Carnegie Mellon University

20 Database Logging

Intro to Database Systems
15-445/15-645
Fall 2020

Andy Pavlo
Computer Science
Carnegie Mellon University

ADMINISTRIVIA

Project #3 is due Sun Nov 22nd @ 11:59pm.

Project #4 will be released this week.

Homework #5 will be released next week.

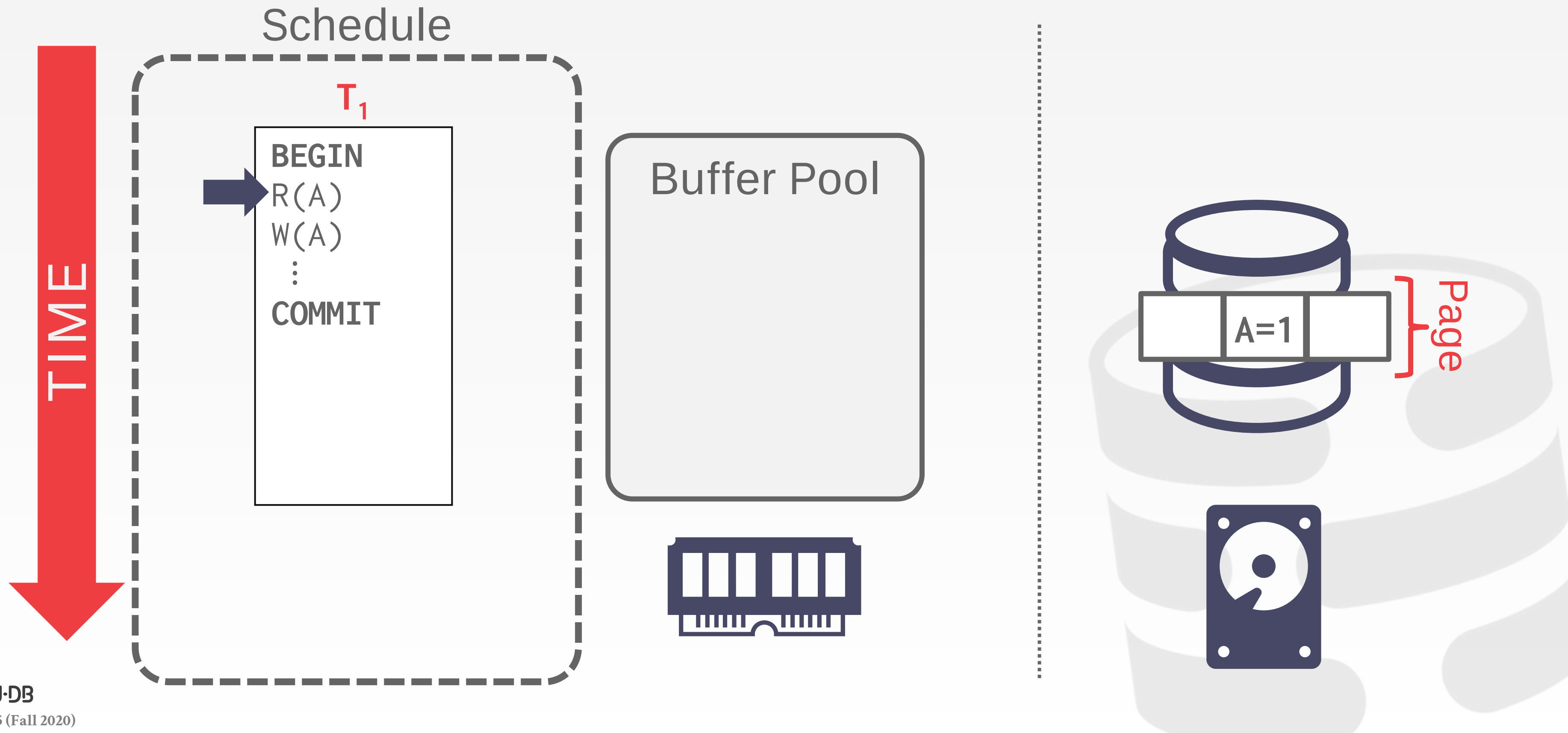
UPCOMING DATABASE TALKS

FaunaDB Serverless DBMS
→ Monday Nov 16th @ 5pm ET

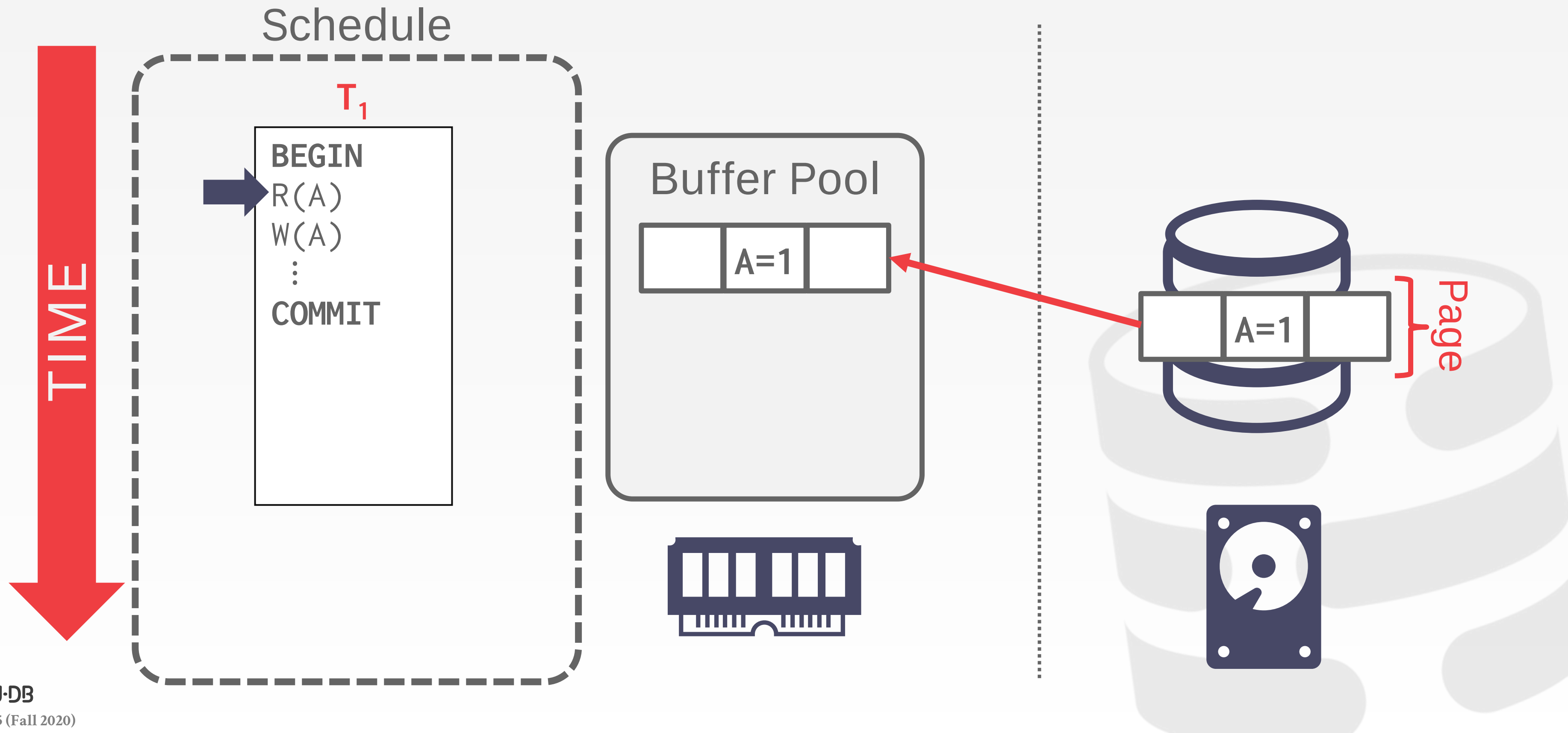
Confluent ksqlDB (Kafka)
→ Monday Nov 23rd @ 5pm ET

Microsoft SQL Server Optimizer
→ Monday Nov 30th @ 5pm ET

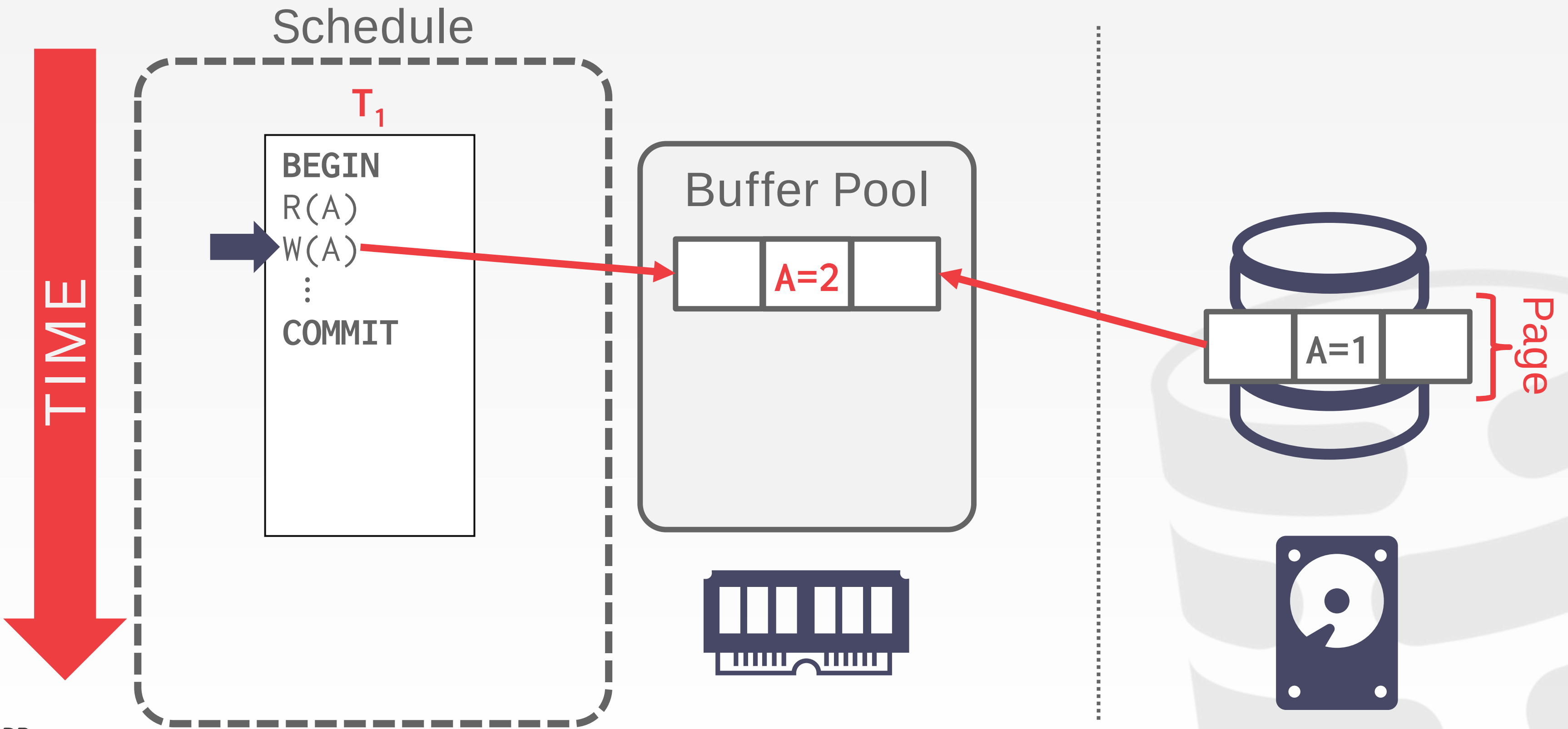
MOTIVATION



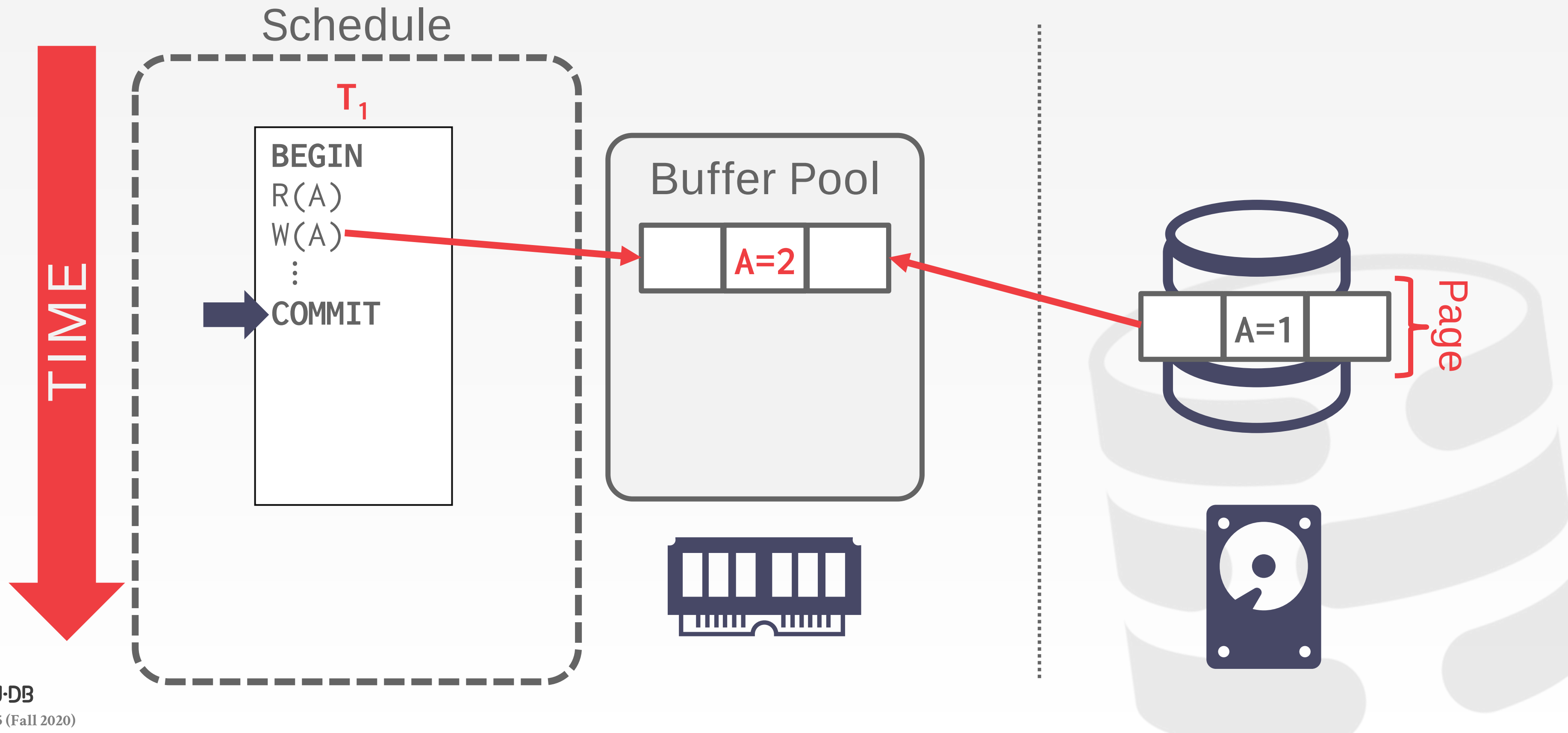
MOTIVATION



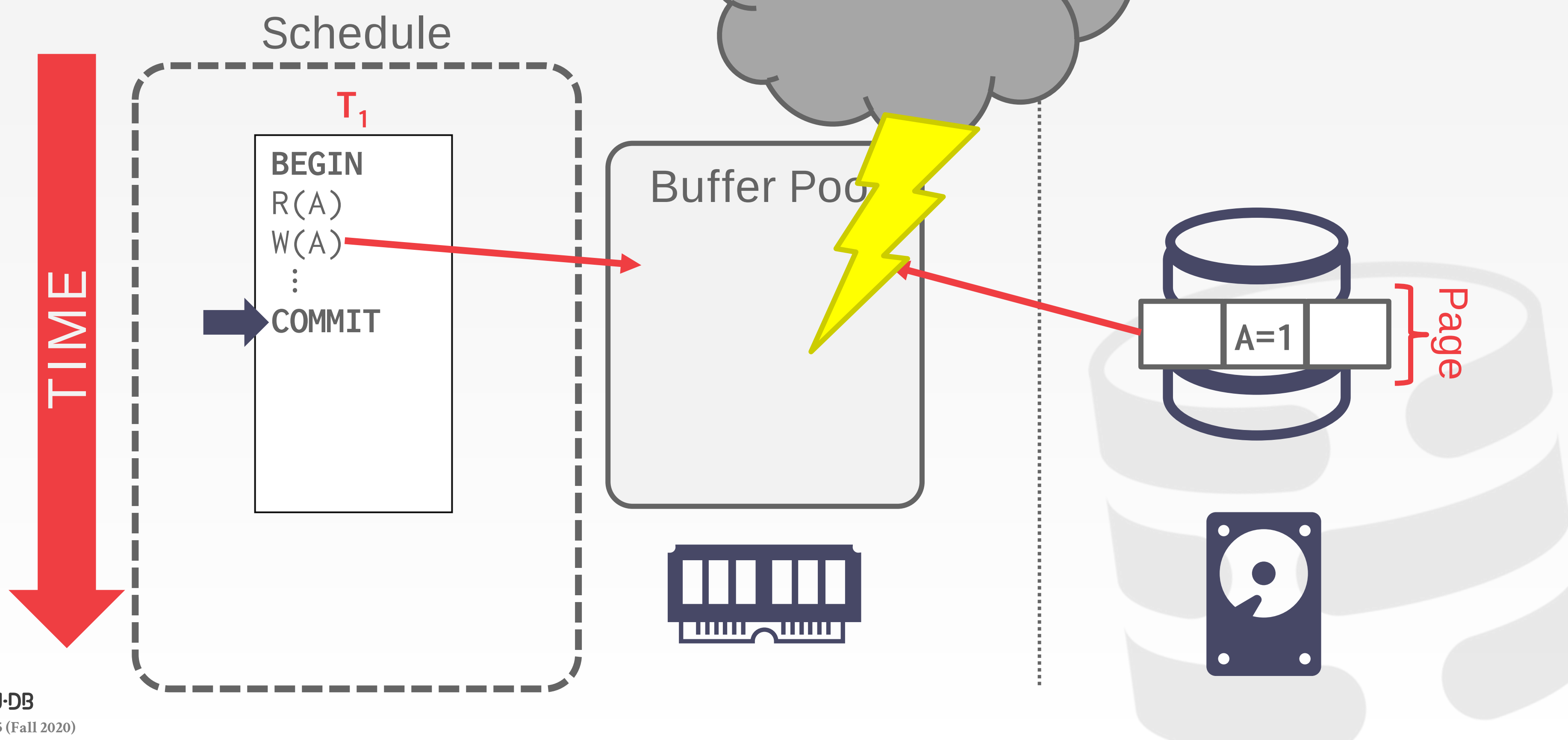
MOTIVATION



MOTIVATION



MOTI



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.

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

OBSERVATION

The DBMS needs to ensure the following guarantees:

→ 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-instating the effects of a committed txn for durability.

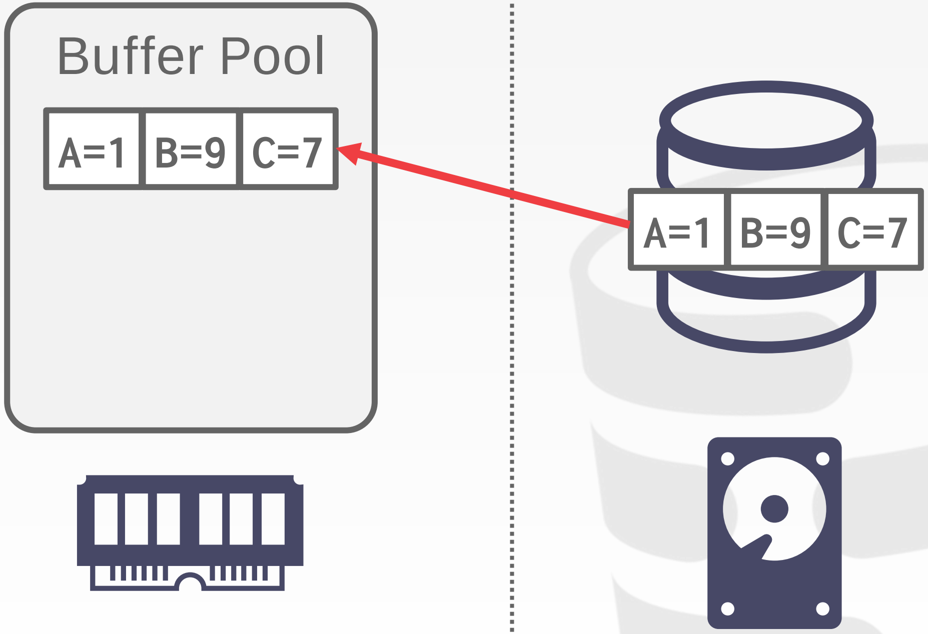
How the DBMS supports this functionality depends on how it manages the buffer pool…

