

 Intro to Database Systems (15-445/645)

# 19 Database Logging

Carnegie  
Mellon  
University

FALL  
2022

Andy  
Pavlo

# ADMINISTRIVIA

---

**Homework #3** is due **Sun Nov 13<sup>th</sup>** @ 11:59pm

**Project #3** is due **Wed Nov 16<sup>th</sup>** @ 11:59pm

**Live Call-in Q&A Lecture** on **Thu Dec 8<sup>th</sup>**

# UPCOMING DATABASE TALKS

## EdgeDB

→ Thursday Nov 10<sup>th</sup> @ 3:00pm

EDGE | DB

## Gaia (Database for Robots)

→ Monday Nov 14<sup>th</sup> @ 4:30pm

 GAIA

## TigerBeetle

→ Monday Nov 21<sup>st</sup> @ 4:30pm

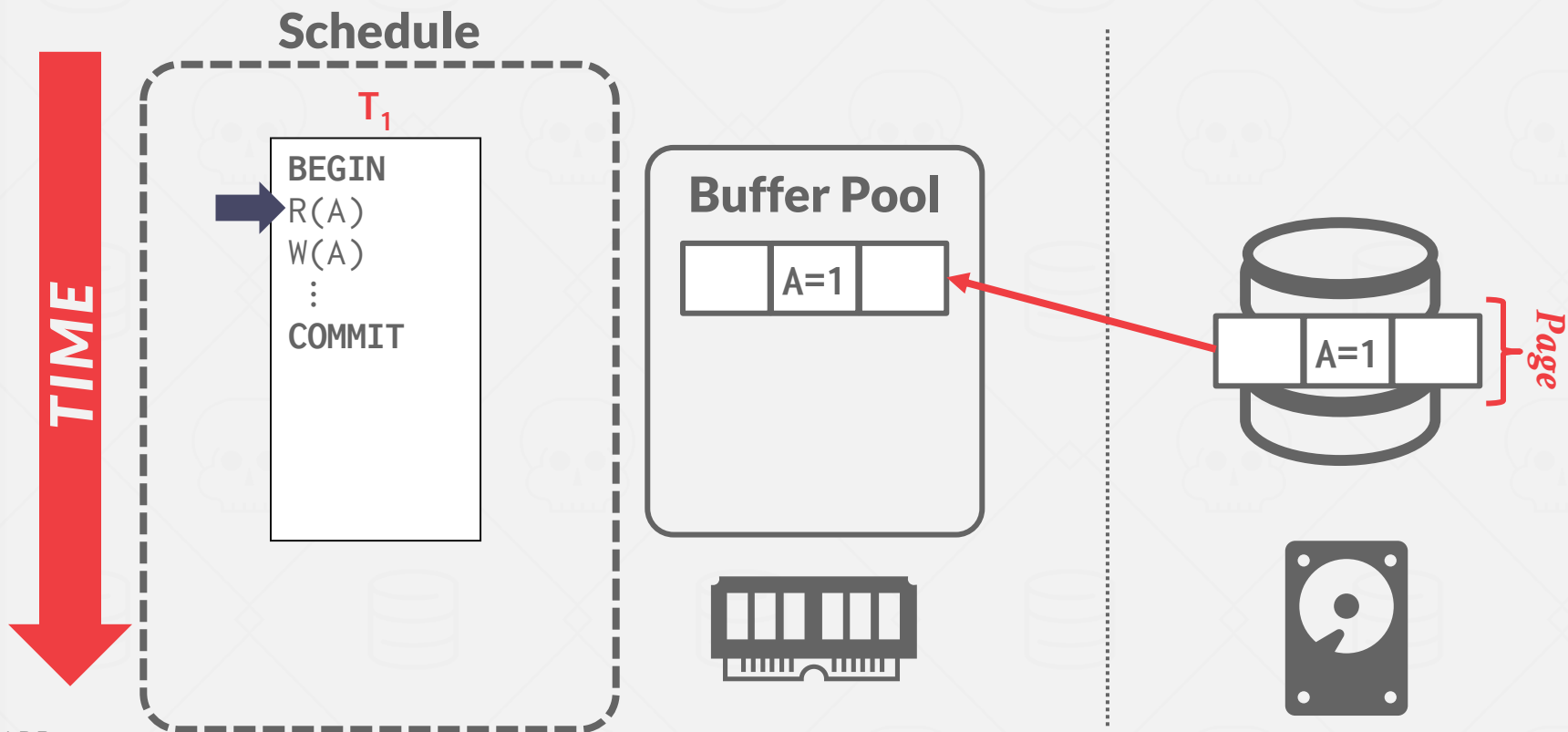
 *TigerBeetle*

## VMWare SplinterDB

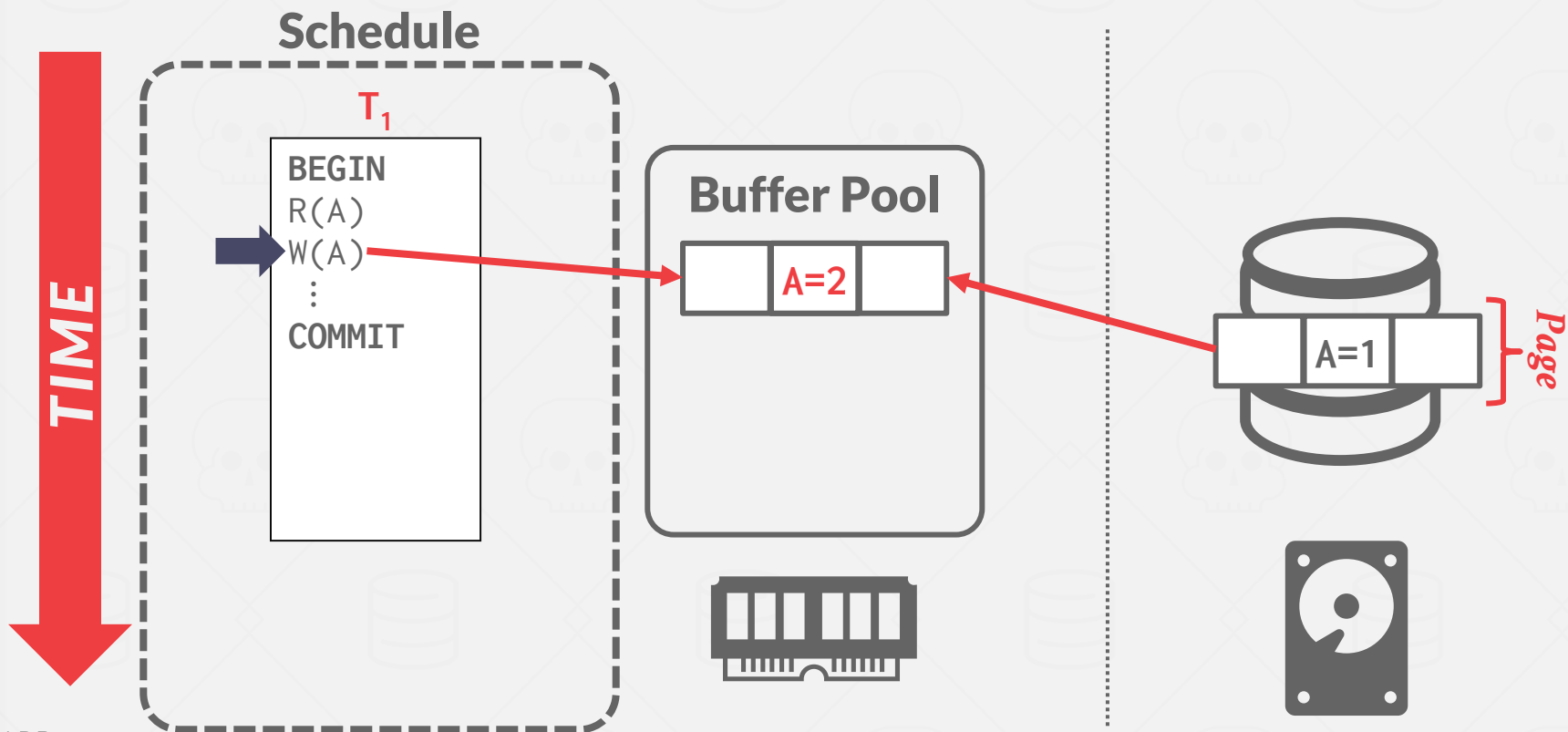
→ Monday Nov 28<sup>th</sup> @ 4:30pm

 SPLINTERDB

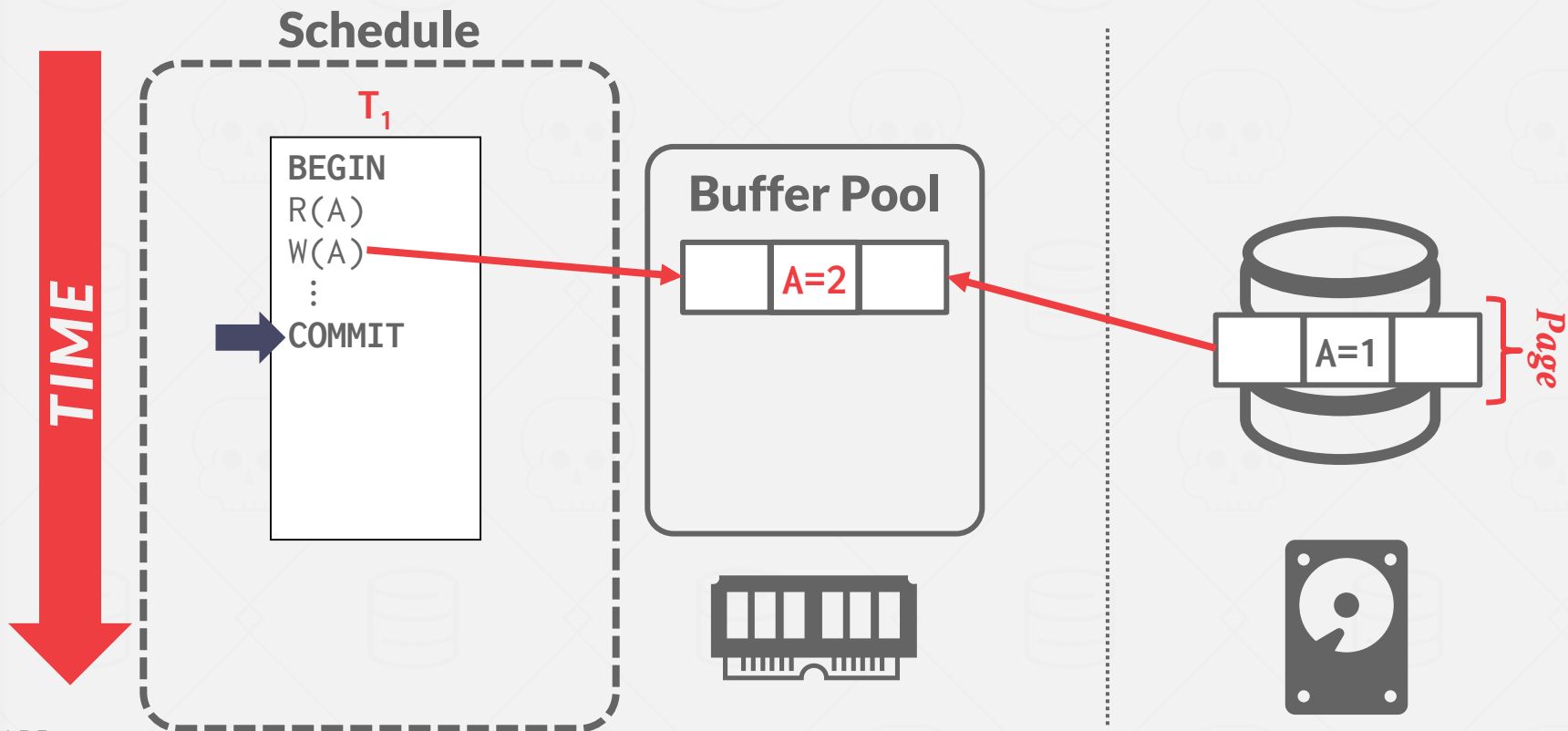
# MOTIVATION



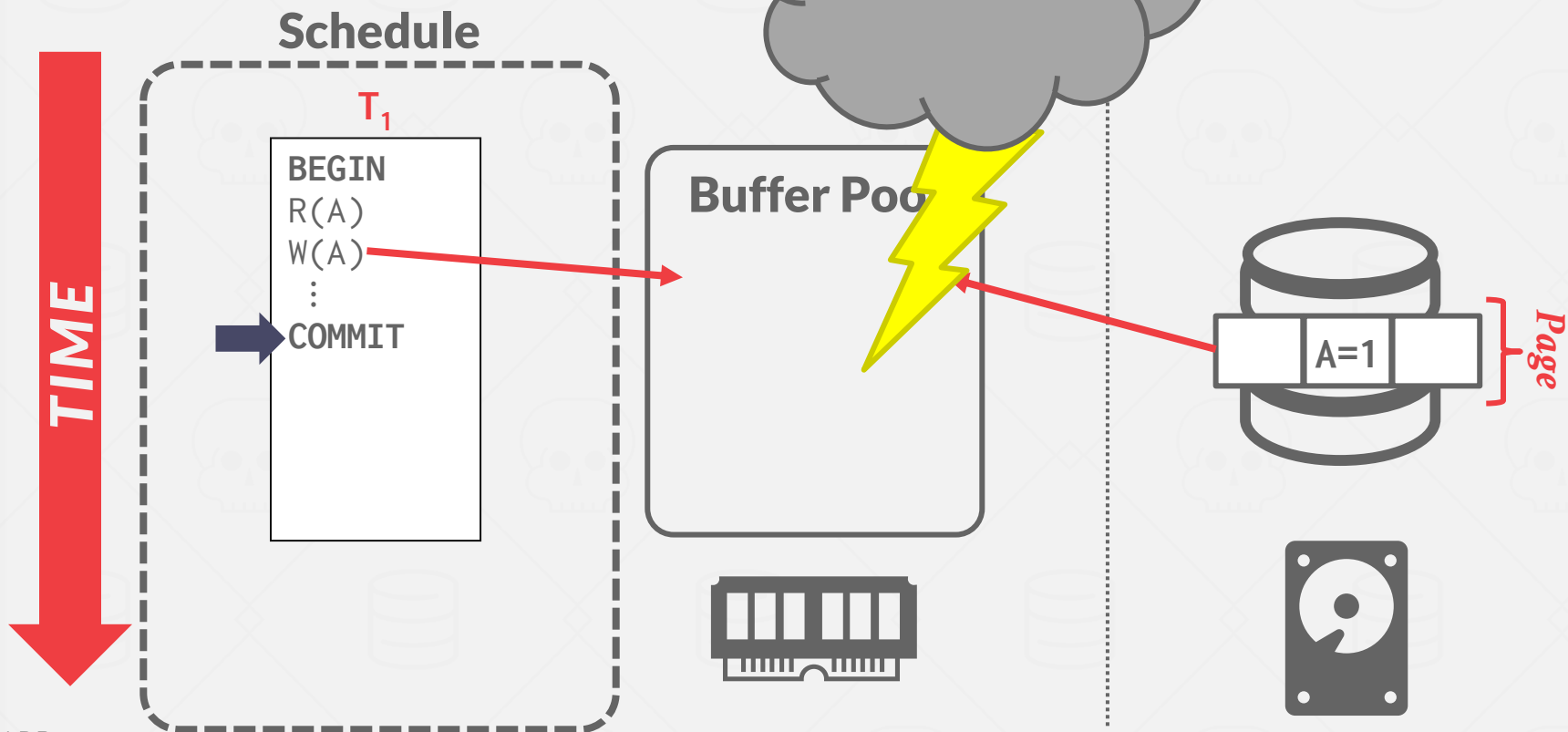
# MOTIVATION



# MOTIVATION



# MOTIVATION



# 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.

*Today*



# TODAY'S AGENDA

---

Failure Classification

Buffer Pool Policies

Shadow Paging

Write-Ahead Log

Logging Schemes

Checkpoints

# CRASH RECOVERY

---

DBMS is divided into different components based on the underlying storage device.

→ Volatile vs. Non-Volatile

We must also classify the different types of failures that the DBMS needs to handle.

# 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.

# 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).

No DBMS can recover from this! Database must be restored from 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

