Carnegie Mellon University

# Logging Schemes

Lecture #21

Database Systems
15-445/15-645
Fall 2017

**Andy Pavlo**
Computer Science Dept.
Carnegie Mellon Univ.

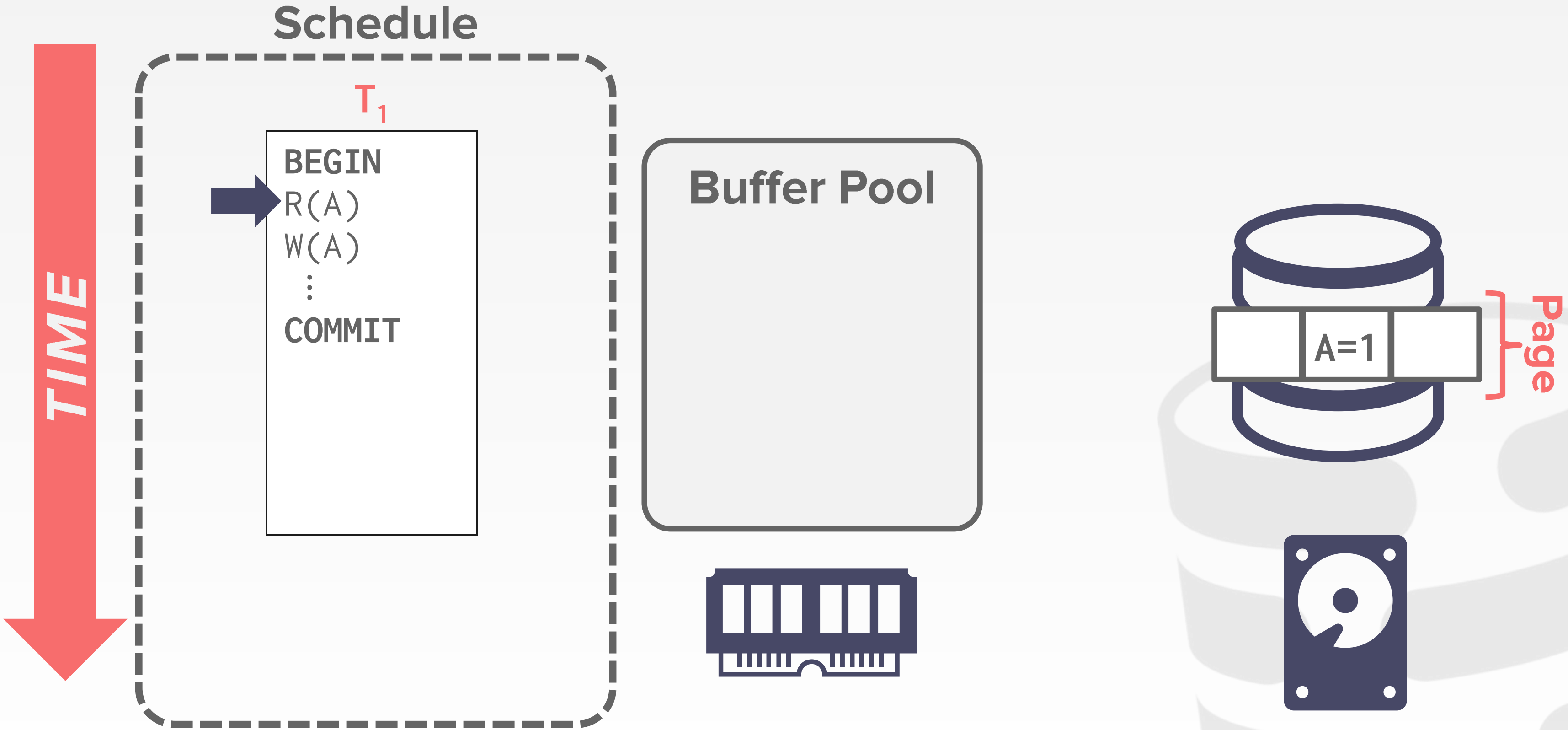
# ADMINISTRIVIA

**Homework #5**: Today @ 11:59pm

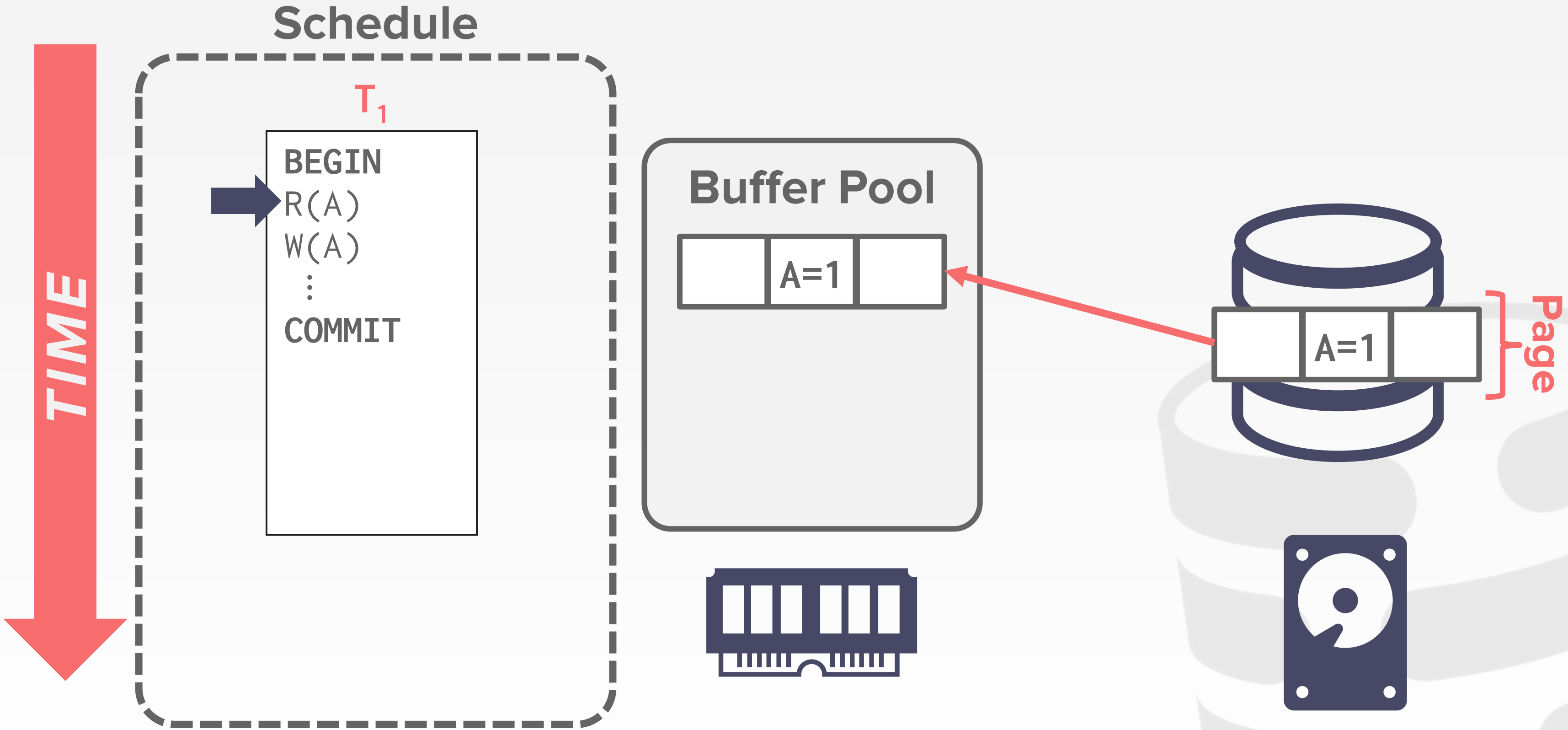
**Project #3**: Wednesday November $15^{th}$ @ 11:59am