BUFFER POOL

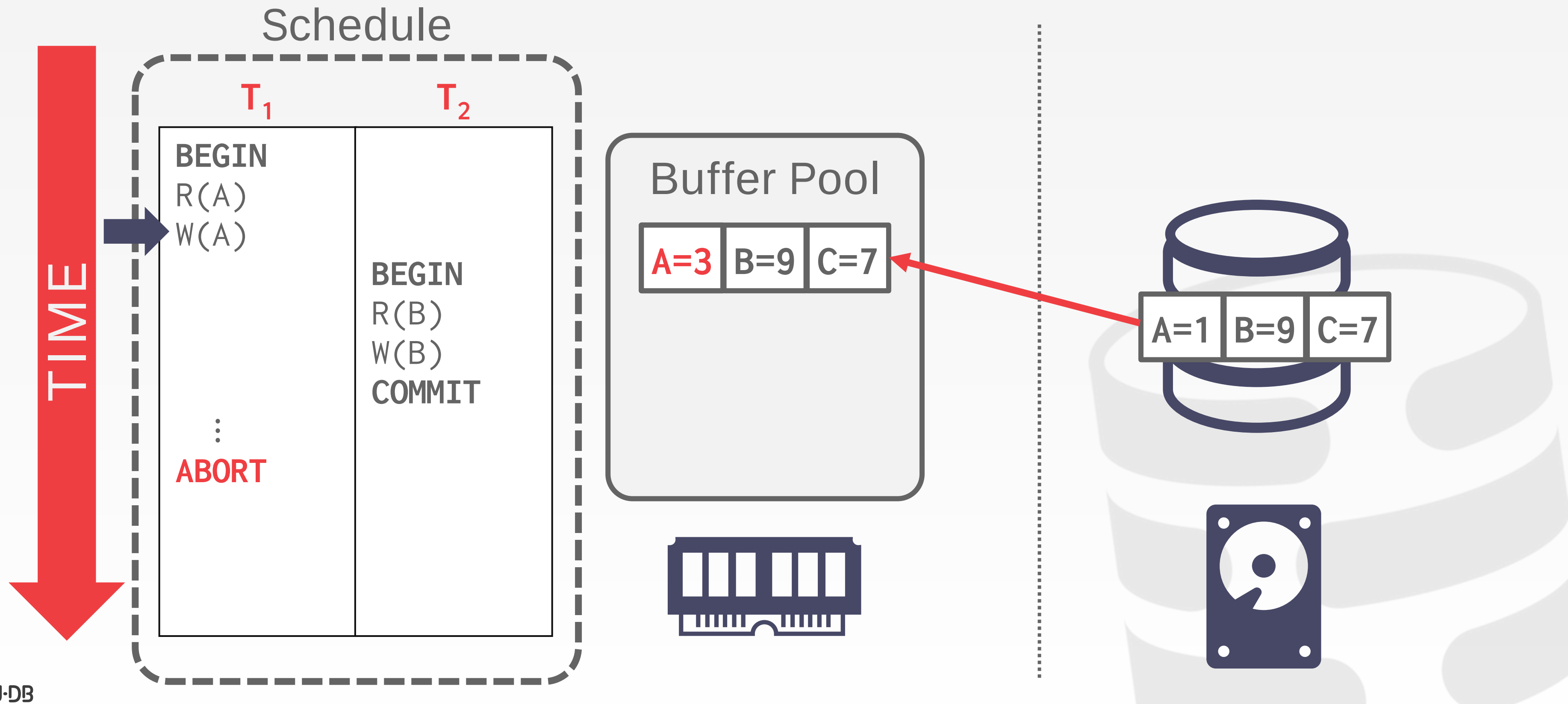
Schedule

TIME

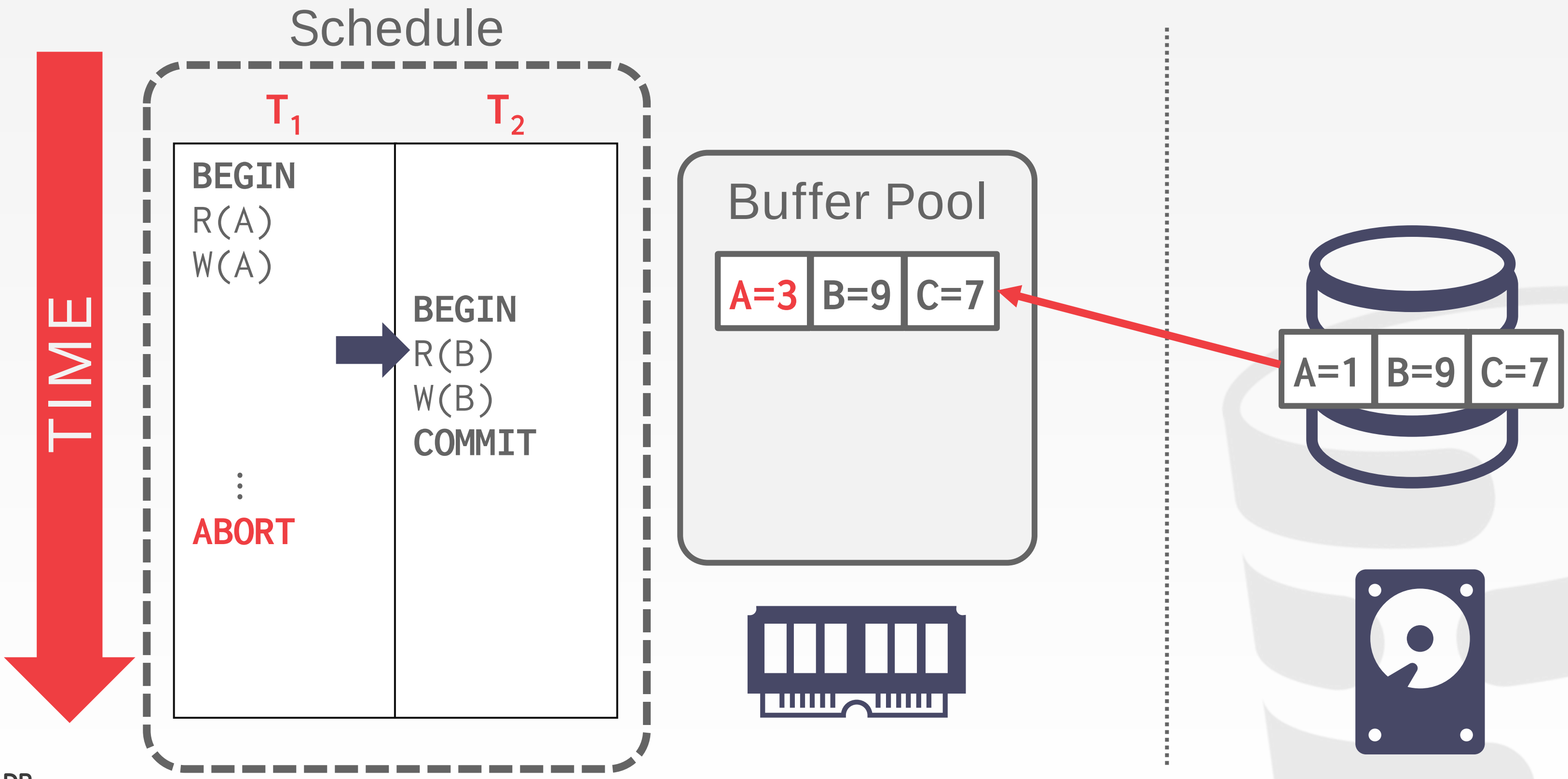
T_1	T_2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(B)
	W(B)
	COMMIT
⋮	
ABORT	



BUFFER POOL


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

BUFFER POOL

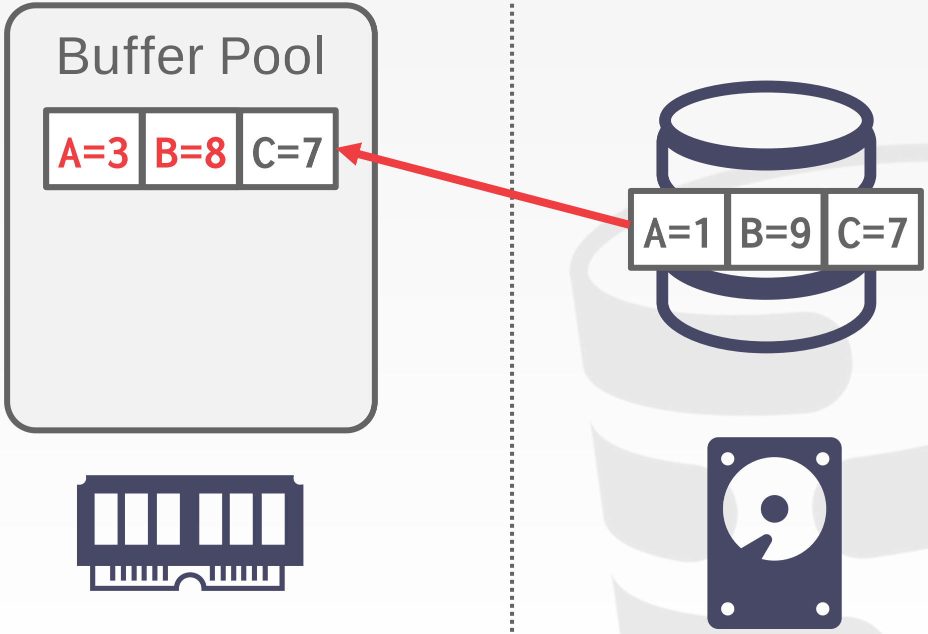

Schedule
TIME
T1
T2
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
⋮
ABORT
Buffer Pool
A=3
B=9
C=7
A=1
B=9
C=7

BUFFER POOL

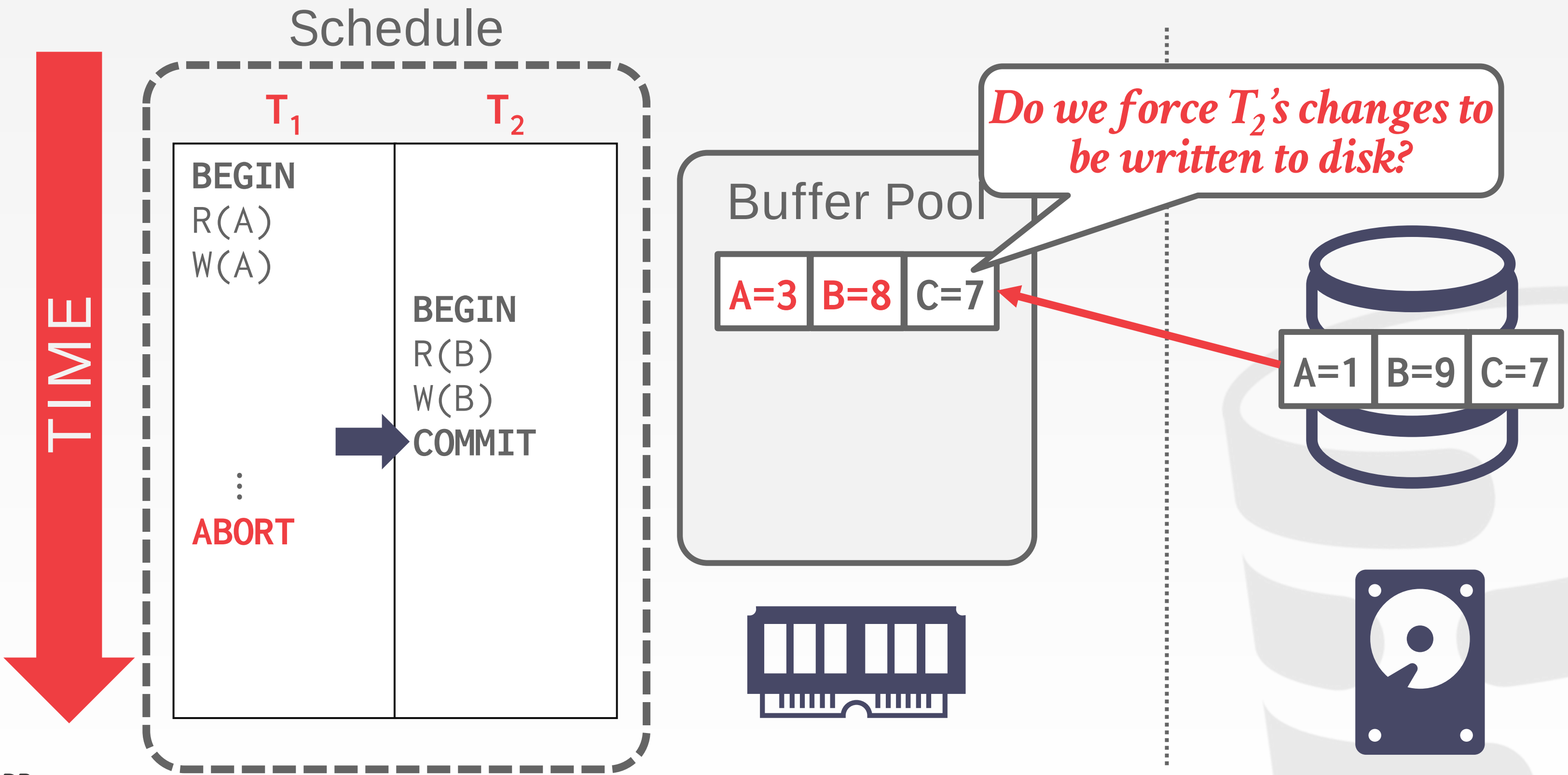
TIME

Schedule

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

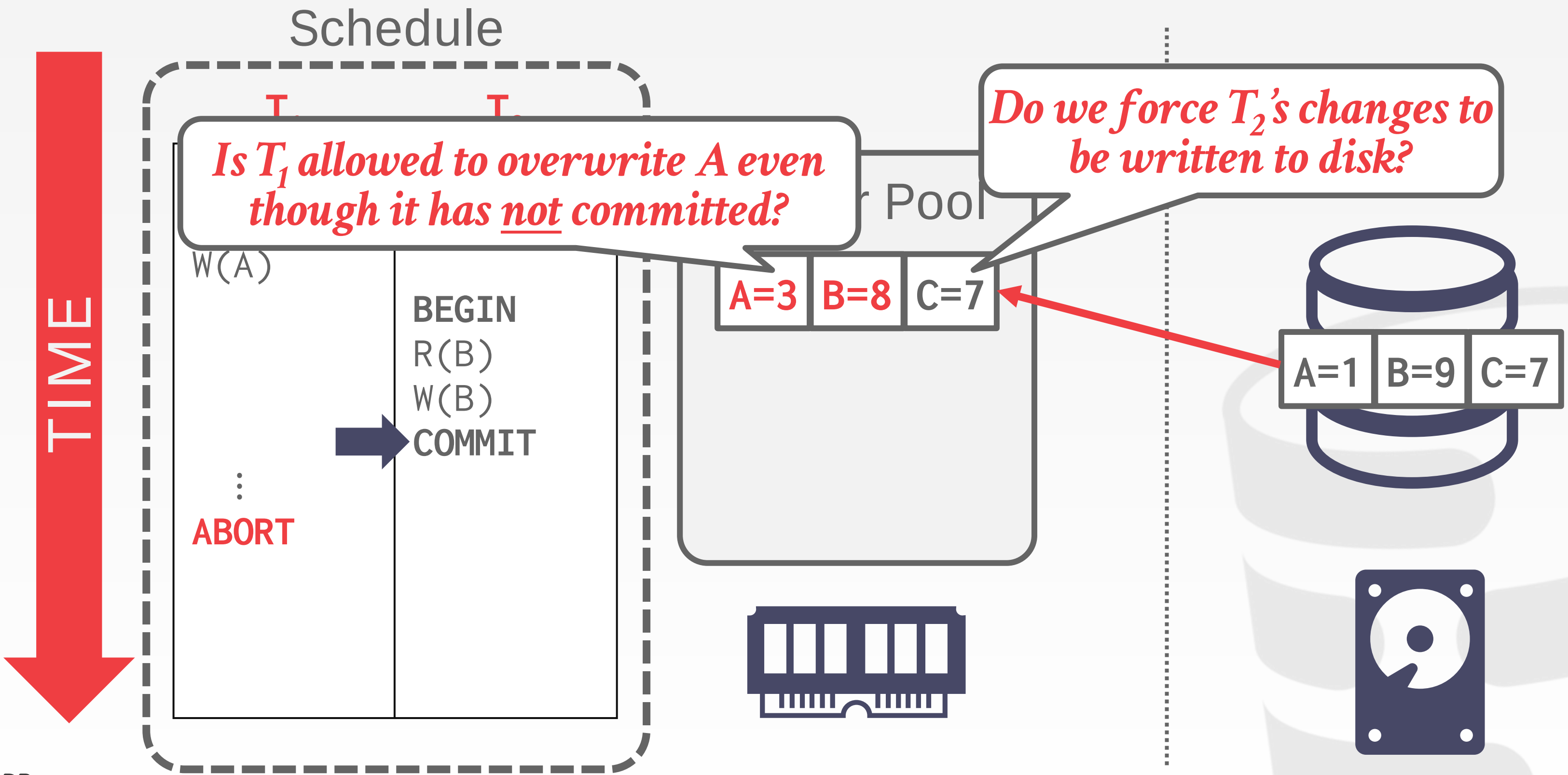


BUFFER POOL

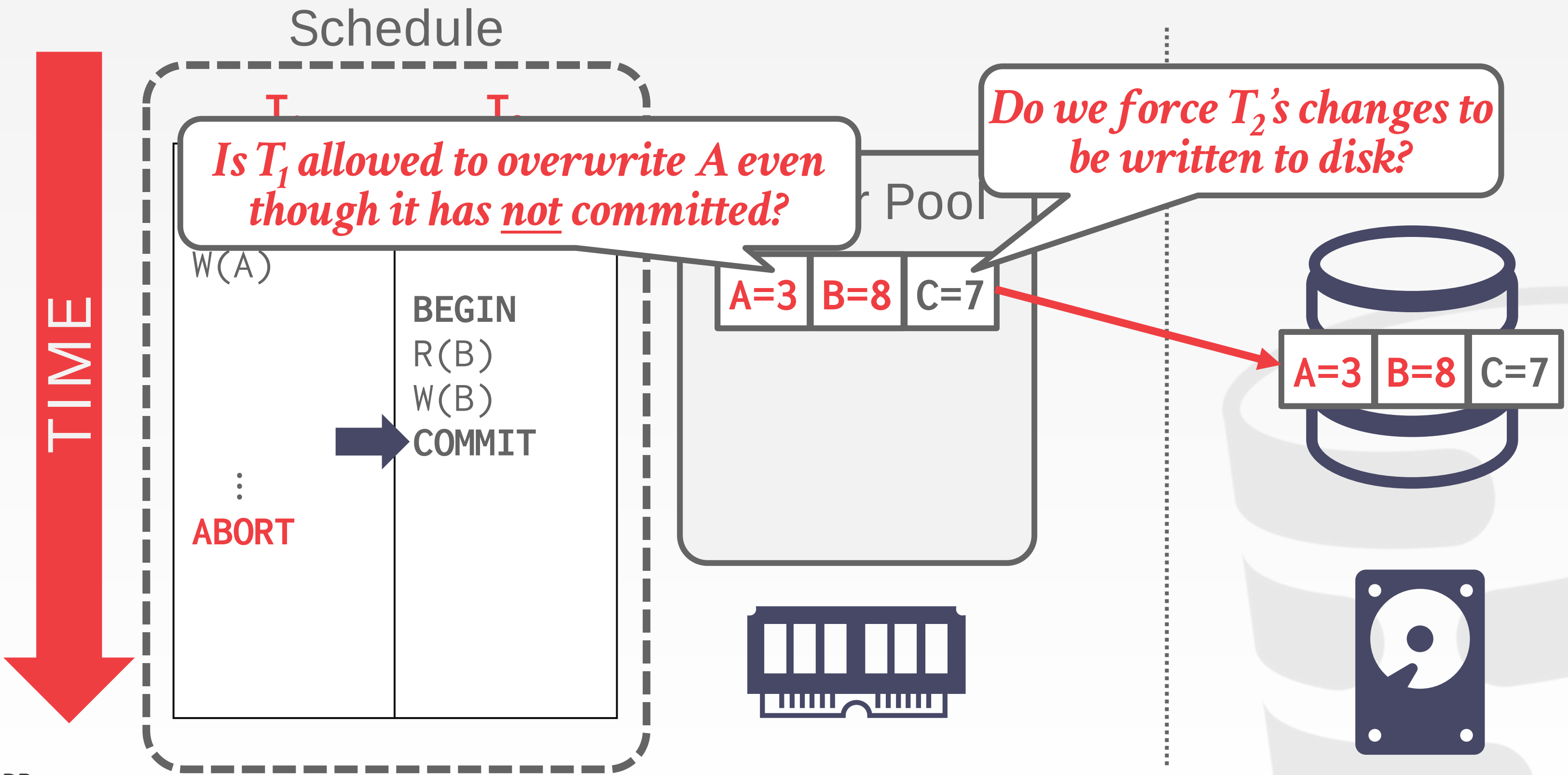


Schedule
TIME
T_1
T_2
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
⋮
ABORT
Buffer Pool
A=3
B=8
C=7
Do we force T_2's changes to be written to disk?
A=1
B=9
C=7

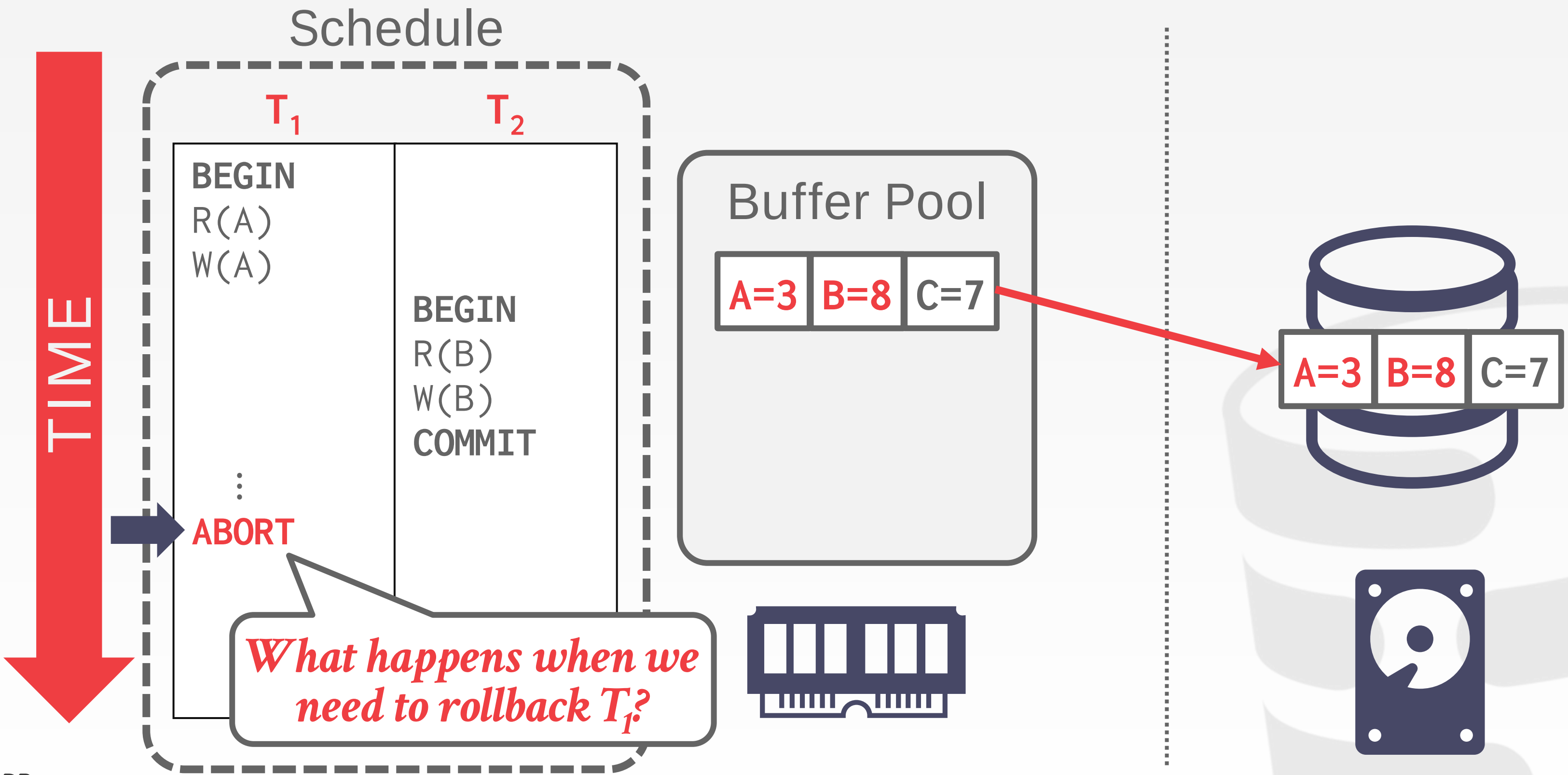
BUFFER POOL


Schedule
TIME
W(A)
ABORT
BEGIN
R(B)
W(B)
COMMIT
Is T_1 allowed to overwrite A even though it has not committed?
Pool
Do we force T_2's changes to be written to disk?
A=3
B=8
C=7
A=1
B=9
C=7

BUFFER POOL


Schedule
TIME
Is T_1 allowed to overwrite A even though it has not committed?
W(A)
BEGIN
R(B)
W(B)
COMMIT
⋮
ABORT
Pool
Do we force T_2's changes to be written to disk?
A=3
B=8
C=7
A=3
B=8
C=7

BUFFER POOL


Schedule
TIME
T_1
T_2
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
⋮
ABORT
What happens when we need to rollback T_1?
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.