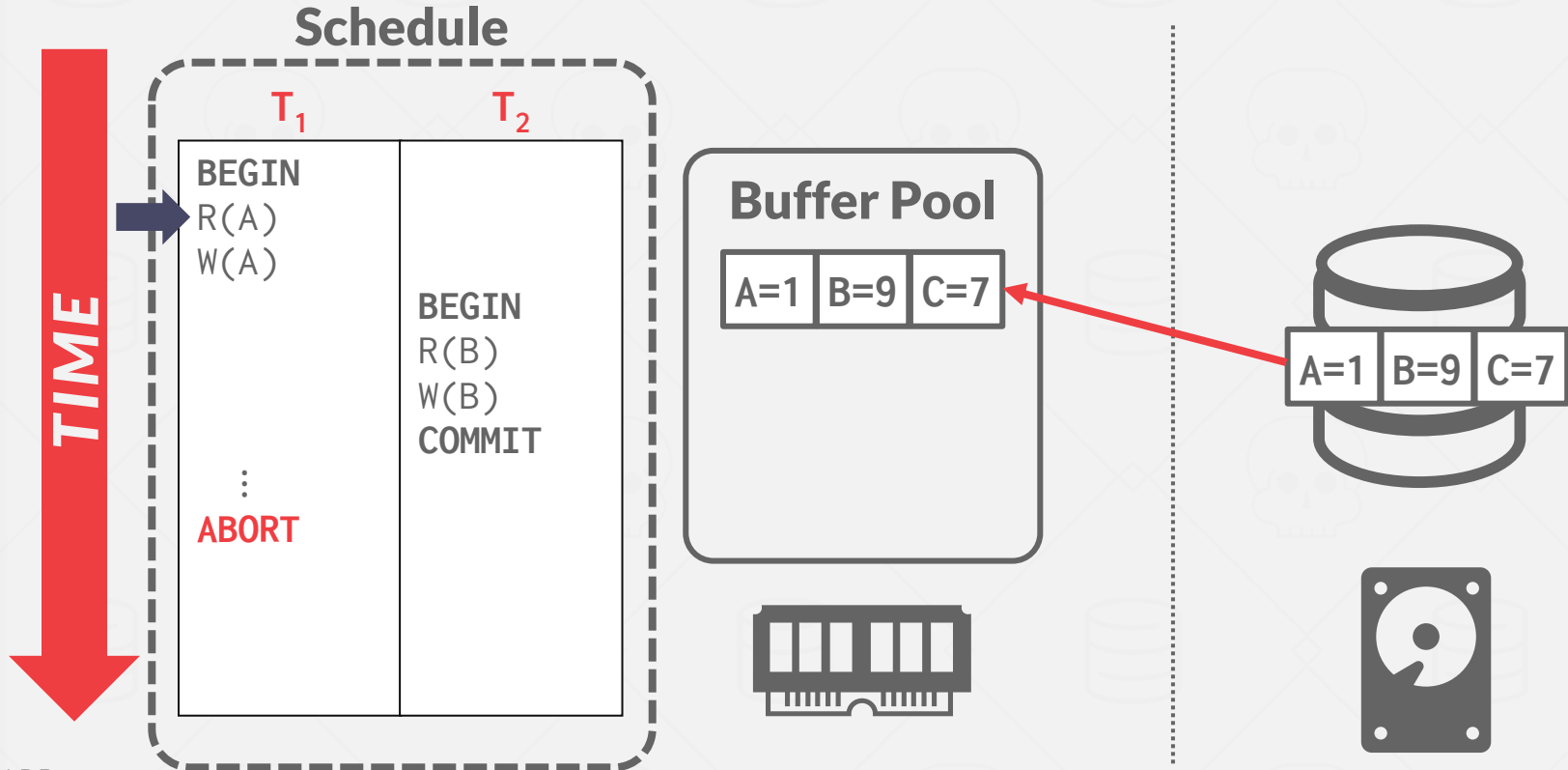
---

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

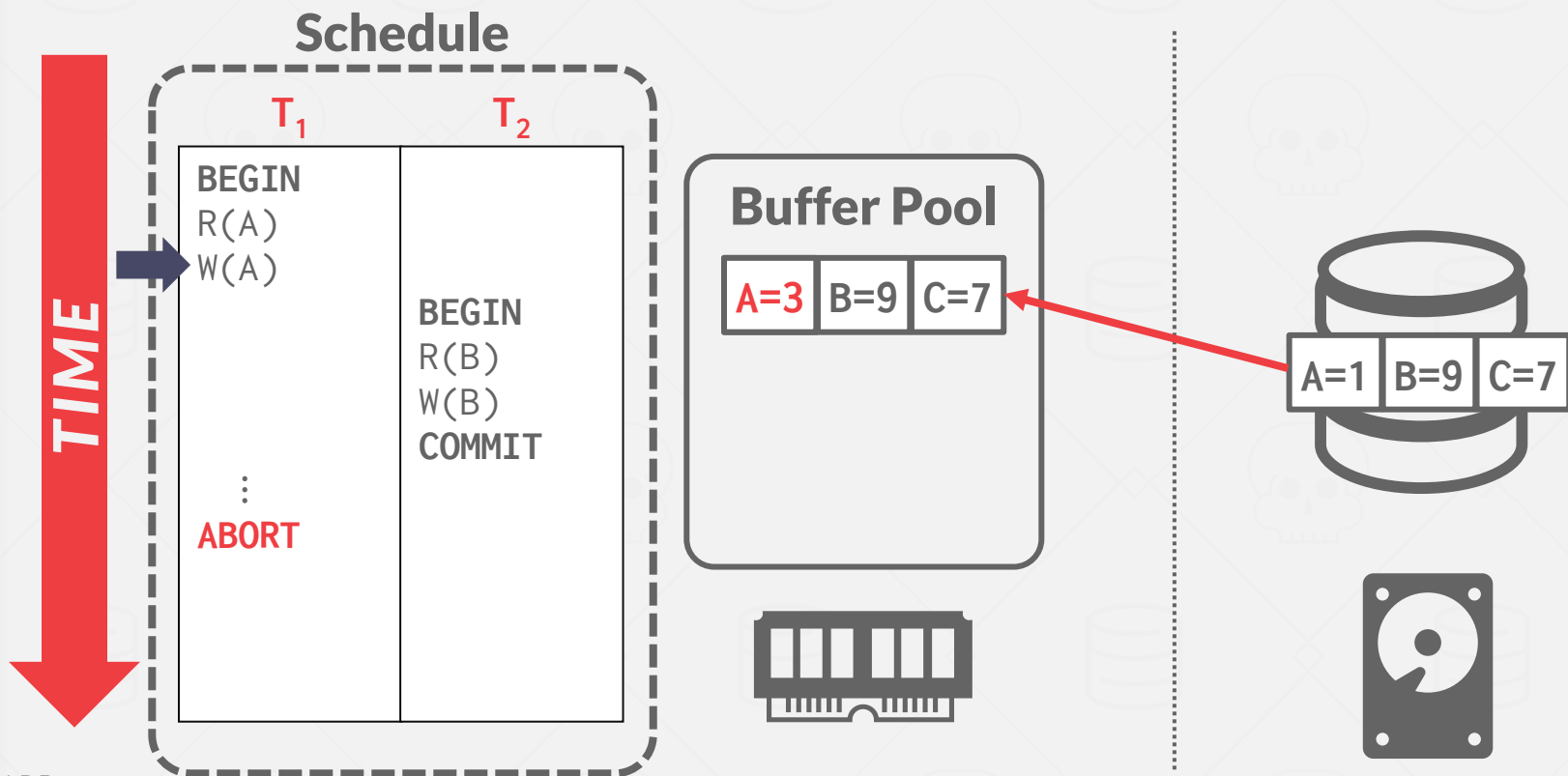
**Redo:** The process of re-applying the effects of a committed txn for durability.