**No class on Wednesday November $15^{th}$**

# MOTIVATION



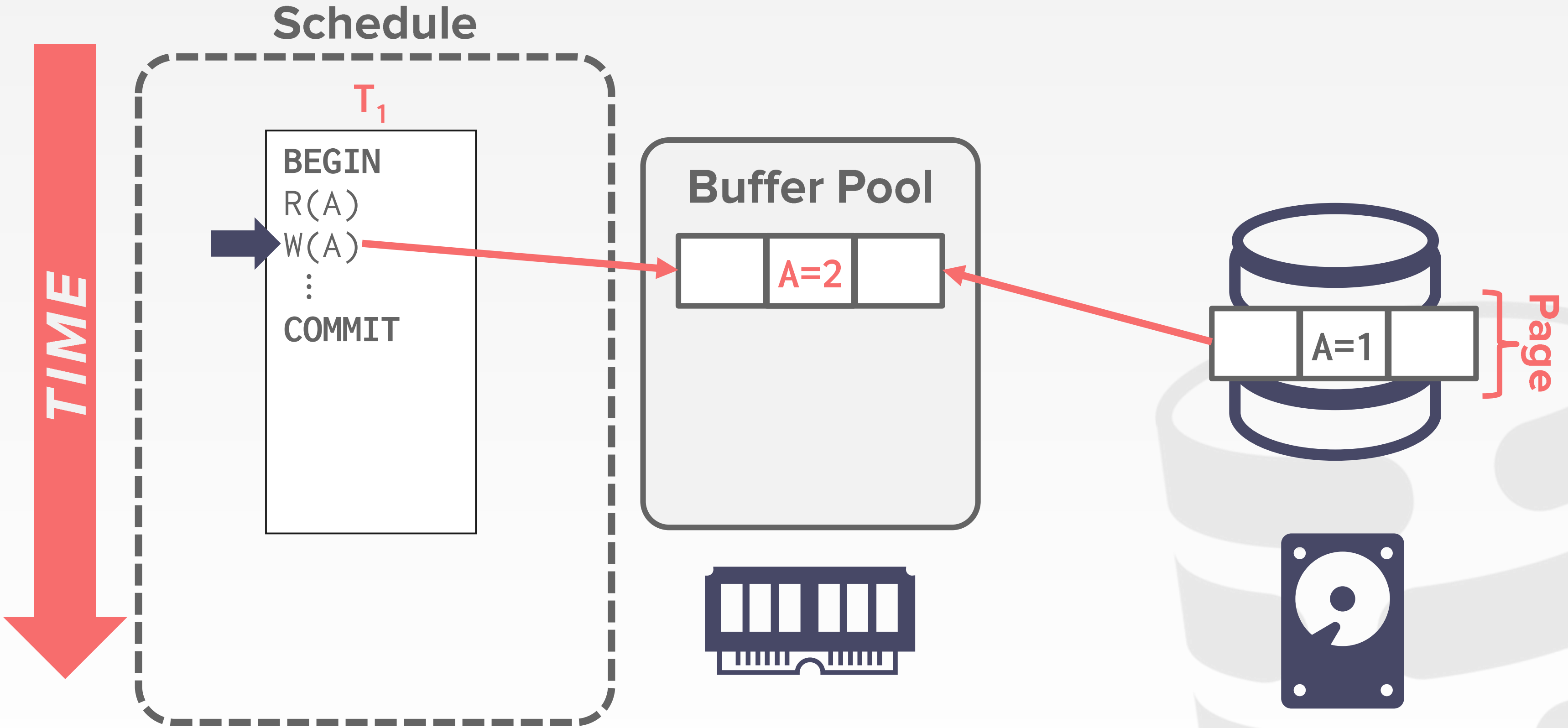
# MOTIVATION






CARNEGIE MELLON
DATABASE GROUP

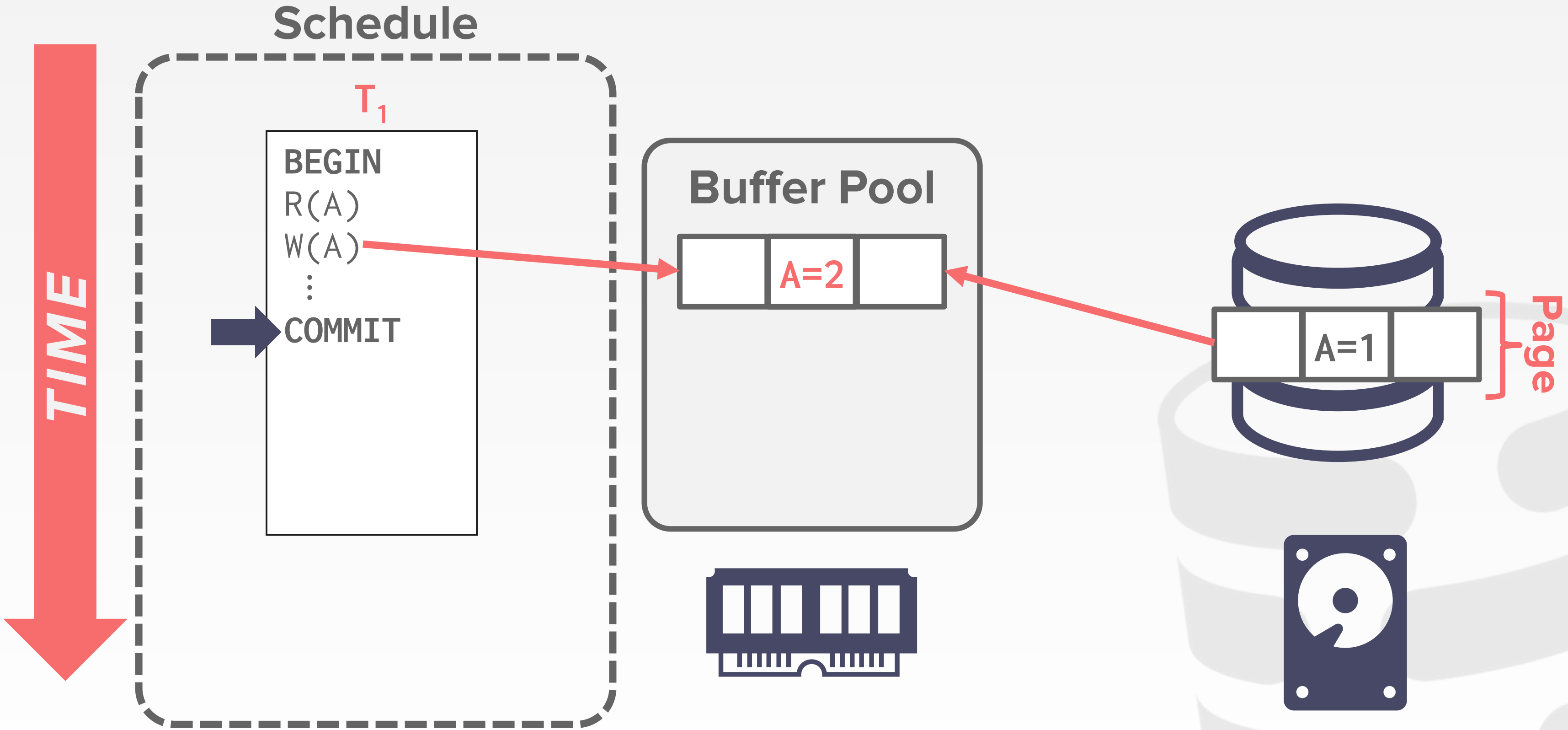
# MOTIVATION






CARNEGIE MELLON DATABASE GROUP

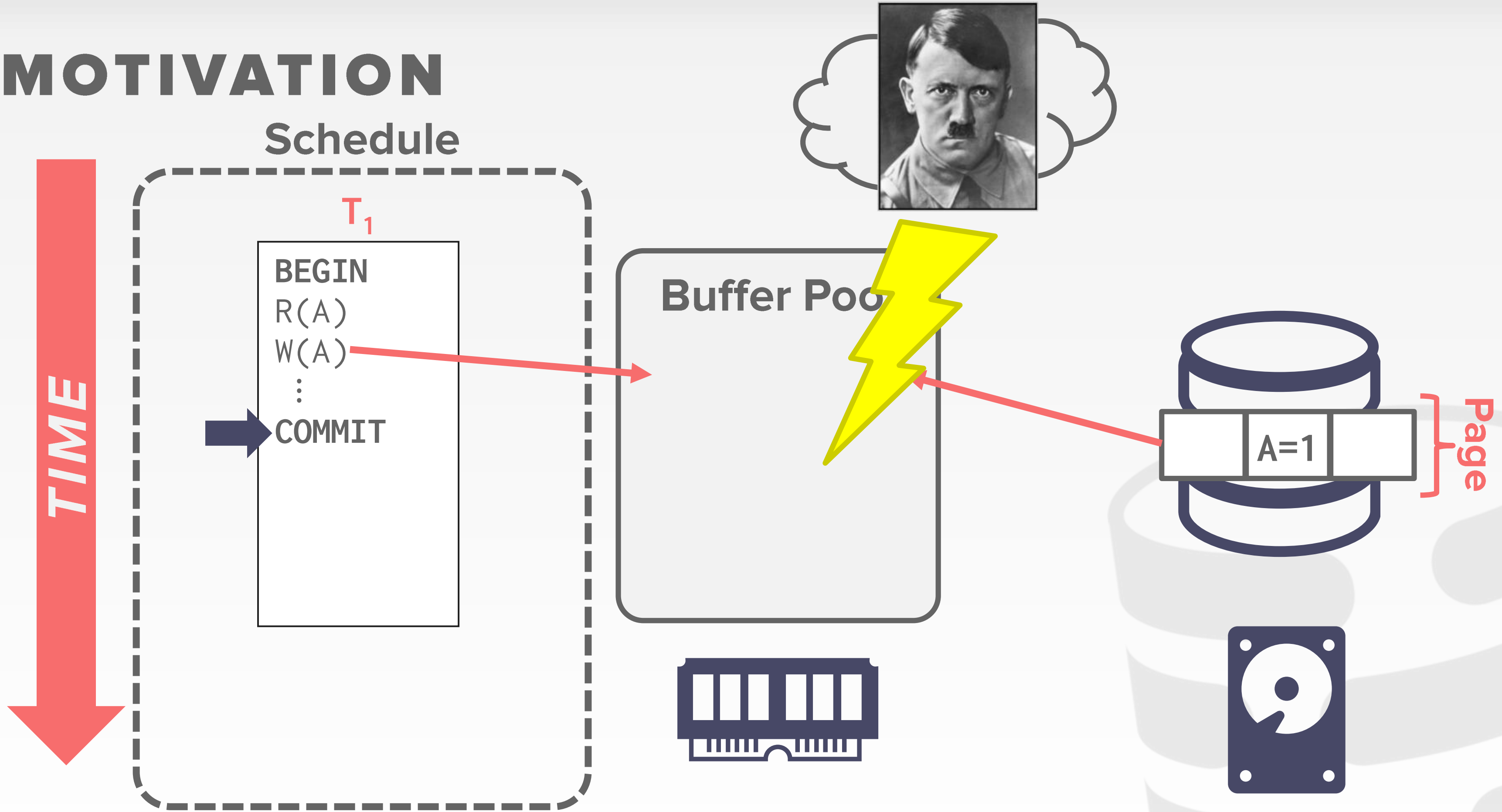
# MOTIVATION





CARNEGIE MELLON DATABASE GROUP

# 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

Checkpoints

Logging Schemes

# CRASH RECOVERY

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

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 is cut.
→ Examples: DRAM, SRAM

## Non-volatile Storage:
→ Data persists after losing power.
→ Examples: HDD, SDD

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

Use multiple storage devices to approximate.

# 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 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 changes are durable if the txn aborted.