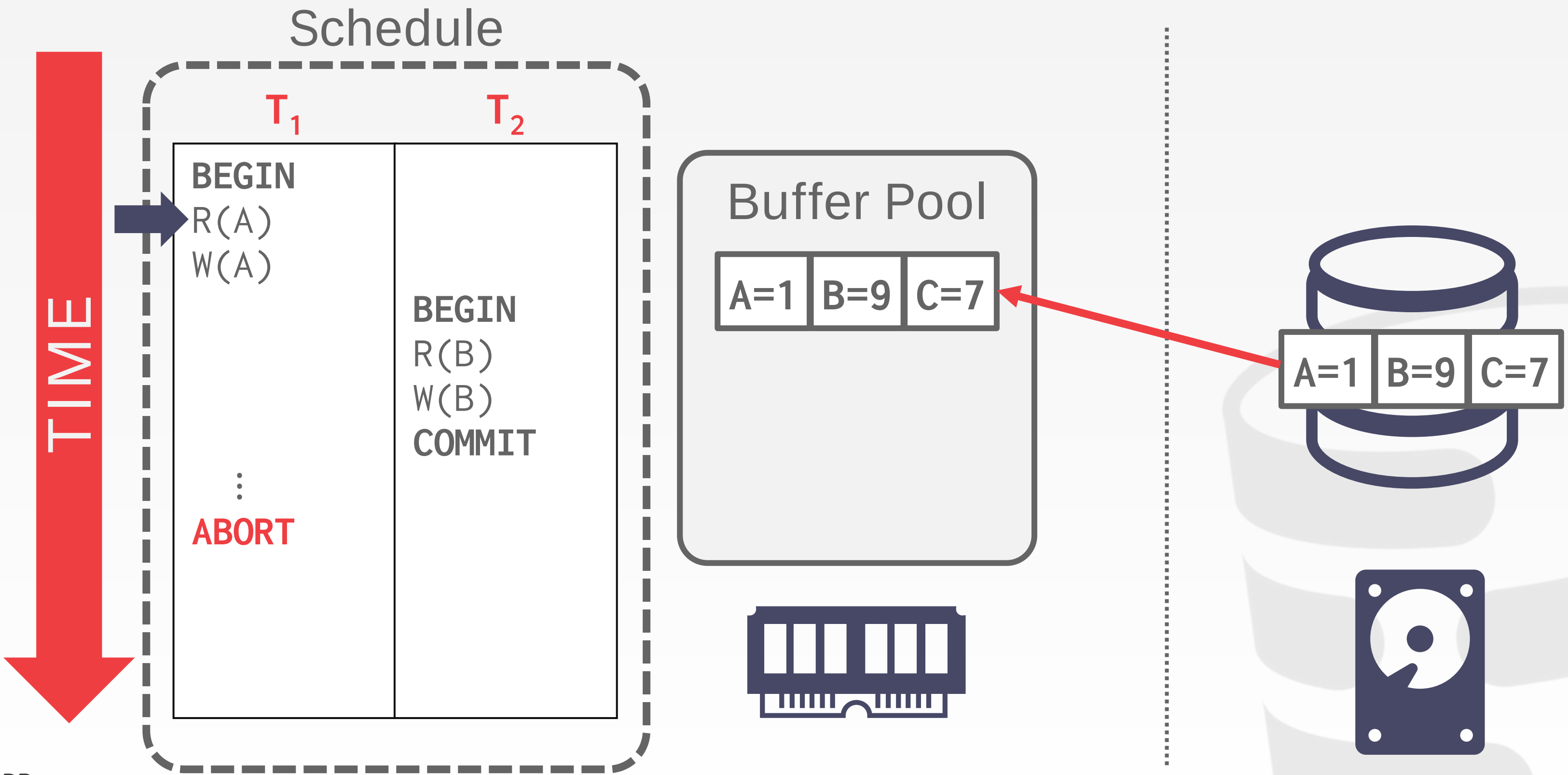
FORCE POLICY

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

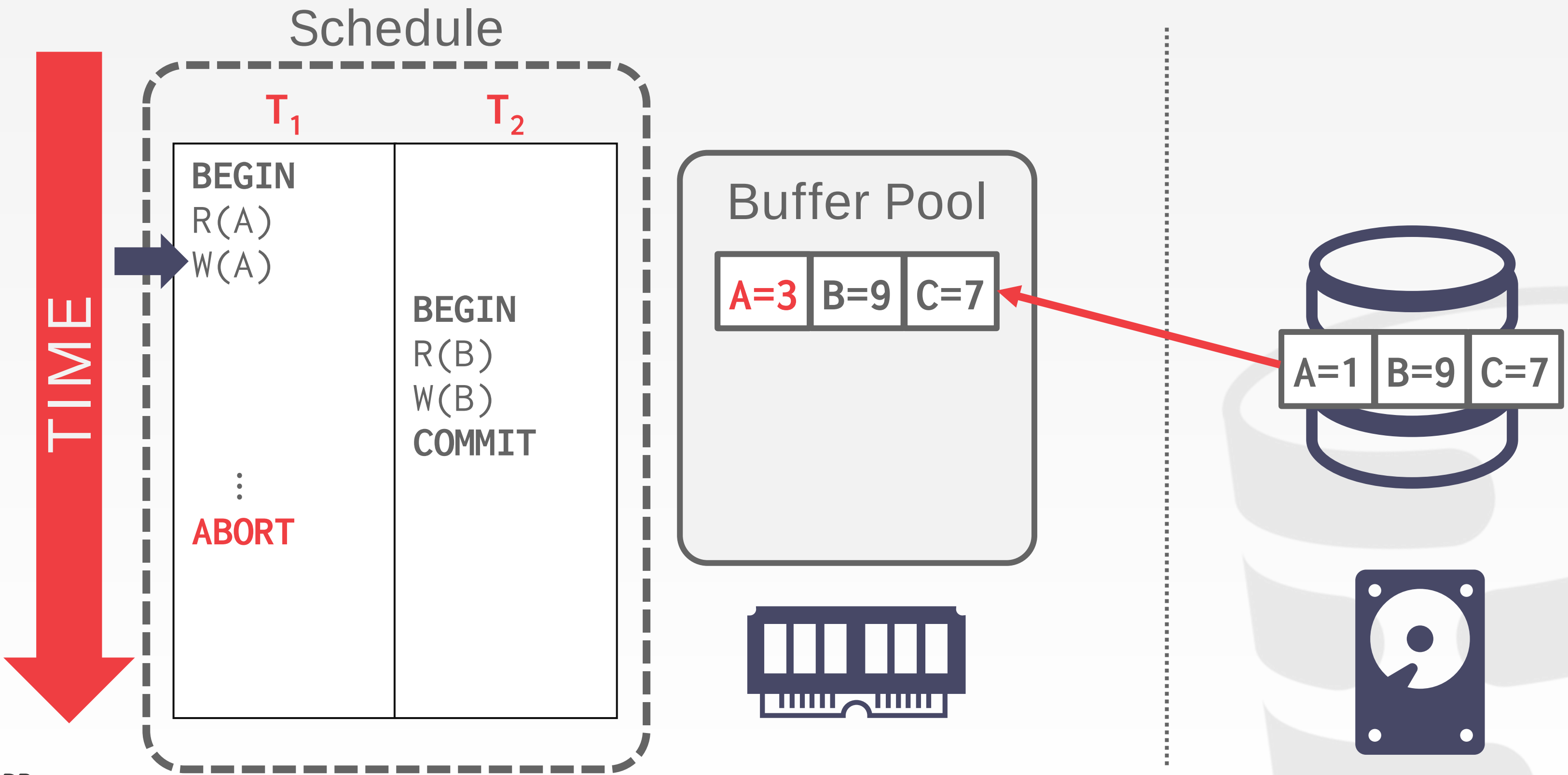
FORCE: Is required.

NO-FORCE: Is not required.