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

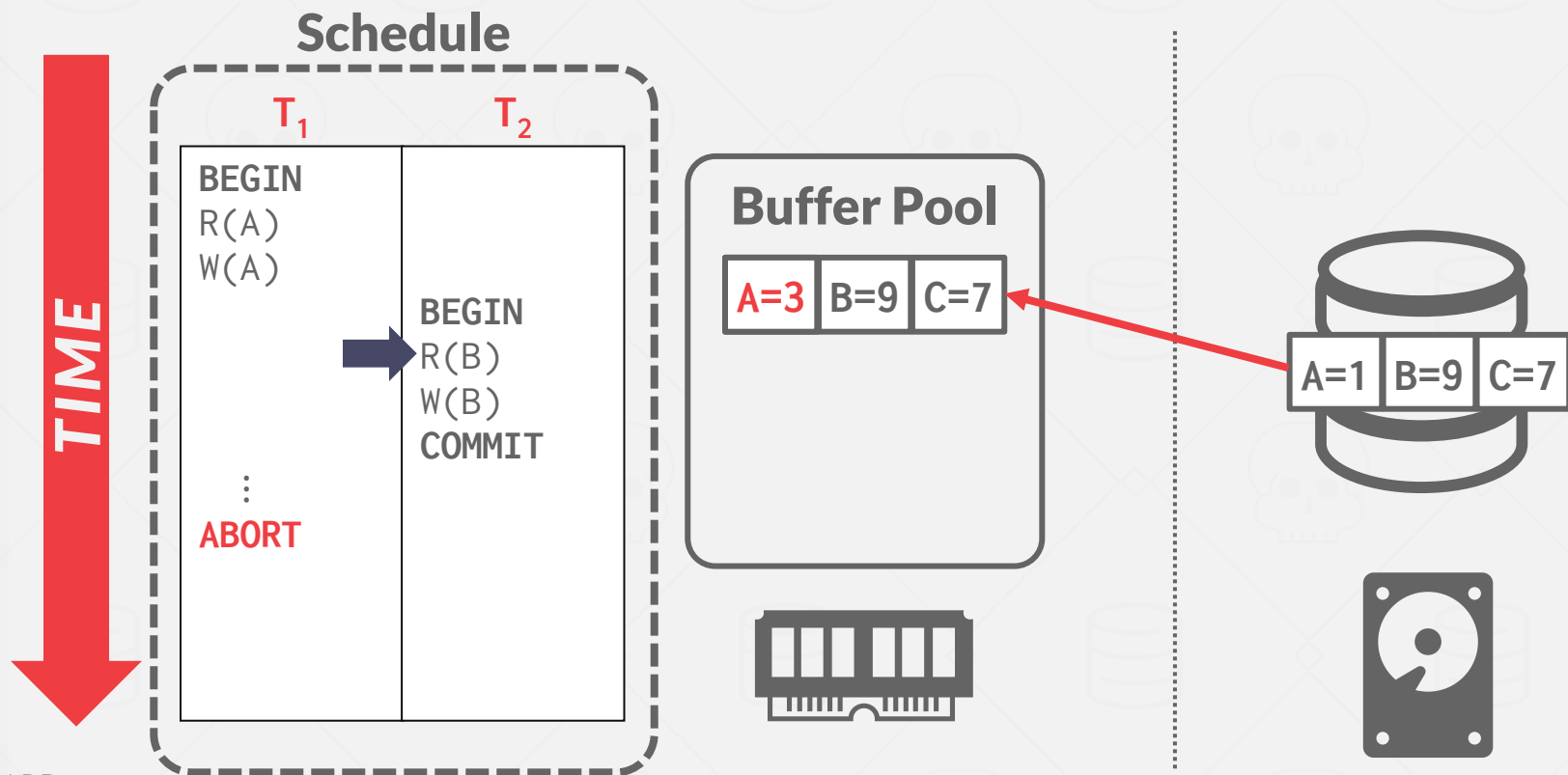
# BUFFER POOL



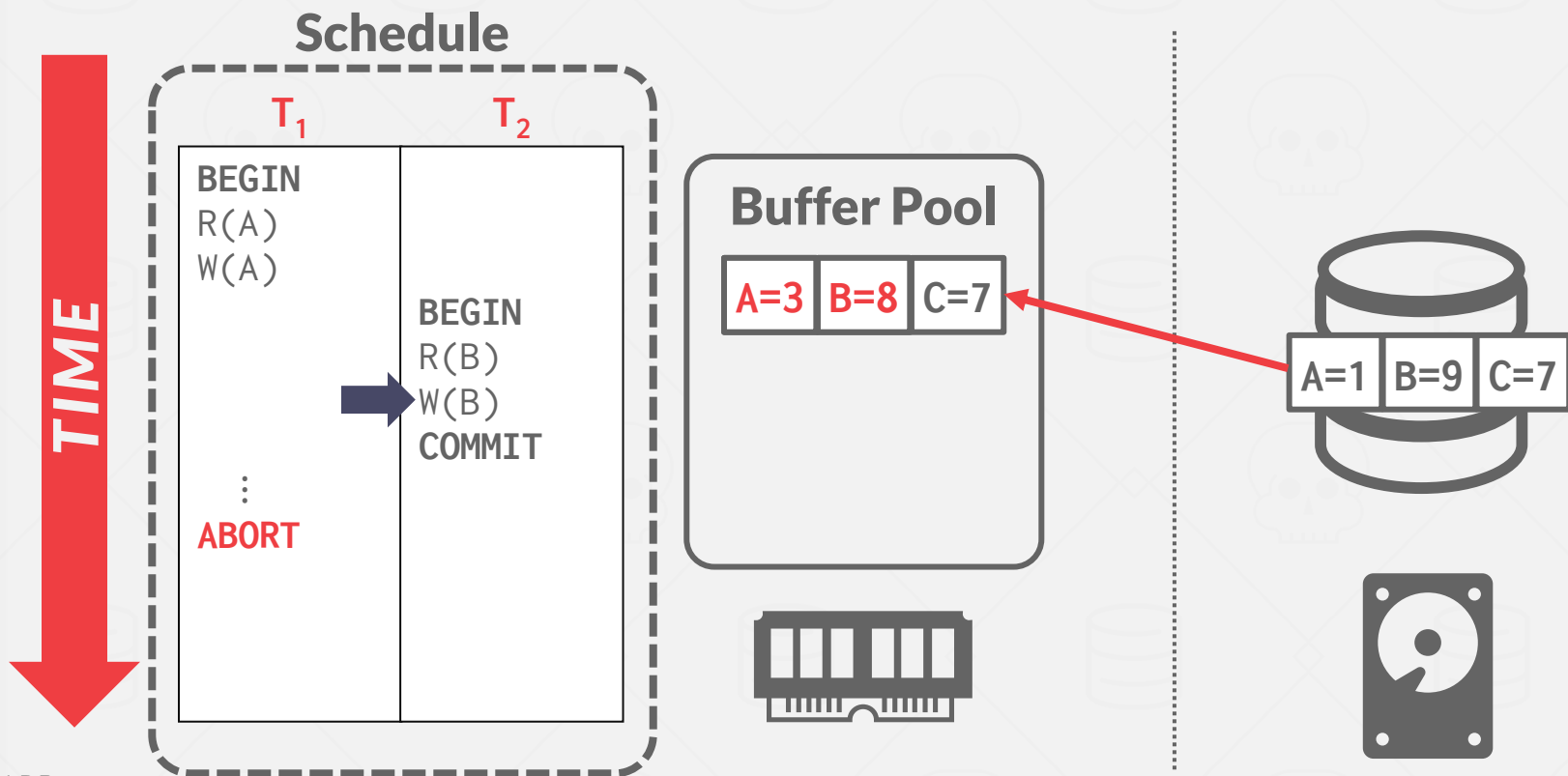
# BUFFER POOL



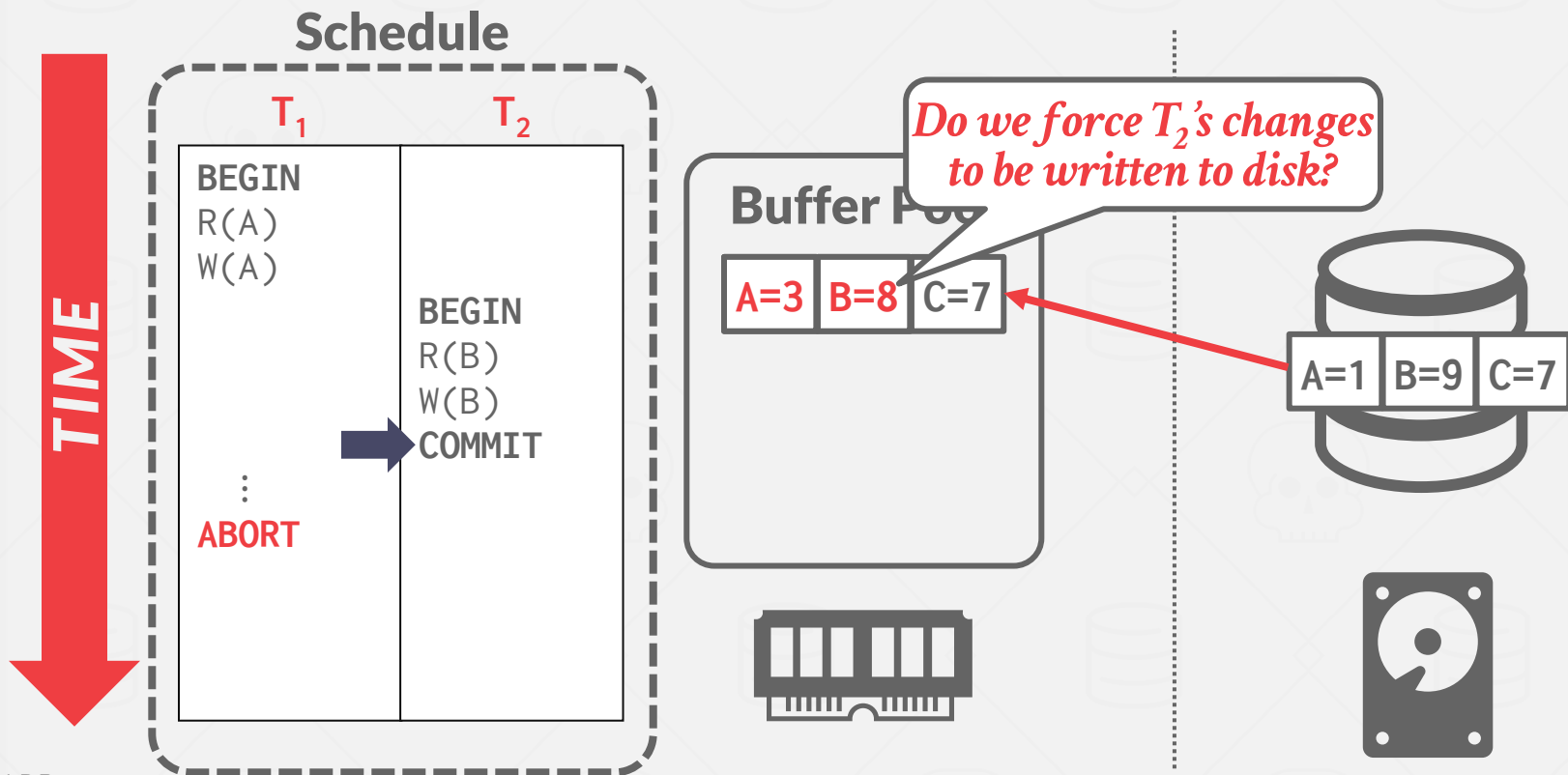
# BUFFER POOL



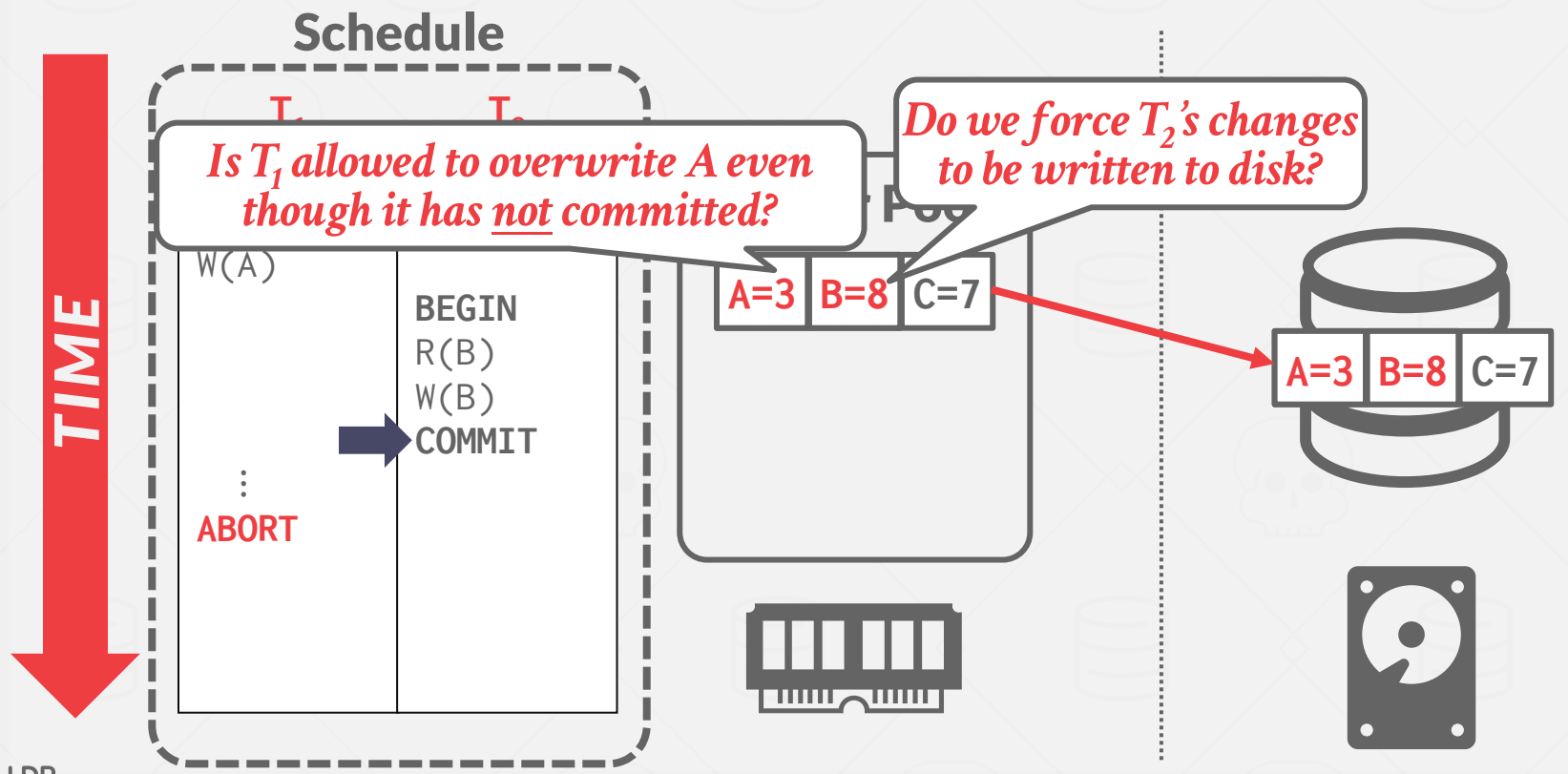
# BUFFER POOL



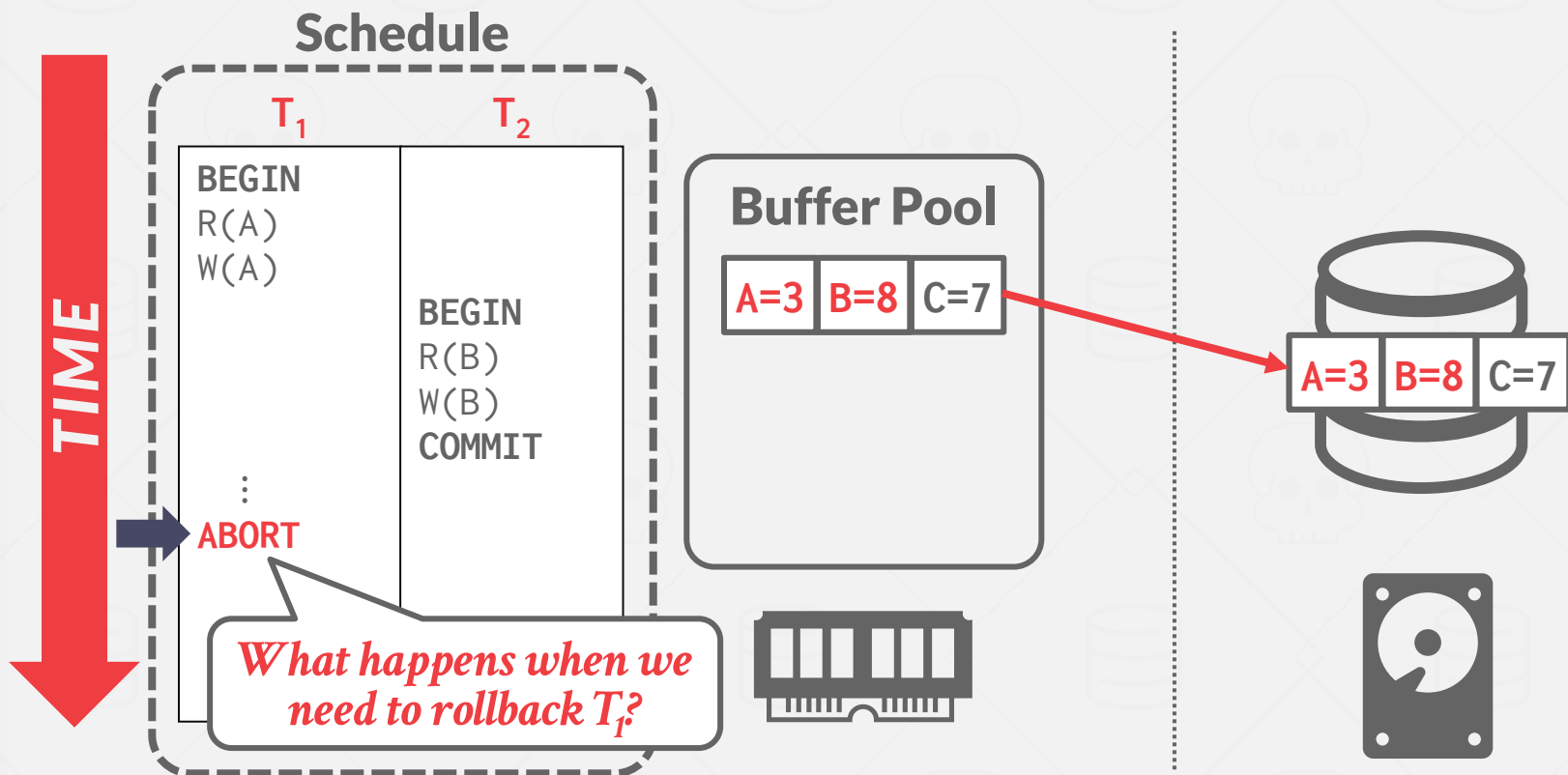
# BUFFER POOL



# BUFFER POOL



# 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

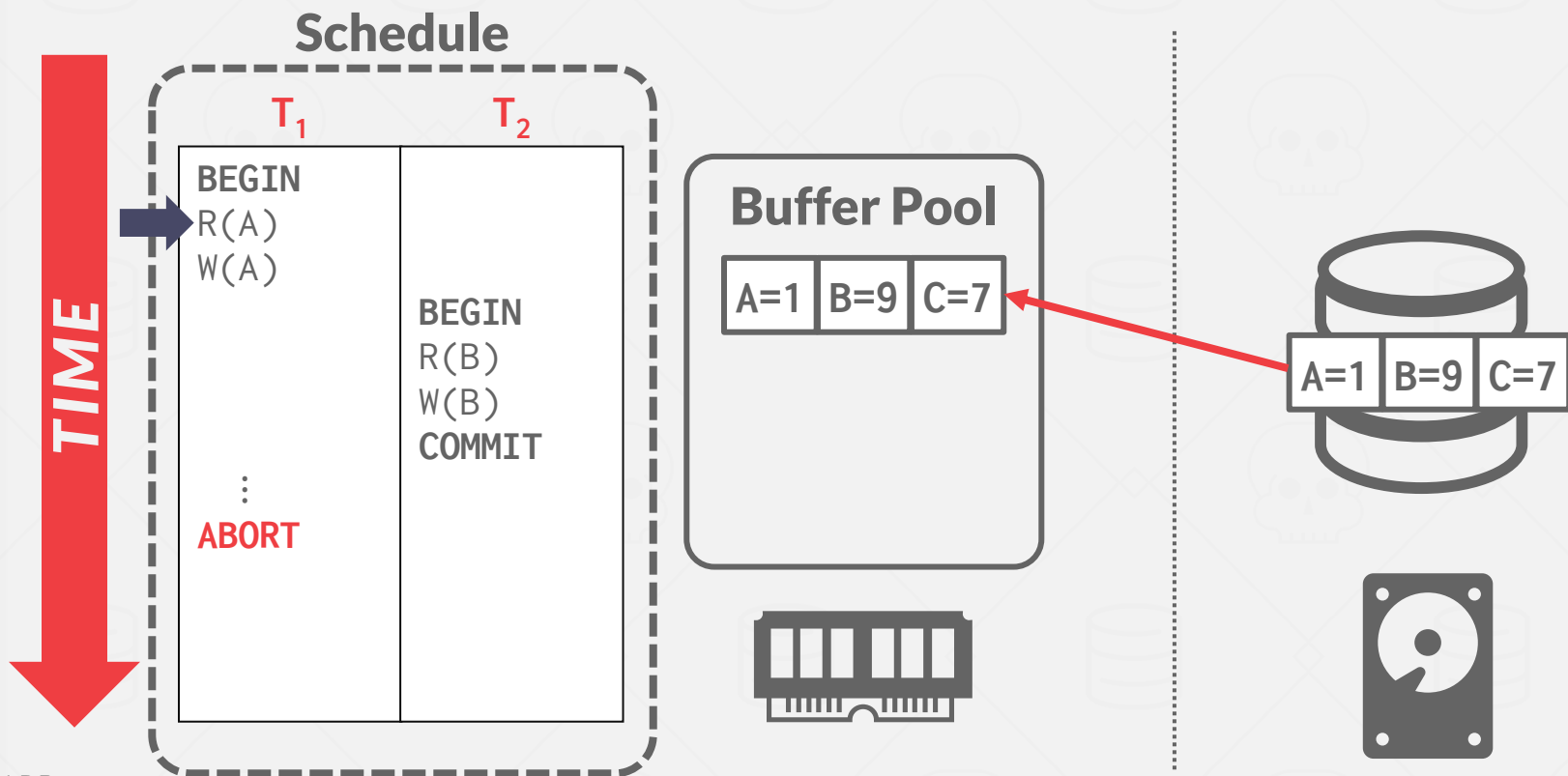
---

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

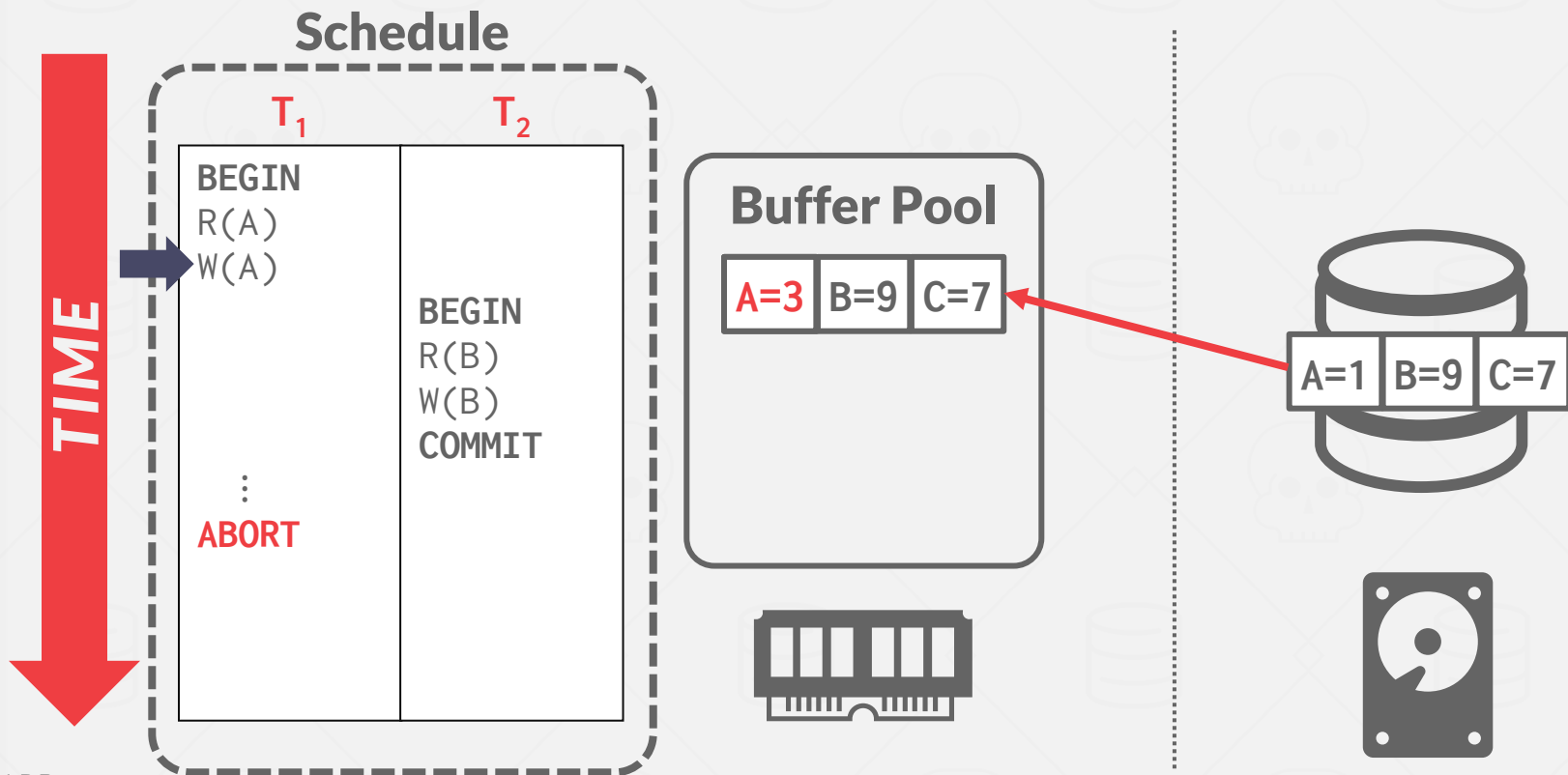
**FORCE:** Is required.