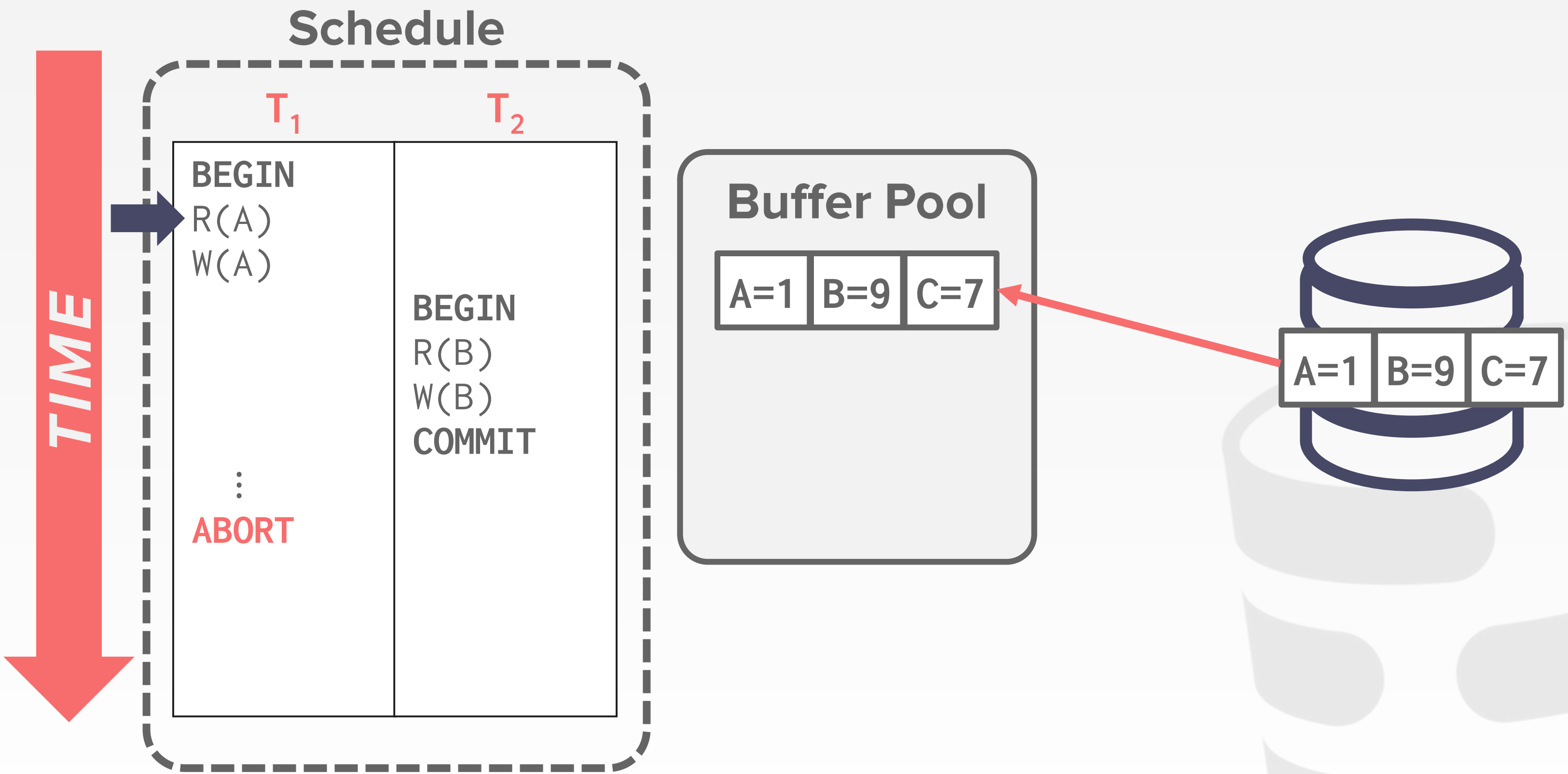
# UNDO VS. REDO

**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.

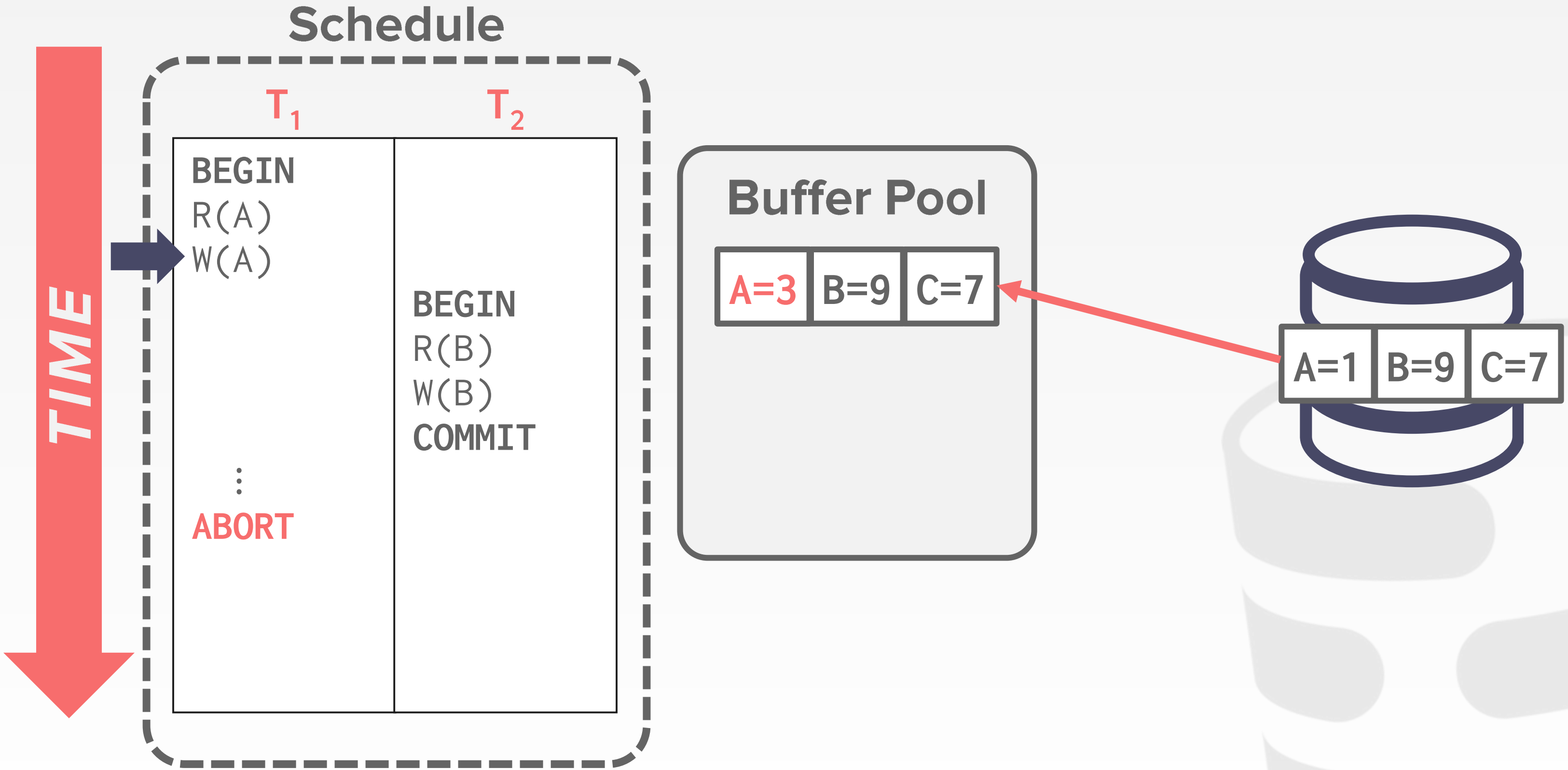
How the DBMS supports this functionality depends on how it manages the buffer pool…

# BUFFER POOL



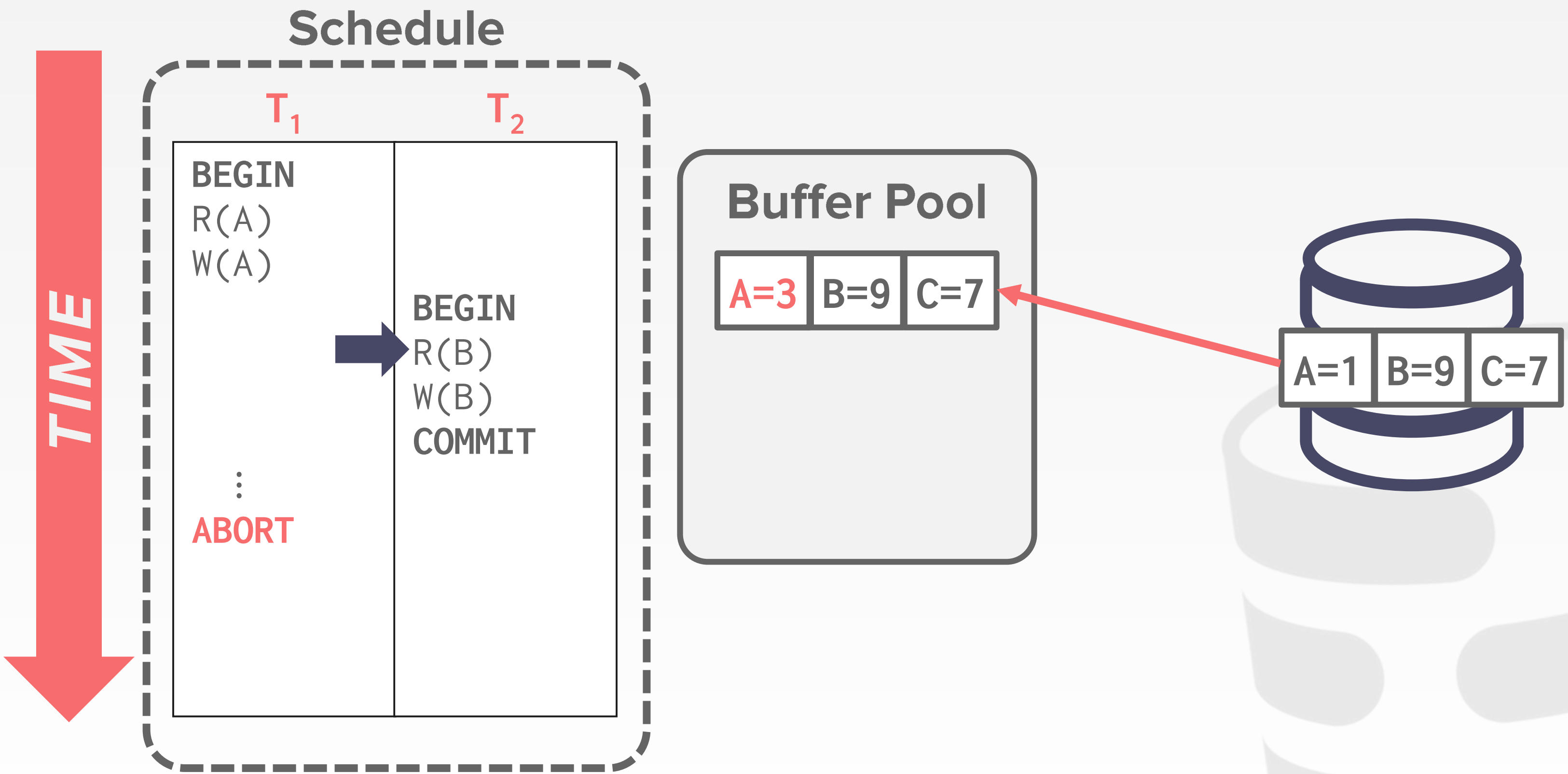
Schedule
TIME
T1
T2
BEGIN
R(A)
W(A)
⋮
ABORT
BEGIN
R(B)
W(B)
COMMIT
Buffer Pool
A=1
B=9
C=7
A=1
B=9
C=7