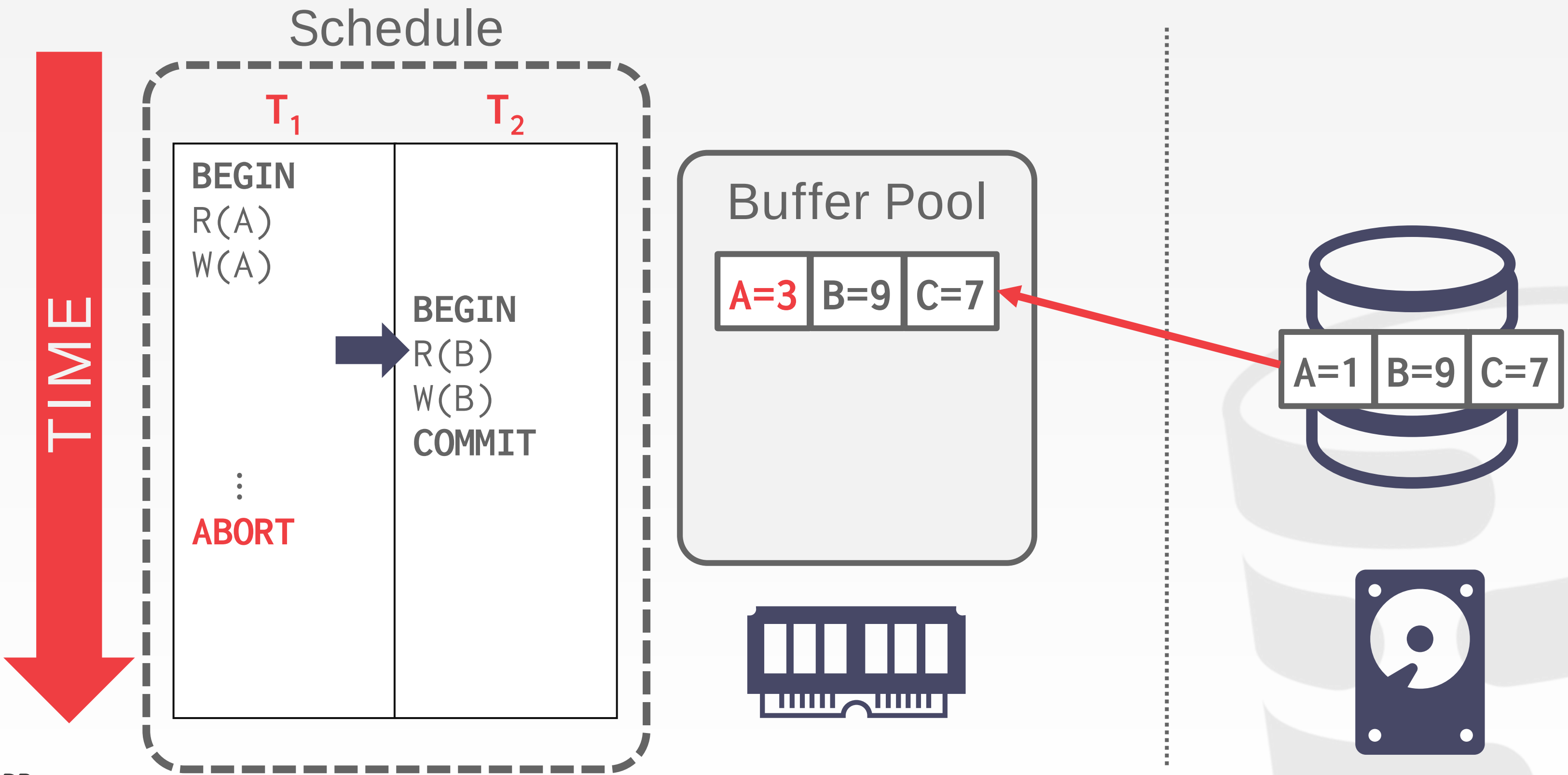
NO-STEAL + FORCE



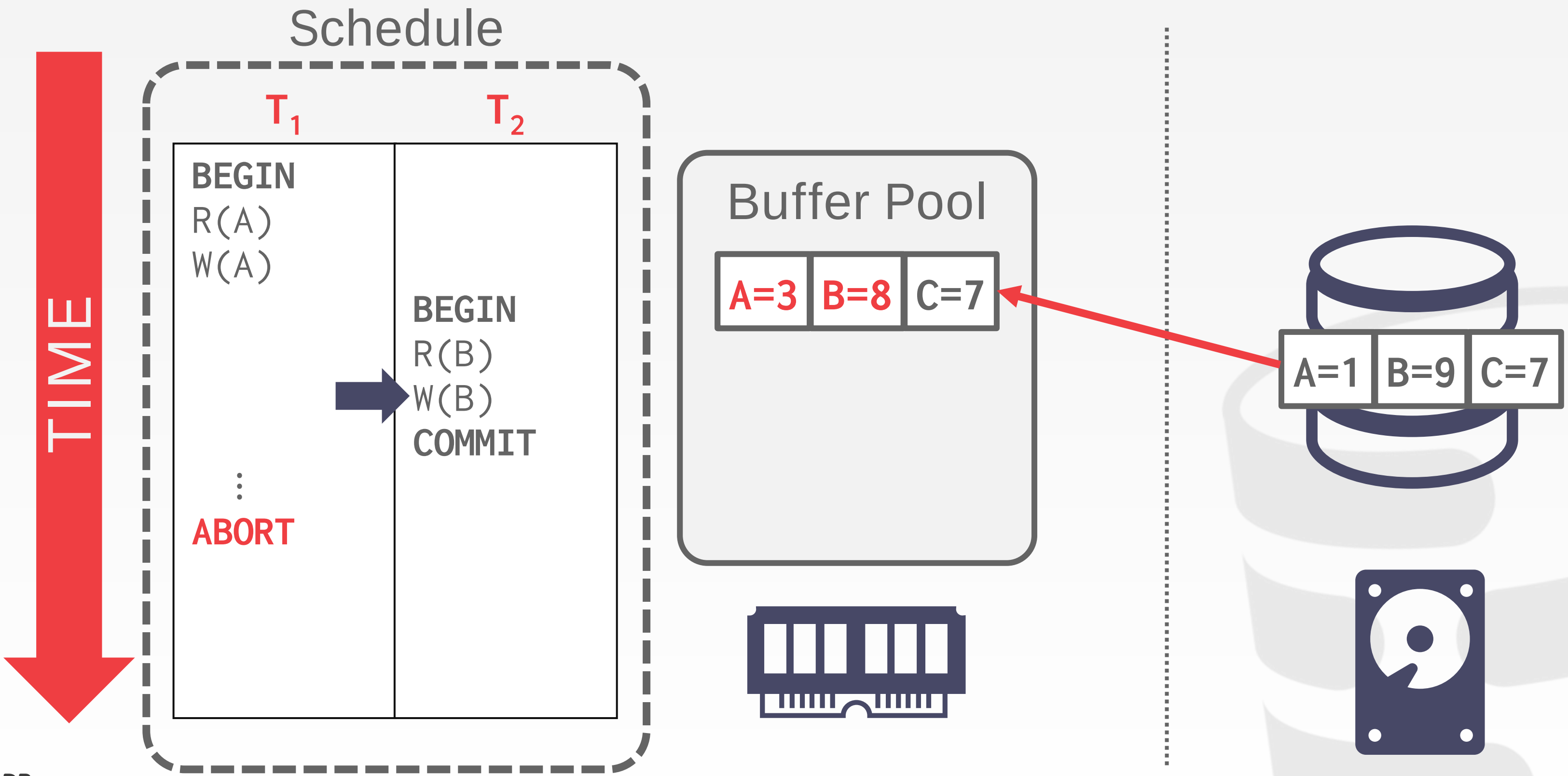
NO-STEAL + FORCE



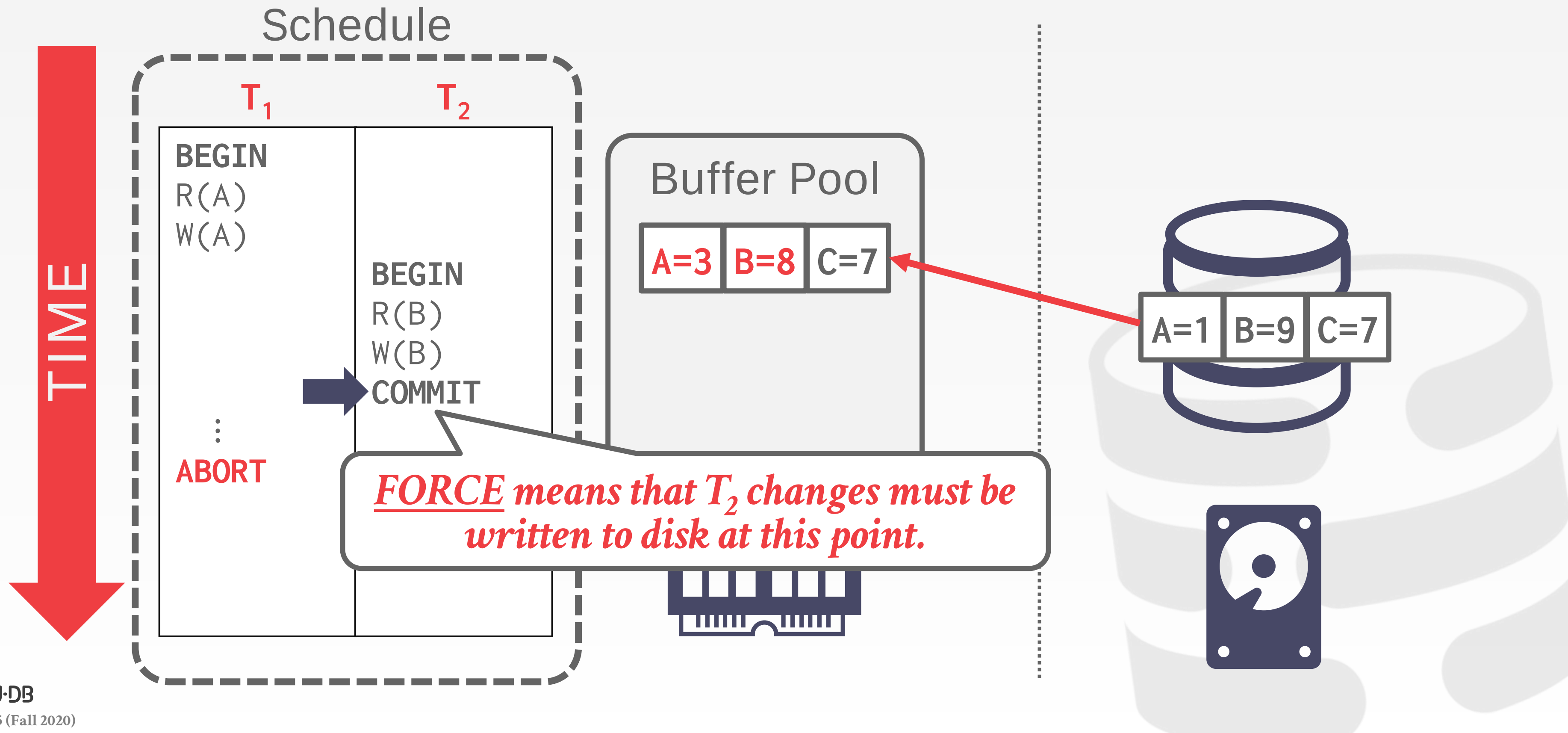
NO-STEAL + FORCE



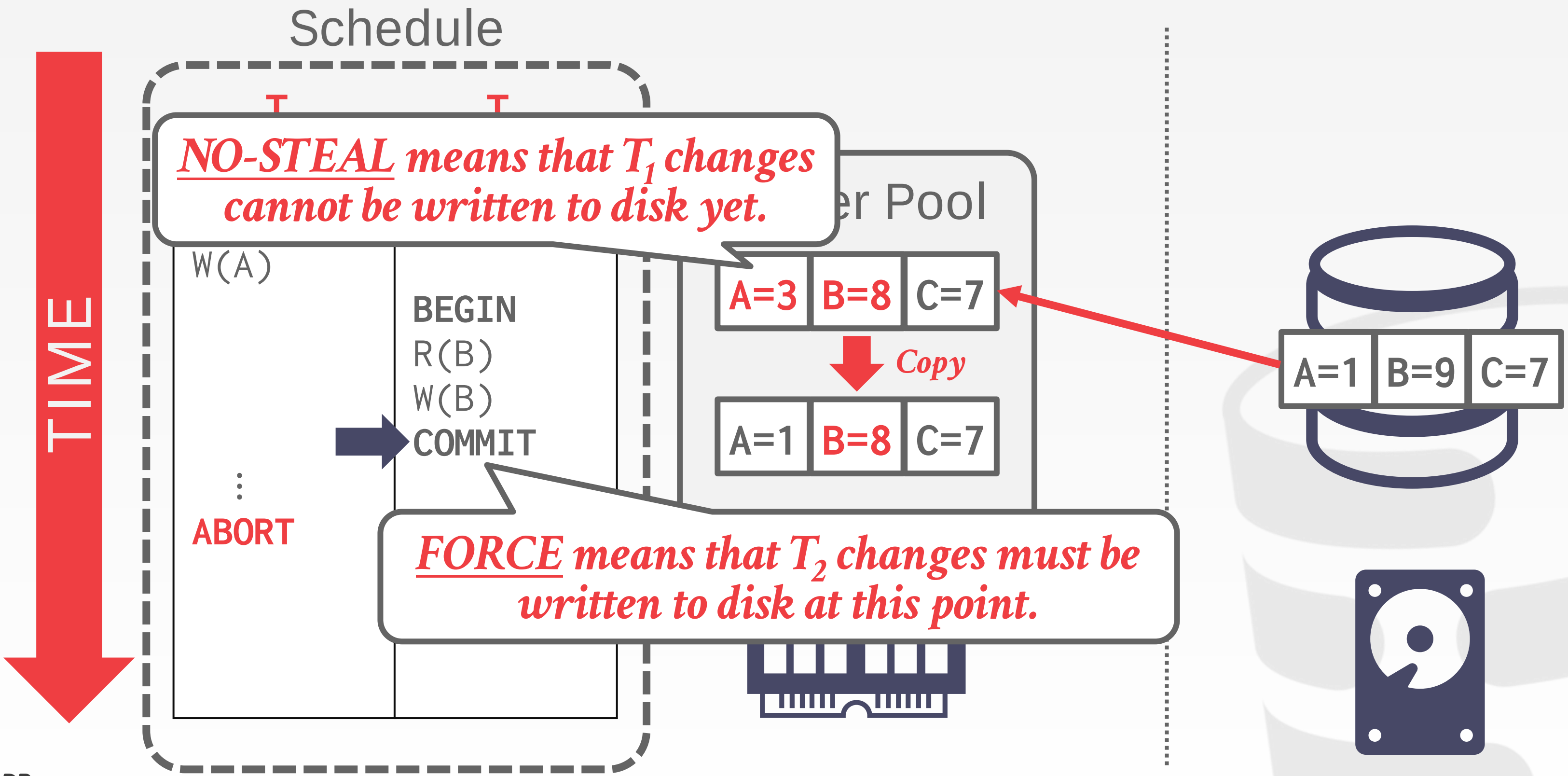
NO-STEAL + FORCE



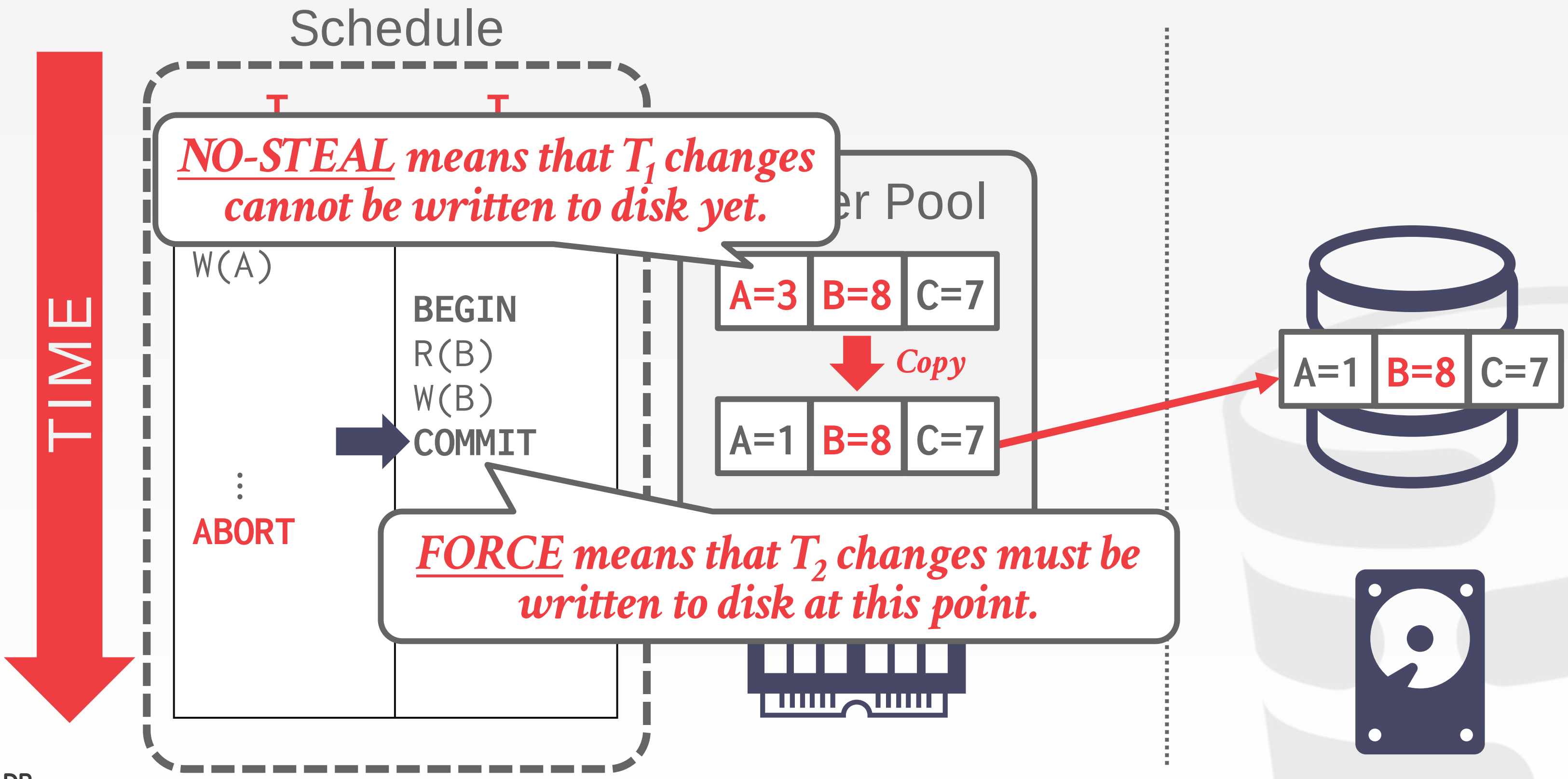
NO-STEAL + FORCE



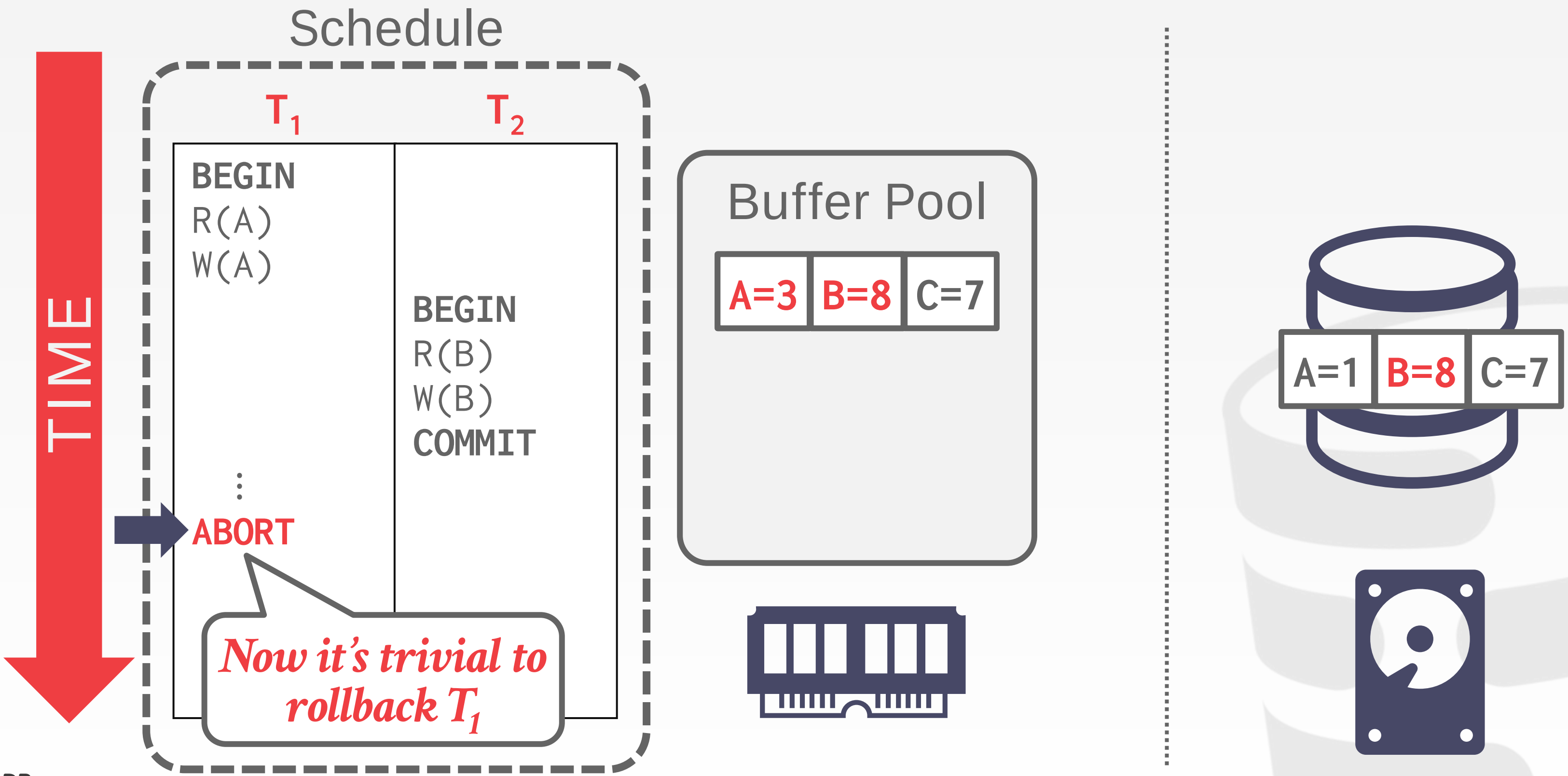
NO-STEAL + FORCE



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

Maintain two separate copies of the database:

→ **Master**: Contains only changes from committed txns.

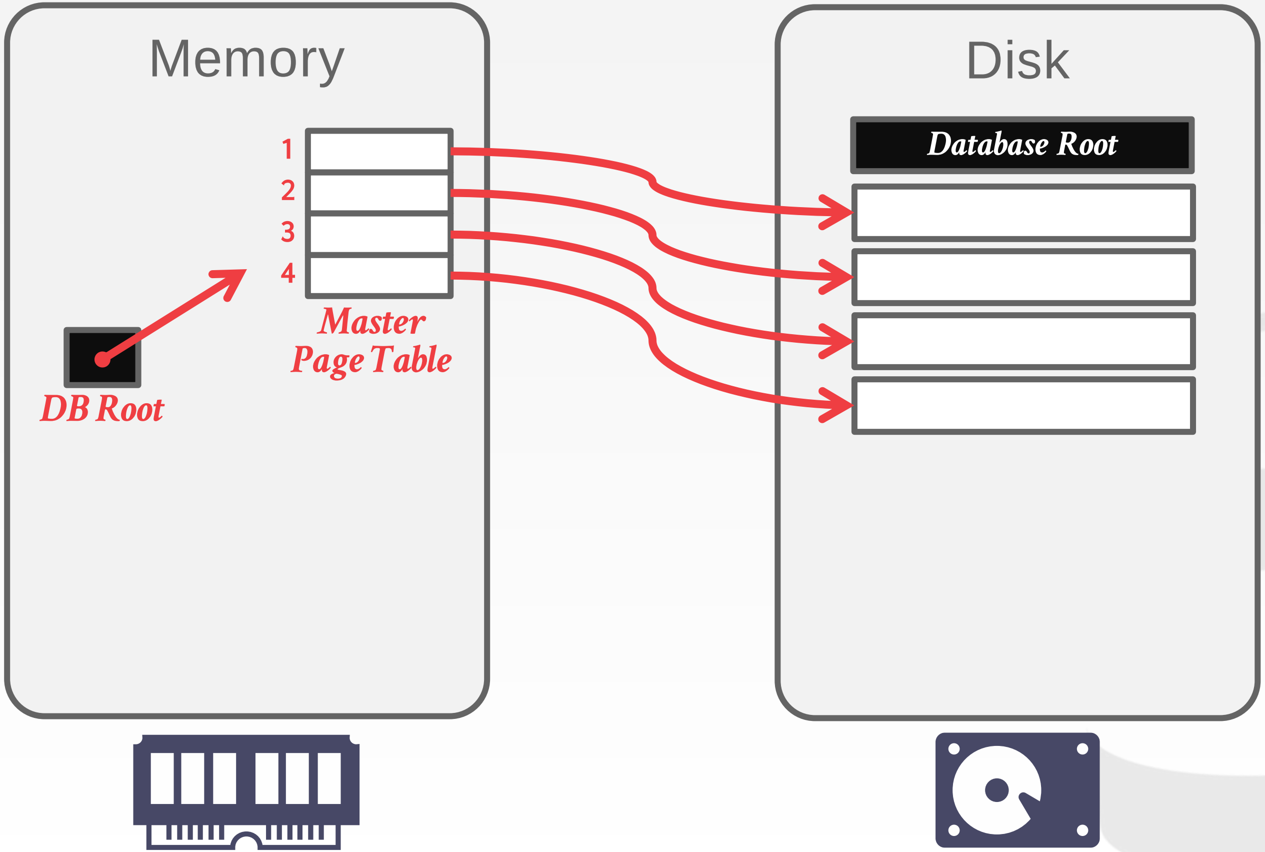
→ **Shadow**: Temporary database with changes made from uncommitted txns.

Txns only make updates in the shadow copy.

When a txn commits, atomically switch the shadow to become the new master.

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

SHADOW PAGING – EXAMPLE



SHADOW PAGING

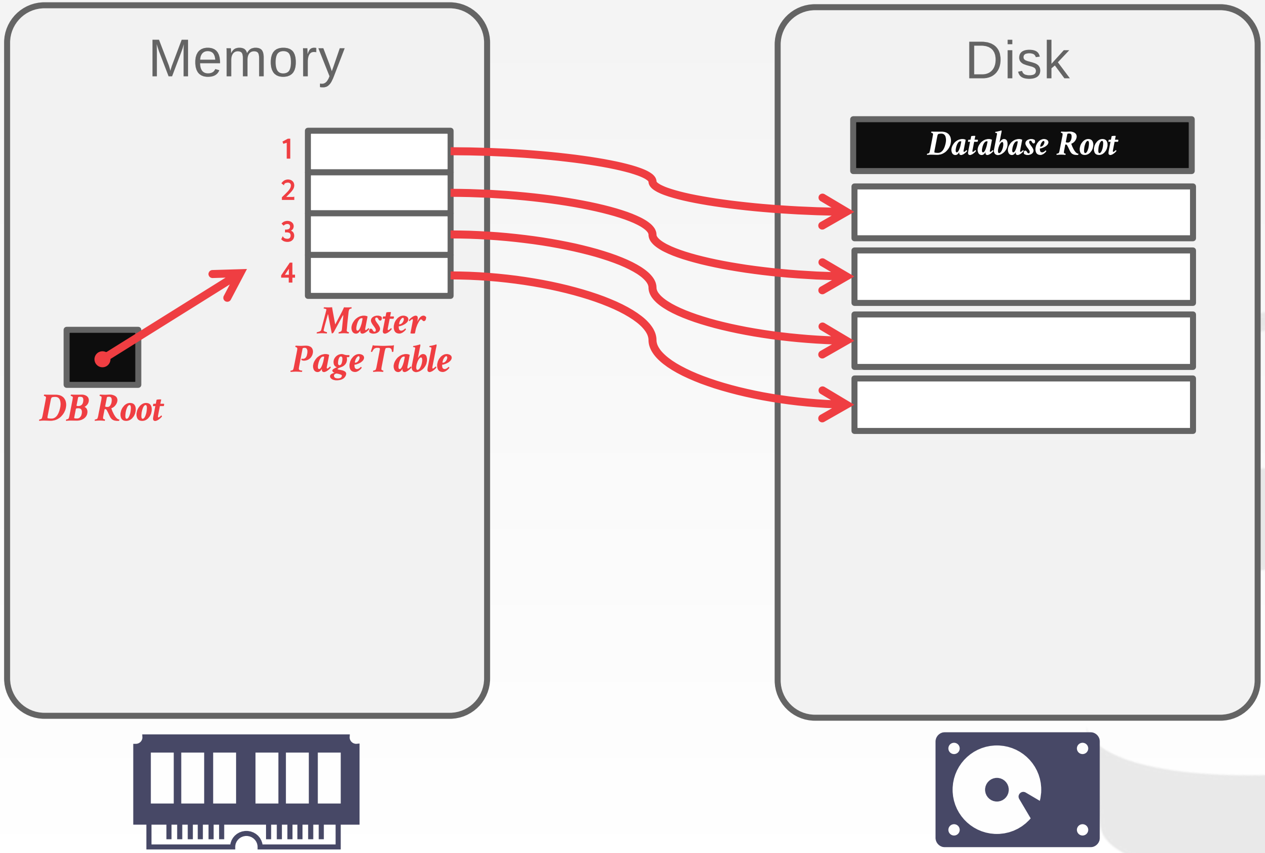
To install the updates, overwrite the root so it points to the shadow, thereby swapping the master and shadow:

→ Before overwriting the root, none of the txn's updates are part of the disk-resident database

→ After overwriting the root, all the txn's updates are part of the disk-resident database.

