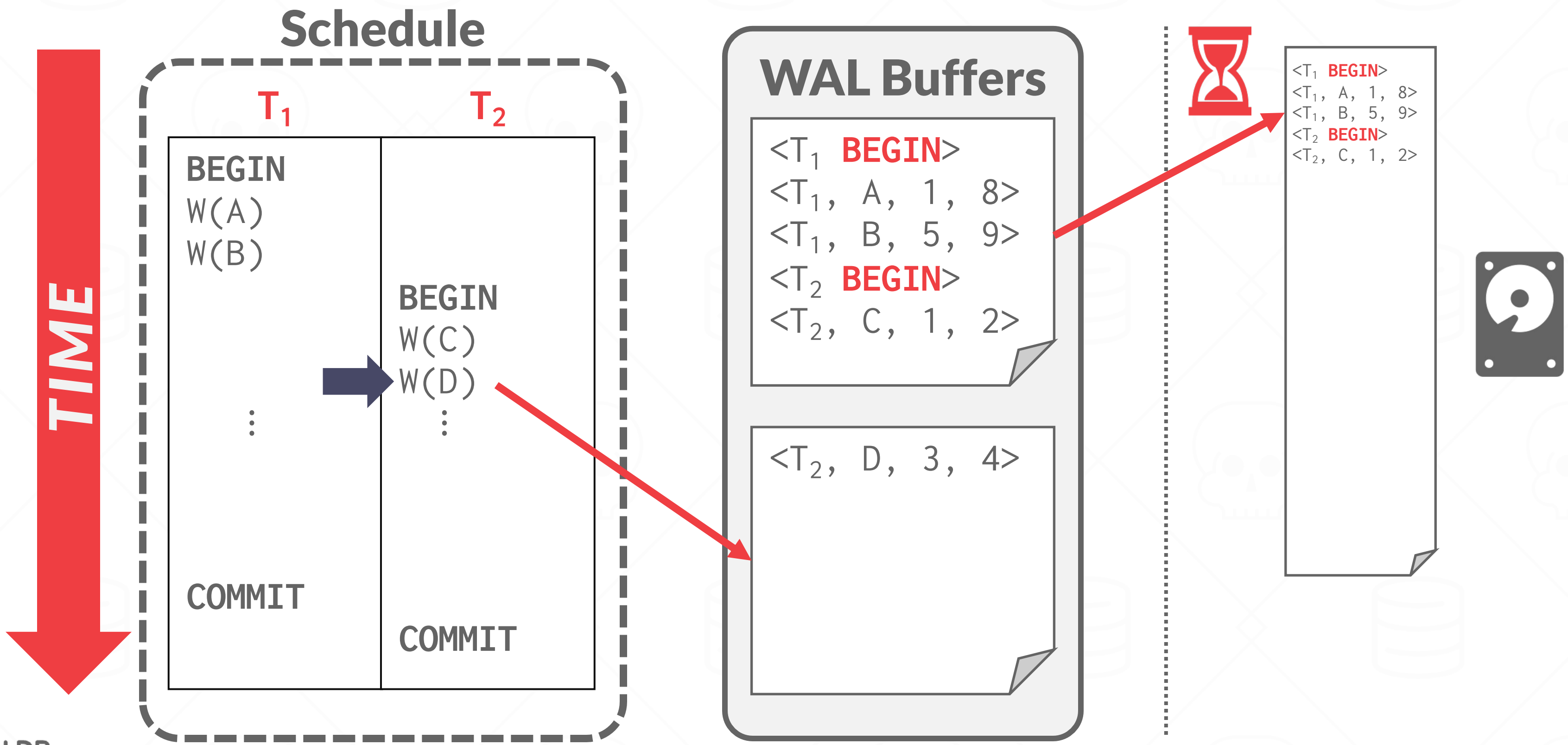
WAL – GROUP COMMIT



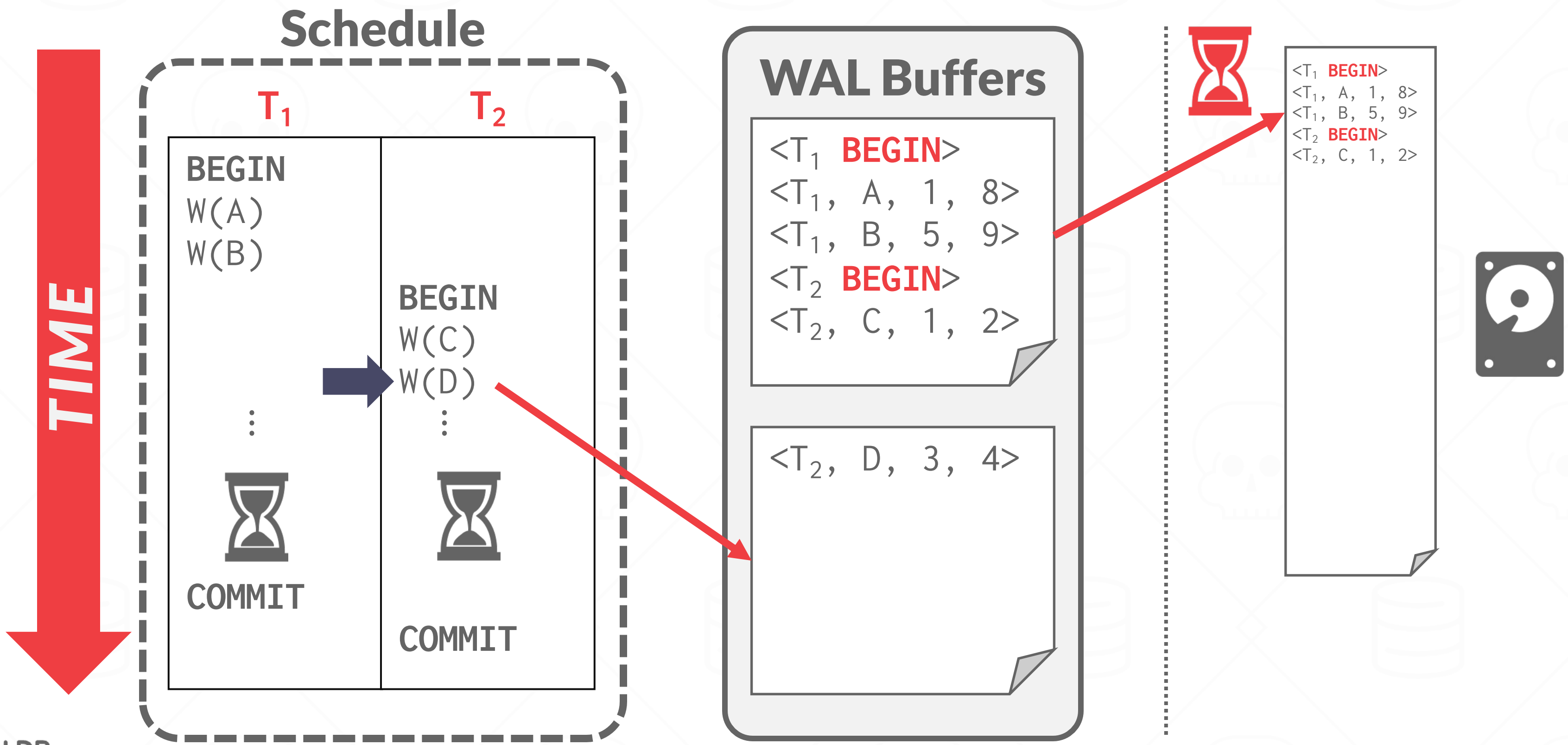
WAL – GROUP COMMIT



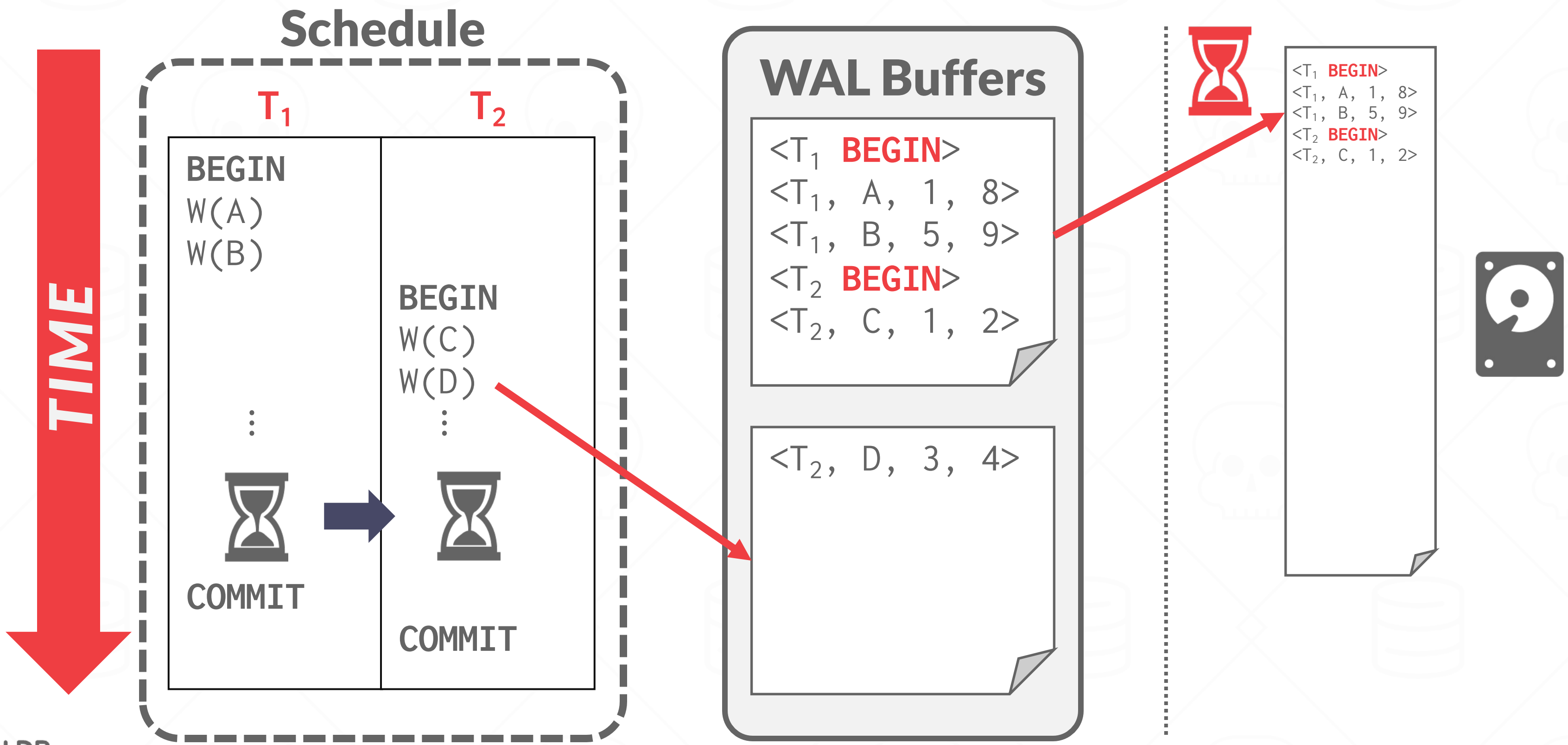
WAL – GROUP COMMIT