# BUFFER POOL



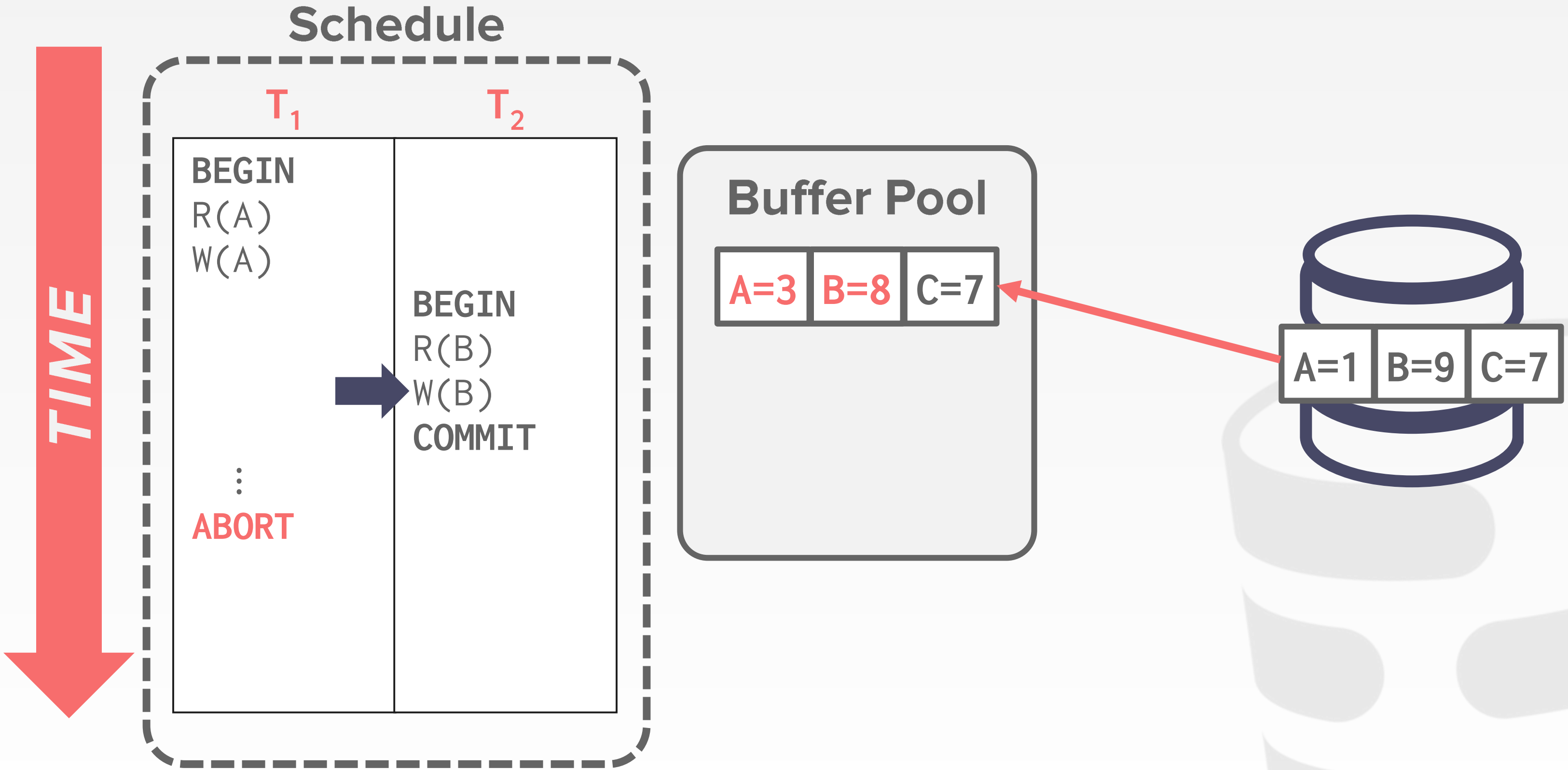
TIME
Schedule
$T_1$
$T_2$
BEGIN
R(A)
W(A)
⋮
ABORT
BEGIN
R(B)
W(B)
COMMIT
Buffer Pool
A=3
B=9
C=7
A=1
B=9
C=7