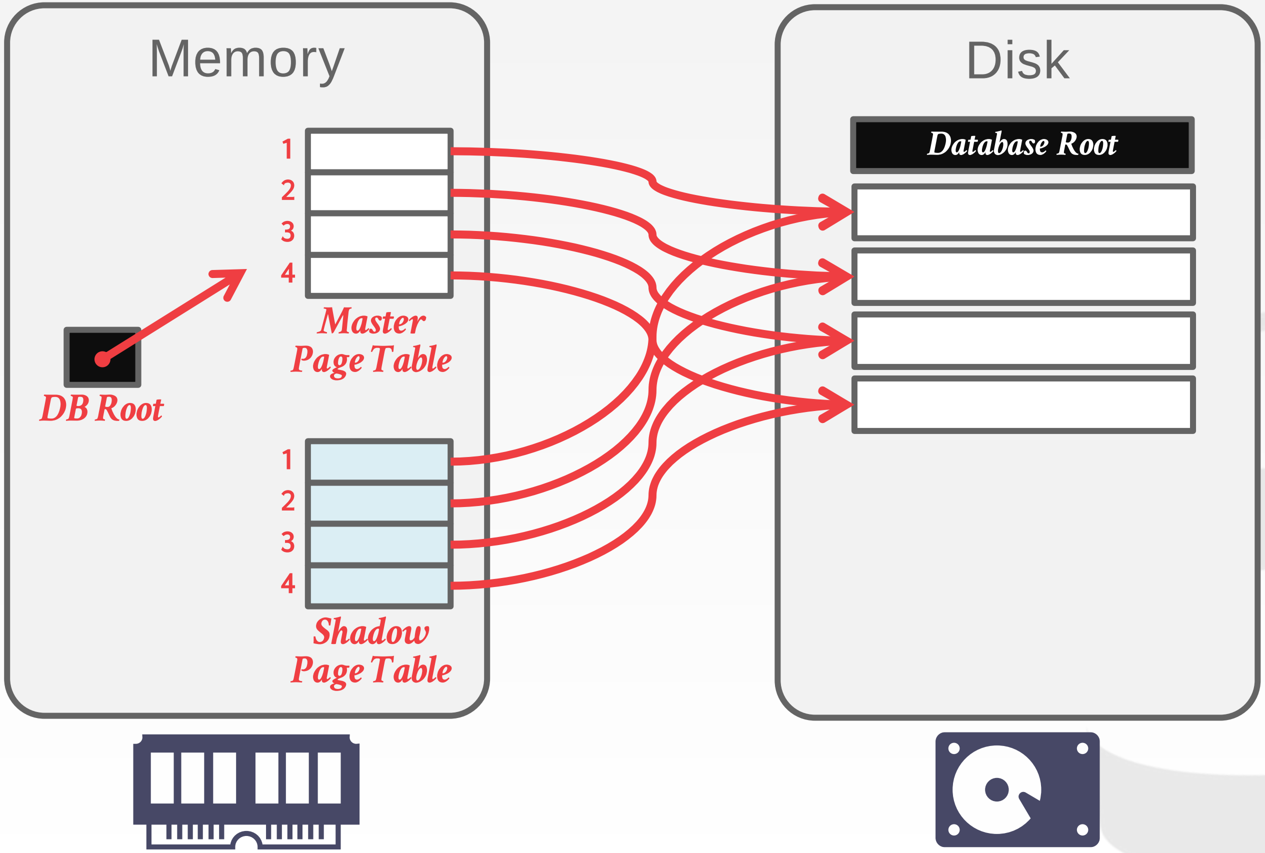
Source: The Great Phil Bernstein

SHADOW PAGING – EXAMPLE



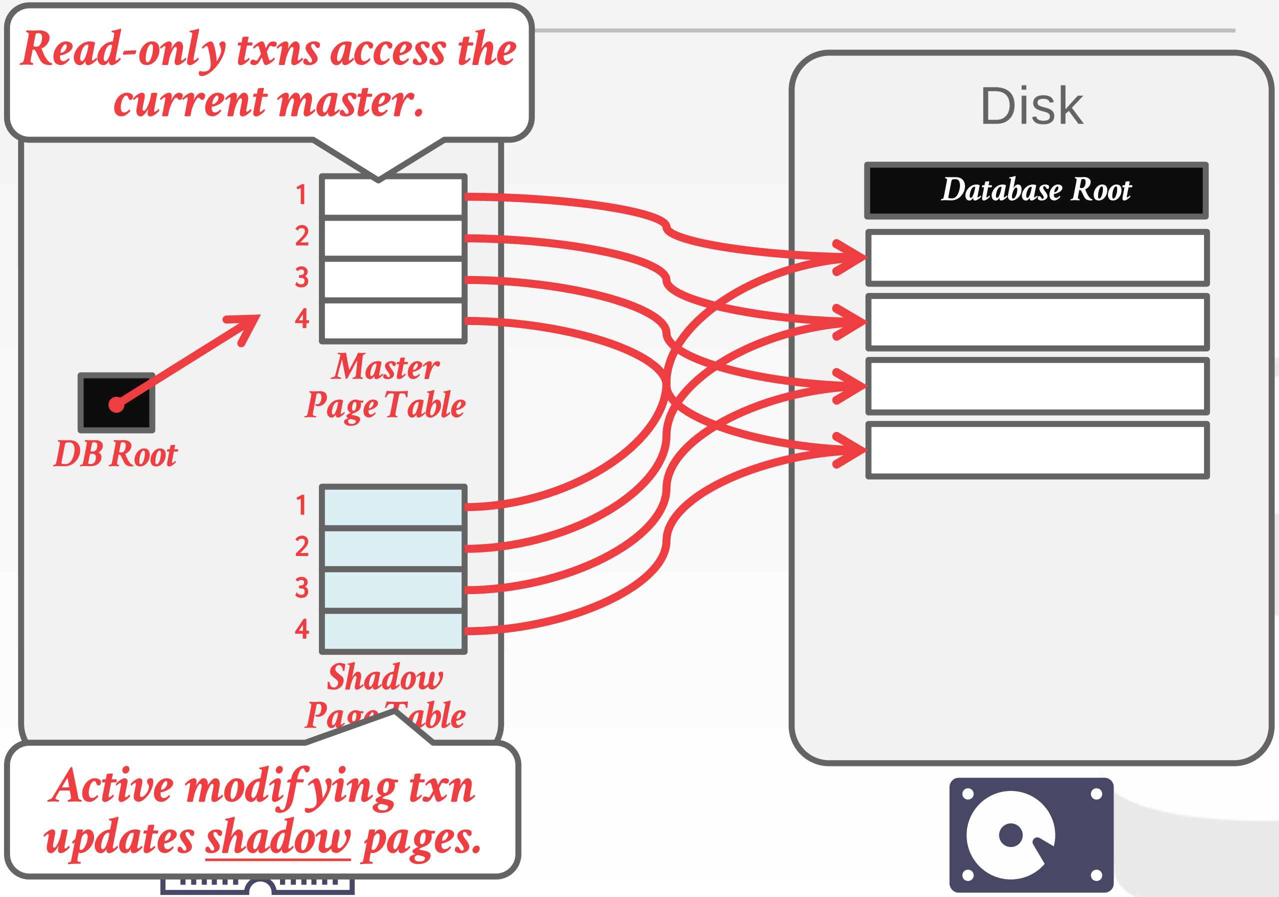
Txn T_1

SHADOW PAGING – EXAMPLE



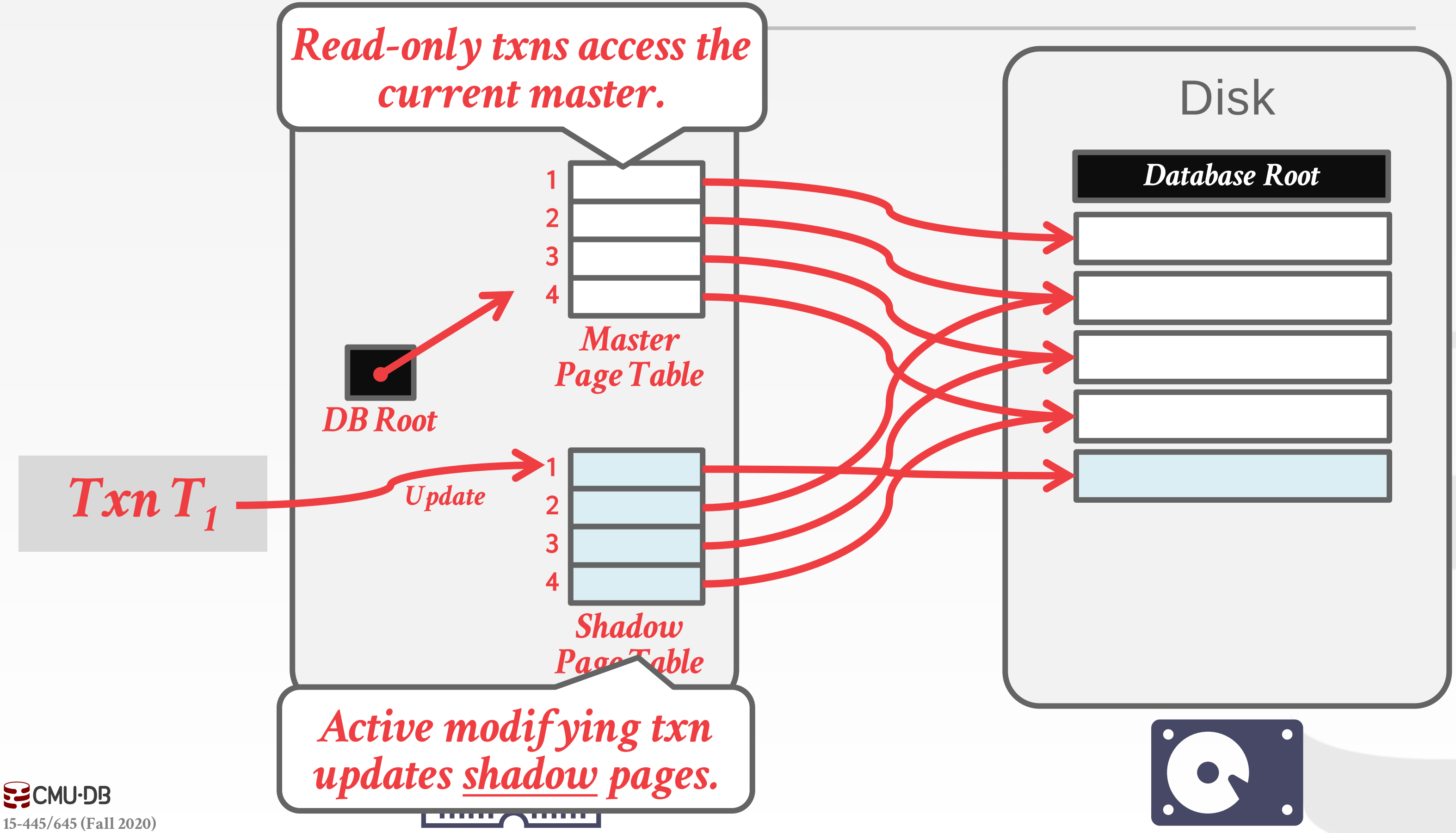
Txn T_1

SHADOW PAGING – EXAMPLE

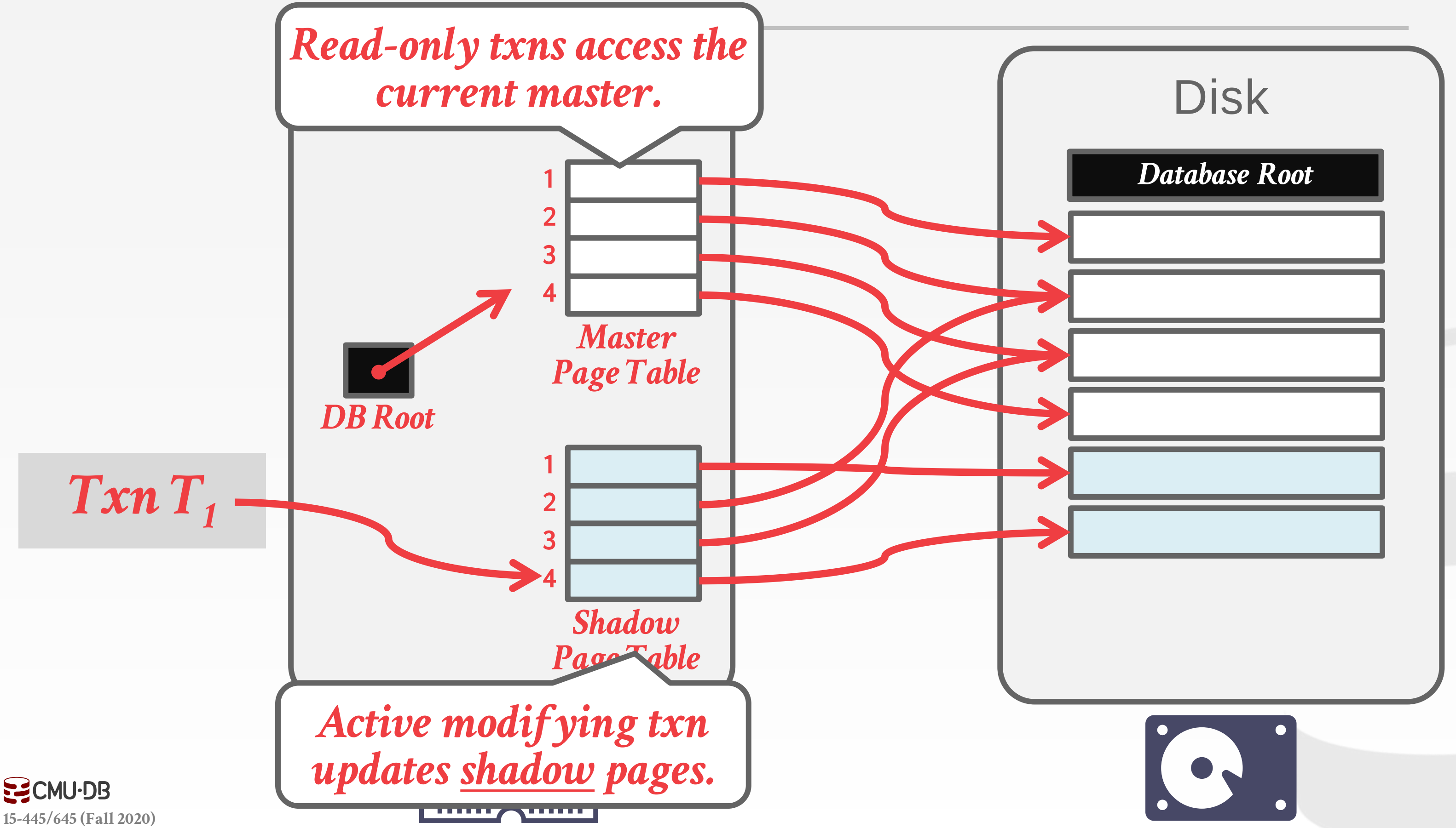


Txn T_1

SHADOW PAGING – EXAMPLE

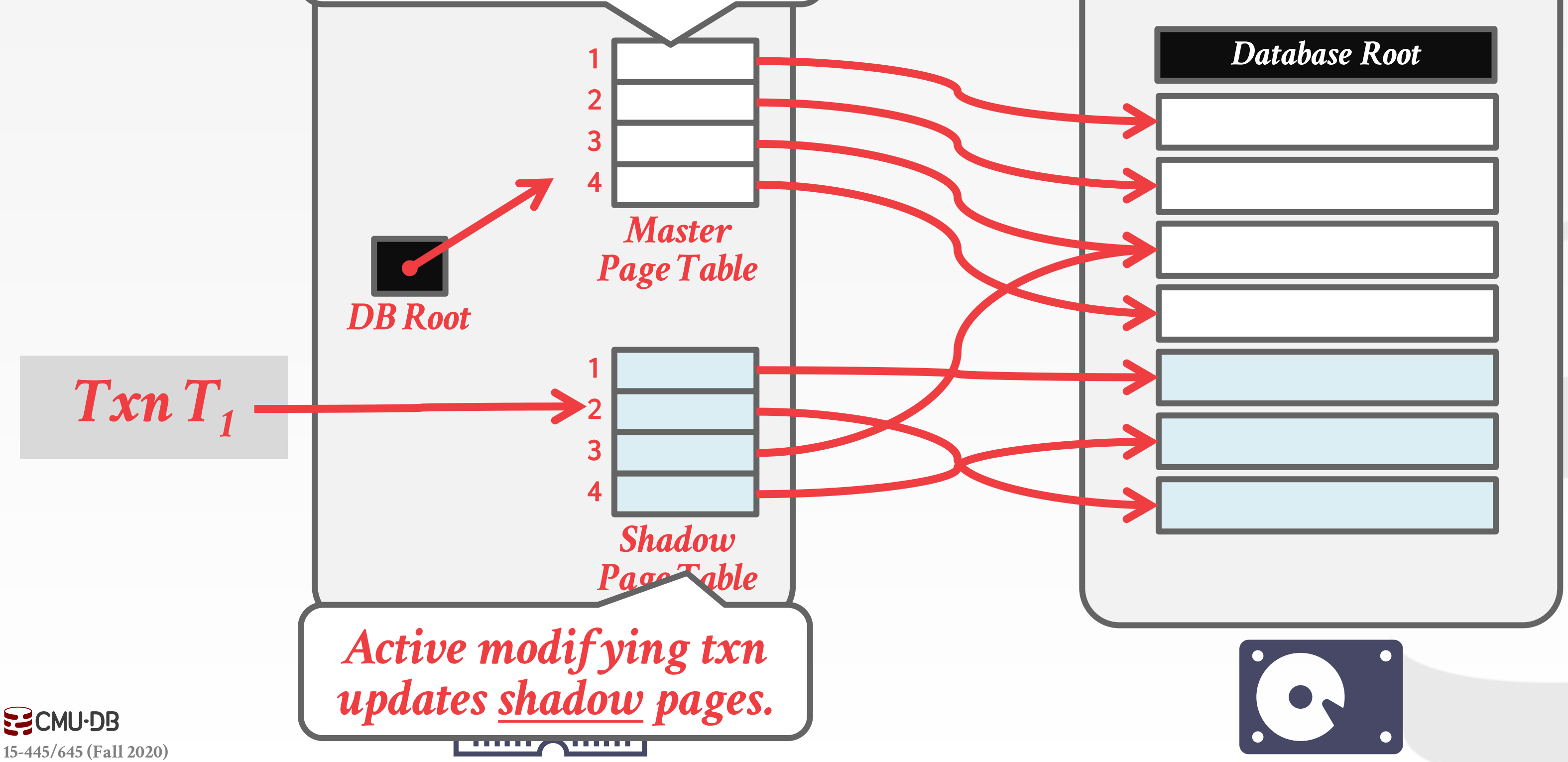


SHADOW PAGING – EXAMPLE



SHADOW PAGING – EXAMPLE

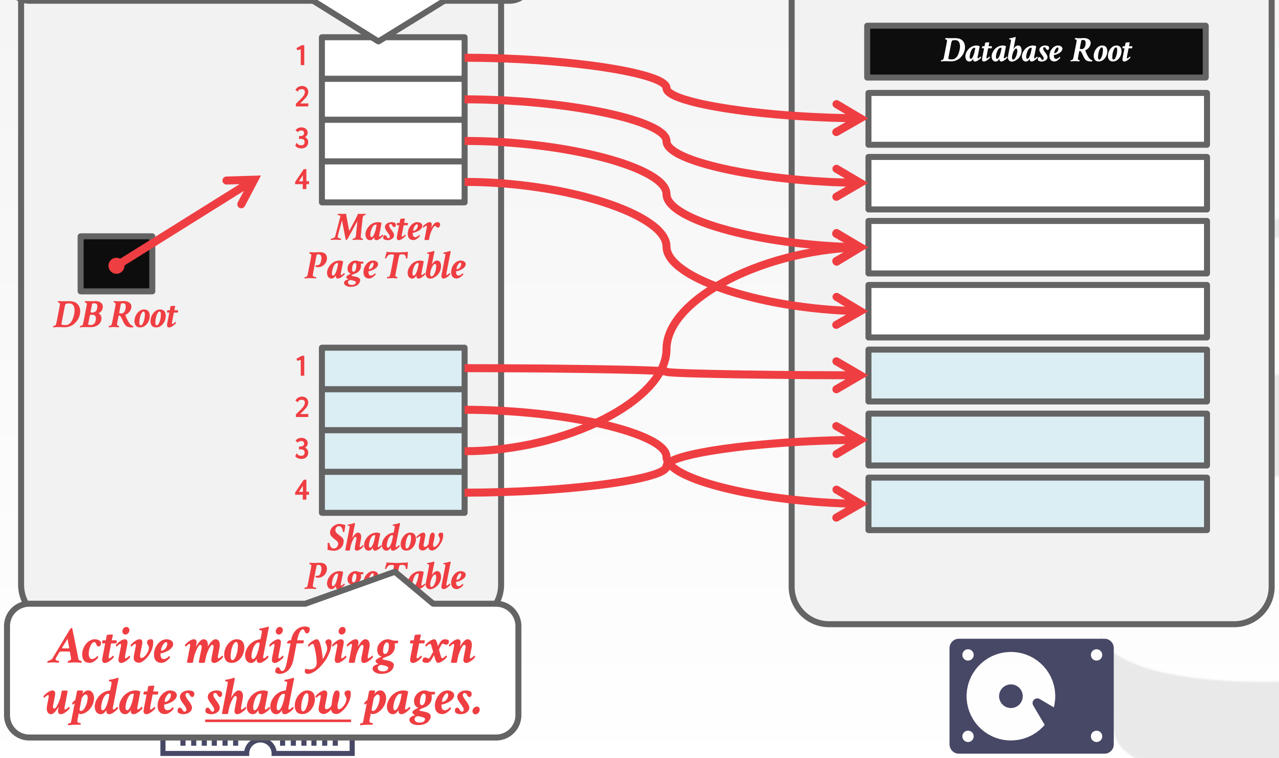
Read-only txns access the current master.



Active modifying txn updates shadow pages.

SHADOW PAGING – EXAMPLE

Read-only txns access the current master.



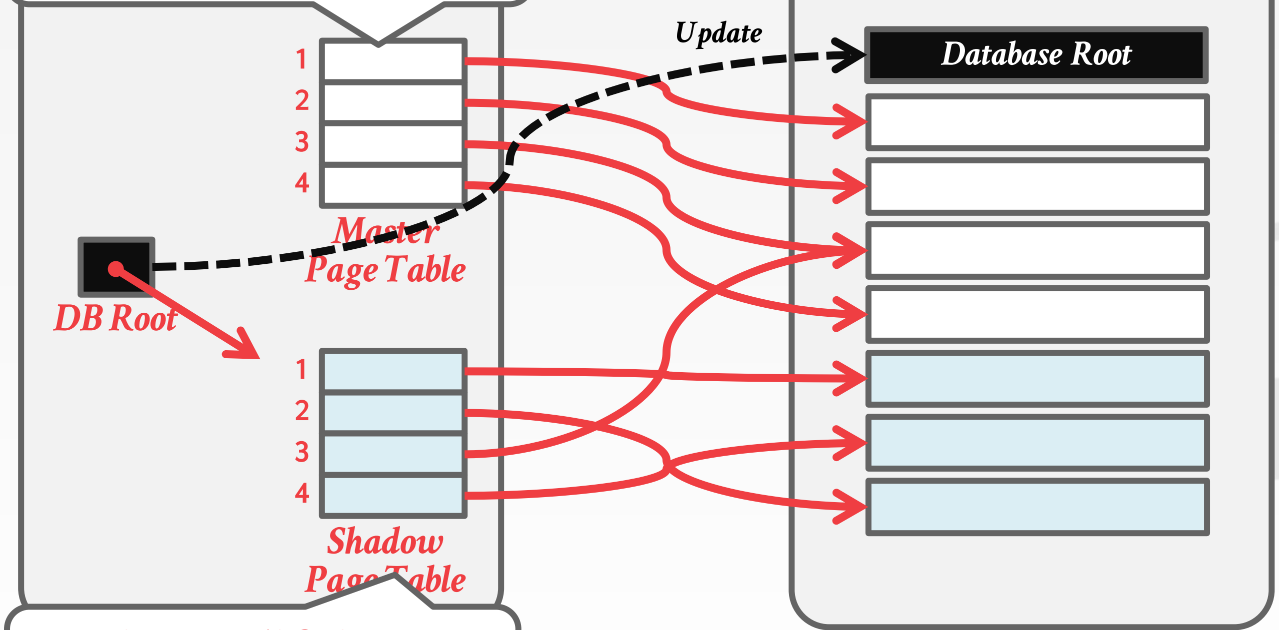
Txn T_1

COMMIT

Active modifying txn updates shadow pages.

SHADOW PAGING – EXAMPLE

Read-only txns access the current master.



Txn T_1

COMMIT

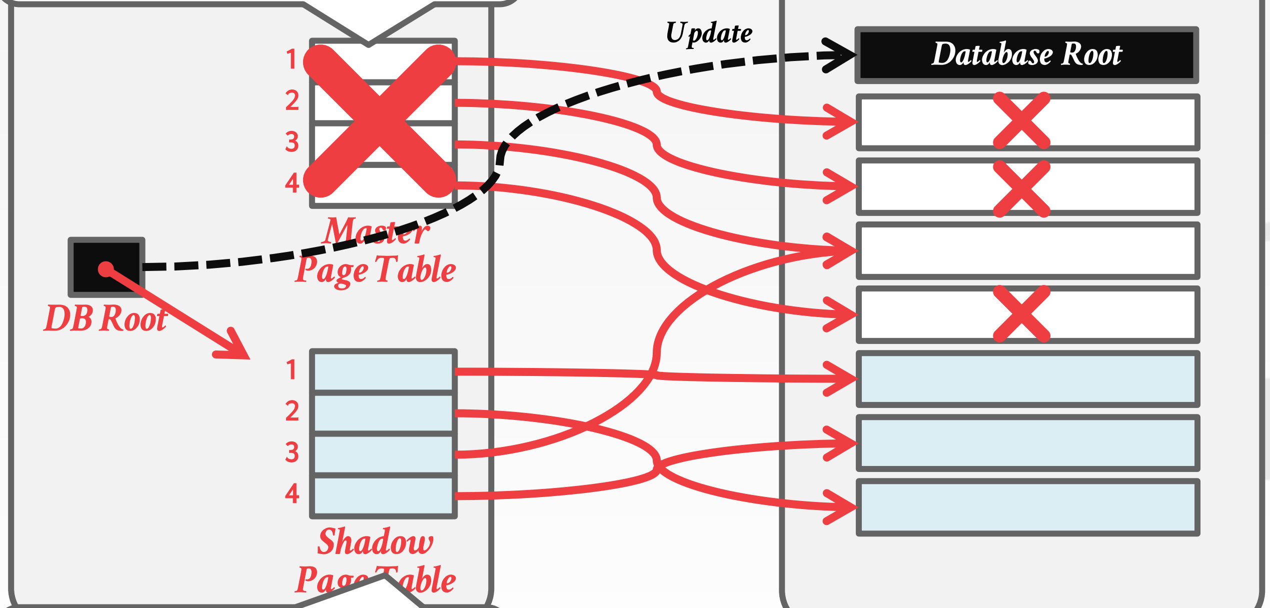
Active modifying txn updates shadow pages.

SHADOW PAGING – EXAMPLE

Read-only txns access the current master.

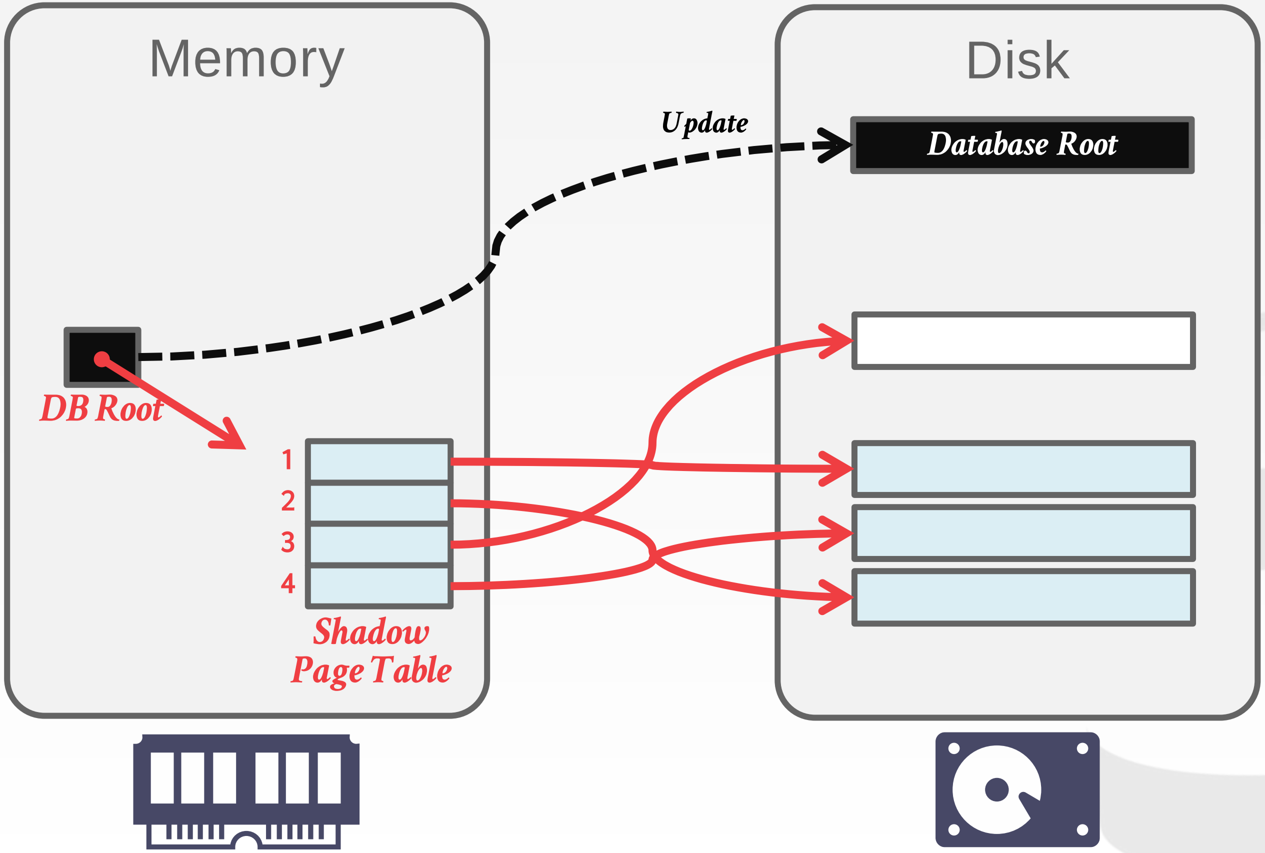
Txn T_1

COMMIT



Active modifying txn updates shadow pages.

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

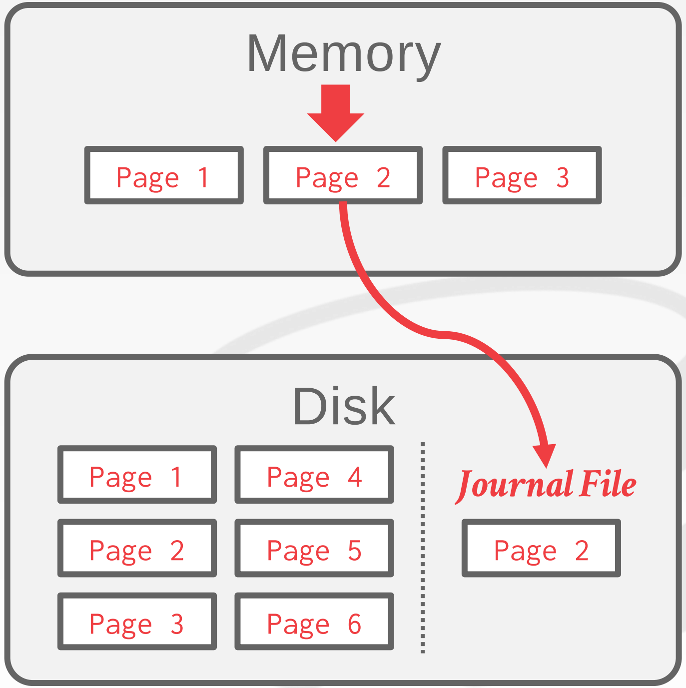
Copying the entire page table is expensive:
→ Use a page table structured like a B+tree.
→ No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes.