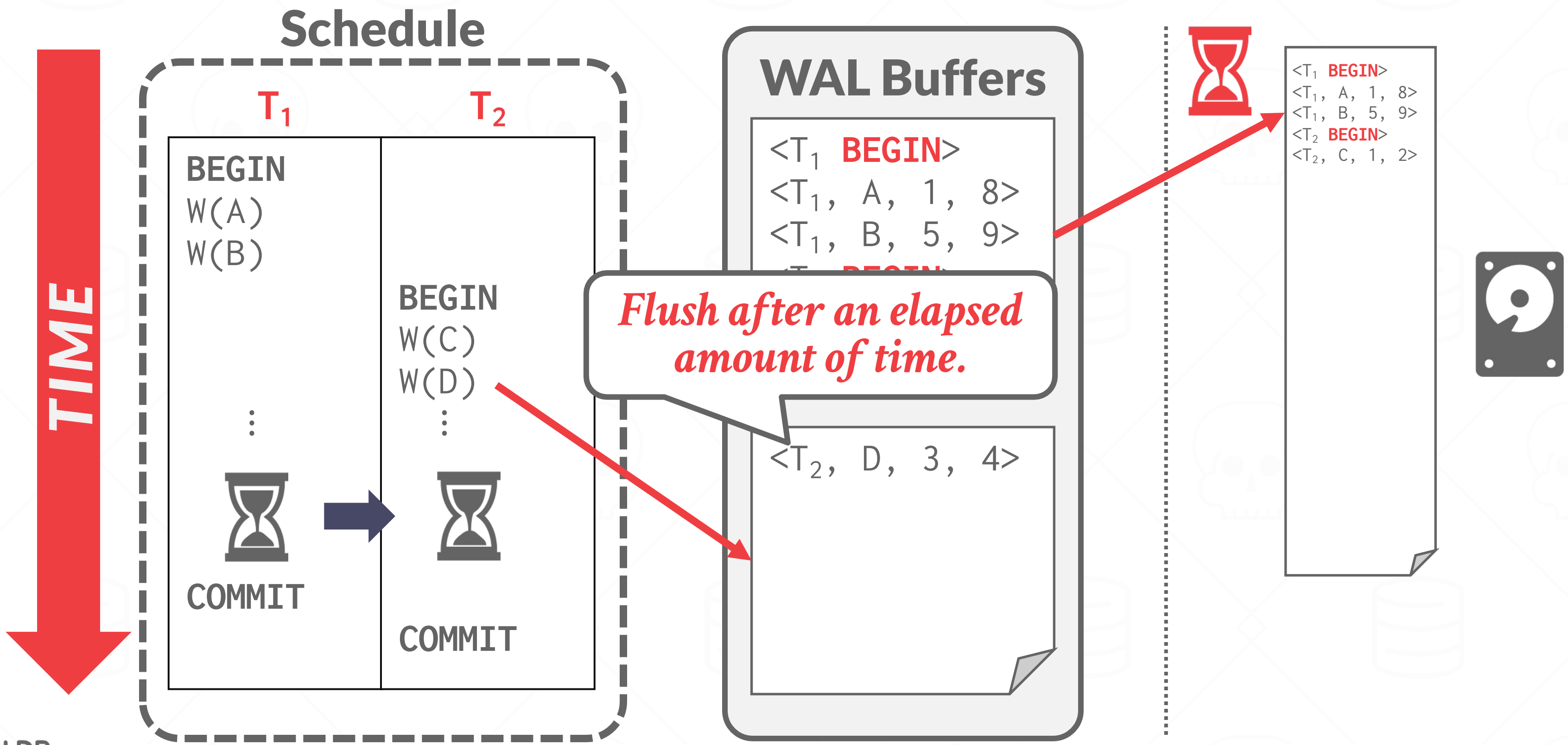
WAL – GROUP COMMIT



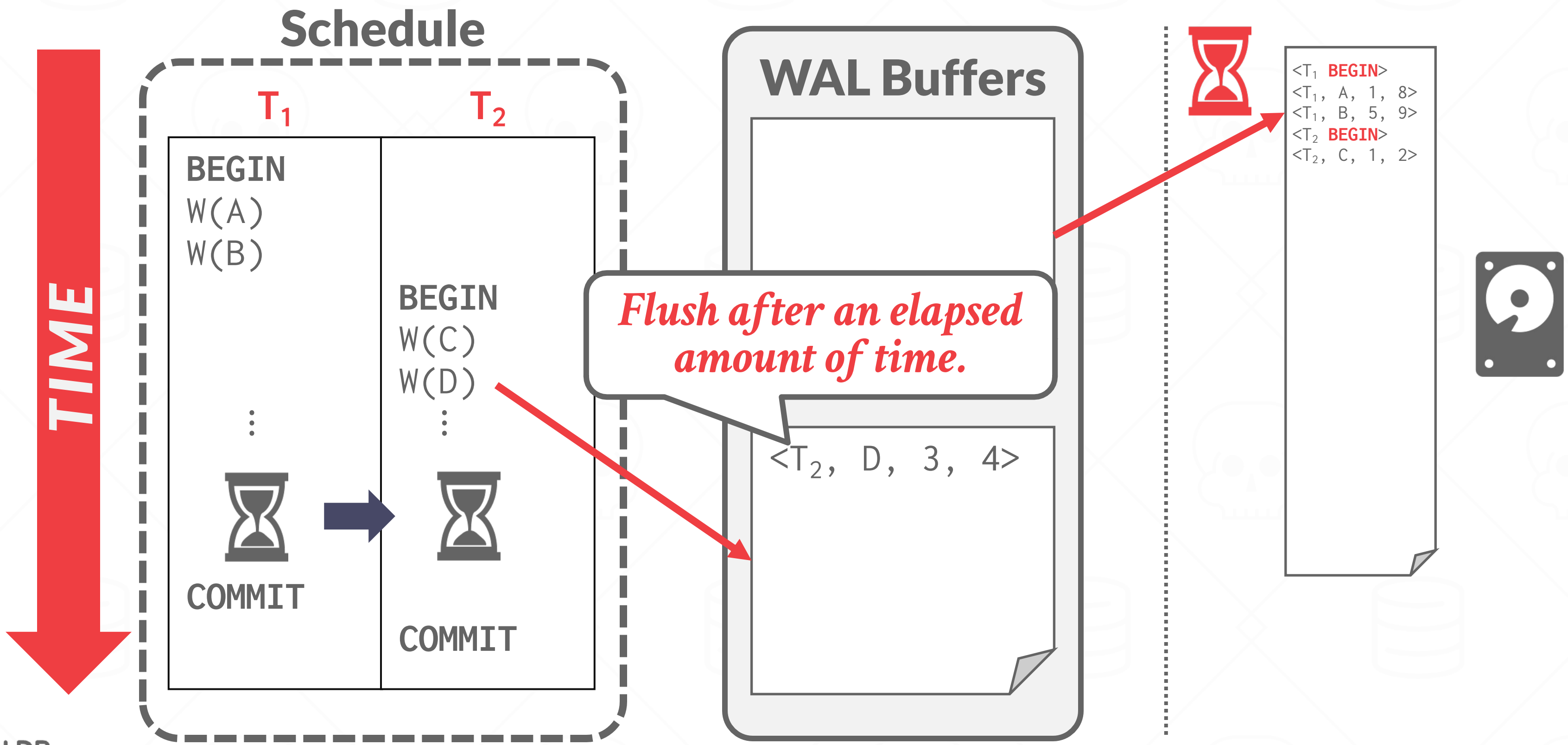
WAL – GROUP COMMIT



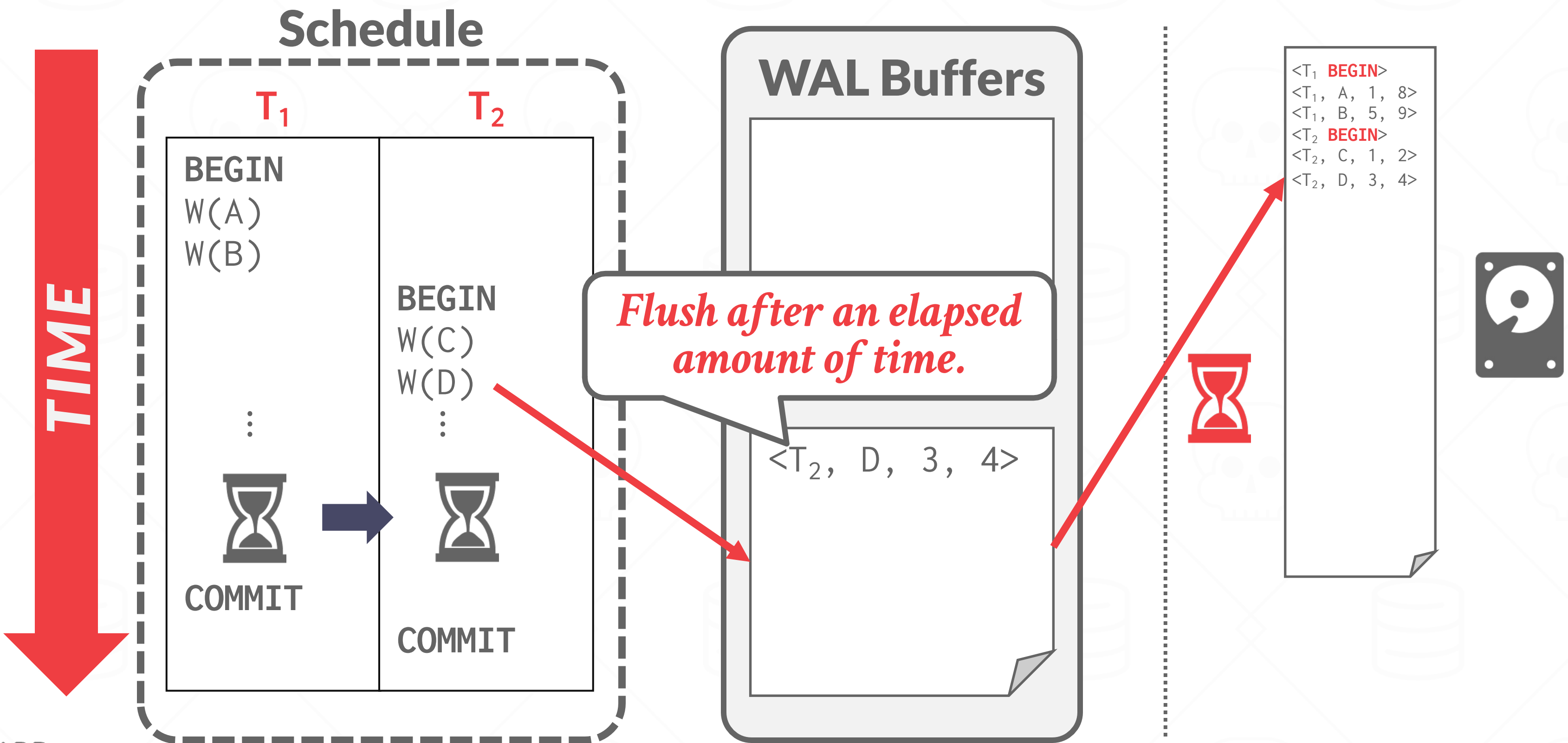
WAL – GROUP COMMIT



WAL – GROUP COMMIT



WAL – GROUP COMMIT



BUFFER POOL POLICIES

Almost every DBMS uses **NO-FORCE** + **STEAL**

Runtime Performance

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

Recovery Performance

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

Undo + Redo

No Undo + No Redo

LOGGING SCHEMES

Physical Logging

→ Record the byte-level changes made to a specific page.

→ Example: `git diff`

Logical Logging

→ Record the high-level operations executed by txns.

→ Example: `UPDATE`, `DELETE`, and `INSERT` queries.

Physiological Logging

→ Physical-to-a-page, logical-within-a-page.

→ Hybrid approach with byte-level changes for a single tuple identified by page id + slot number.

→ Does not specify organization of the page.