# BUFFER POOL



TIME
Schedule
$T_1$
$T_2$
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

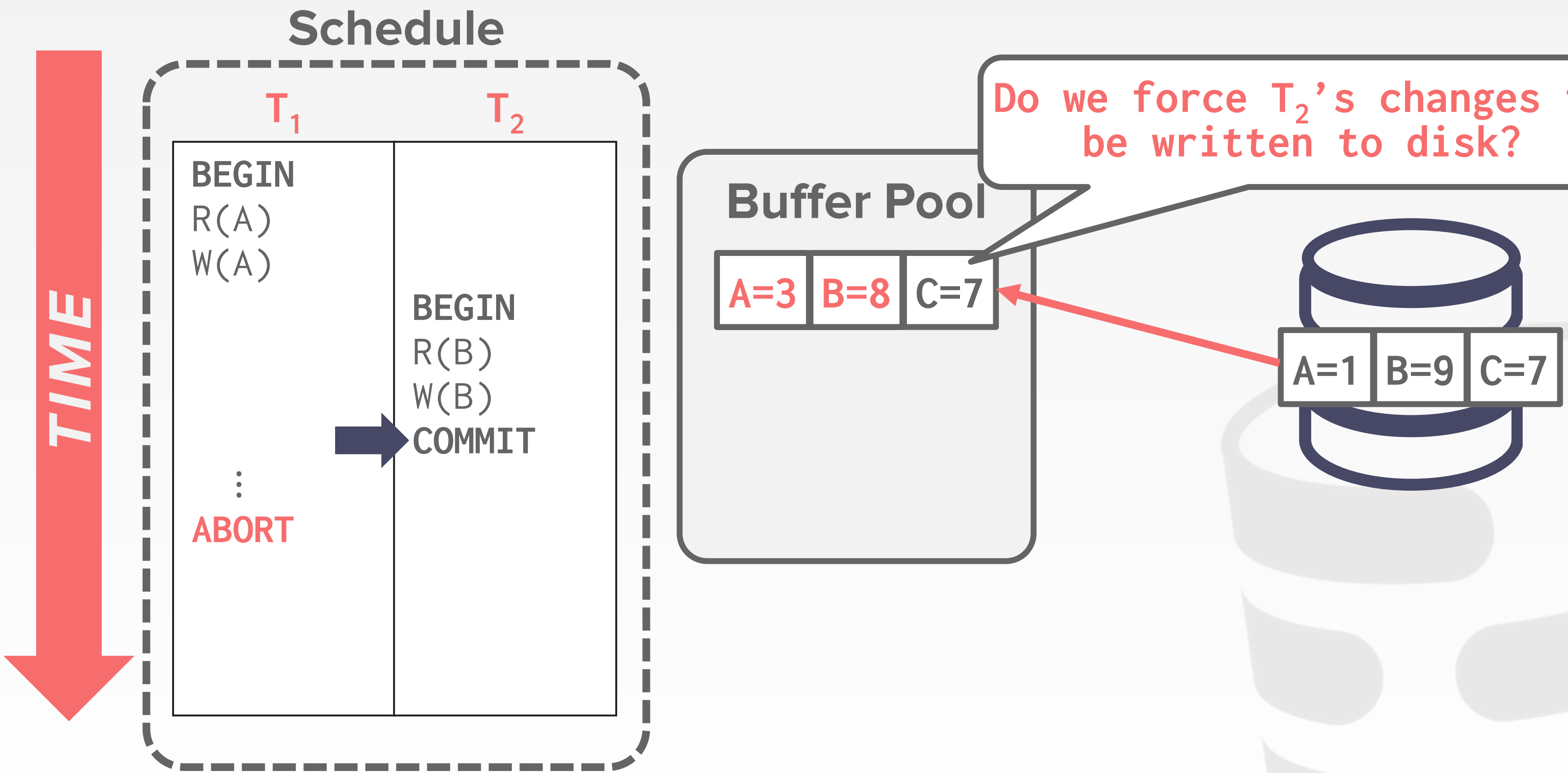
# BUFFER POOL





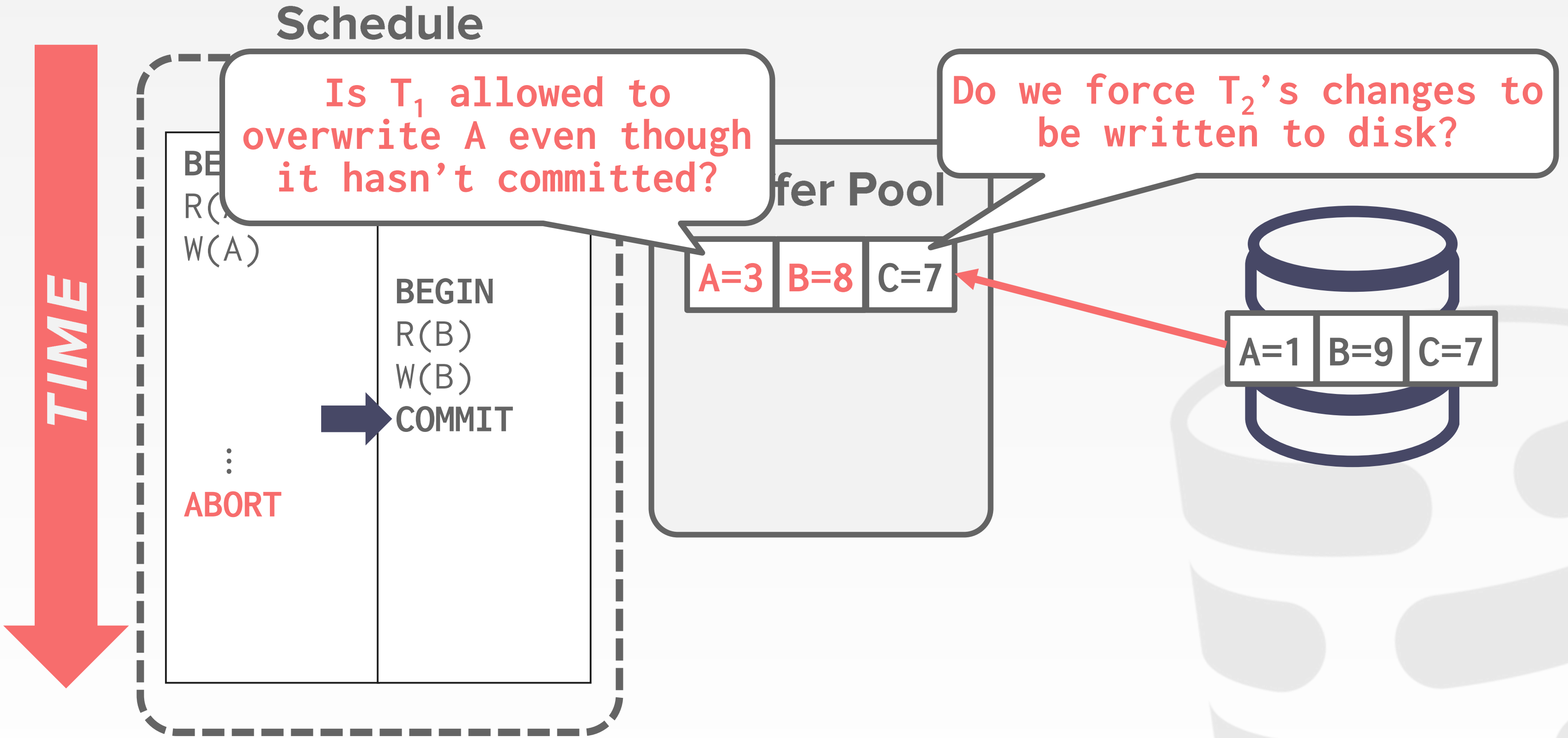
CARNEGIE MELLON DATABASE GROUP

# BUFFER POOL



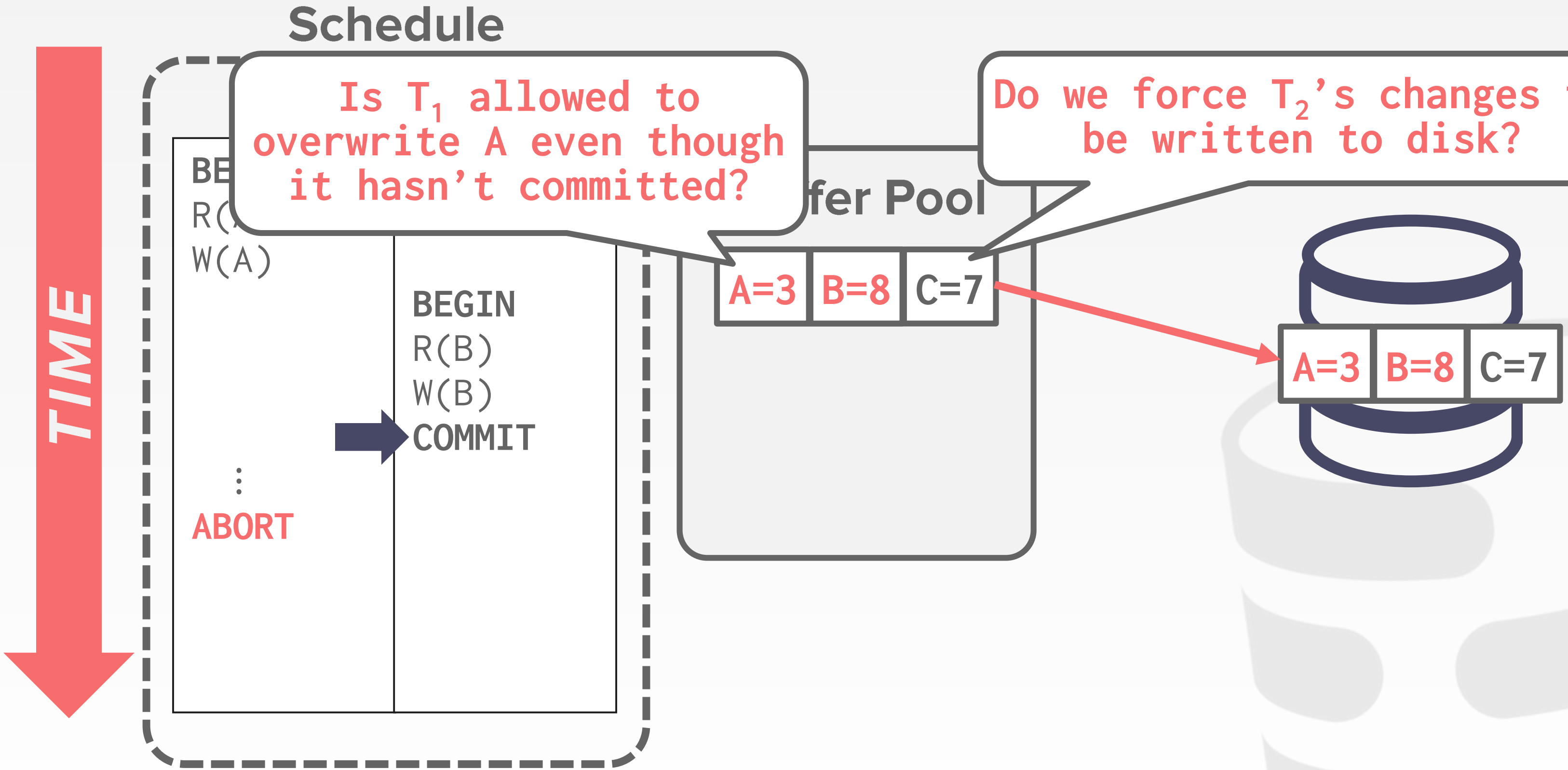
Schedule
TIME
$T_1$
$T_2$
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
⋮
ABORT
Buffer Pool
A=3
B=8
C=7
Do we force $T_2$'s changes to be written to disk?
A=1
B=9
C=7

# BUFFER POOL



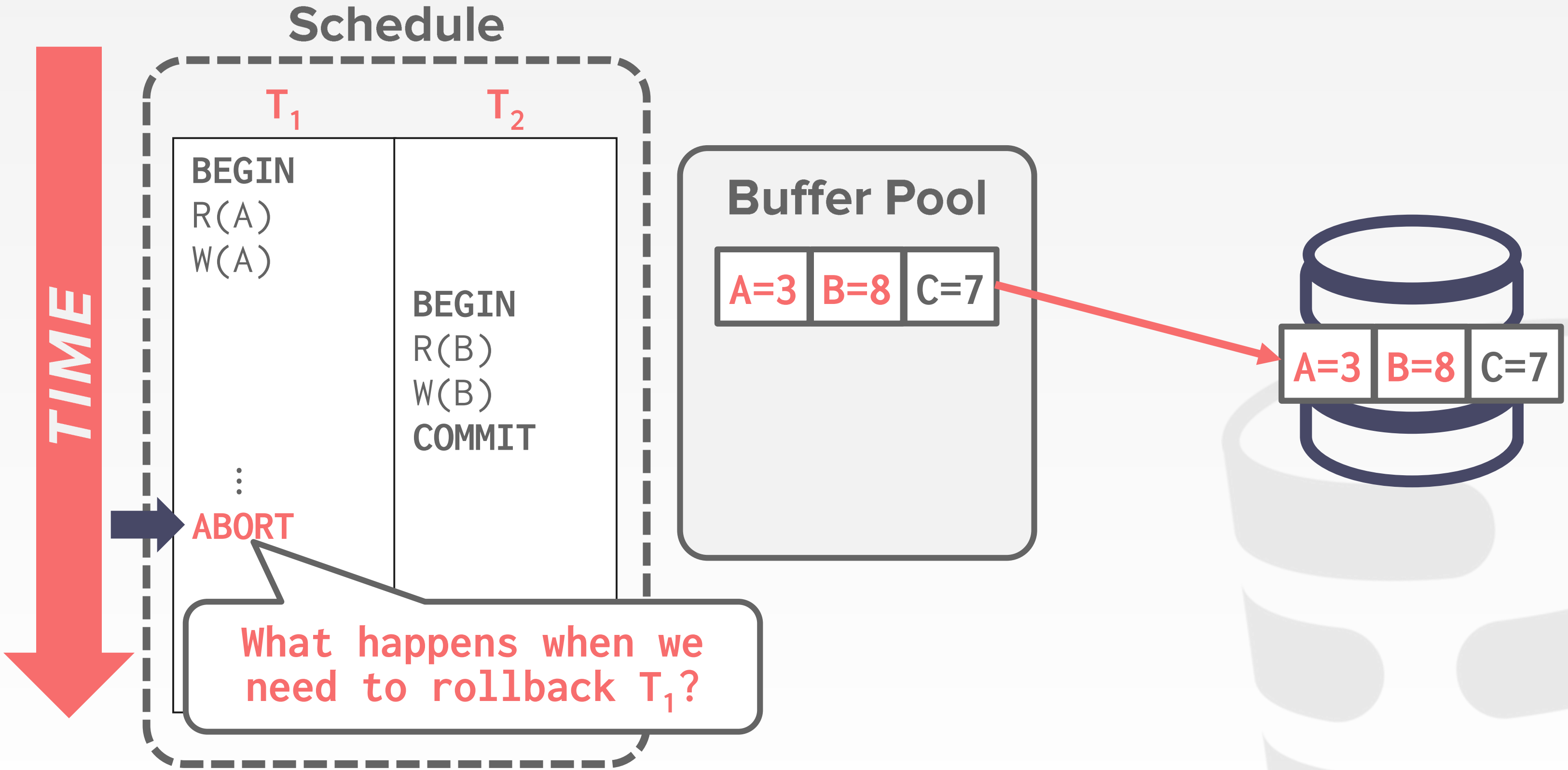
Schedule
TIME
BE
R(
W(A)
BEGIN
R(B)
W(B)
COMMIT
⋮
ABORT
Is T1 allowed to overwrite A even though it hasn't committed?
fer Pool
Do we force T2's changes to be written to disk?
A=3
B=8
C=7
A=1
B=9
C=7