Commit overhead is high:
→ Flush every updated page, page table, and root.
→ Data gets fragmented.
→ 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.

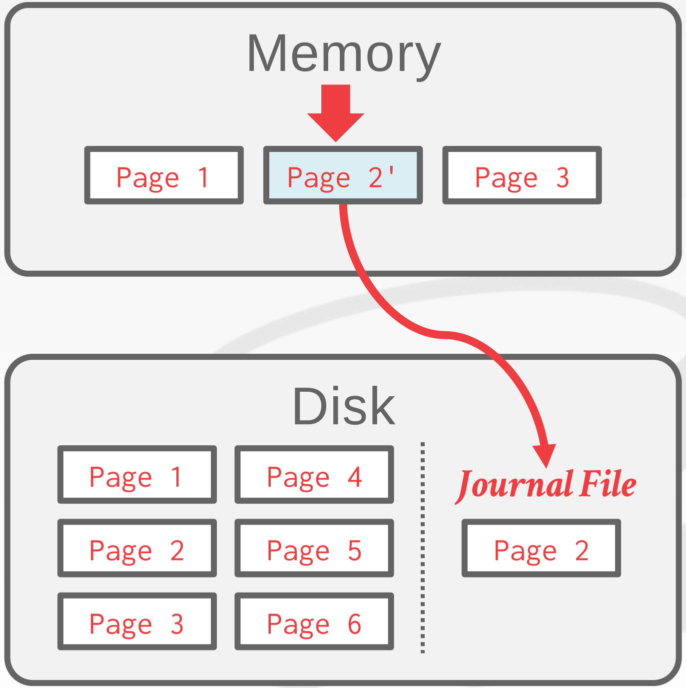
After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



SQLITE (PRE-2010)

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

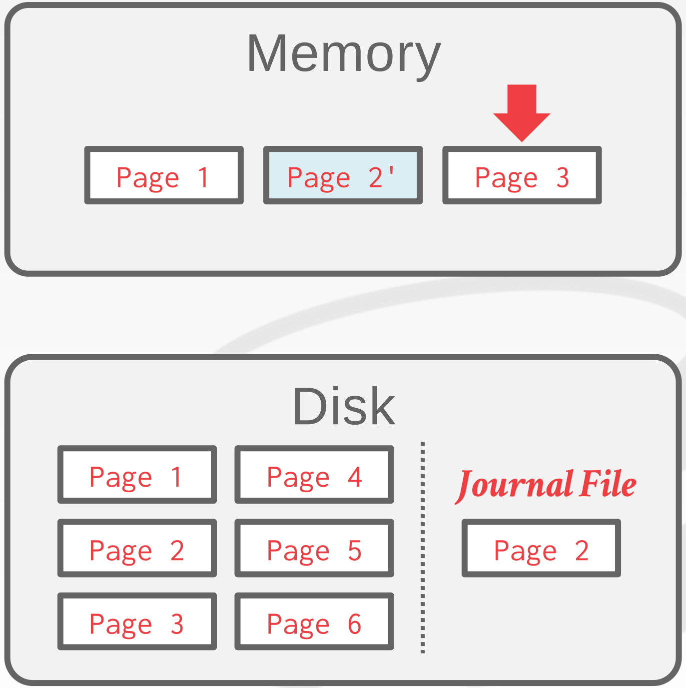
After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



SQLITE (PRE-2010)

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

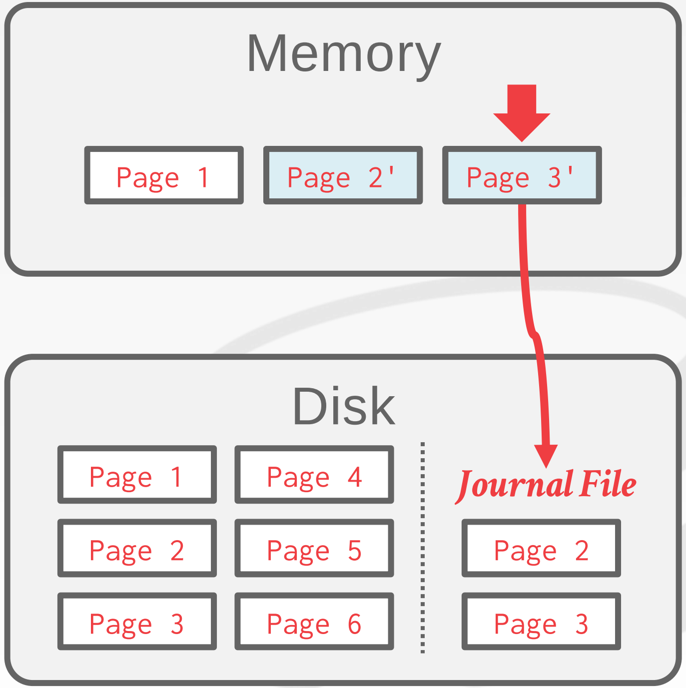
After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



SQLITE (PRE-2010)

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

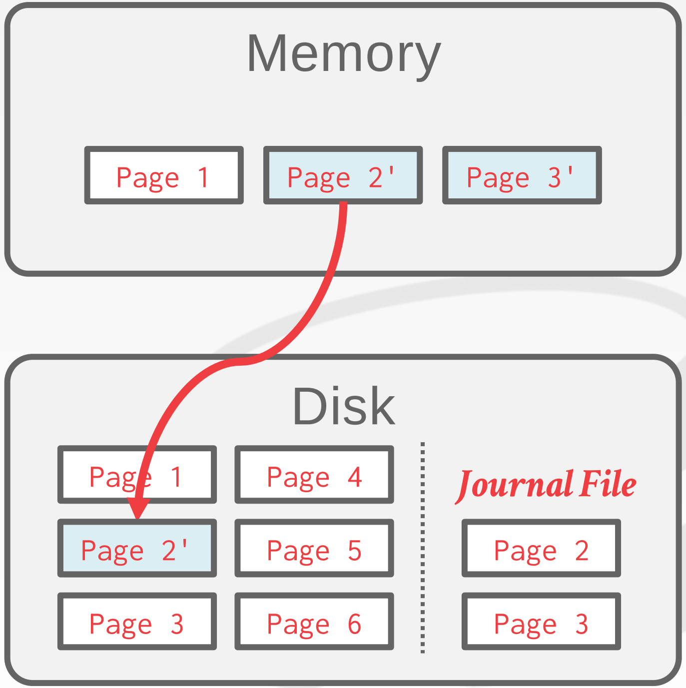
After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



SQLITE (PRE-2010)

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

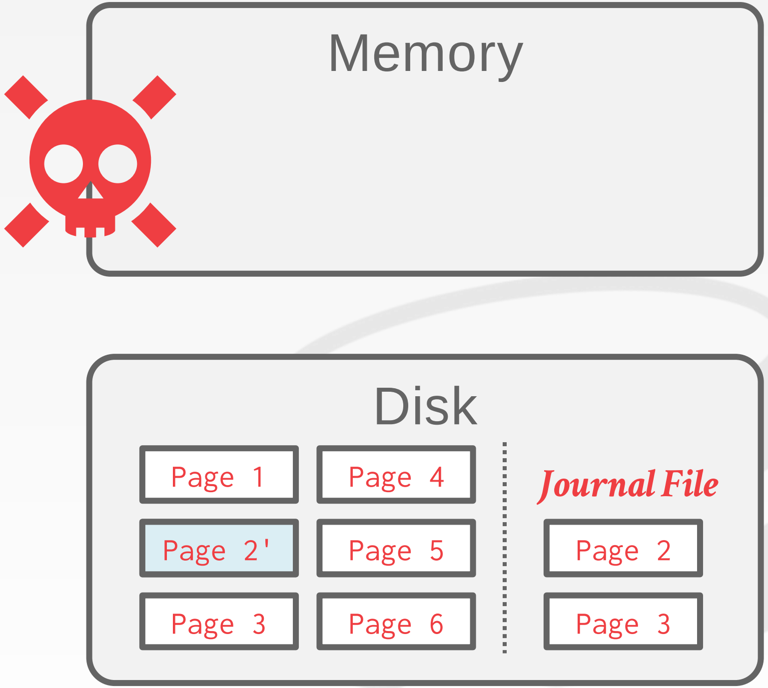
After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



SQLITE (PRE-2010)

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

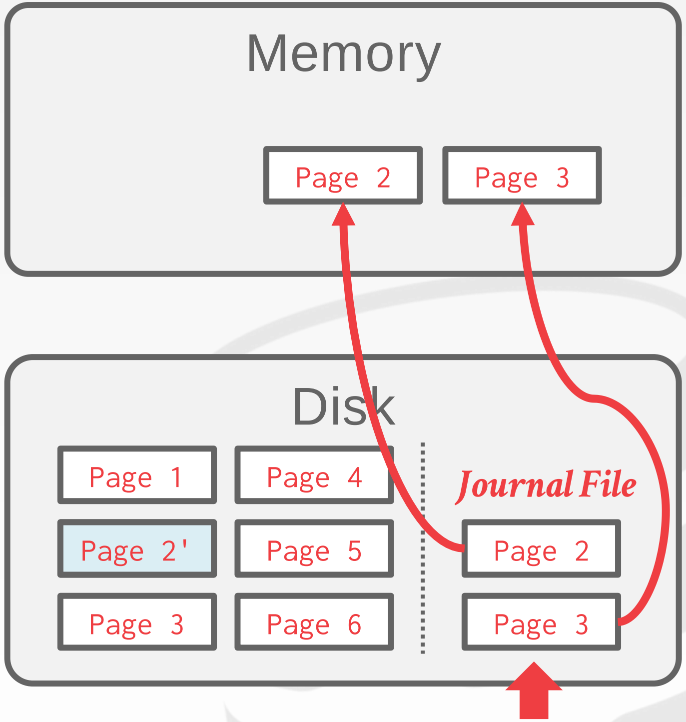
After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



SQLITE (PRE-2010)

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

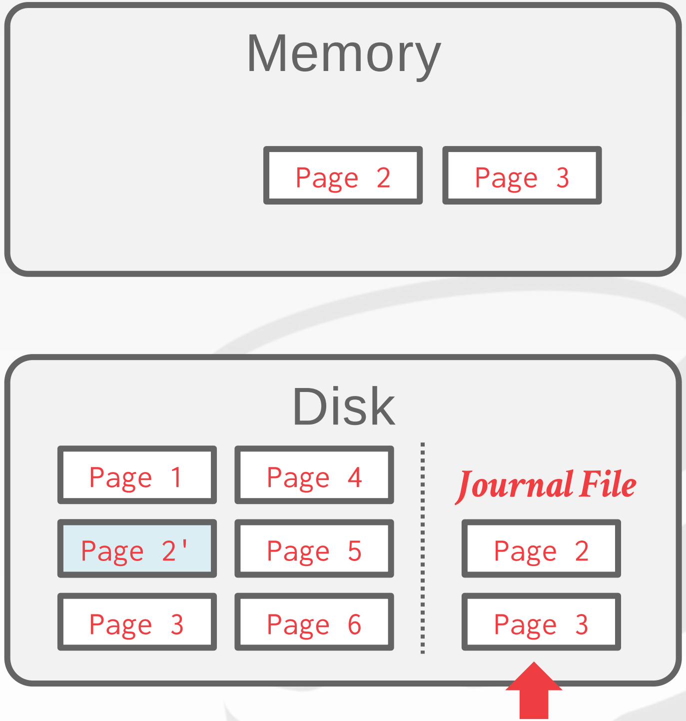
After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



SQLITE (PRE-2010)

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

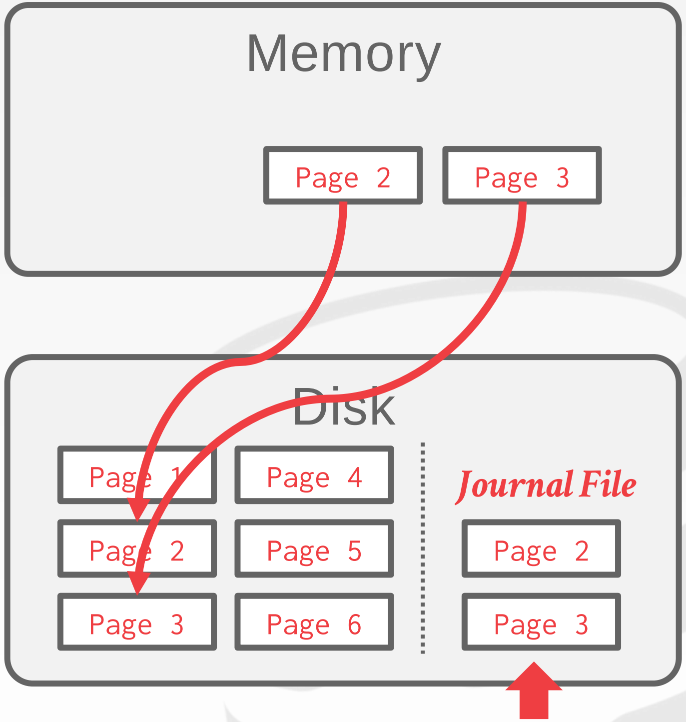
After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.



SQLITE (PRE-2010)

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

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 <u>before</u> the page itself is over-written in non-volatile storage.

A txn is not considered committed until <u>all</u> 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

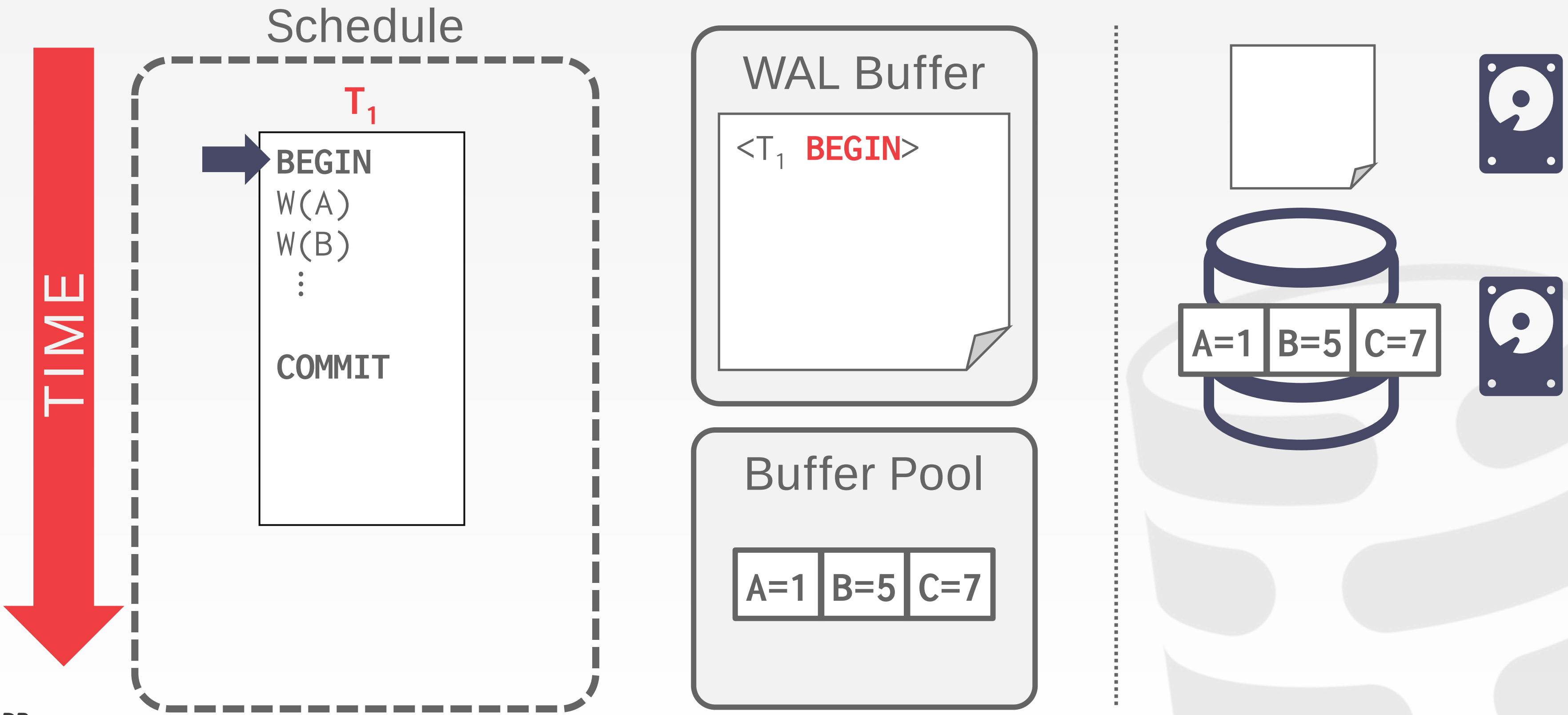
→ Make sure that all log records are flushed before it returns an acknowledgement to application.

WAL PROTOCOL

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

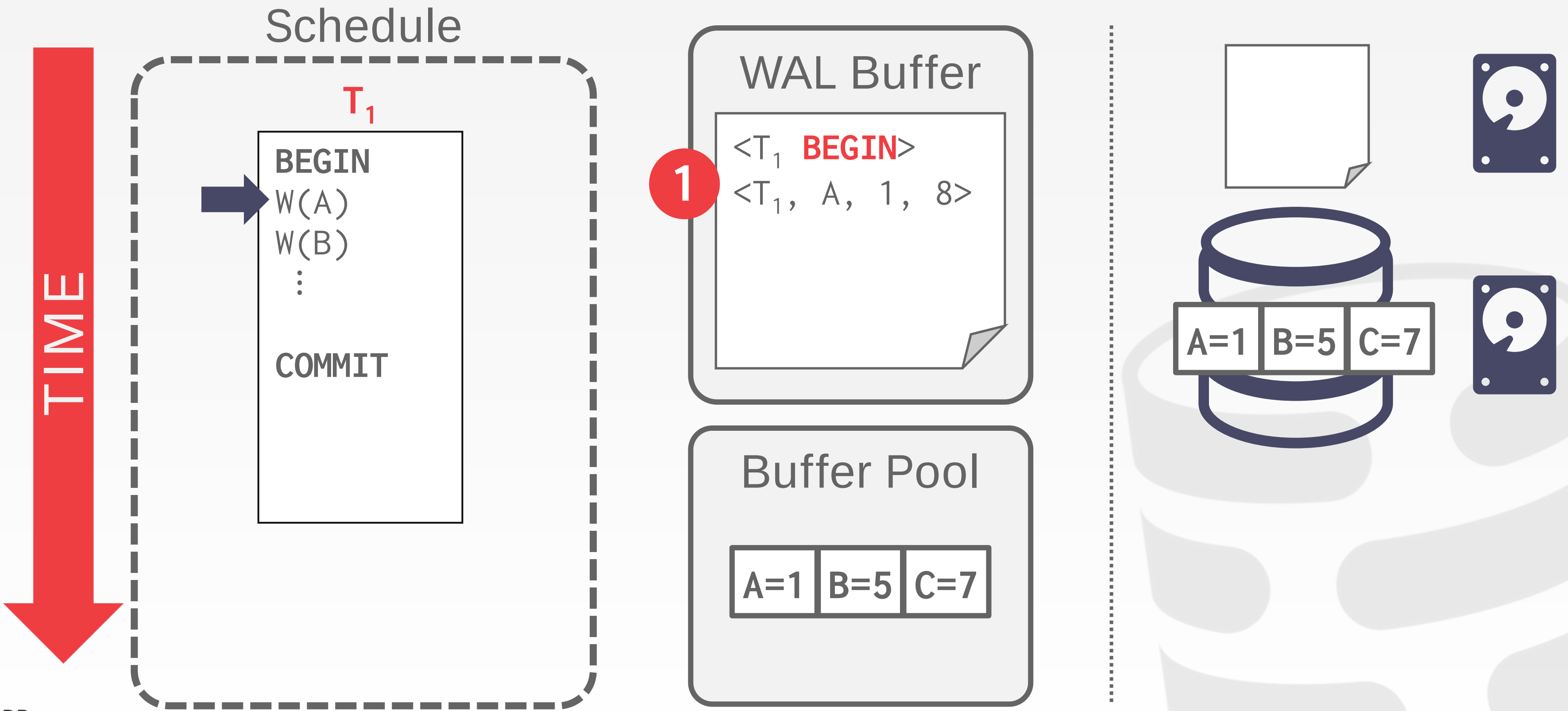
→ Transaction Id
→ Object Id
→ Before Value (UNDO)
→ After Value (REDO)

WAL – EXAMPLE



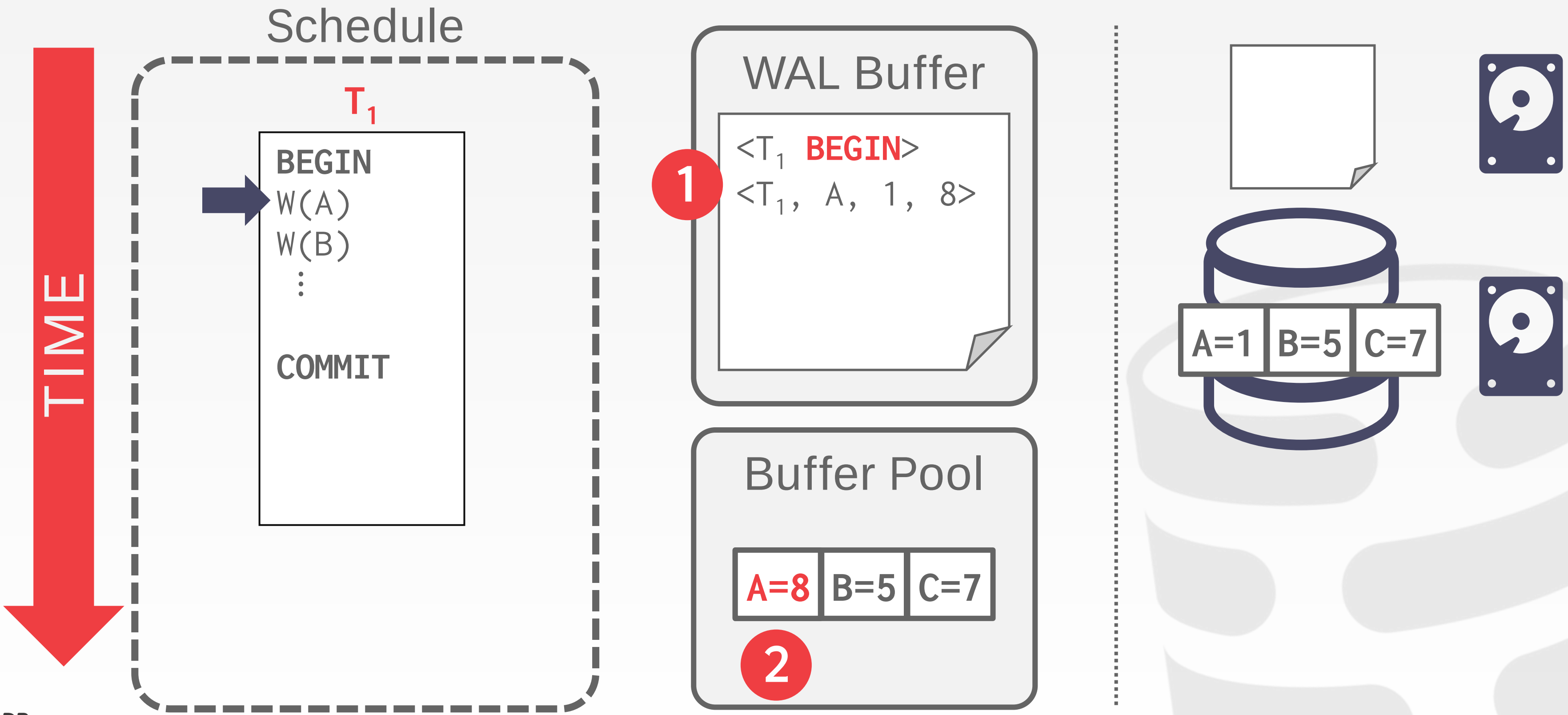
Schedule
TIME
T1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL Buffer
<T1 BEGIN>
Buffer Pool
A=1 B=5 C=7
A=1 B=5 C=7