**NO-FORCE:** Is not required.

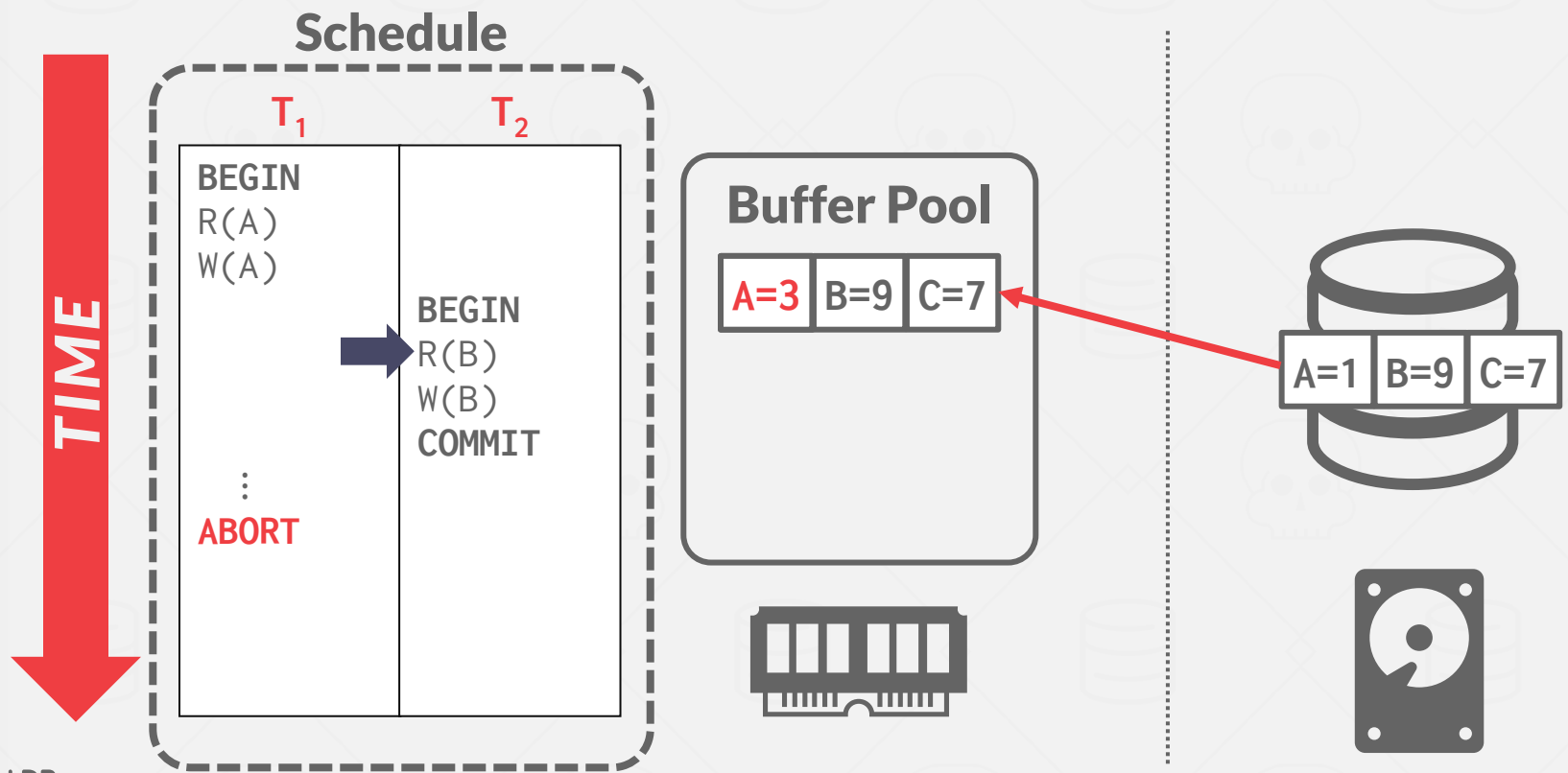
# NO-STEAL + FORCE



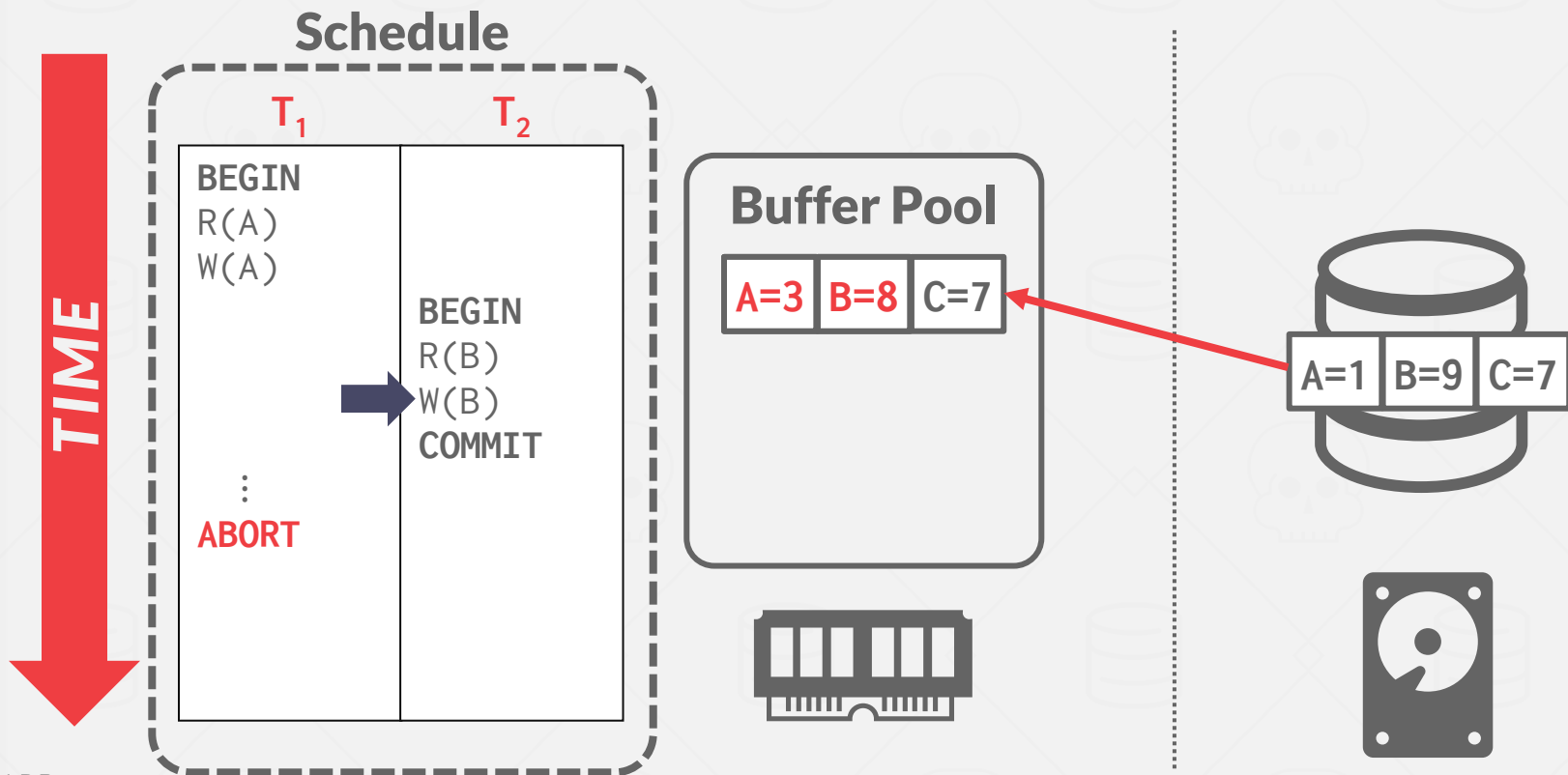
# NO-STEAL + FORCE



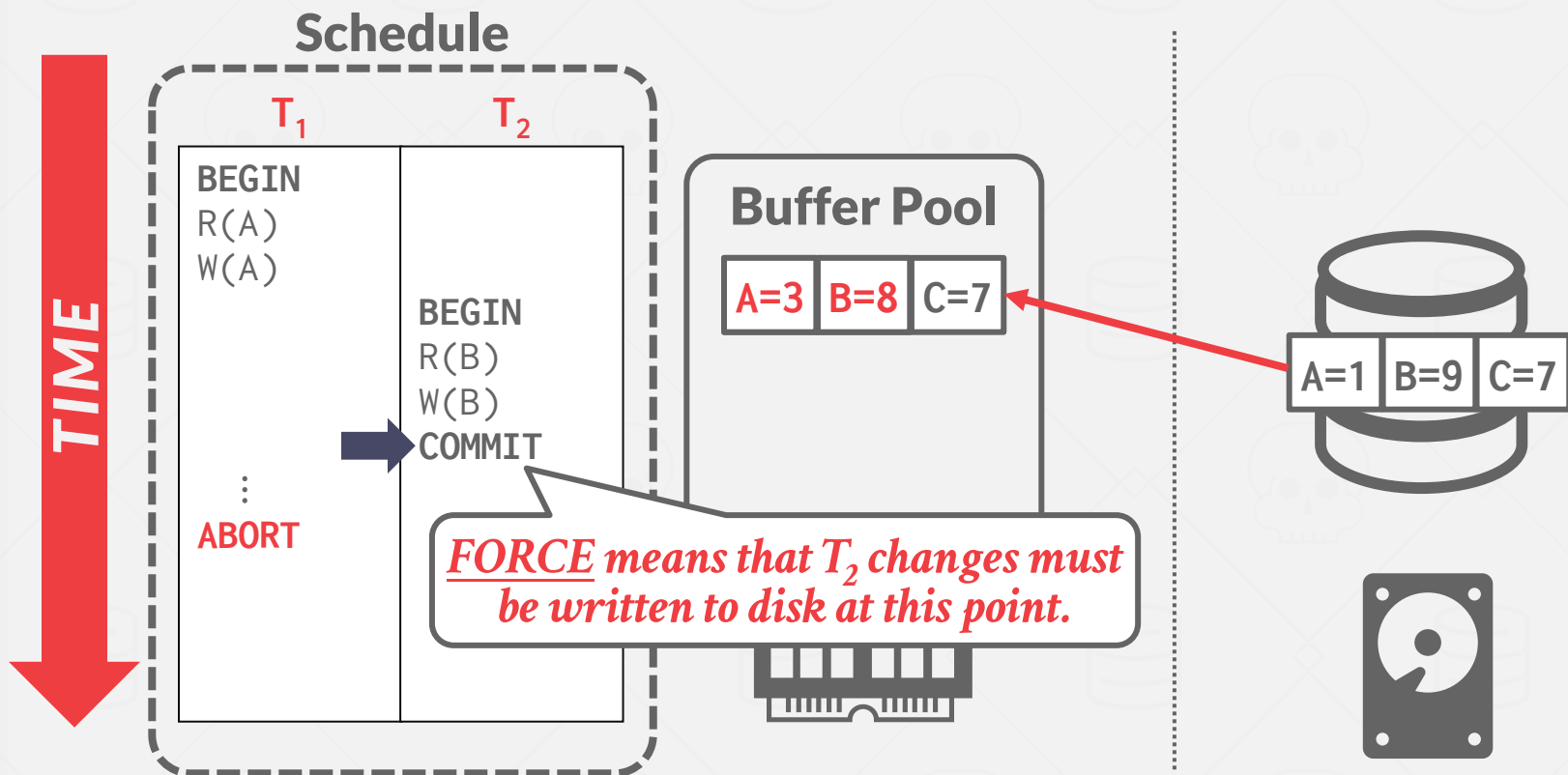
# NO-STEAL + FORCE



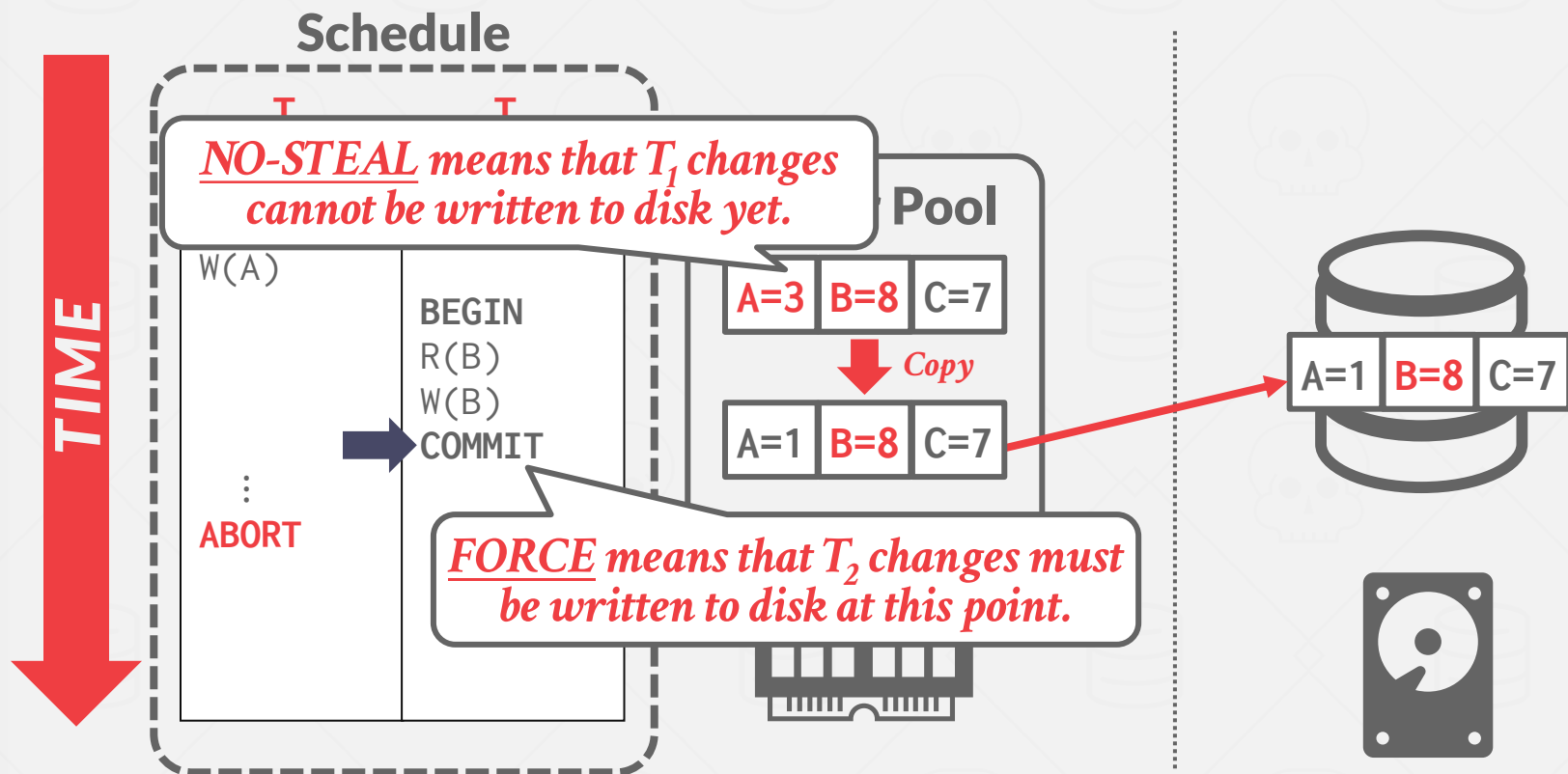
# NO-STEAL + FORCE



# NO-STEAL + FORCE

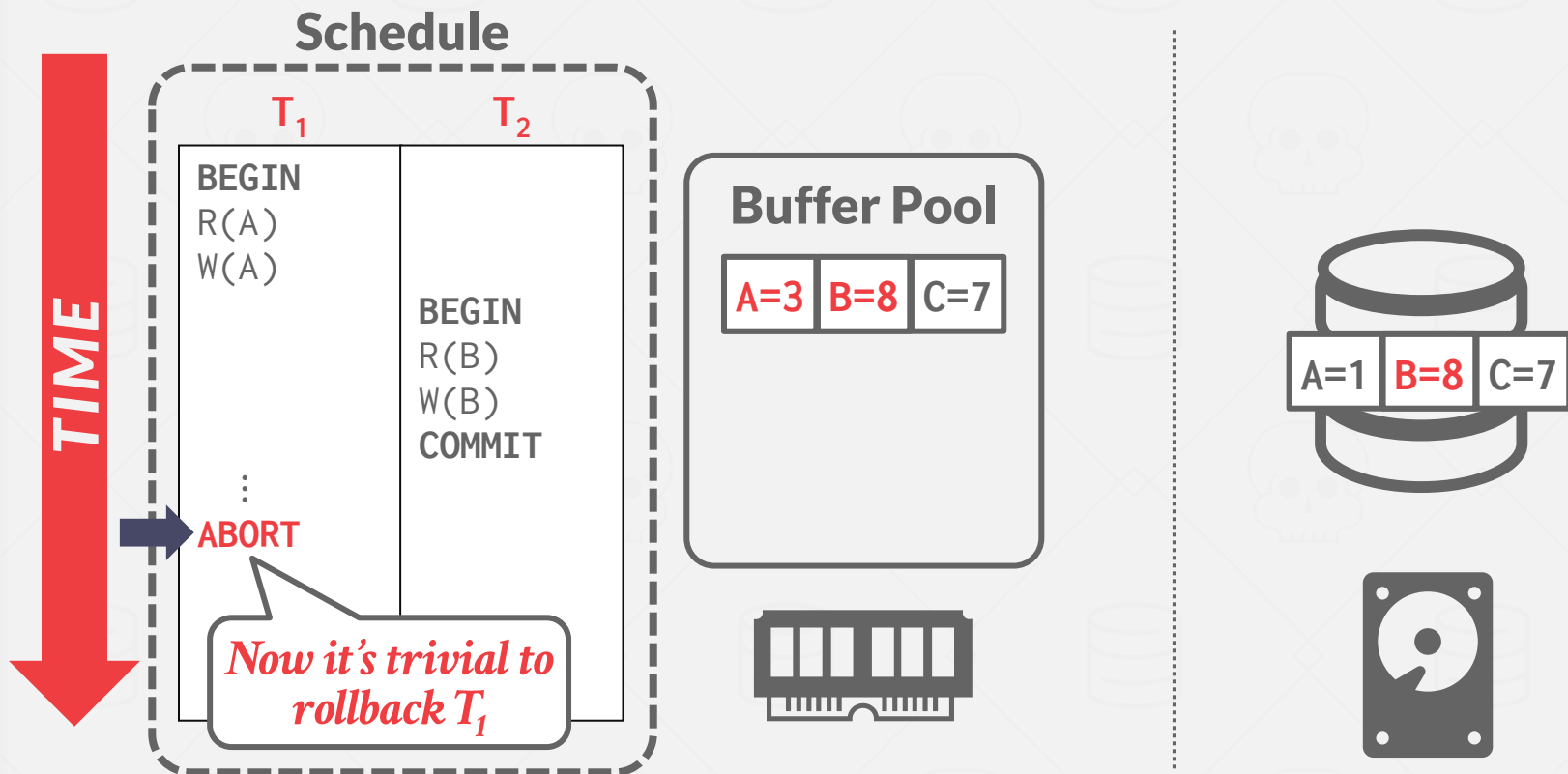


# NO-STEAL + FORCE





# 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

---

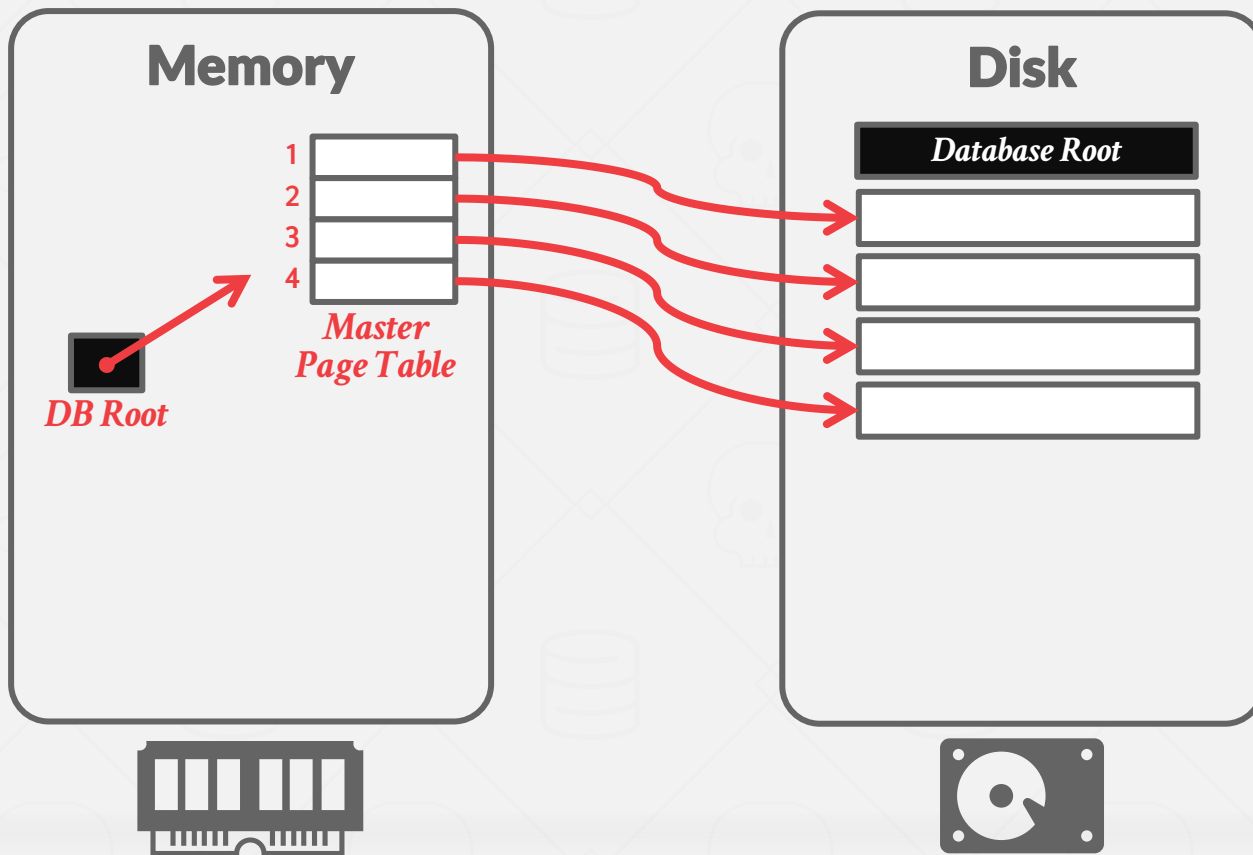
Instead of copying the entire database, the DBMS copies pages on write to create two versions:

- **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.

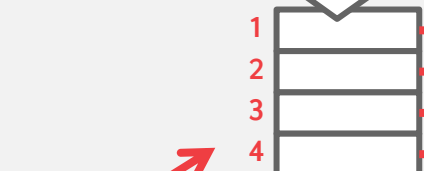
Buffer Pool Policy: **NO-STEAL** + **FORCE**

# SHADOW PAGING - EXAMPLE



# SHADOW PAGING - EXAMPLE

*Read-only txns access the current master.*



*Master  
Page Table*

*DB Root*



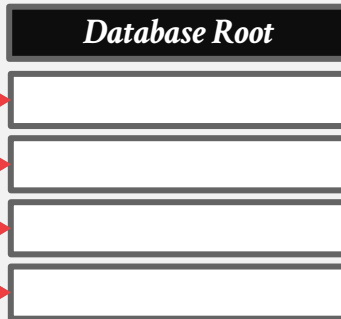
*Shadow  
Page Table*

*Txn  $T_1$*

*Active modifying txn updates shadow pages.*

**Disk**

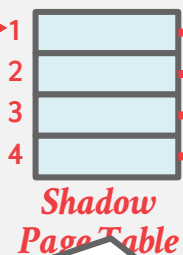
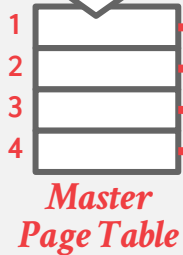
*Database Root*



# SHADOW PAGING - EXAMPLE

*Read-only txns access the current master.*

*DB Root*



*Txn T<sub>1</sub>*

*Update*

*Active modifying txn updates shadow pages.*

**Disk**

*Database Root*



# SHADOW PAGING - EXAMPLE

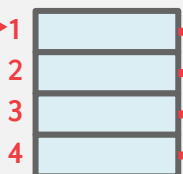
*Read-only txns access the current master.*

DB Root



Master Page Table

*Txn T<sub>1</sub>* Update

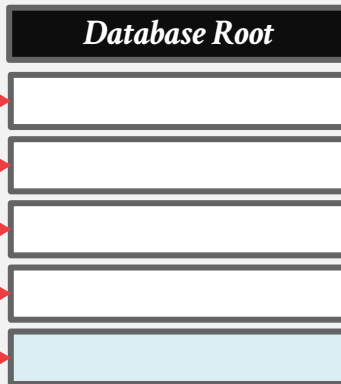


Shadow Page Table

*Active modifying txn updates shadow pages.*

Disk

Database Root



# SHADOW PAGING - EXAMPLE

*Read-only txns access the current master.*



DB Root



Master Page Table



Shadow Page Table

*Txn T<sub>1</sub>*

*Active modifying txn updates shadow pages.*

Disk

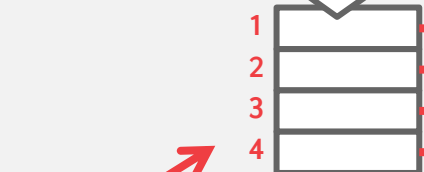
Database Root





# SHADOW PAGING - EXAMPLE

*Read-only txns access the current master.*



*Master Page Table*

*DB Root*

*Txn  $T_1$*

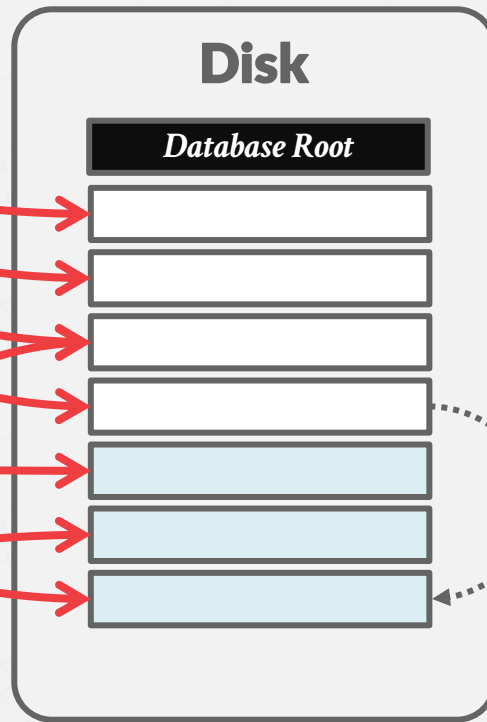


*Shadow Page Table*

*Active modifying txn updates shadow pages.*

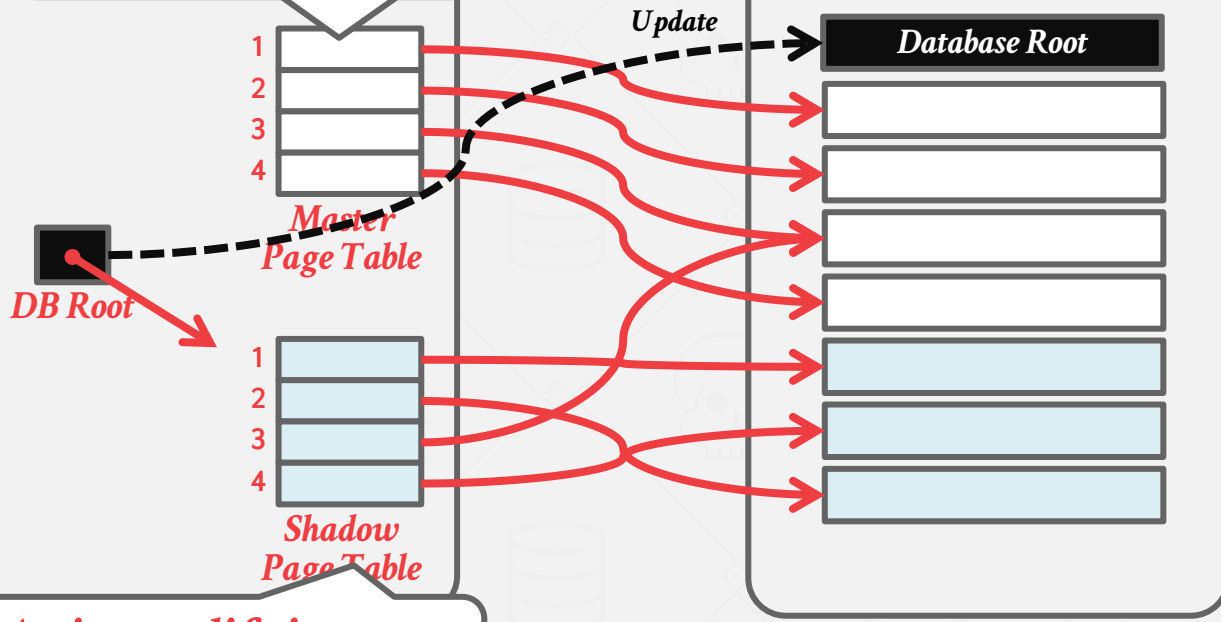
**Disk**

*Database Root*



# SHADOW PAGING - EXAMPLE

*Read-only txns access the current master.*



*Txn  $T_1$*

**COMMIT**

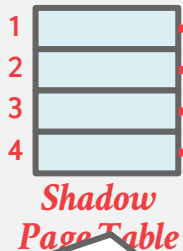
*Active modifying txn updates shadow pages.*