LOGGING SCHEMES

`UPDATE foo SET val = XYZ WHERE id = 1;`

Physical

<T_1,
Table=X,
Page=99,
Offset=1024,
Before=ABC,
After=XYZ>
<T_1,
Index=X_PKEY,
Page=45,
Offset=9,
Key=(1,Record1)>

Logical

<T_1,
Query="**UPDATE** foo
SET val=XYZ
WHERE id=1">

Physiological

<T_1,
Table=X,
Page=99,
Slot=1,
Before=ABC,
After=XYZ>
<T_1,
Index=X_PKEY,
IndexPage=45,
Key=(1,Record1)>

PHYSICAL VS. LOGICAL LOGGING

Logical logging requires less data written in each log record than physical logging.

Difficult to implement recovery with logical logging if you have concurrent txns running at lower isolation levels.

→ Hard to determine which parts of the database may have been modified by a query before crash.

→ Also takes longer to recover because you must re-execute every txn all over again.

LOG-STRUCTURED SYSTEMS

Log-structured DBMSs do not have dirty pages.
→ Any page retrieved from disk is immutable.

The DBMS buffers log records in in-memory pages (MemTable). If this buffer is full, it must be flushed to disk. But it may contain changes uncommitted txns.

These DBMSs still maintain a separate WAL to recreate the MemTable on crash.

CHECKPOINTS

The WAL will grow forever.

After a crash, the DBMS must replay the entire log, which will take a long time.

The DBMS periodically takes a checkpoint where it flushes all buffers out to disk.
→ This provides a hint on how far back it needs to replay the WAL after a crash.

CHECKPOINTS

Blocking / Consistent Checkpoint Protocol:

→ Pause all queries.

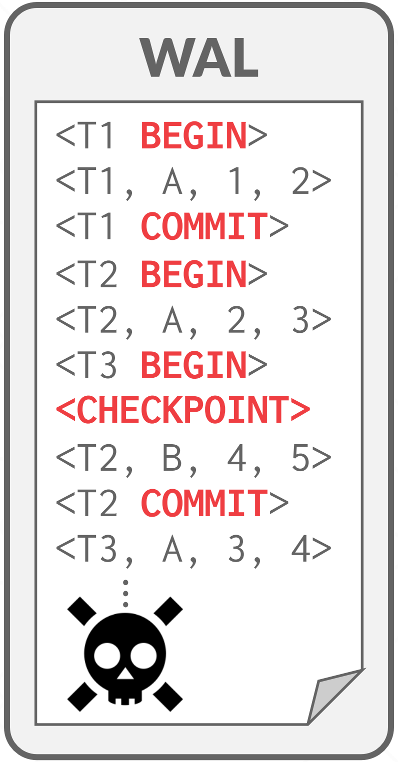
→ Flush all WAL records in memory to disk.

→ Flush all modified pages in the buffer pool to disk.

→ Write a `<CHECKPOINT>` entry to WAL and flush to disk.

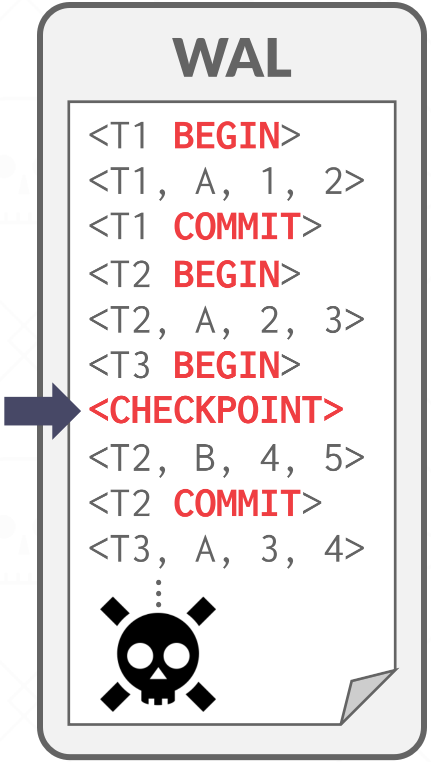
→ Resume queries.