# BUFFER POOL



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

# BUFFER POOL






CARNEGIE MELLON DATABASE GROUP

# 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.

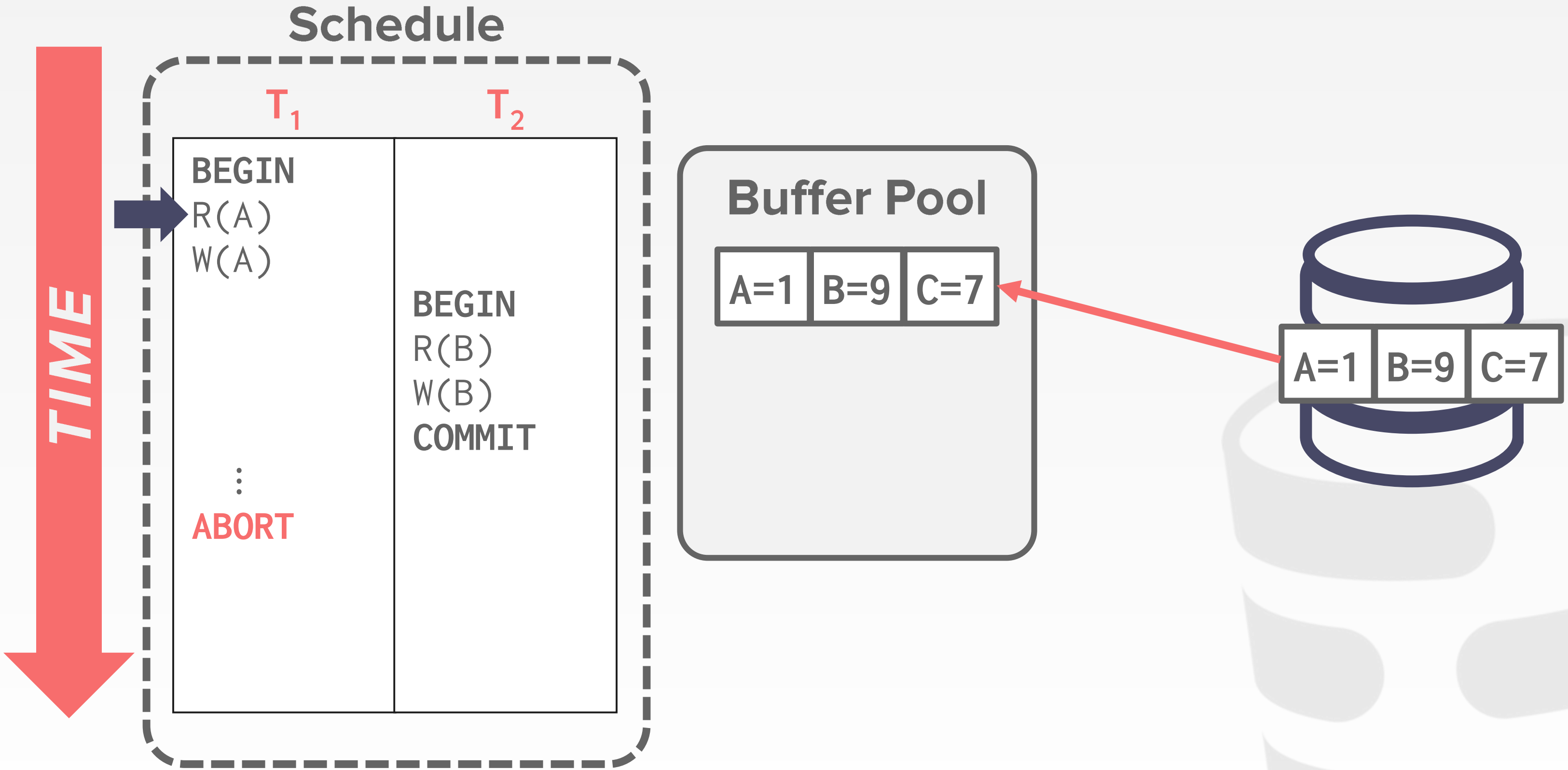
# FORCE POLICY

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

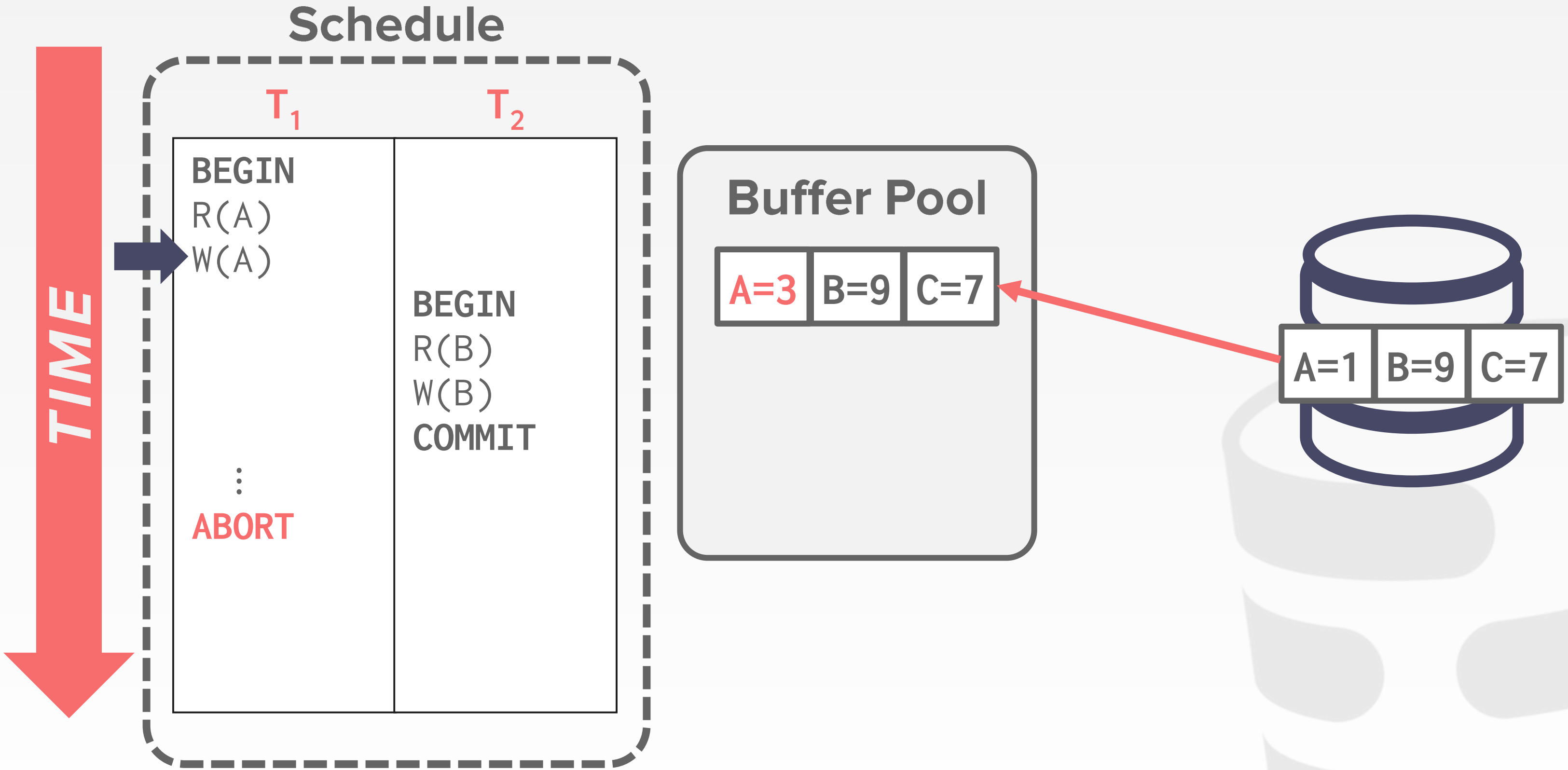
→ **FORCE**: Is enforced.
→ **NO-FORCE**: Is not enforced.

Force writes makes it easier to recover but results in poor runtime performance.