# SHADOW PAGING - EXAMPLE

*Read-only txns access the current master.*



*Master Page Table*

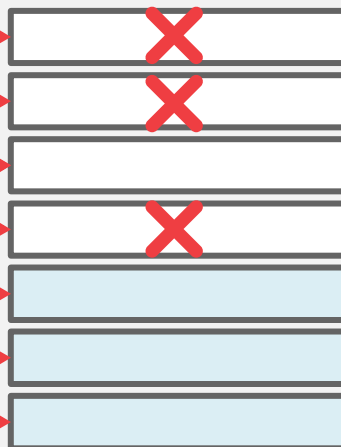


*Active modifying txn updates shadow pages.*

*Update*

**Disk**

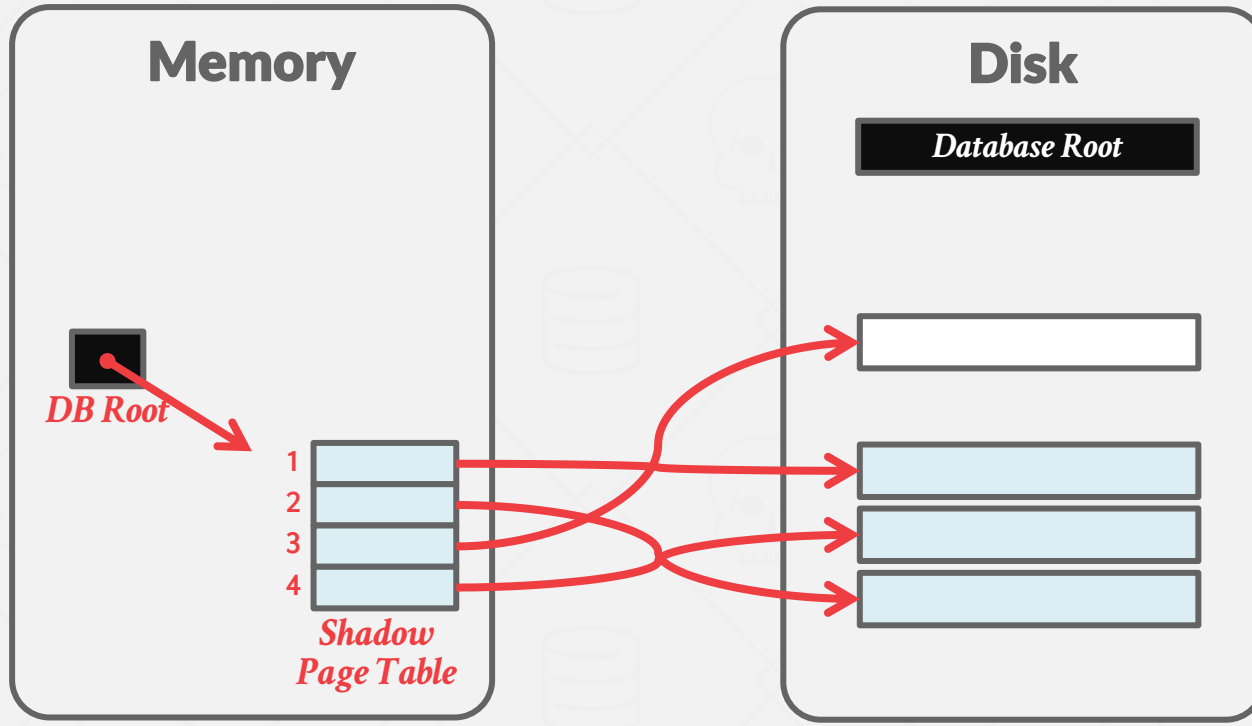
*Database Root*



*Txn  $T_1$*   
**COMMIT**



# SHADOW PAGING - EXAMPLE



*Txn T<sub>1</sub>*

**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.

## Memory

Page 1

Page 2

Page 3

## Disk

Page 1

Page 4

Page 2

Page 5

Page 3

Page 6

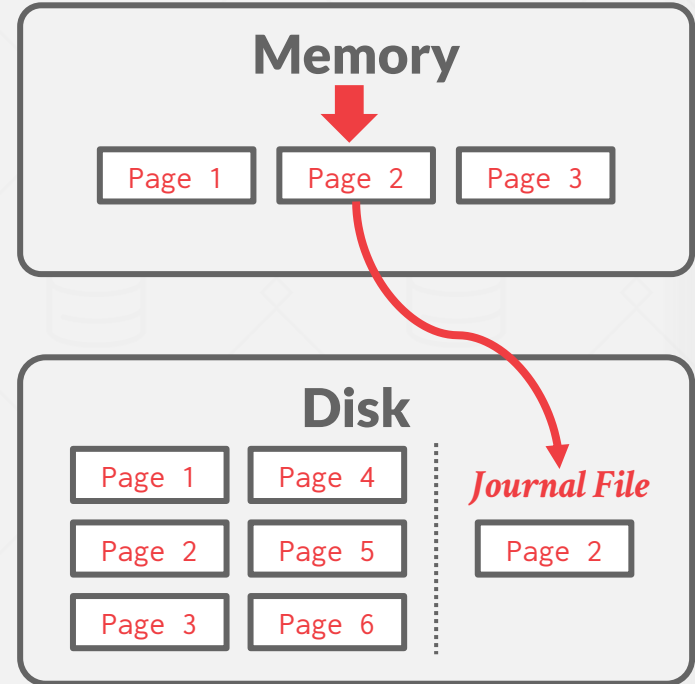
*Journal File*

# 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.



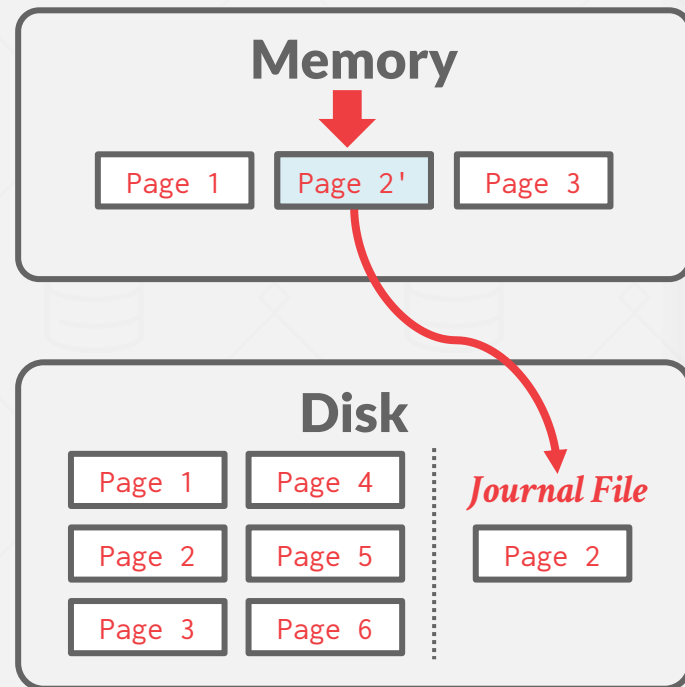


# SQLITE (PRE-2010)

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.

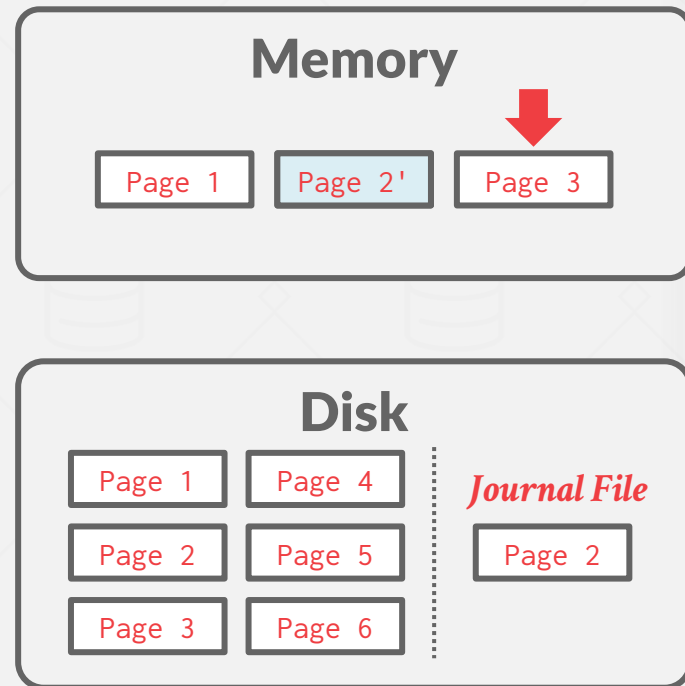


# SQLITE (PRE-2010)

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.

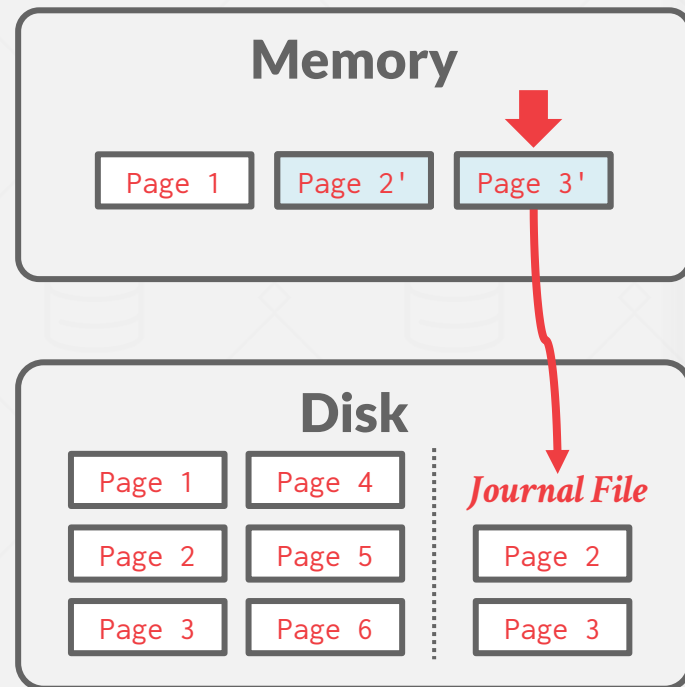


# SQLITE (PRE-2010)

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.

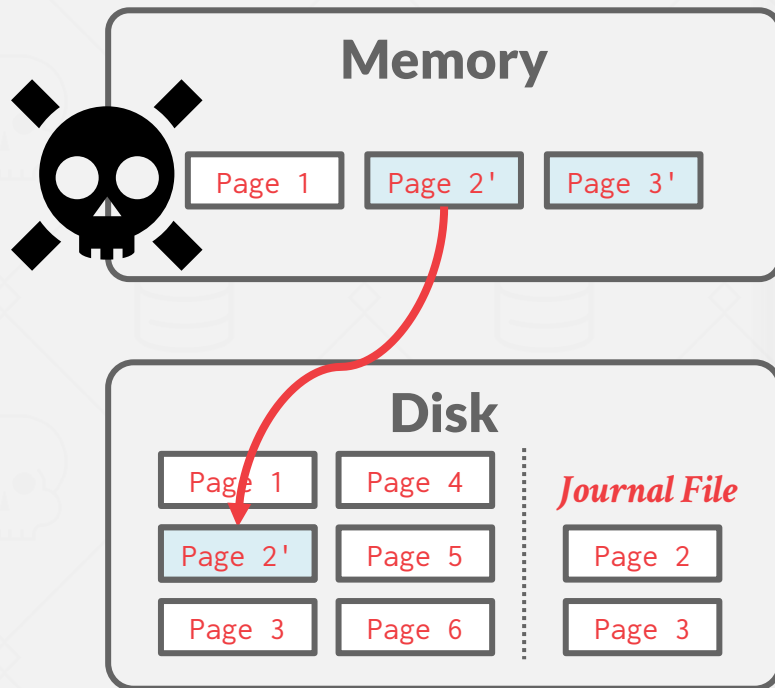


# SQLITE (PRE-2010)

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.



Memory

Disk

Page 1

Page 4

Page 2'