CHECKPOINTS



CHECKPOINTS

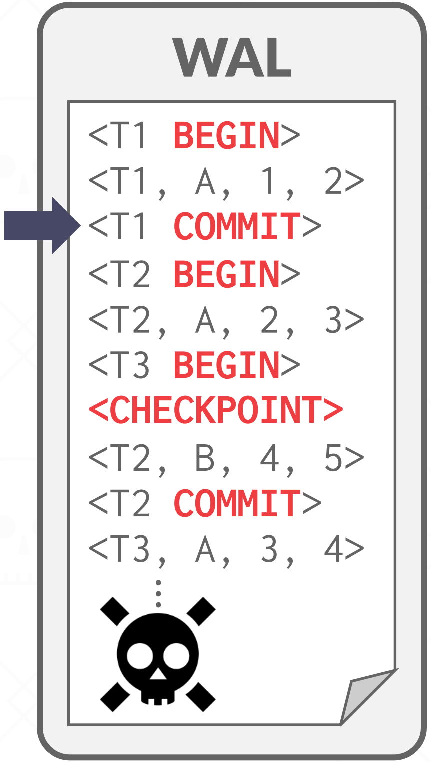
Use the **<CHECKPOINT>** record as the starting point for analyzing the WAL.



CHECKPOINTS

Use the <CHECKPOINT> record as the starting point for analyzing the WAL.

Any txn that committed before the checkpoint is ignored (T_1).



CHECKPOINTS

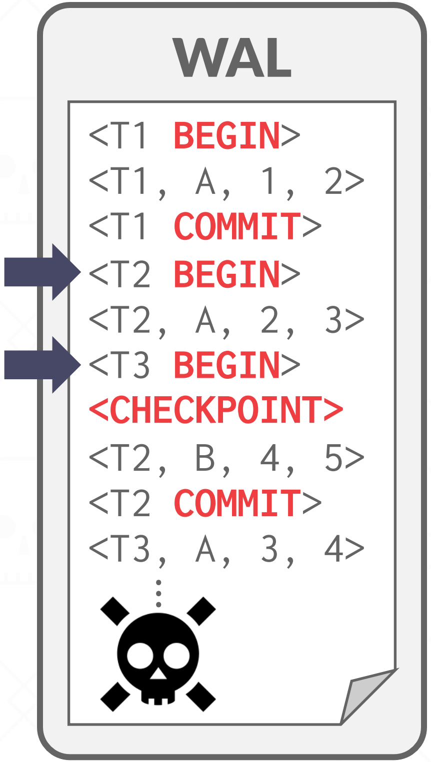
Use the **<CHECKPOINT>** record as the starting point for analyzing the WAL.

Any txn that committed before the checkpoint is ignored (T_1).

T_2 + T_3 did not commit before the last checkpoint.

→ Need to <u>redo</u> T_2 because it committed after checkpoint.

→ Need to <u>undo</u> T_3 because it did not commit before the crash.



CHECKPOINTS – CHALLENGES

In this example, the DBMS must stall txns when it takes a checkpoint to ensure a consistent snapshot.

→ We will see how to get around this problem next class.

Scanning the log to find uncommitted txns can take a long time.

→ Unavoidable but we will add hints to the <CHECKPOINT> record to speed things up next class.

How often the DBMS should take checkpoints depends on many different factors…

CHECKPOINTS – FREQUENCY

Checkpointing too often causes the runtime performance to degrade.
→ System spends too much time flushing buffers.

But waiting a long time is just as bad:
→ The checkpoint will be large and slow.
→ Makes recovery time much longer.

Tunable option that depends on application recovery time requirements.

CONCLUSION

Write-Ahead Logging is (almost) always the best approach to handle loss of volatile storage.

Use incremental updates (**STEAL** + **NO-FORCE**) with checkpoints.

On Recovery: undo uncommitted txns + redo committed txns.

NEXT CLASS

Better Checkpoint Protocols.

Recovery with ARIES.