WAL – EXAMPLE



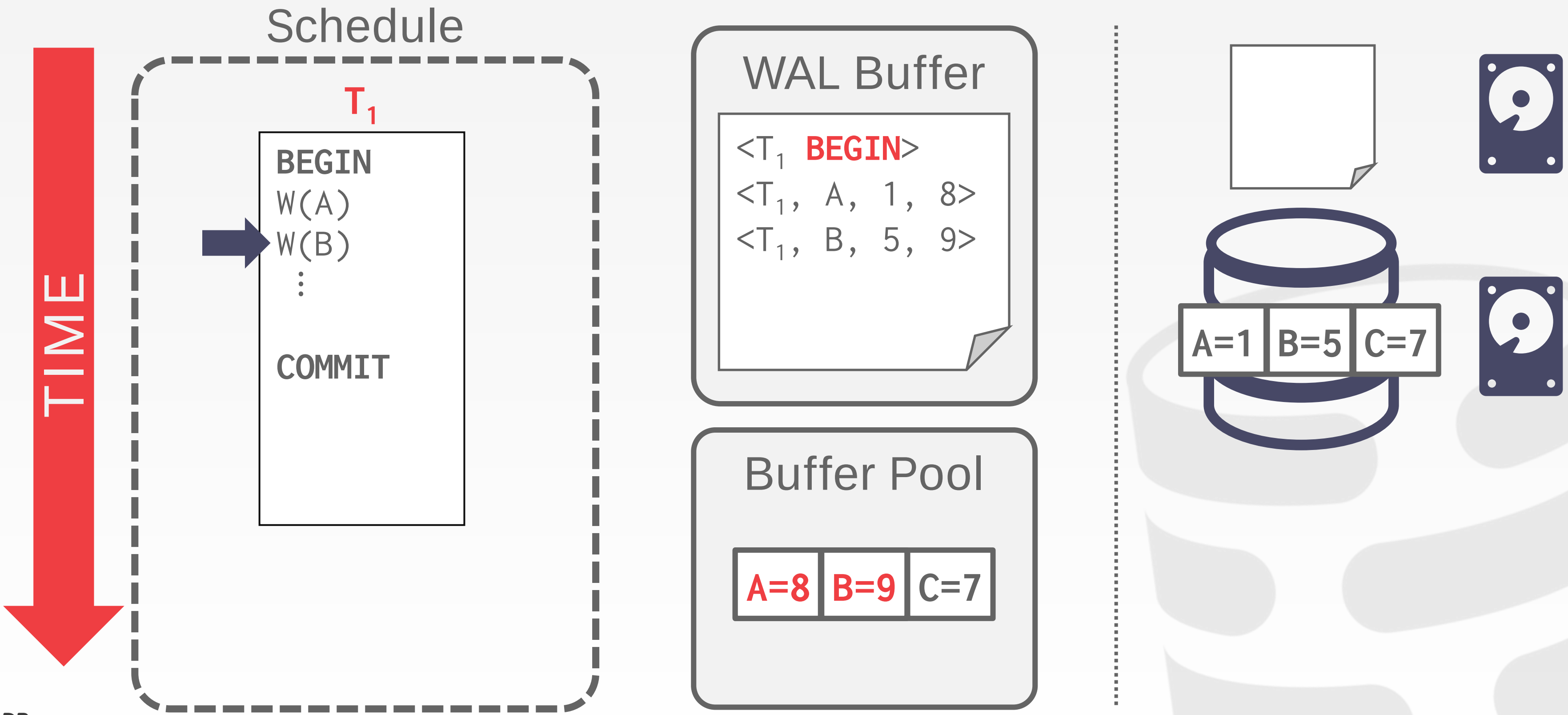
Schedule
TIME
T1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL Buffer
1
<T1 BEGIN>
<T1, A, 1, 8>
Buffer Pool
A=1
B=5
C=7
A=1
B=5
C=7

WAL – EXAMPLE



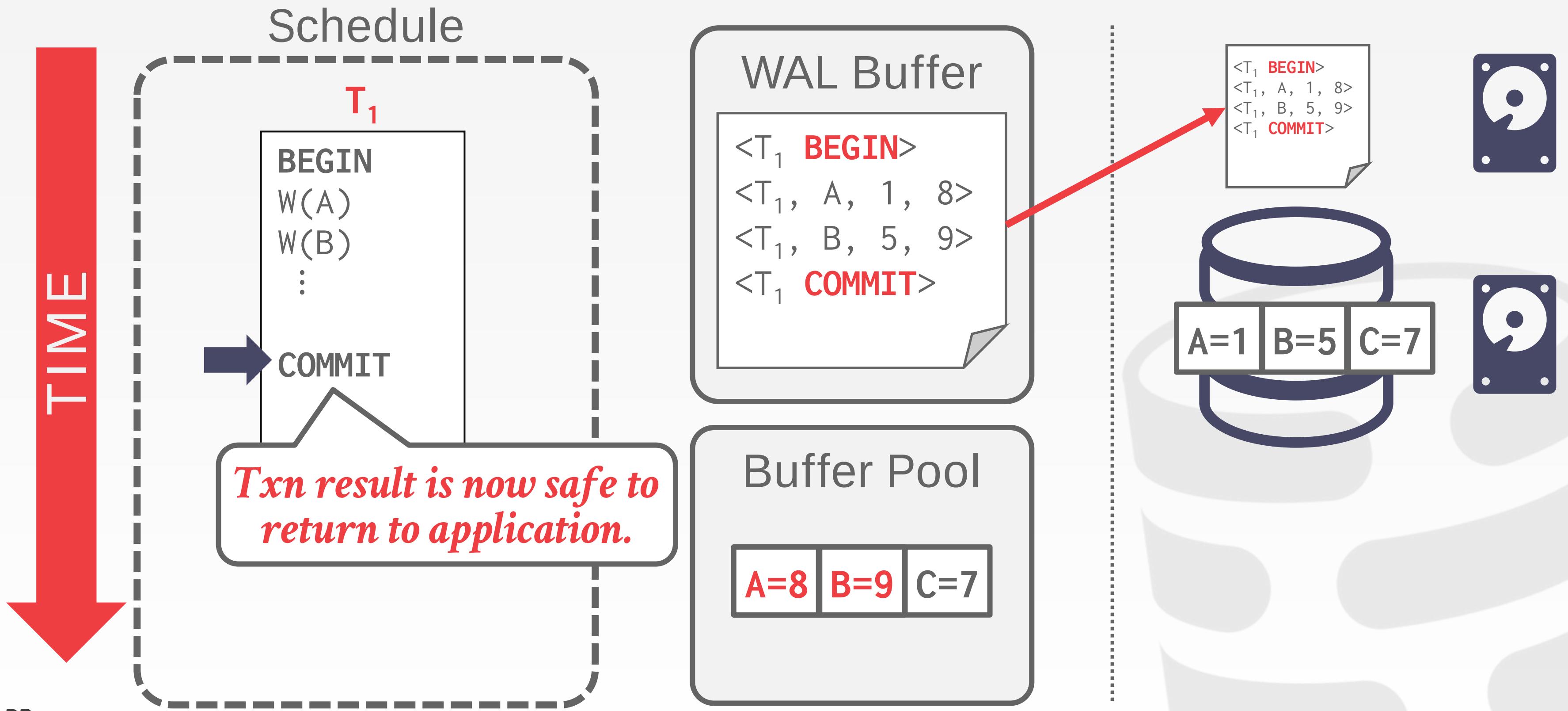
Schedule
TIME
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
WAL Buffer
1
<T_1 BEGIN>
<T_1, A, 1, 8>
Buffer Pool
A=8
B=5
C=7
2
A=1
B=5
C=7

WAL – EXAMPLE



Schedule
TIME
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
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

Schedule

TIME

T_1

```
BEGIN
W(A)
W(B)
  ⋮
COMMIT
```

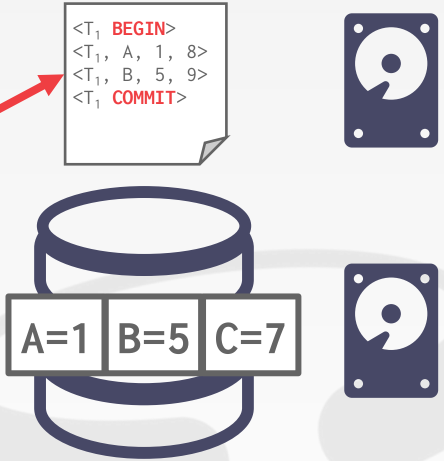
Txn result is now safe to return to application.

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



WAL – EX

Everything we need to restore T_1 is in the log!

Schedule

TIME

T_1

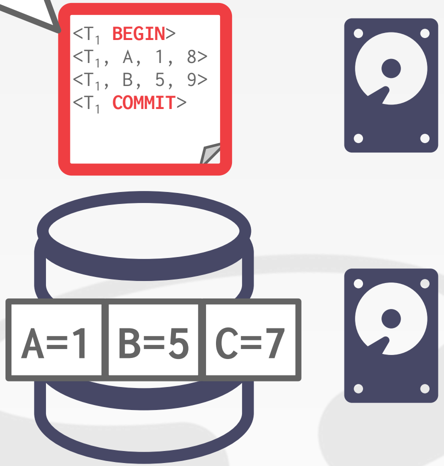
```
BEGIN
W(A)
W(B)
  ⋮

COMMIT
```

Txn result is now safe to return to application.

WAL Buffer

Buffer Pool



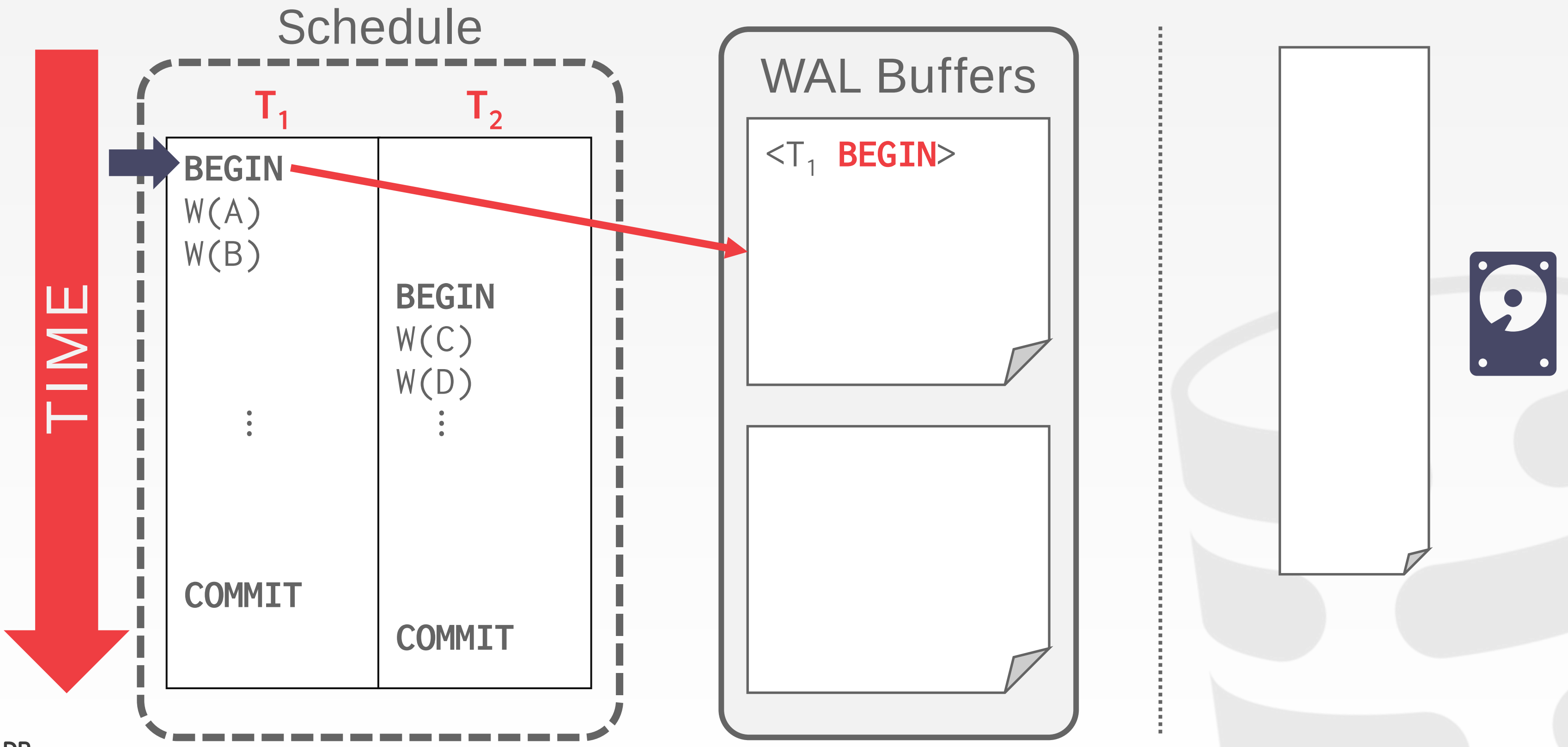
WAL – IMPLEMENTATION

When should the DBMS write log entries to disk?

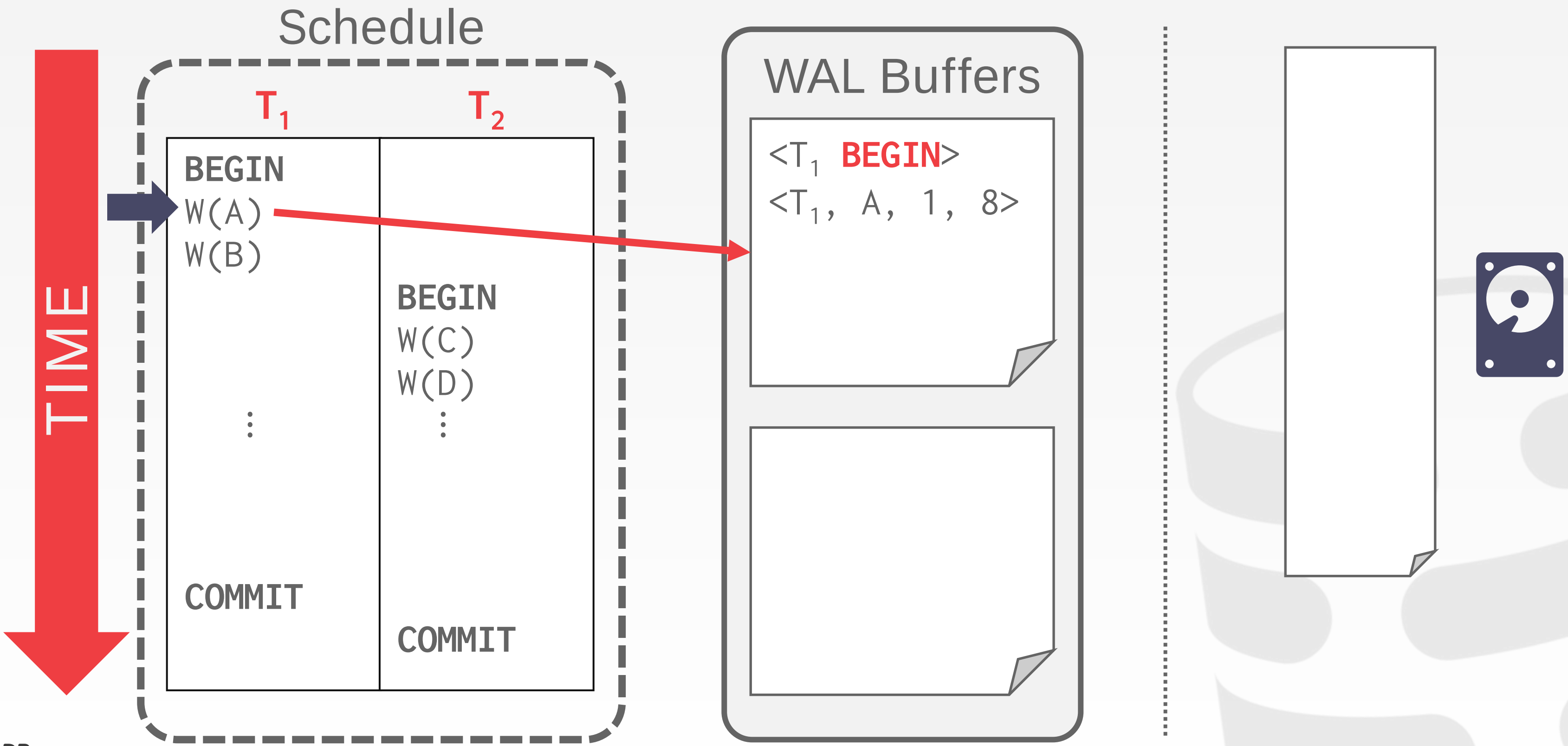
→ When the transaction commits.

→ Can use group commit to batch multiple log flushes together to amortize overhead.