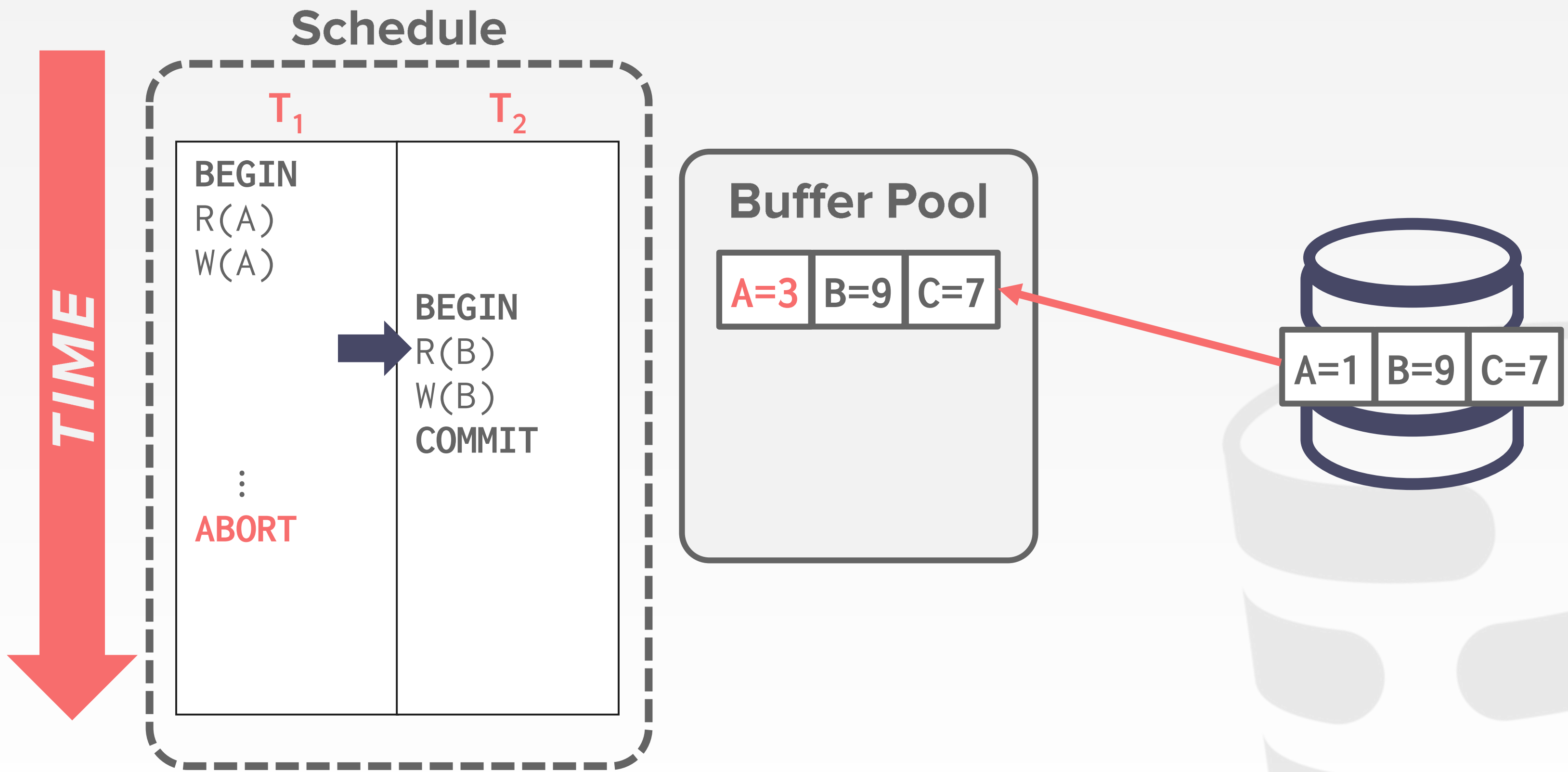
# NO-STEAL + FORCE



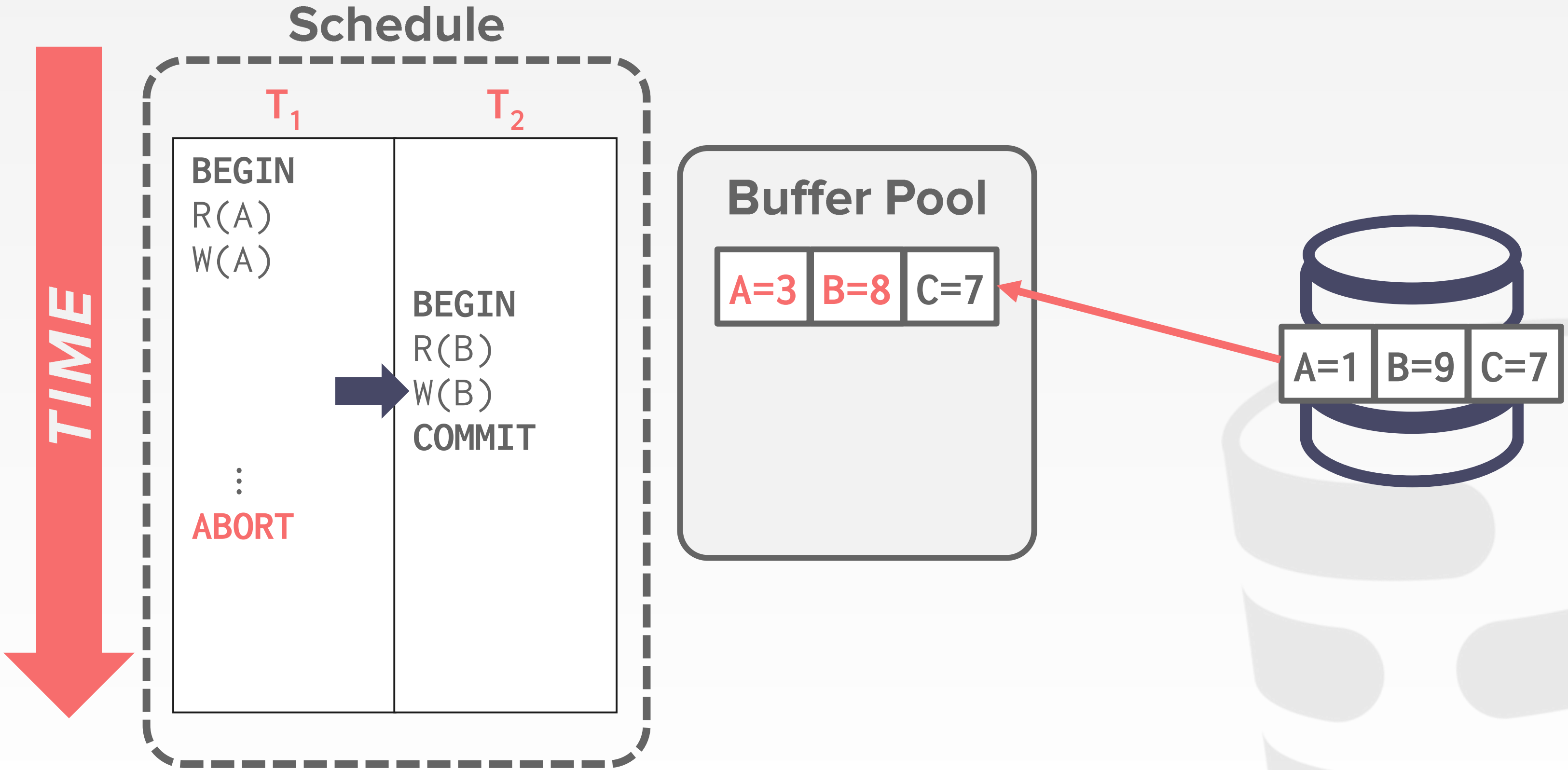
# NO-STEAL + FORCE



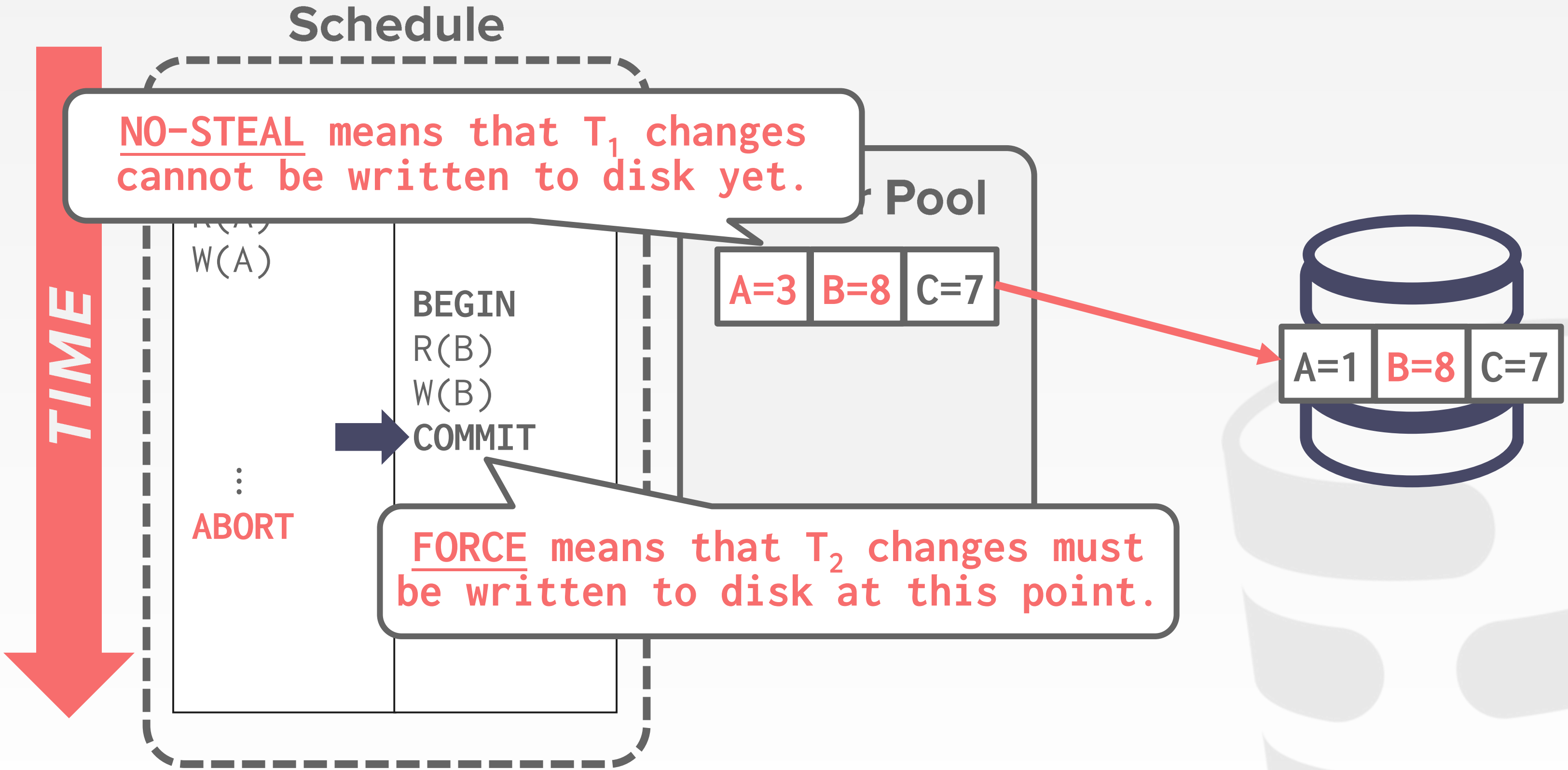
# NO-STEAL + FORCE



# NO-STEAL + FORCE



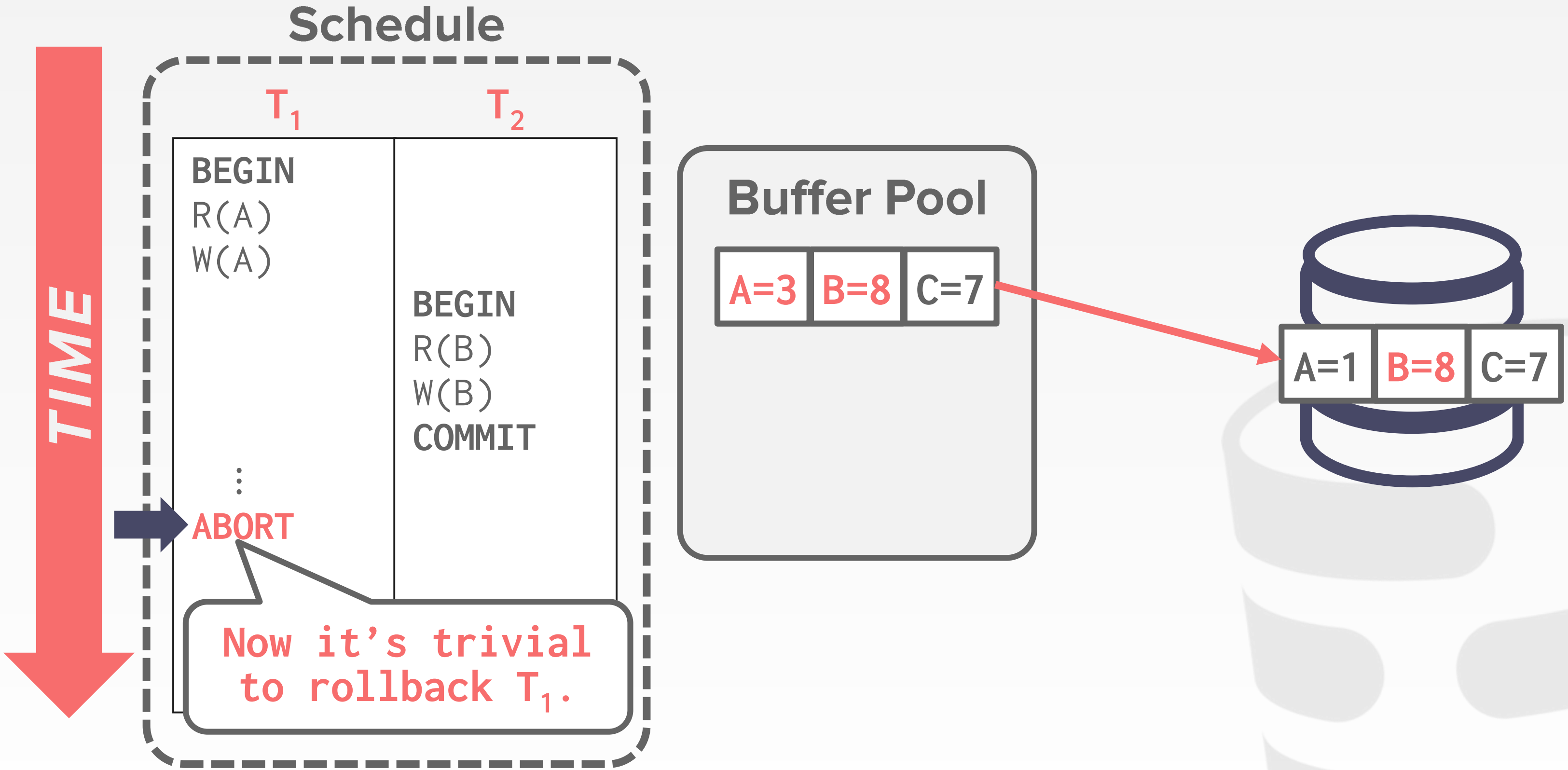
# NO-STEAL + FORCE






CARNEGIE MELLON DATABASE GROUP

# 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.

# SHADOW PAGING

Maintain two separate copies of the database (**master**, **shadow**)

Updates are only made in the shadow copy.