Page 5

Page 3

Page 6

*Journal File*

Page 2

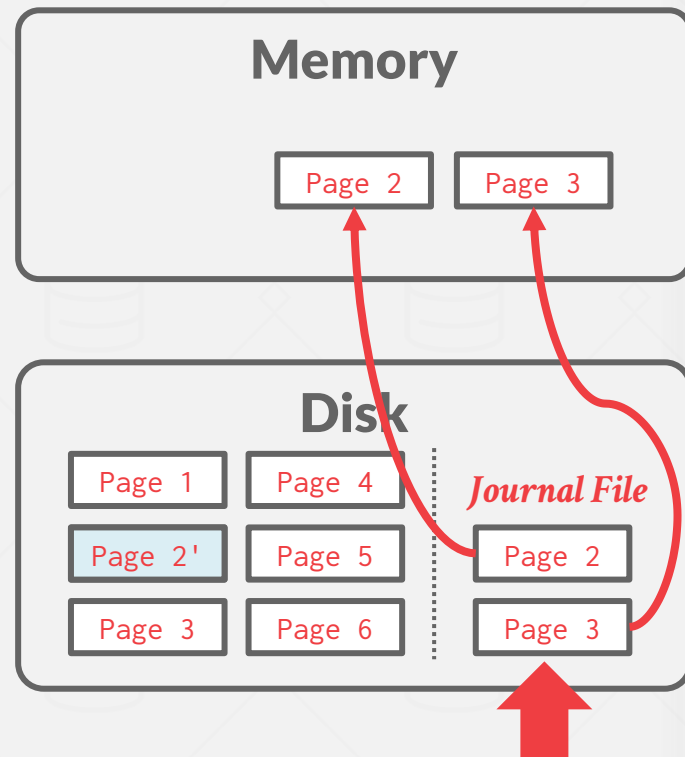
Page 3

# 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.

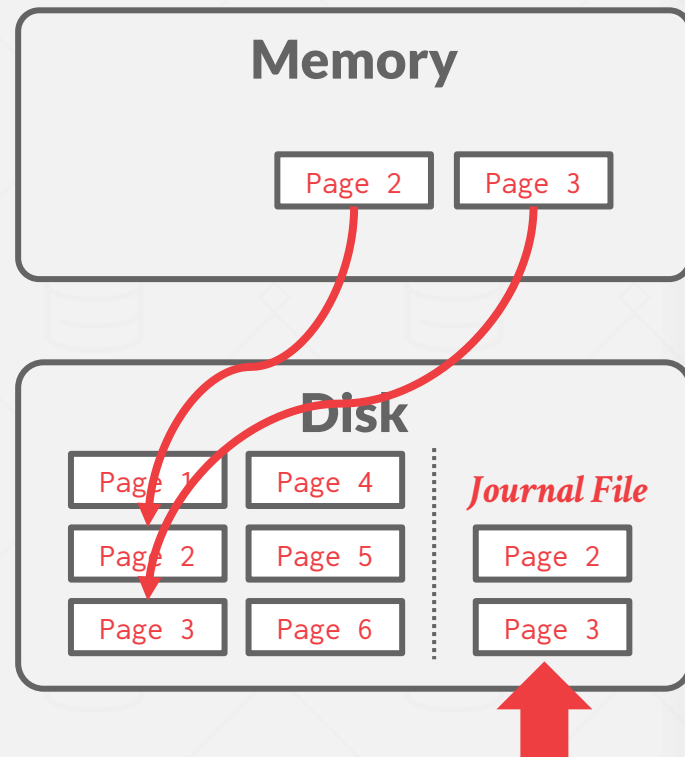


# SQLITE (PRE-2010)

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**

# 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

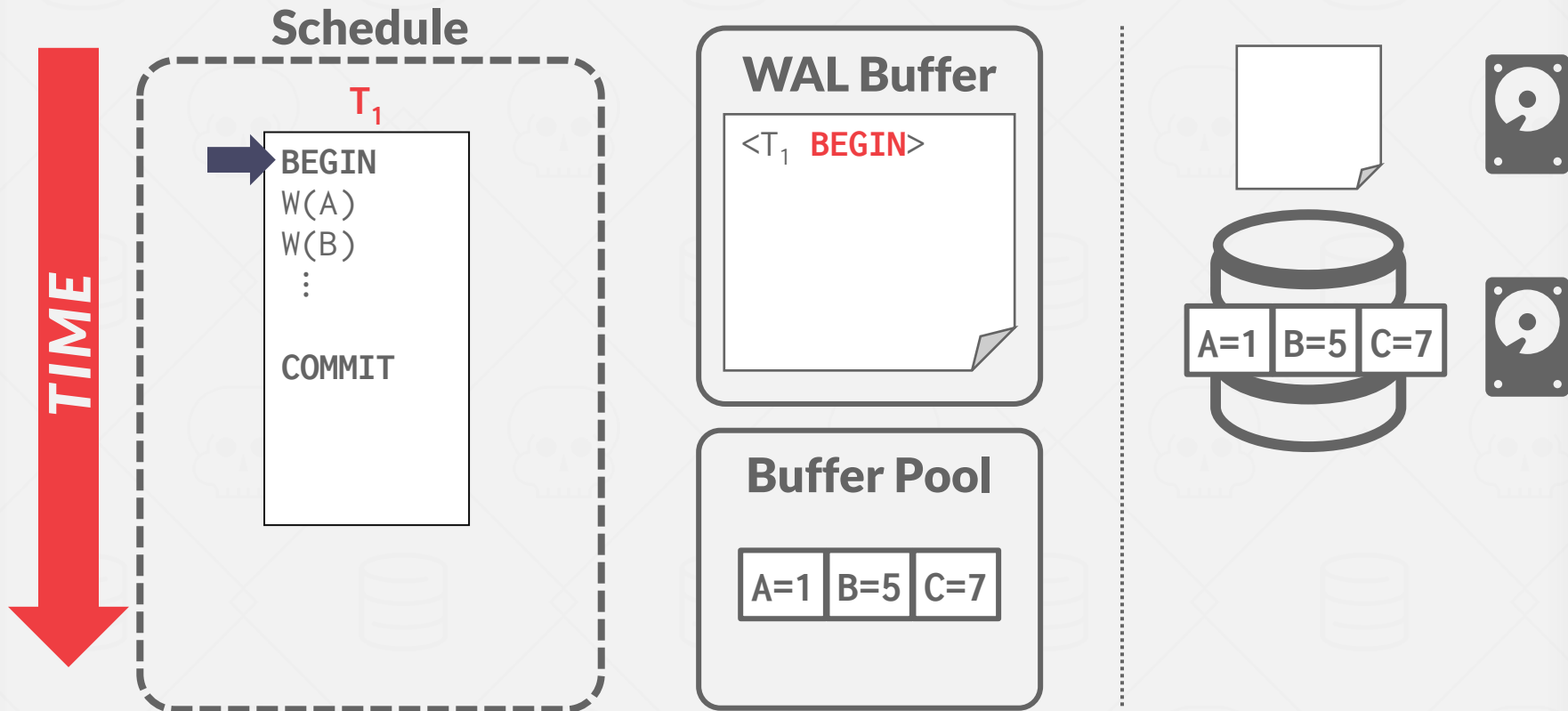
---

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

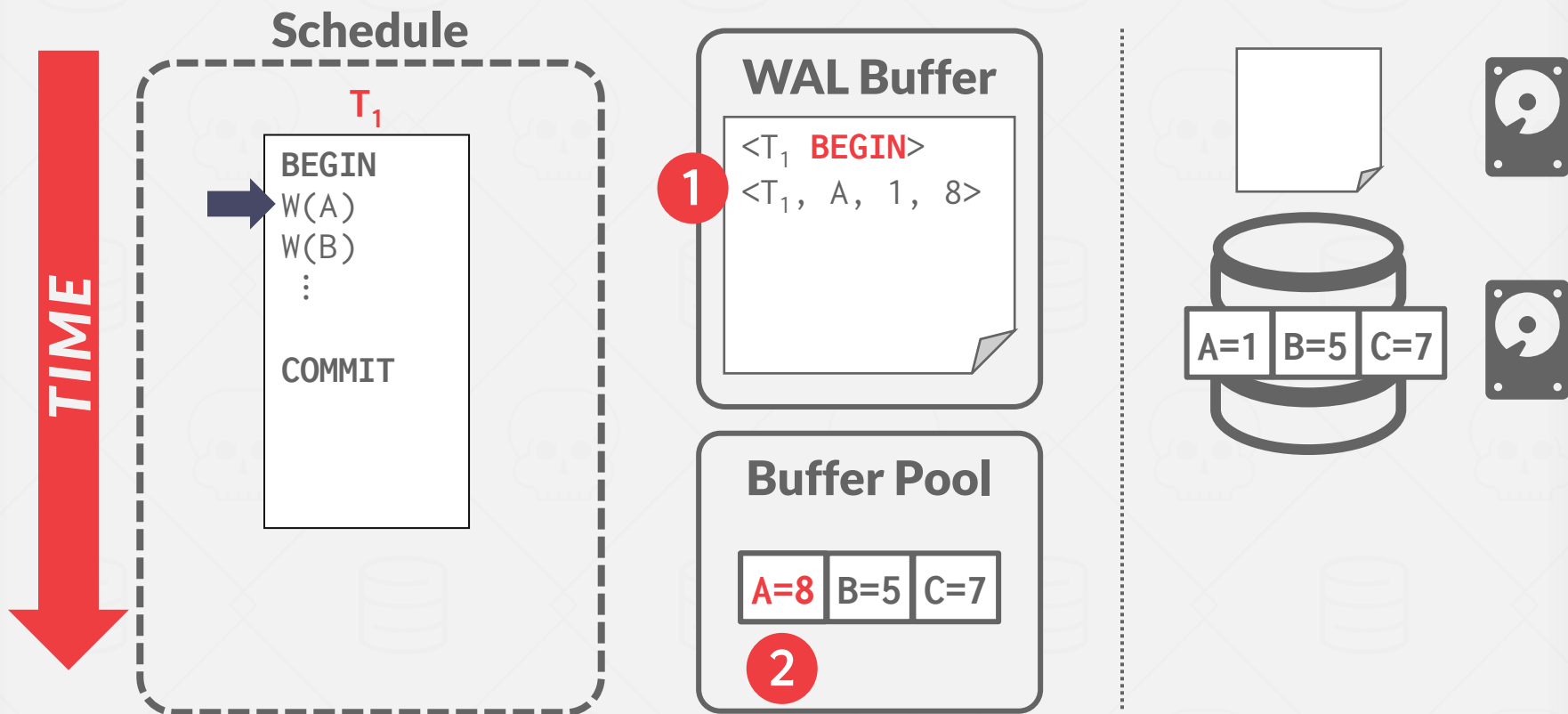
- Transaction Id
- Object Id
- Before Value (**UNDO**)
- After Value (**REDO**)