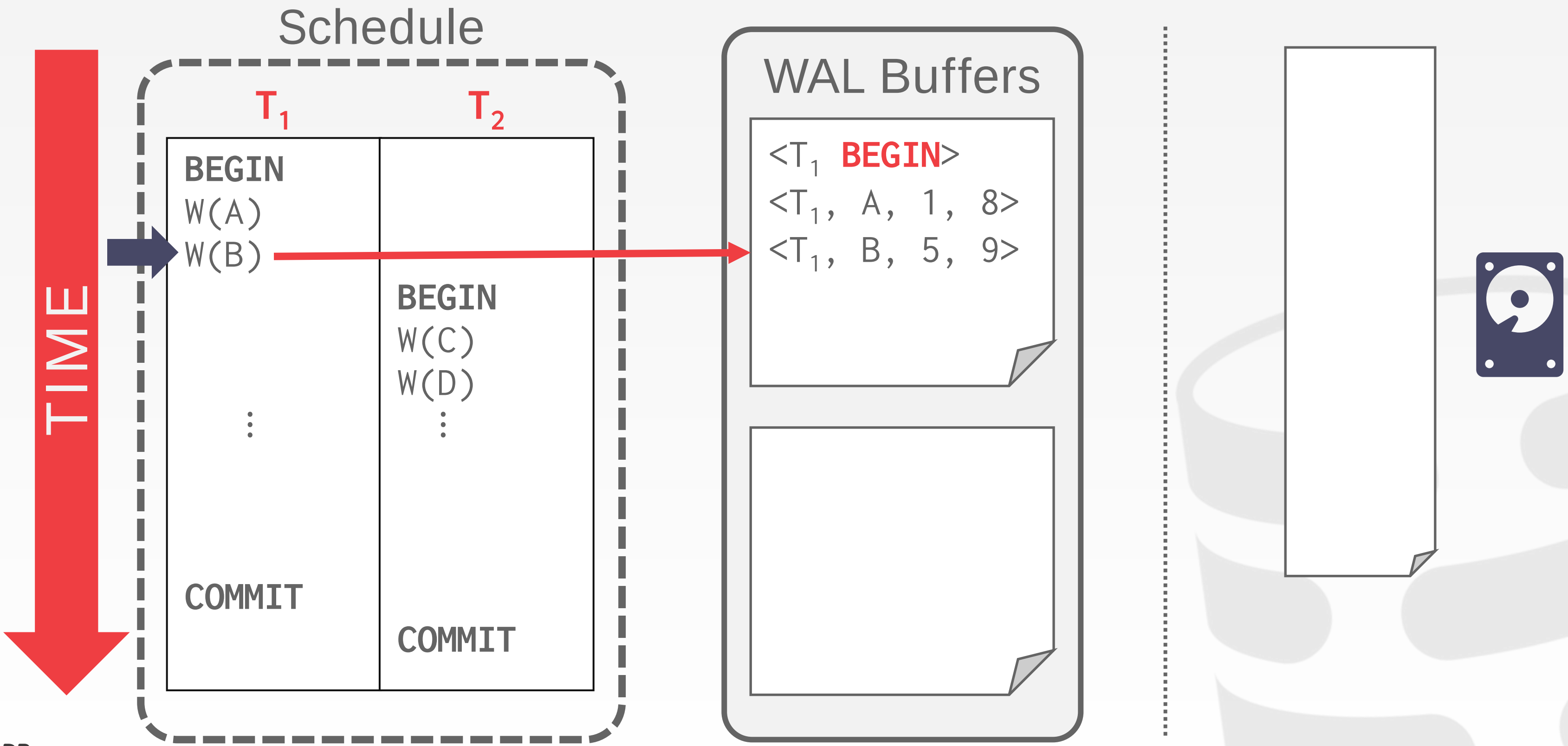
WAL – GROUP COMMIT



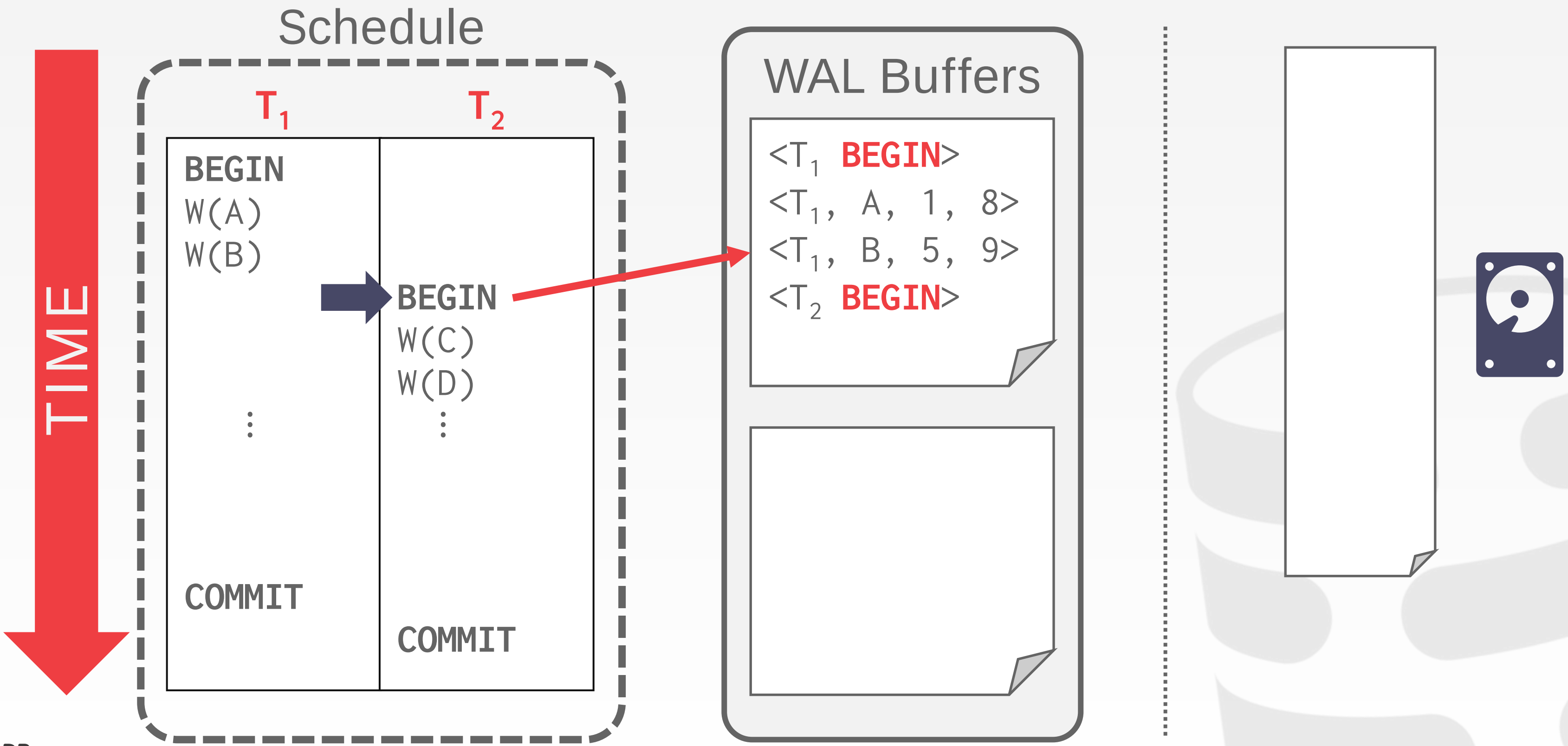
WAL – GROUP COMMIT



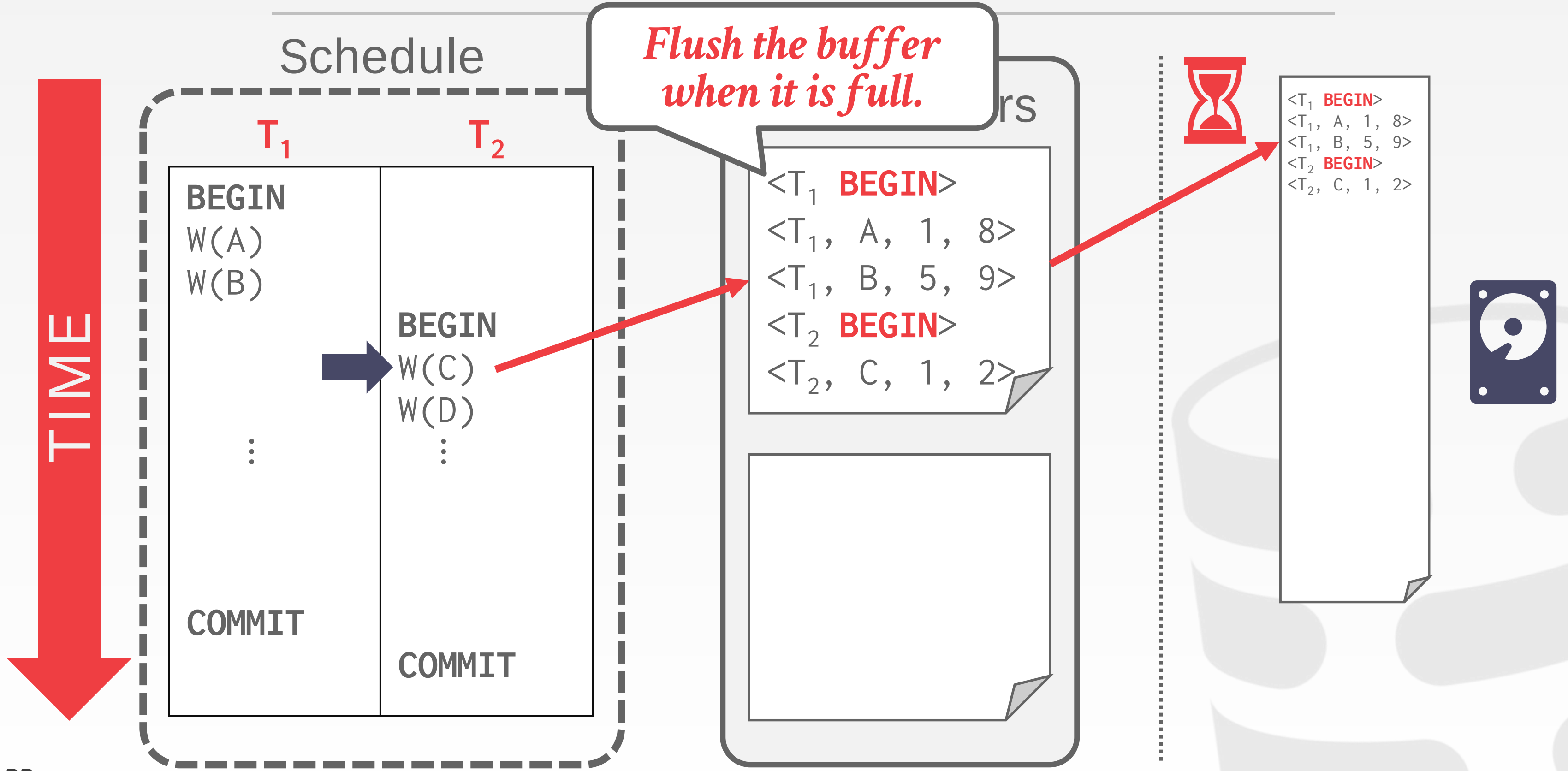
WAL – GROUP COMMIT



WAL – GROUP COMMIT



WAL – GROUP COMMIT



WAL – GROUP COMMIT

Schedule

TIME

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

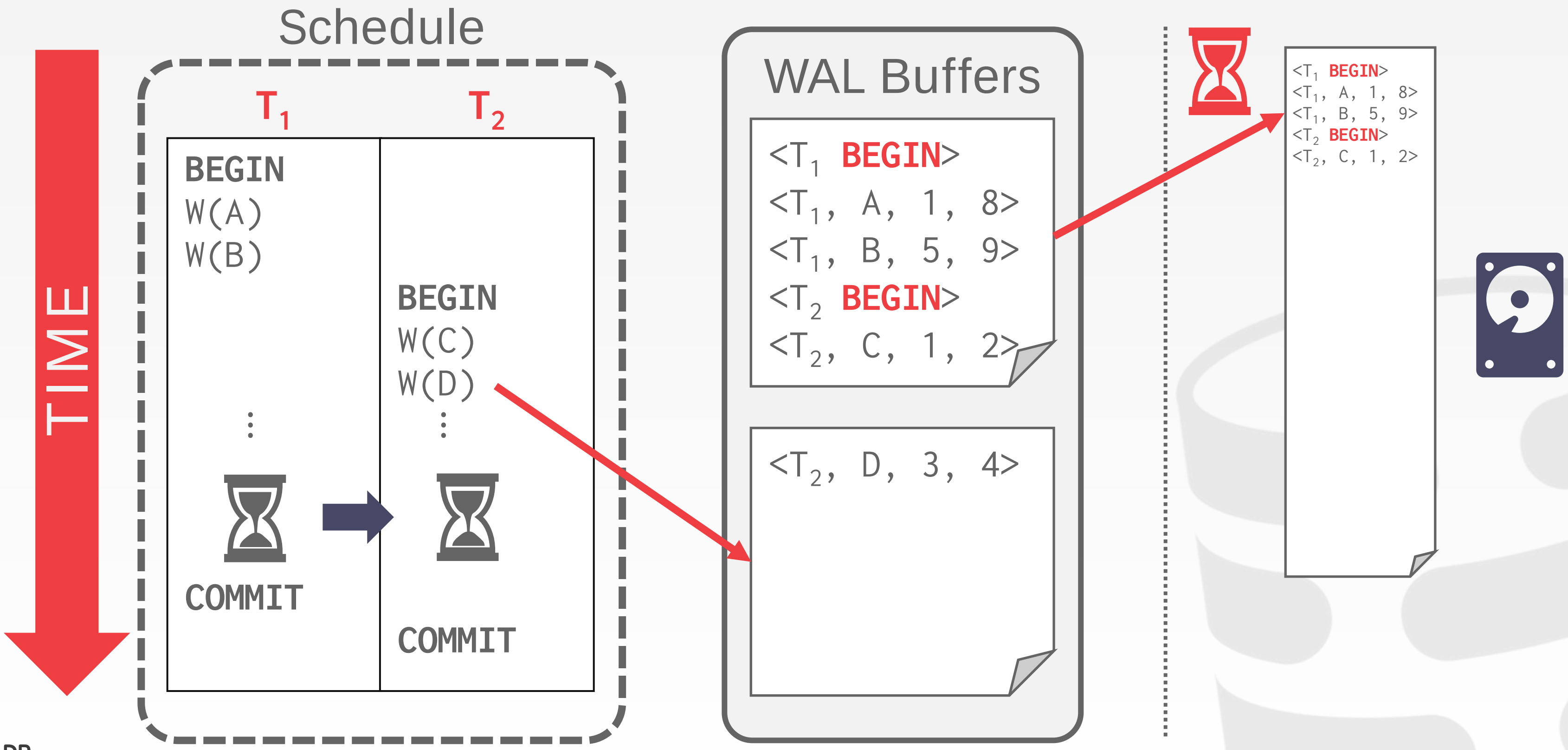
WAL Buffers

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

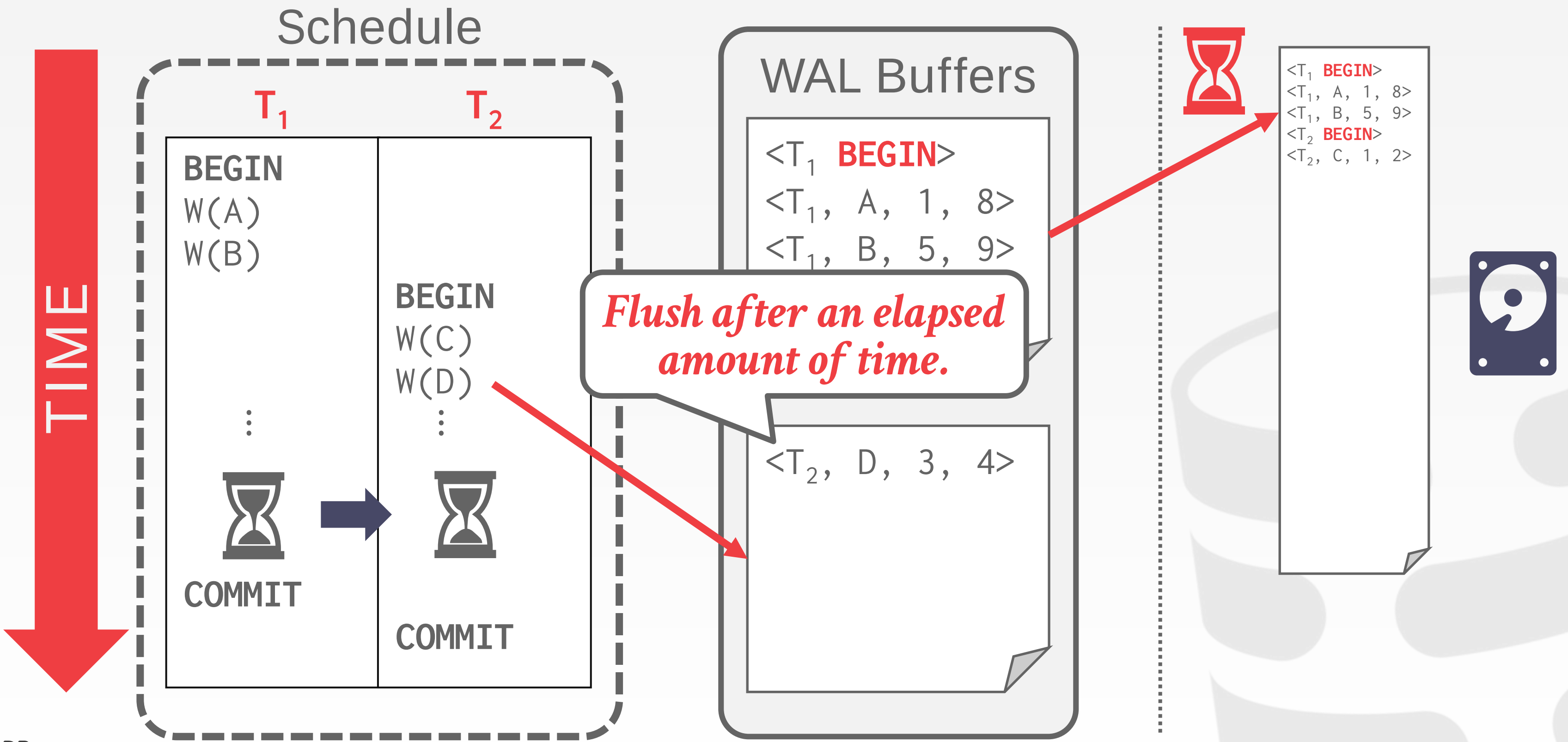
<T_2, D, 3, 4>

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

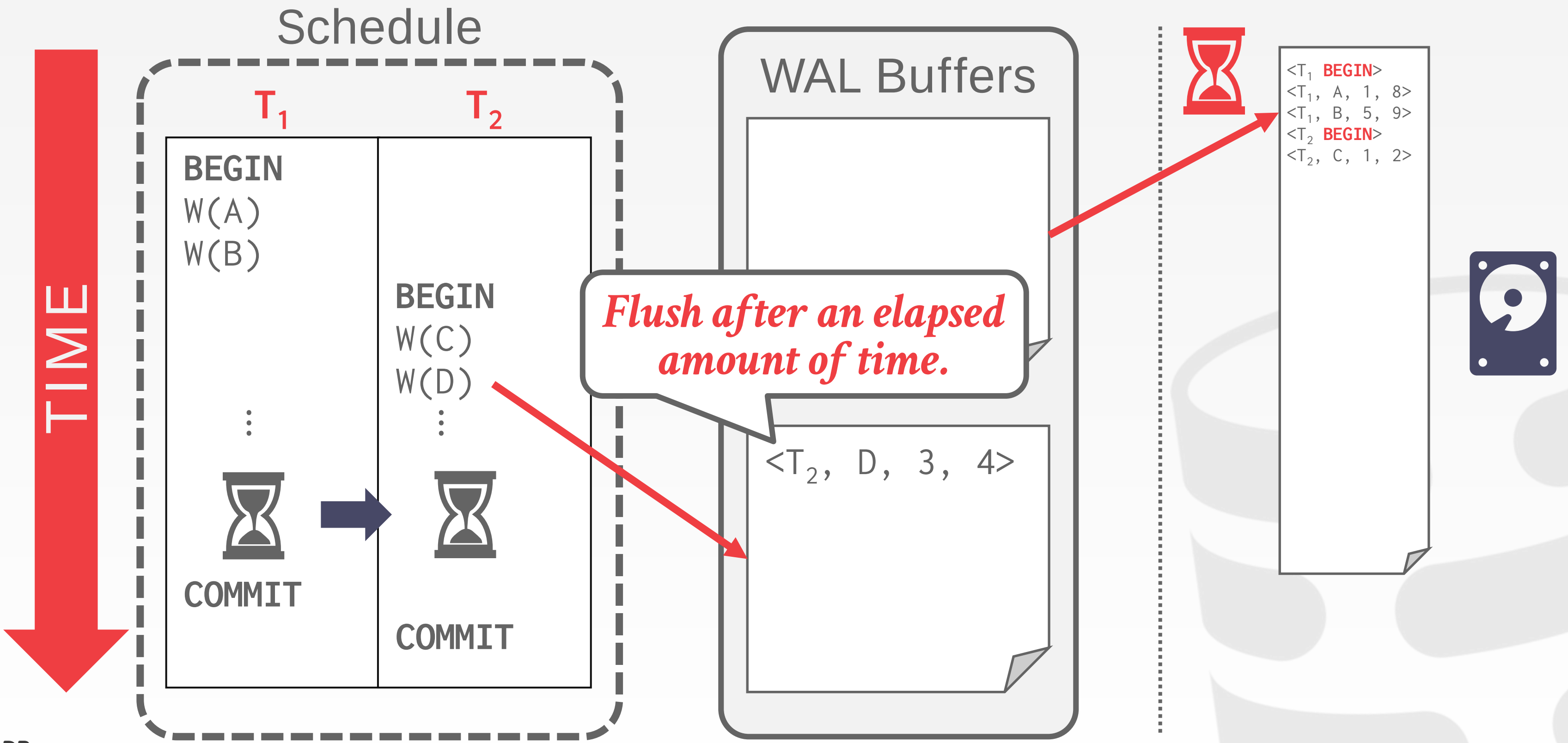
WAL – GROUP COMMIT



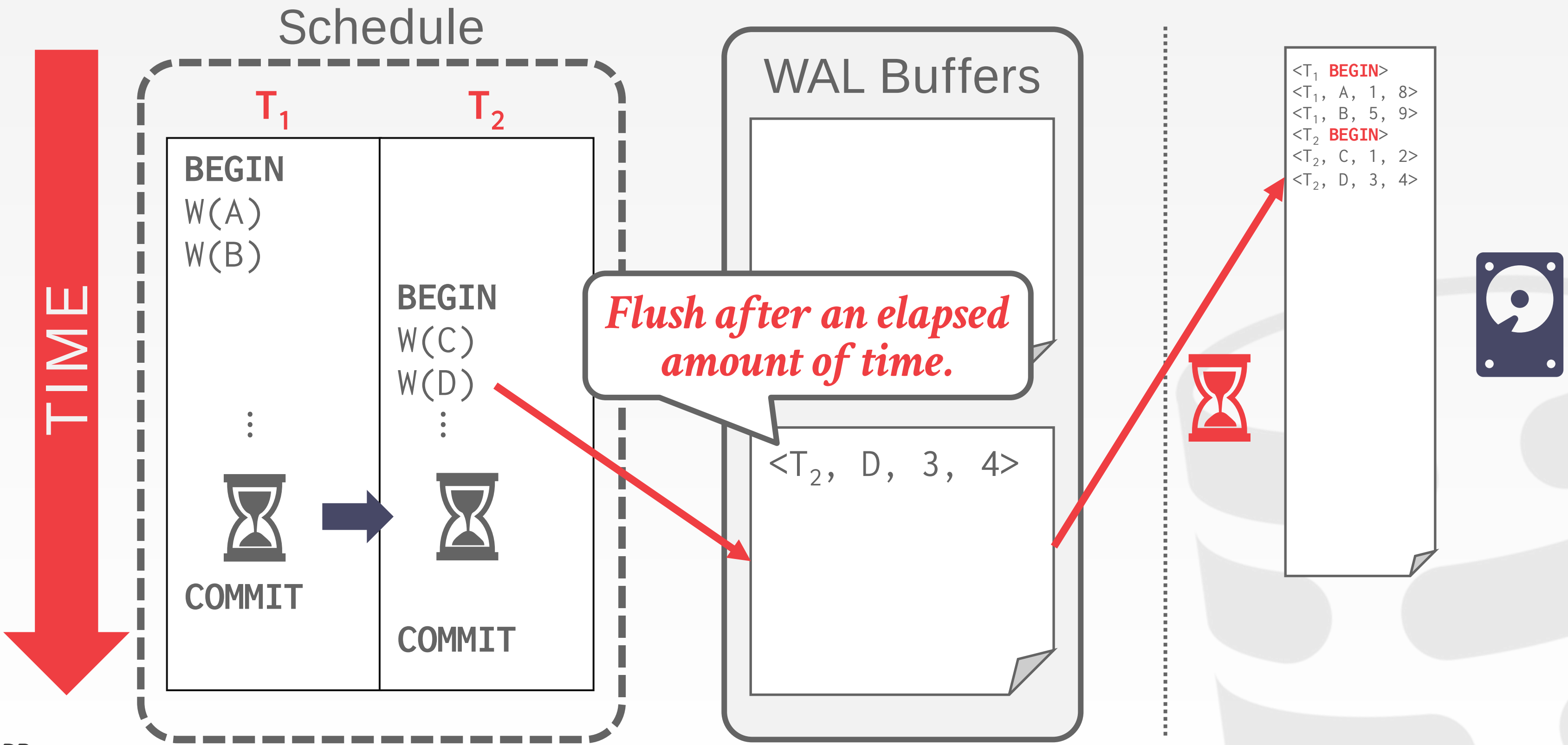
WAL – GROUP COMMIT



WAL – GROUP COMMIT



WAL – GROUP COMMIT



WAL – IMPLEMENTATION

When should the DBMS write log entries to disk?

→ When the transaction commits.

→ Can use group commit to batch multiple log flushes together to amortize overhead.

When should the DBMS write dirty records to disk?

→ Every time the txn executes an update?

→ Once when the txn commits?

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	–

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 changes made to a specific location in the database.

→ Example: `git diff`

Logical Logging

→ Record the high-level operations executed by txns.

→ Not necessarily restricted to single page.

→ Example: The `UPDATE`, `DELETE`, and `INSERT` queries invoked by a txn.

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.

→ 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.

PHYSIOLOGICAL LOGGING

Hybrid approach where log records target a single page but do not specify organization of the page.

→ Identify tuples based on their slot number.

→ Allows DBMS to reorganize pages after a log record has been written to disk.

This is the most popular approach.

LOGGING SCHEMES

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

Physical

```
<T1,
 Table=X,
 Page=99,
 Offset=4,
 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)>
```

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.

CHECKPOINTS

Output onto stable storage all log records currently residing in main memory.

Output to the disk all modified blocks.

Write a <CHECKPOINT> entry to the log and flush to stable storage.

CHECKPOINTS

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

WAL

```
<T1 BEGIN>
<T1, A, 1, 2>
<T1 COMMIT>
<T2 BEGIN>
<T2, A, 2, 3>
<T3 BEGIN>
<CHECKPOINT>
<T2 COMMIT>
<T3, A, 3, 4>
⋮
CRASH!
```

CHECKPOINTS

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

T_2 + T_3 did not commit before the last checkpoint.

→ Need to redo T_2 because it committed after checkpoint.

→ Need to undo T_3 because it did not commit before the crash.

WAL

```
<T1 BEGIN>
<T1, A, 1, 2>
<T1 COMMIT>
<T2 BEGIN>
<T2, A, 2, 3>
<T3 BEGIN>
<CHECKPOINT>
<T2 COMMIT>
<T3, A, 3, 4>
⋮
CRASH!
```

CHECKPOINTS – CHALLENGES

The DBNS must stall txns when it takes a checkpoint to ensure a consistent snapshot.

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

Not obvious how often the DBMS should take a checkpoint…

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.

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.