*Not necessary if using  
append-only MVCC* ←

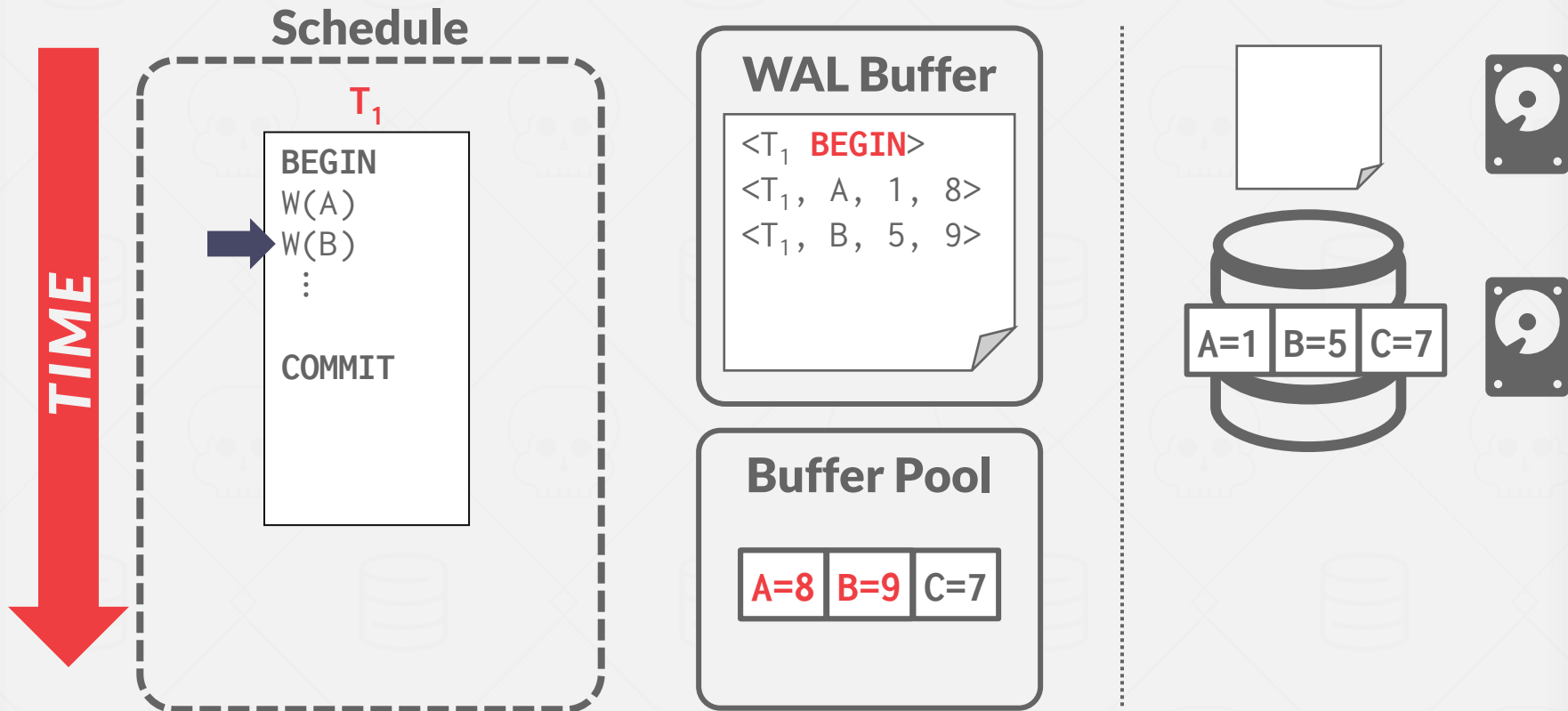
# WAL - EXAMPLE



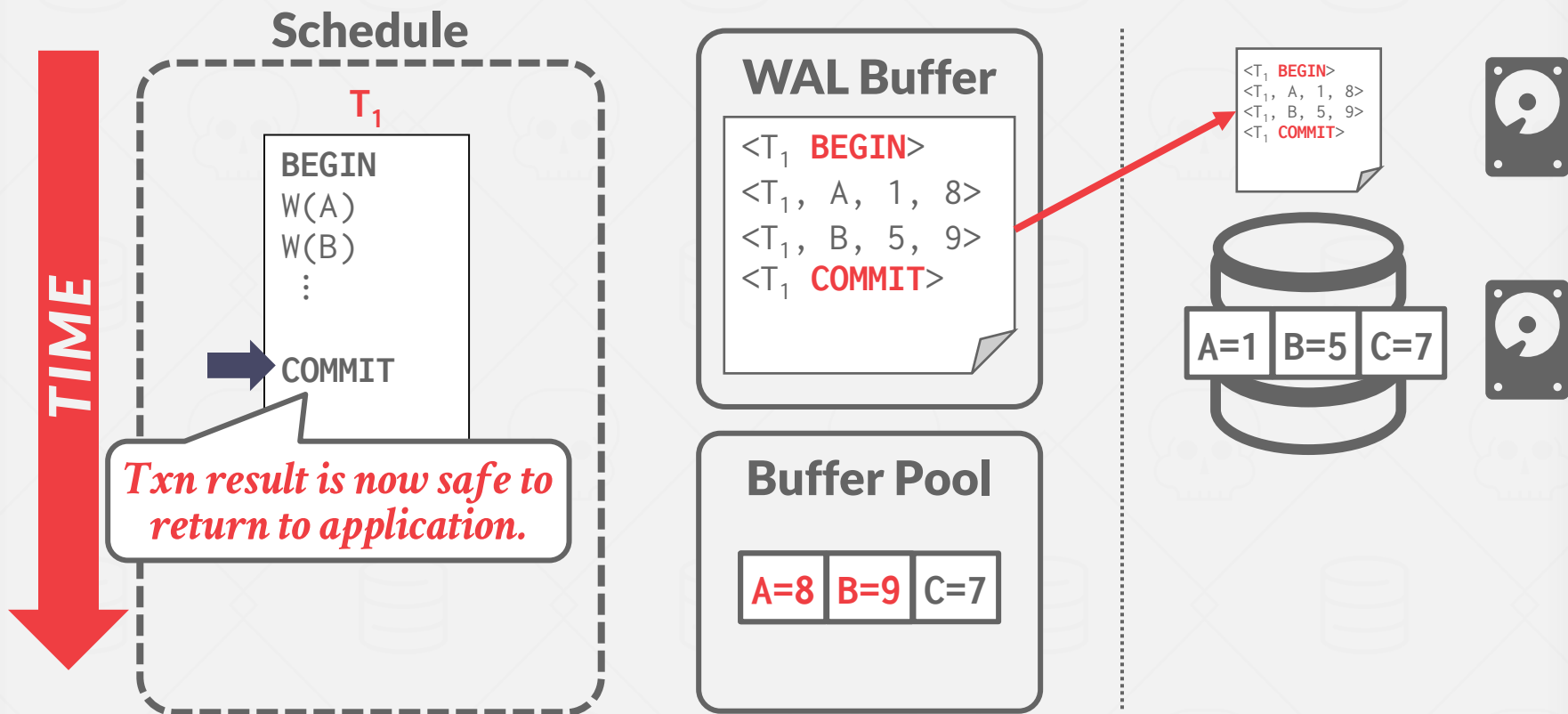
# WAL - EXAMPLE



# WAL - EXAMPLE

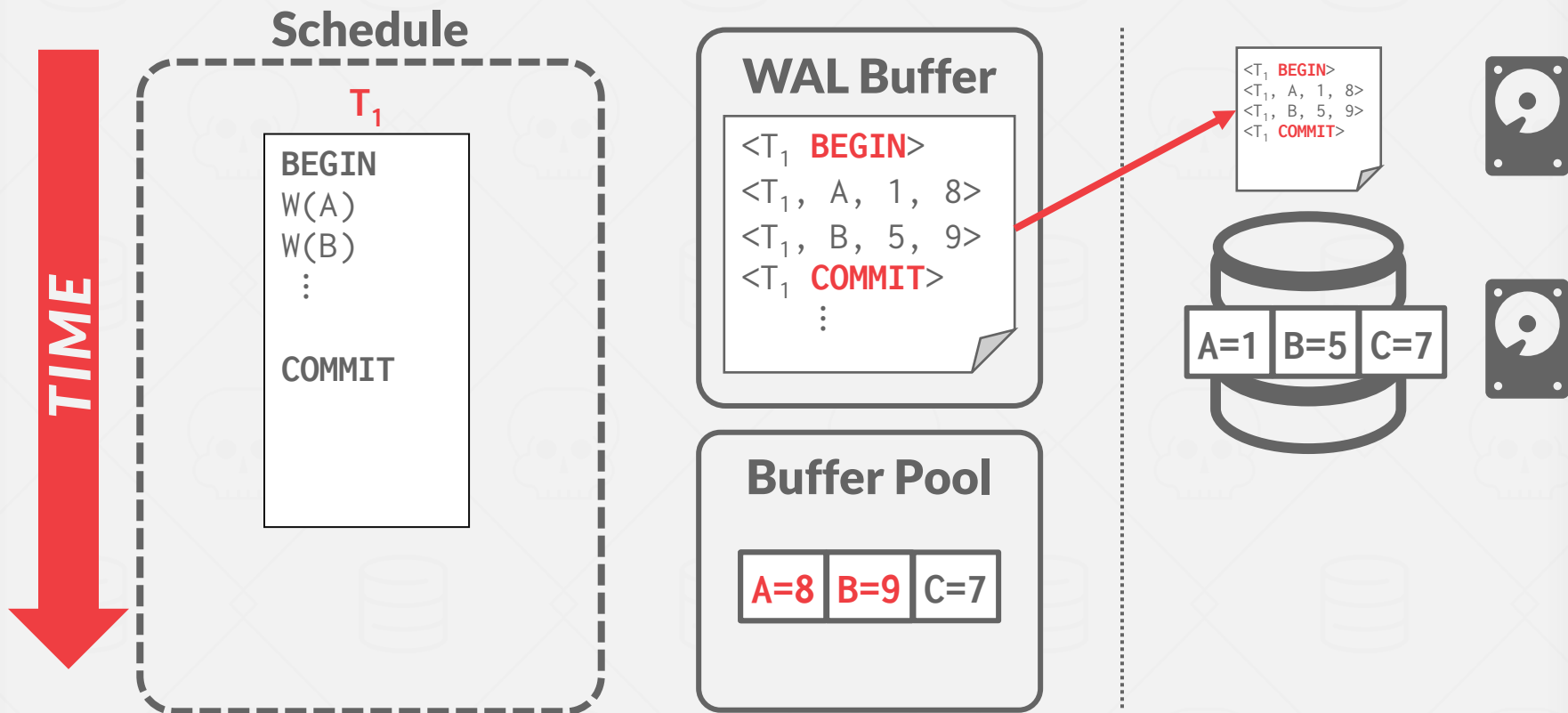


# WAL - EXAMPLE



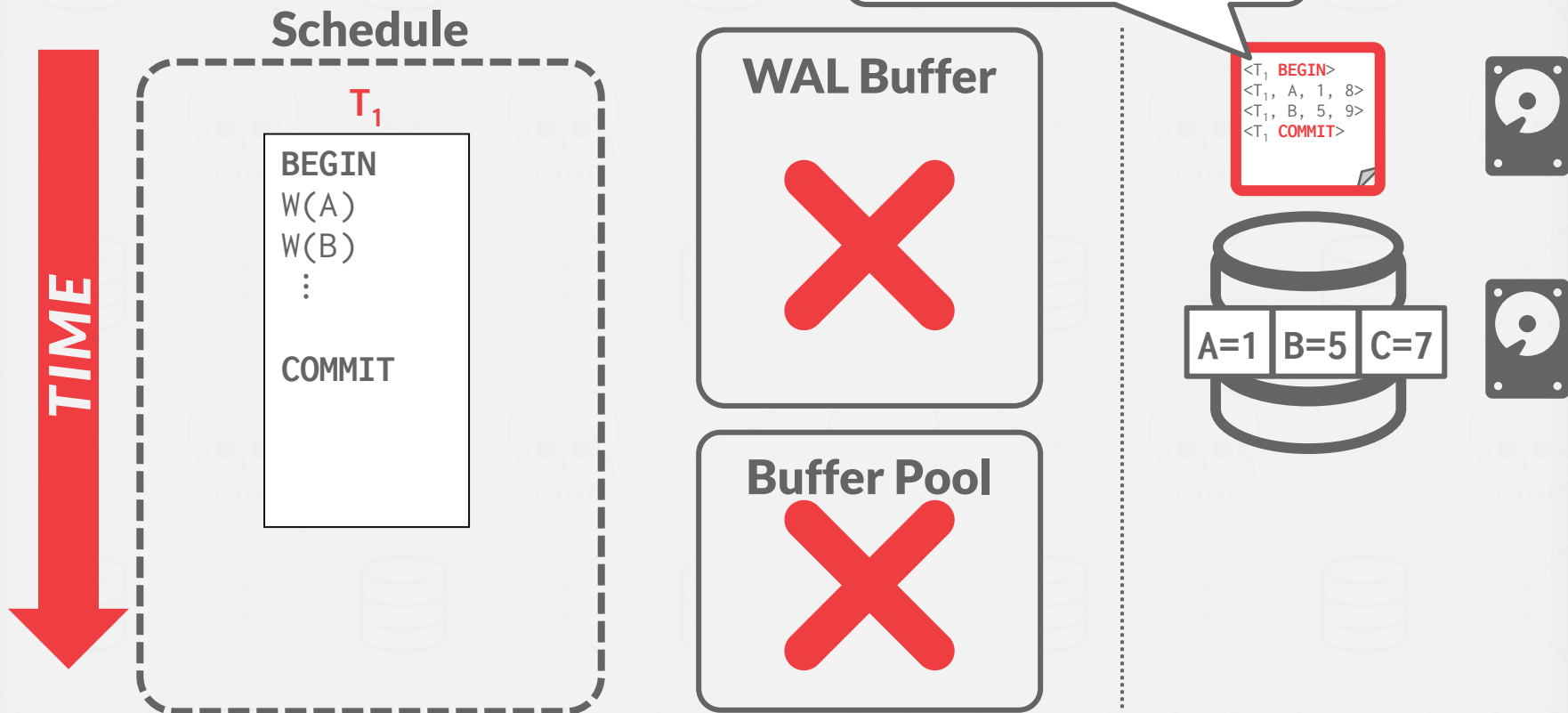


# WAL - EXAMPLE



# WAL - EXA

*Everything we need to restore  $T_1$  is in the log!*



# WAL - IMPLEMENTATION

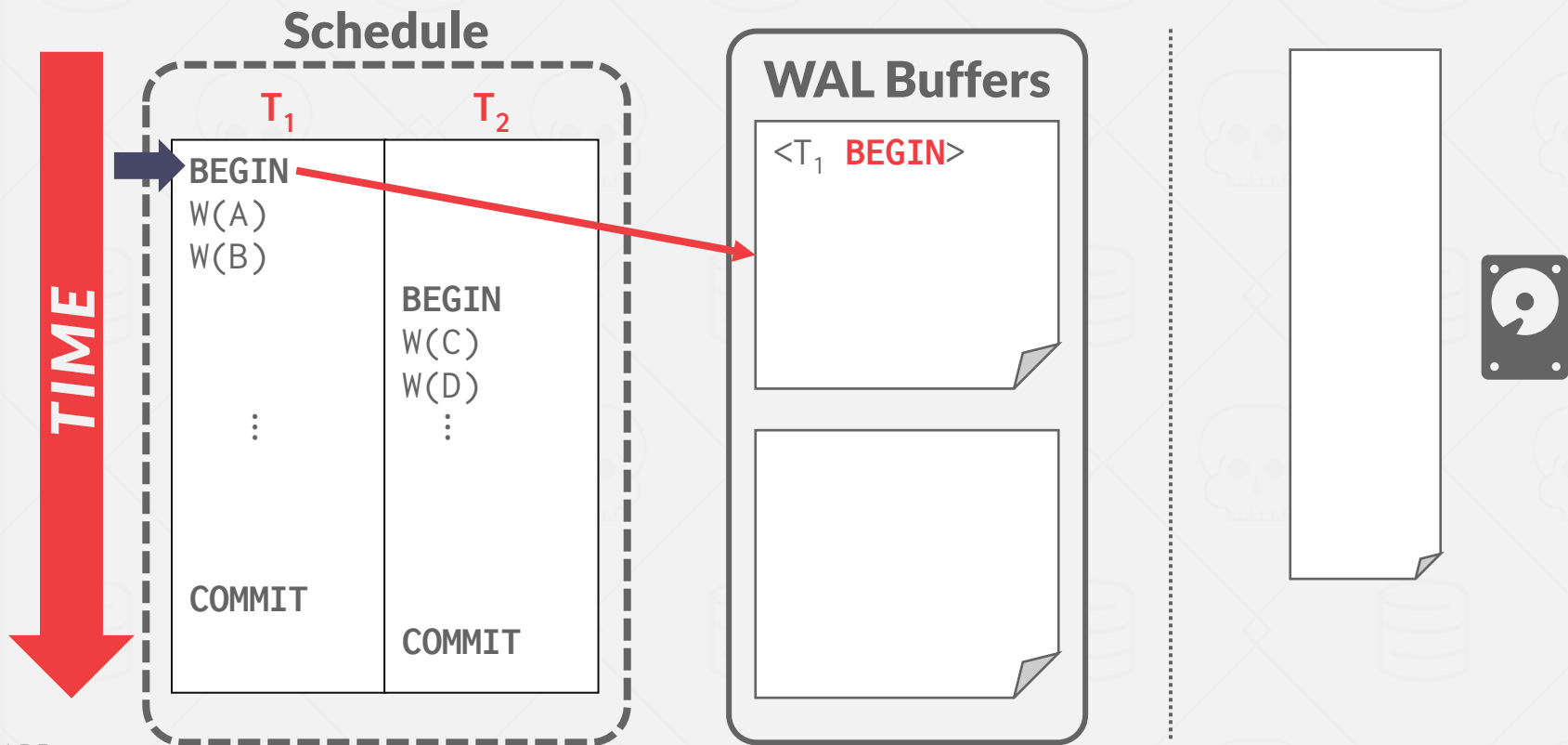
---

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

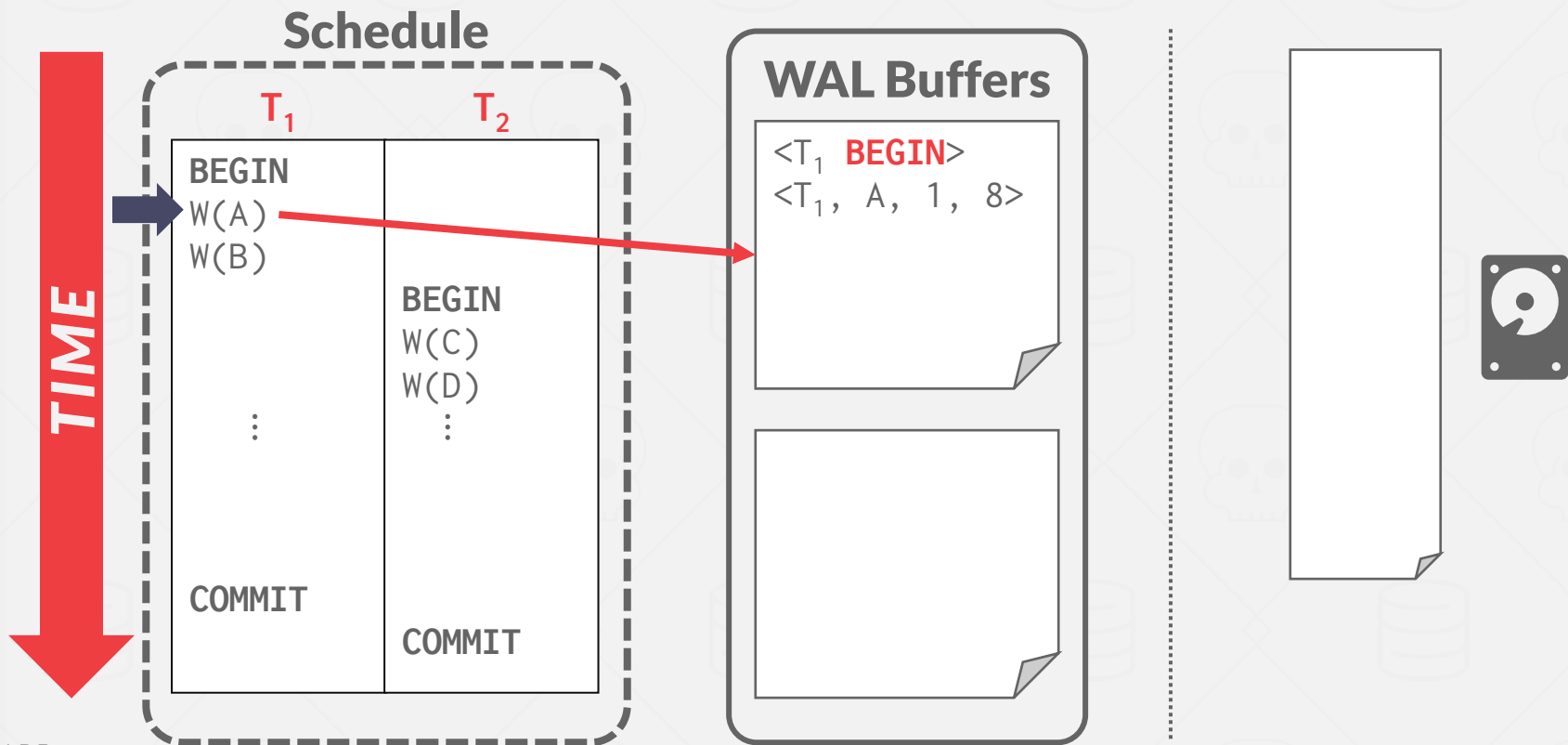
The DBMS can use the group commit optimization to batch multiple log flushes together to amortize overhead.

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

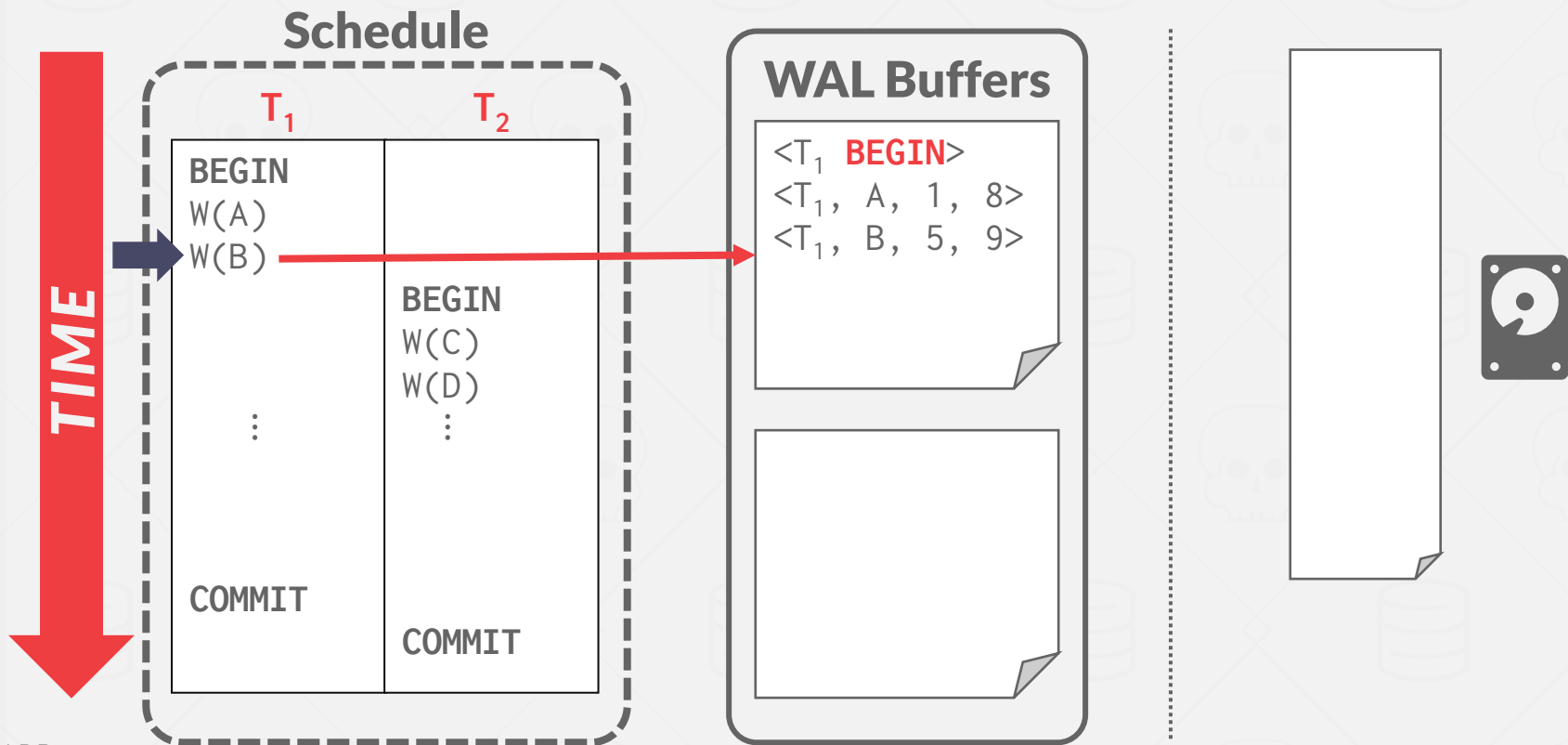
# WAL - GROUP COMMIT



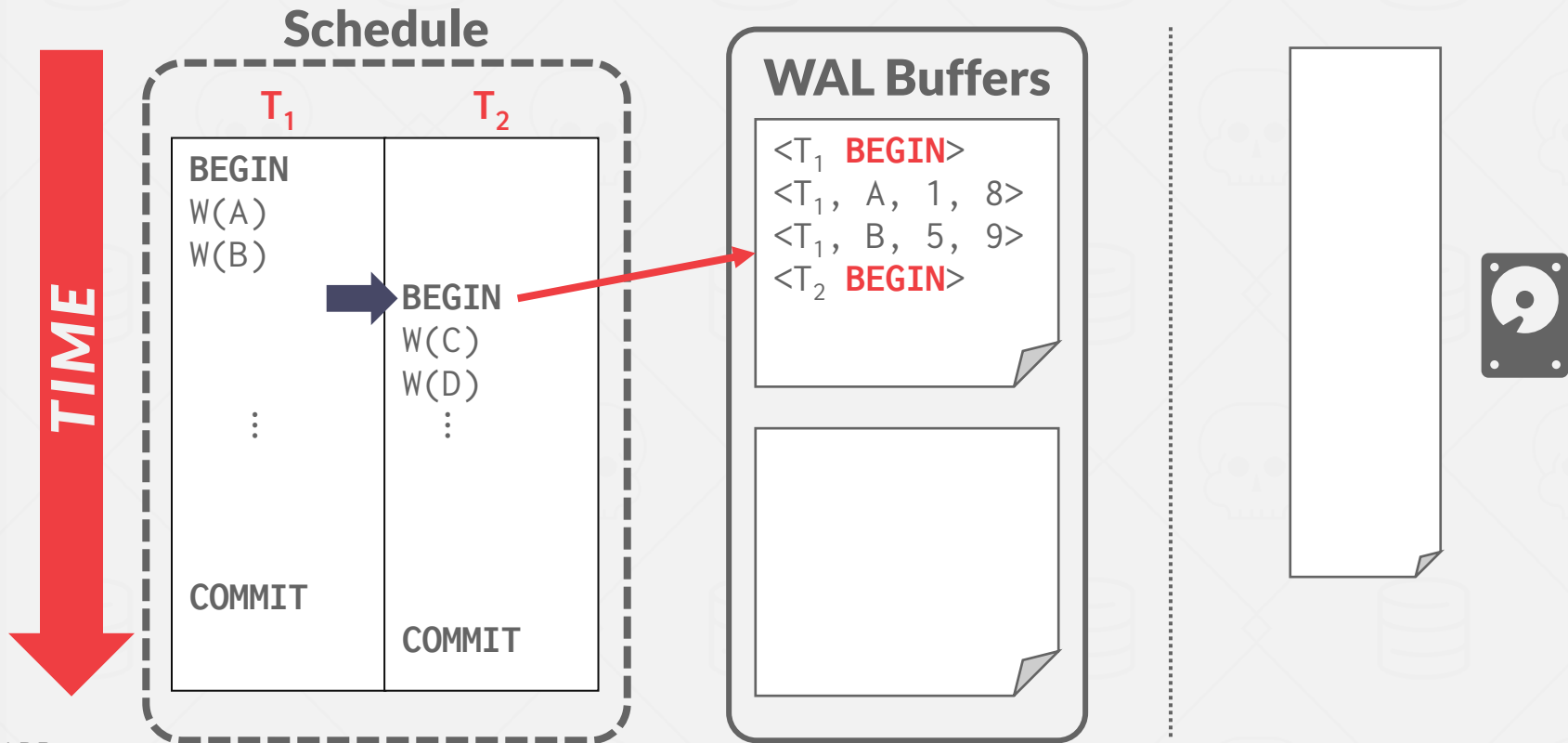
# WAL - GROUP COMMIT



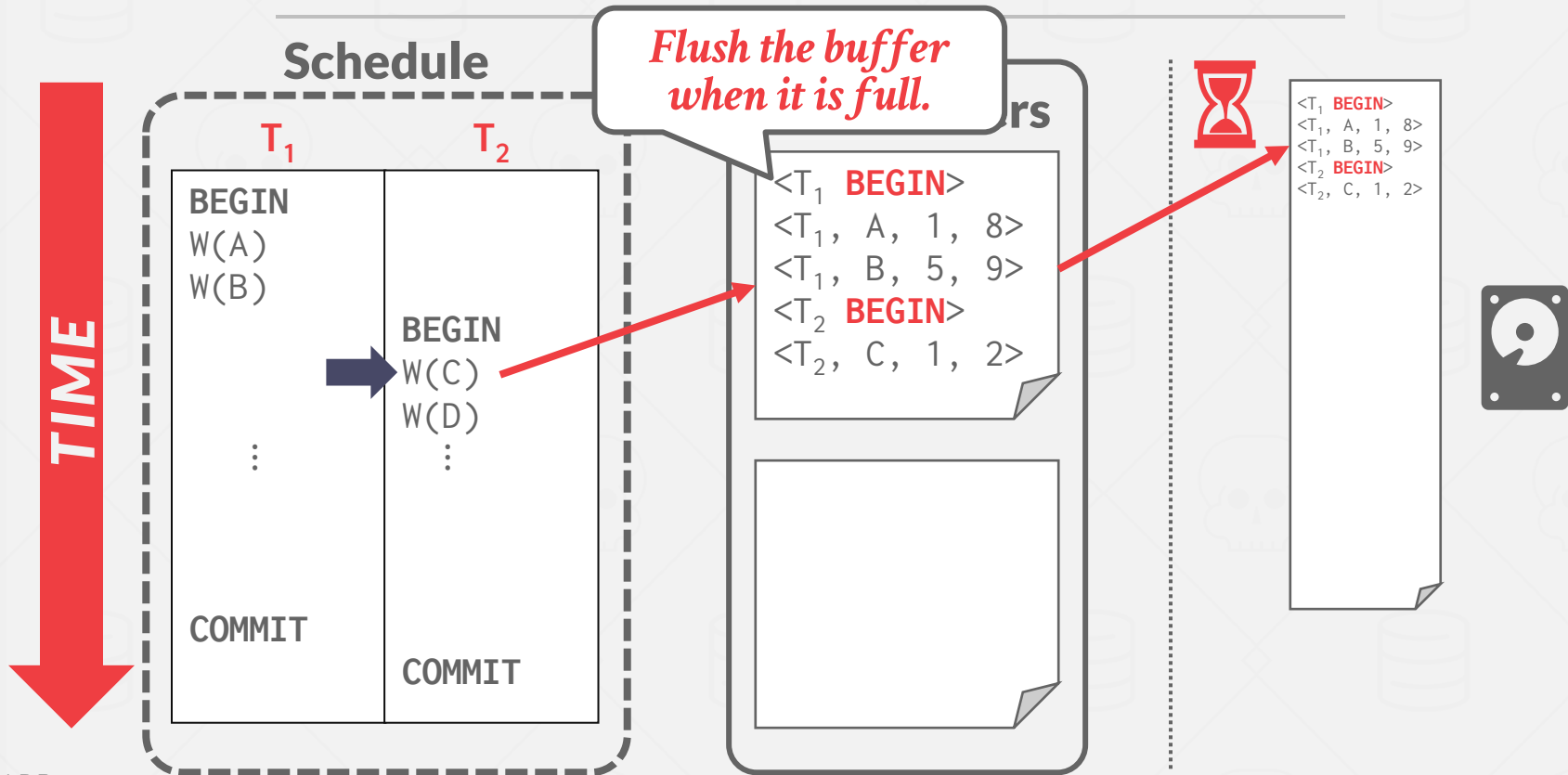
# WAL - GROUP COMMIT



# WAL - GROUP COMMIT

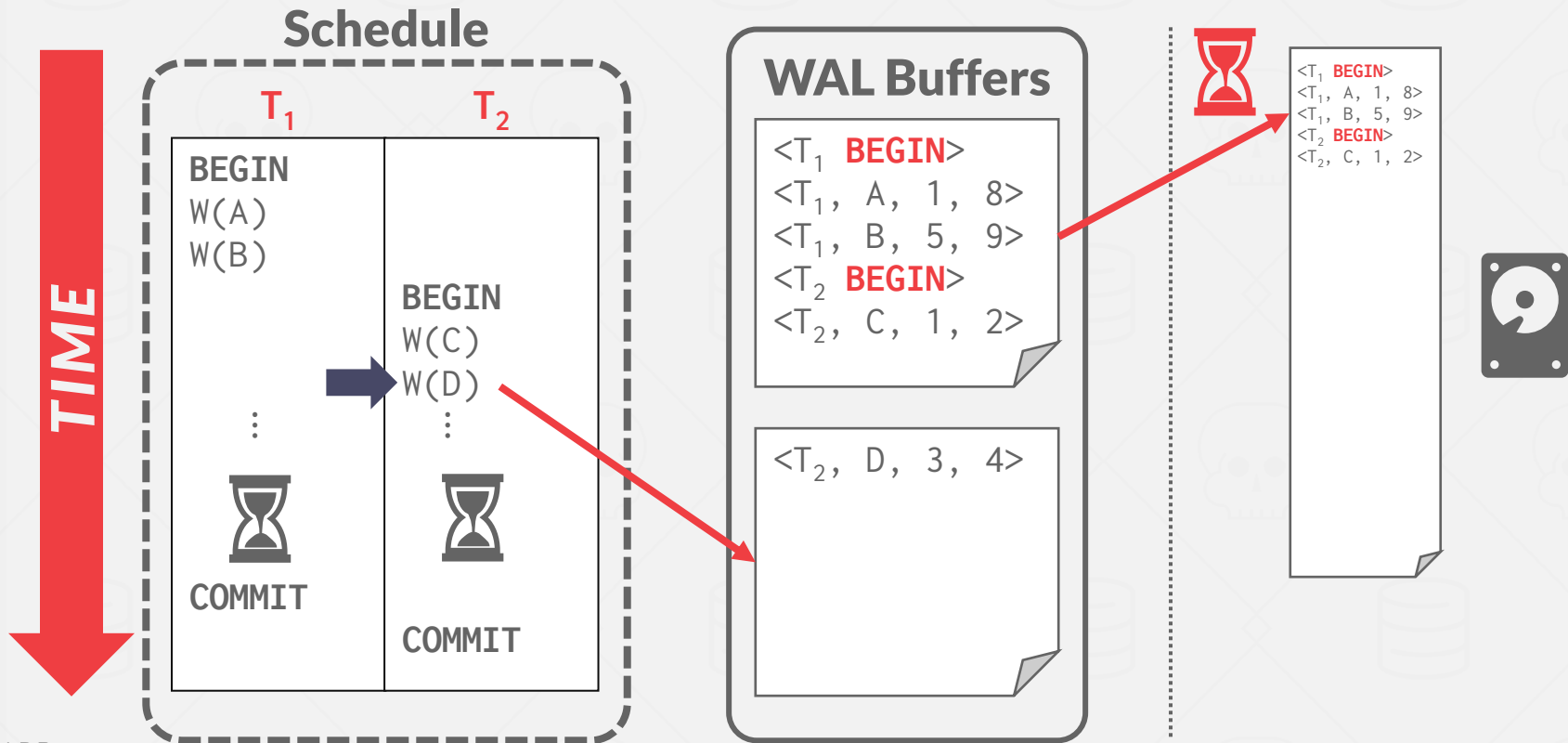


# WAL - GROUP COMMIT

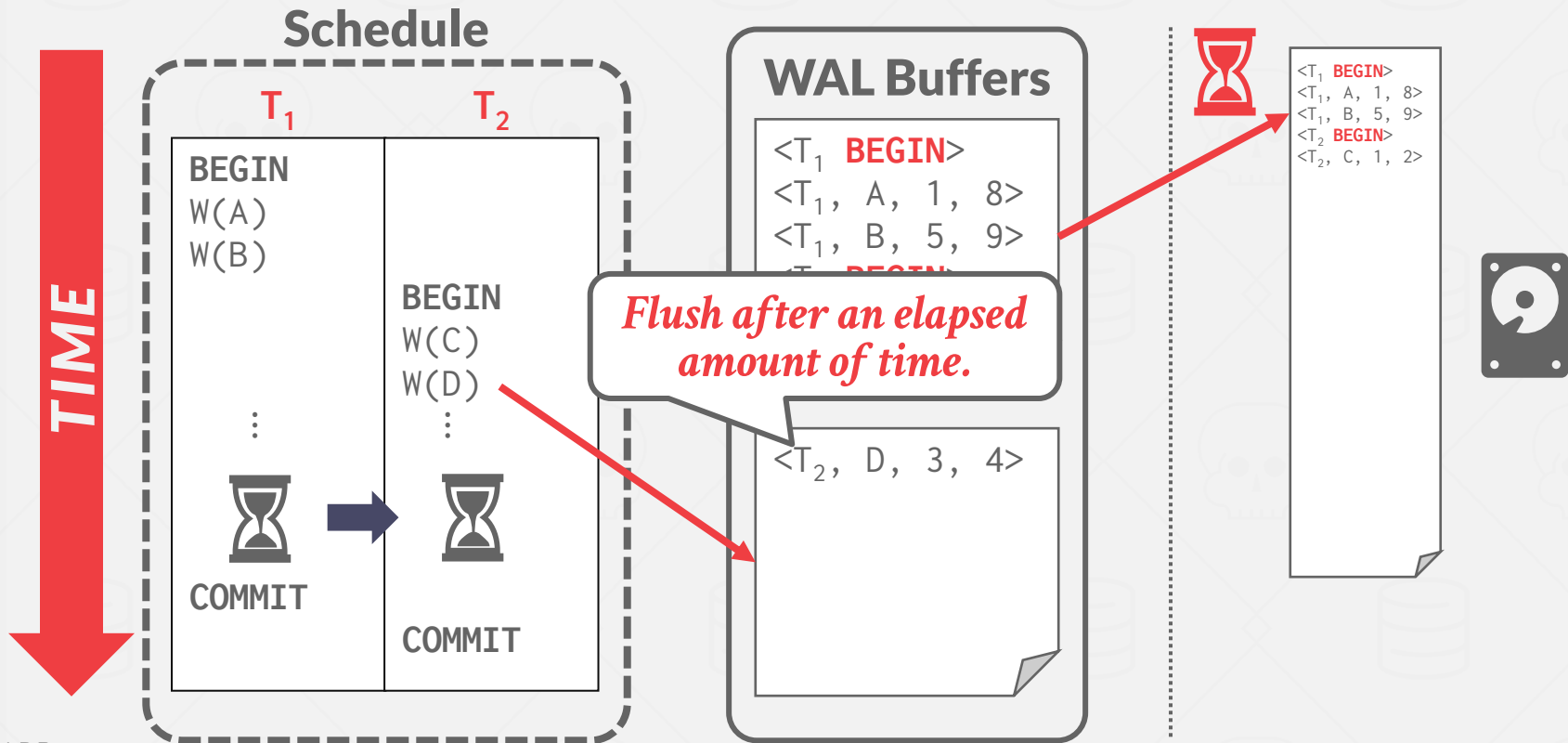




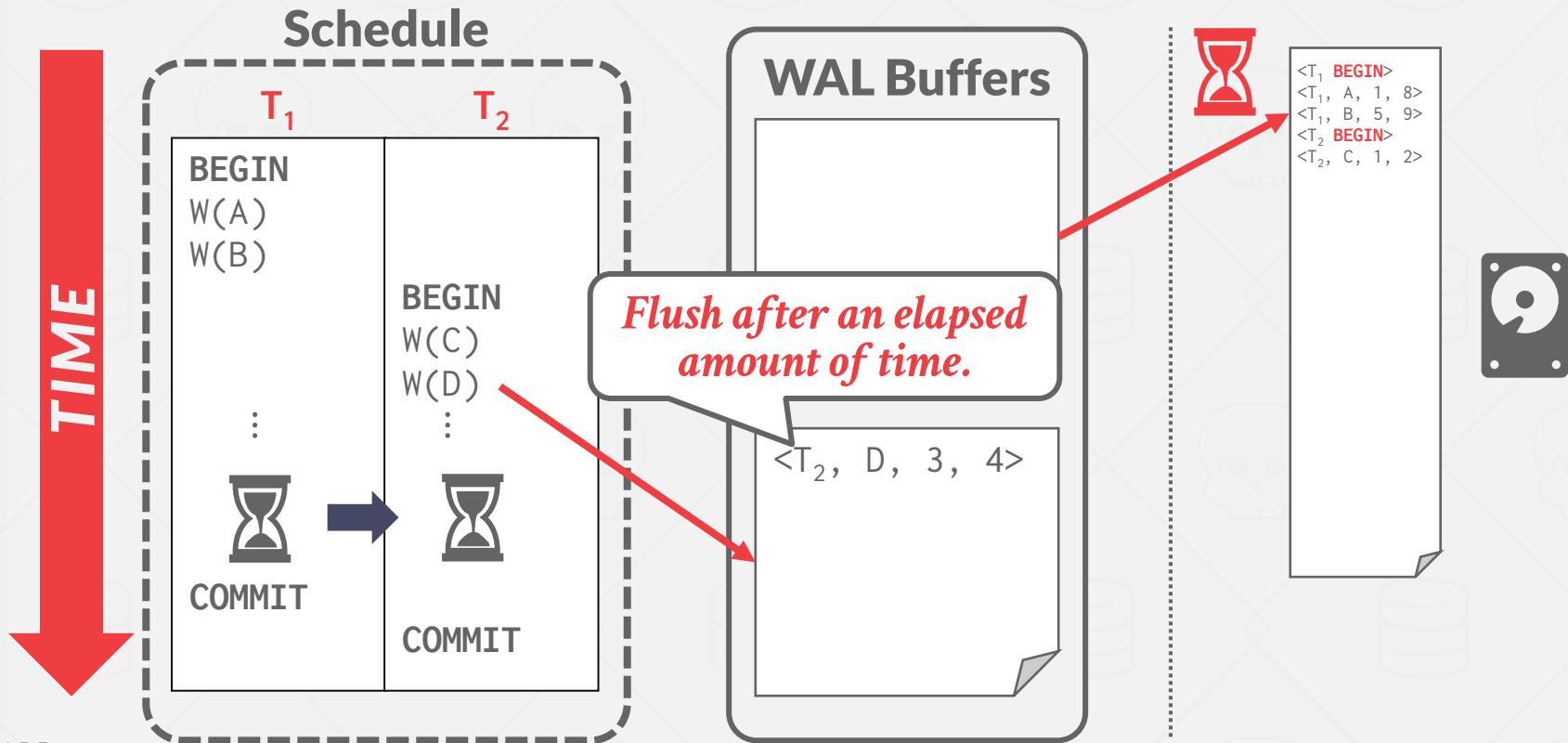
# WAL - GROUP COMMIT



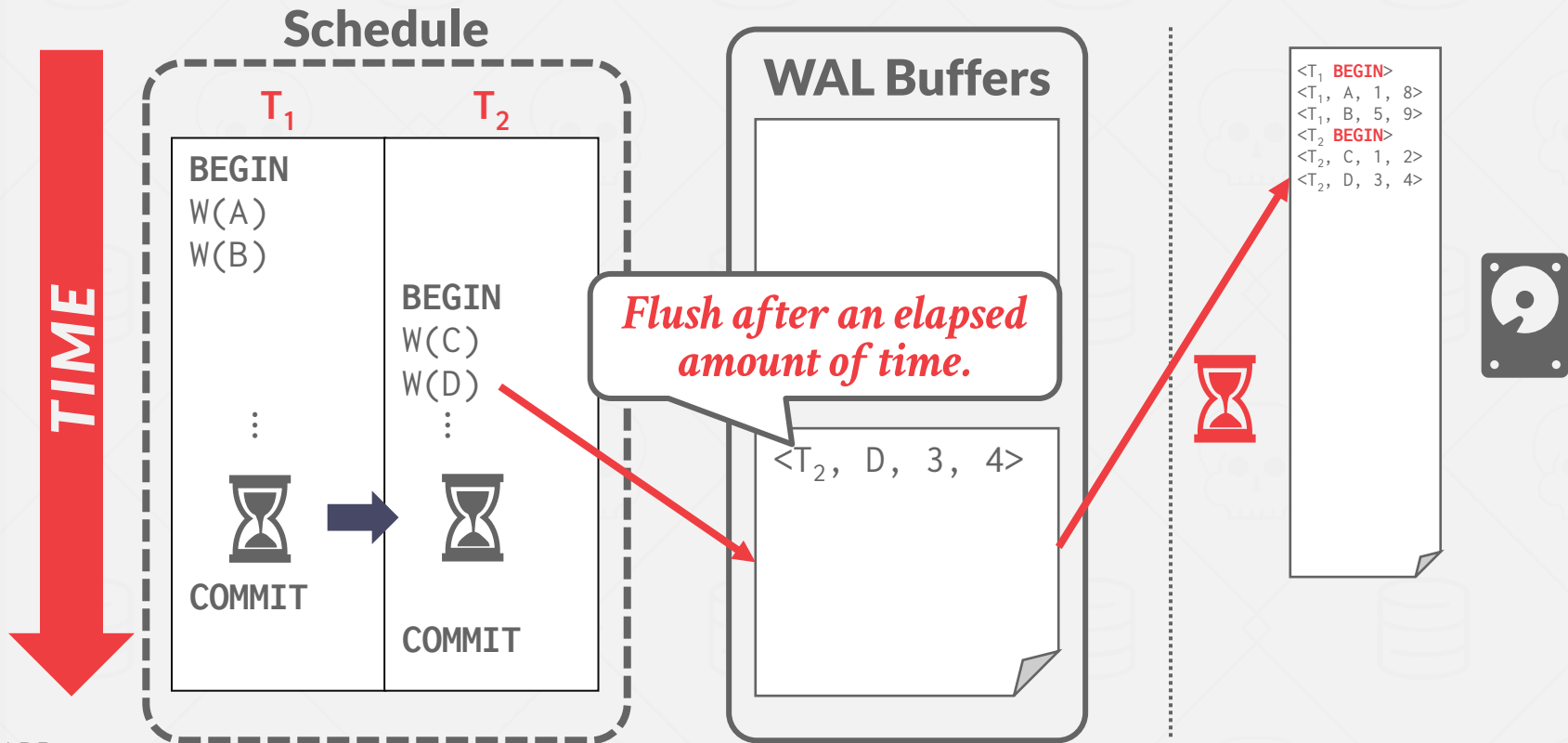
# WAL - GROUP COMMIT



# WAL - GROUP COMMIT



# WAL - GROUP COMMIT



# BUFFER POOL POLICIES

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

## *Runtime Performance*

	<i>NO-STEAL</i>	<i>STEAL</i>
<i>NO-FORCE</i>	–	Fastest
<i>FORCE</i>	Slowest	–

## *Recovery Performance*

	<i>NO-STEAL</i>	<i>STEAL</i>
<i>NO-FORCE</i>	–	Slowest
<i>FORCE</i>	Fastest	–

# 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

- 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

```
<T1,  
Table=X,  
Page=99,  
Offset=1024,  
Before=ABC,  
After=XYZ>  
  
<T1,  
Index=X_PKEY,  
Page=45,  
Offset=9,  
Key=(1,Record1)>
```

## Logical

```
<T1,  
Query="UPDATE foo  
SET val=XYZ  
WHERE id=1">
```

## Physiological

```
<T1,  
Table=X,  
Page=99,  
Slot=1,  
Before=ABC,  
After=XYZ>  
  
<T1,  
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

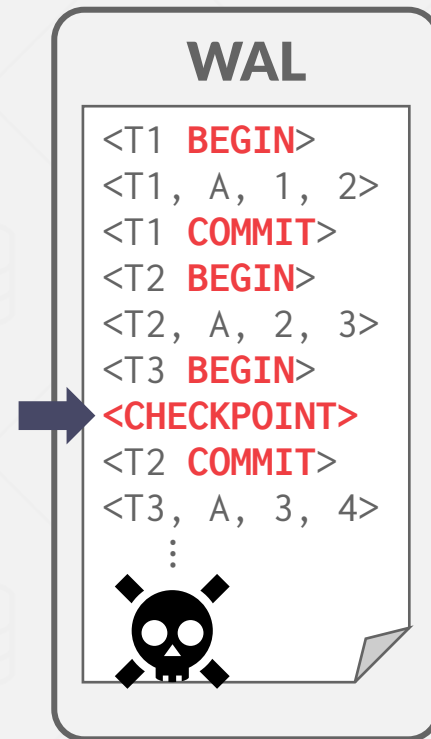
---

## 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

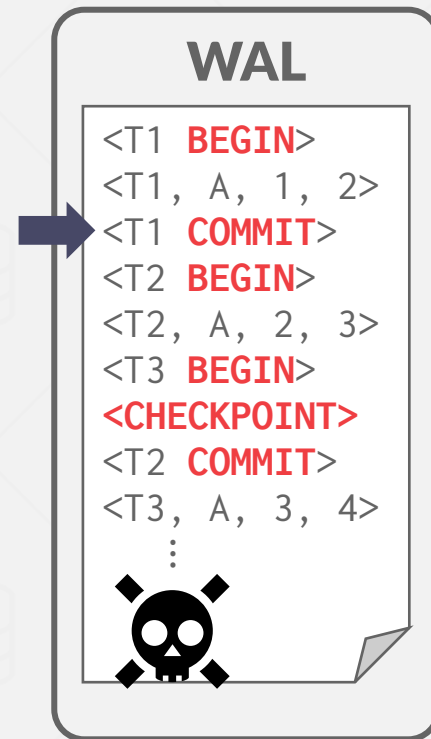
Use the **<CHECKPOINT>** record as the starting point for analyzing the WAL.



# CHECKPOINTS

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

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



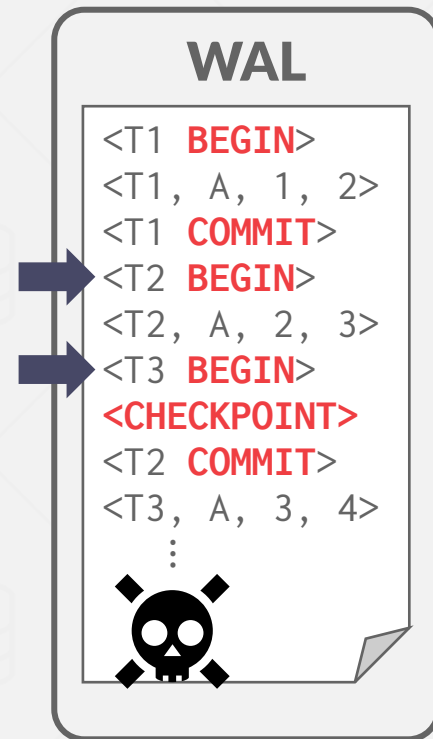
# CHECKPOINTS

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

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

T

$T_2 + T_3$  did not commit before the last checkpoint.



# CHECKPOINTS

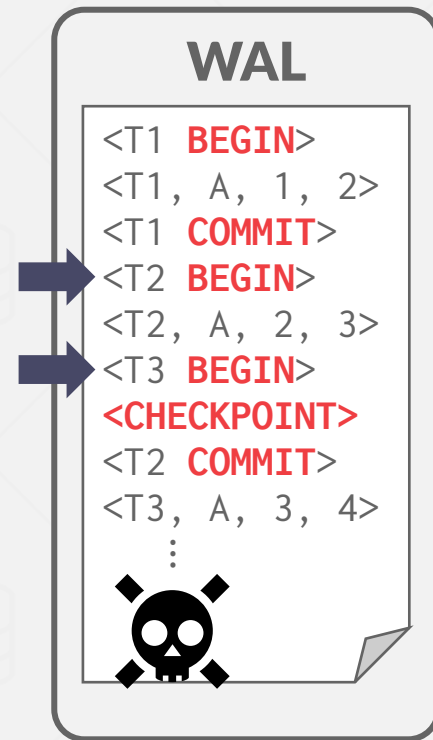
Use the **<CHECKPOINT>** record as the starting point for analyzing the WAL.

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

T

→ Need to redo  $T_2$  because it committed after checkpoint.

→ Need to undo  $T_3$  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.