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

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

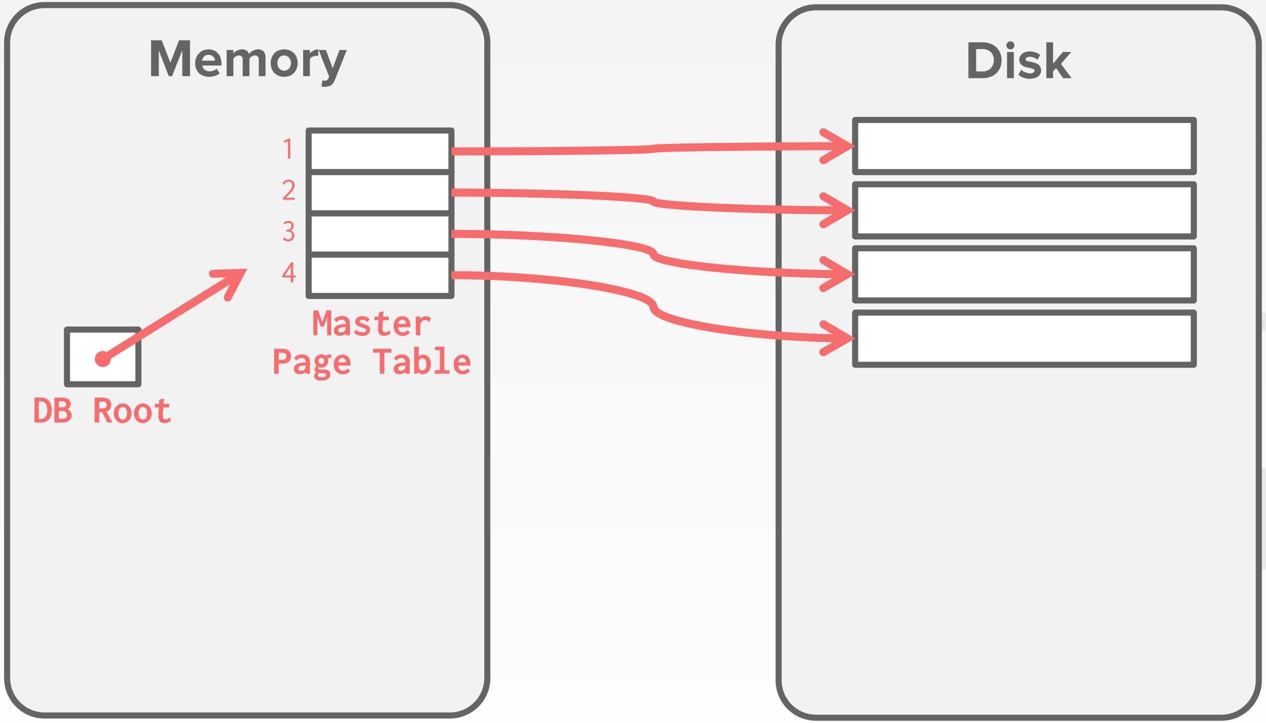
# SHADOW PAGING

Organize the database pages in a tree structure where the root is a single disk page.

There are two copies of the tree, the master and shadow

→ The root points to the master copy.
→ Updates are applied to the shadow copy.

# SHADOW PAGING – EXAMPLE






CARNEGIE MELLON DATABASE GROUP

# SHADOW PAGING

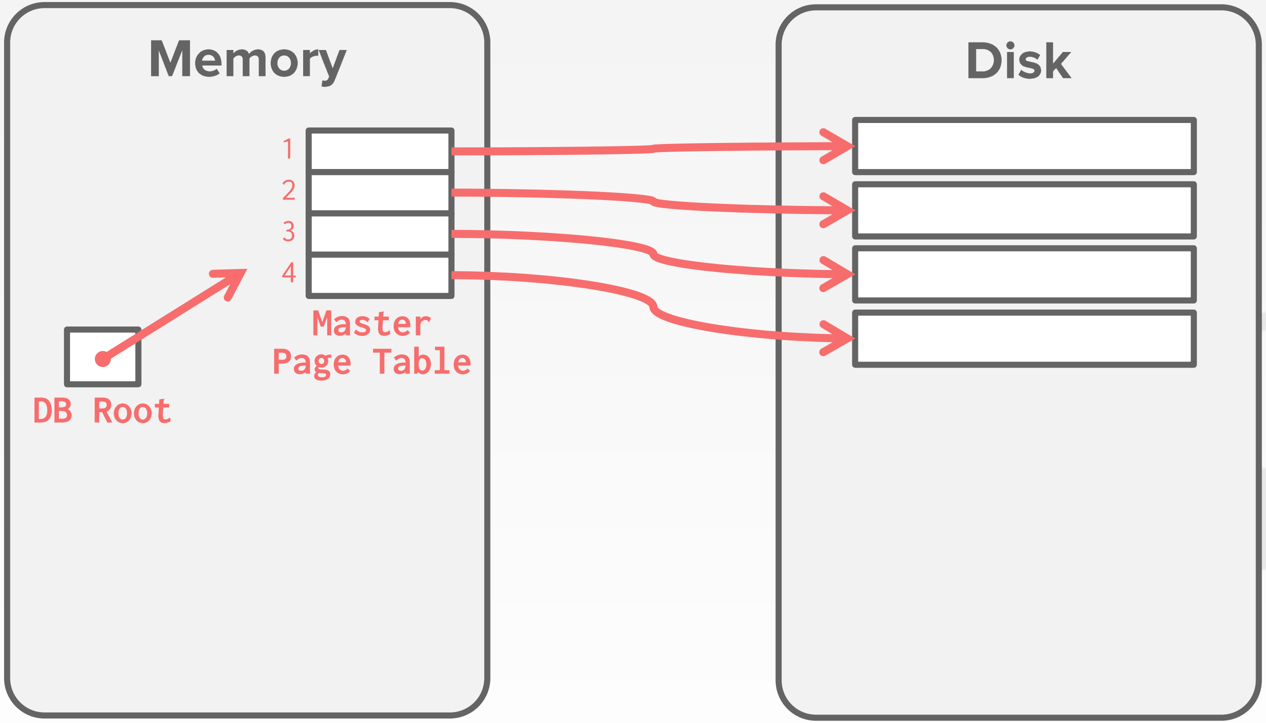
To install the updates, overwrite the root so it points to the shadow, thereby swapping the master and shadow:

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

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

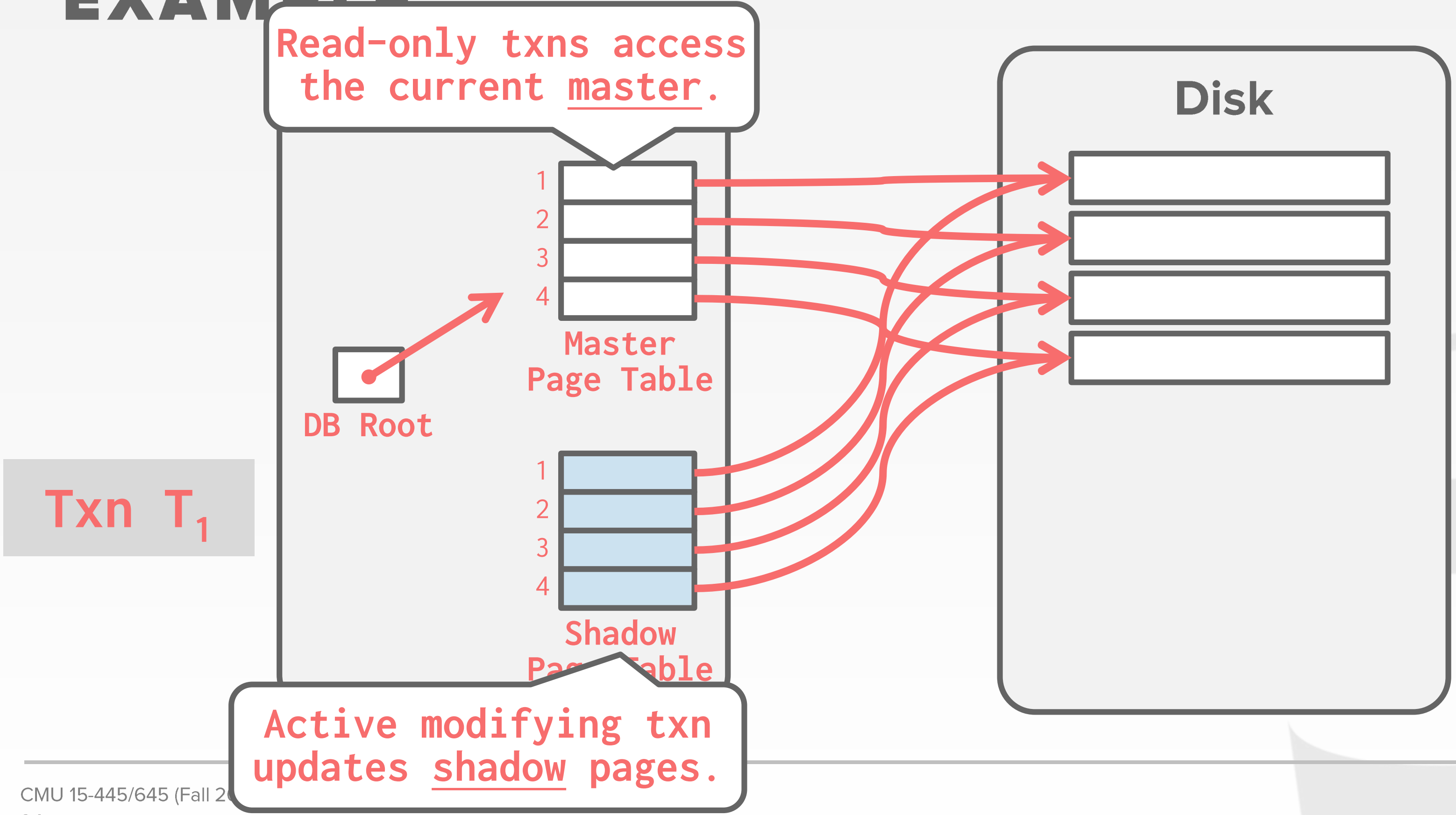
Source: The Great Phil Bernstein

# SHADOW PAGING – EXAMPLE



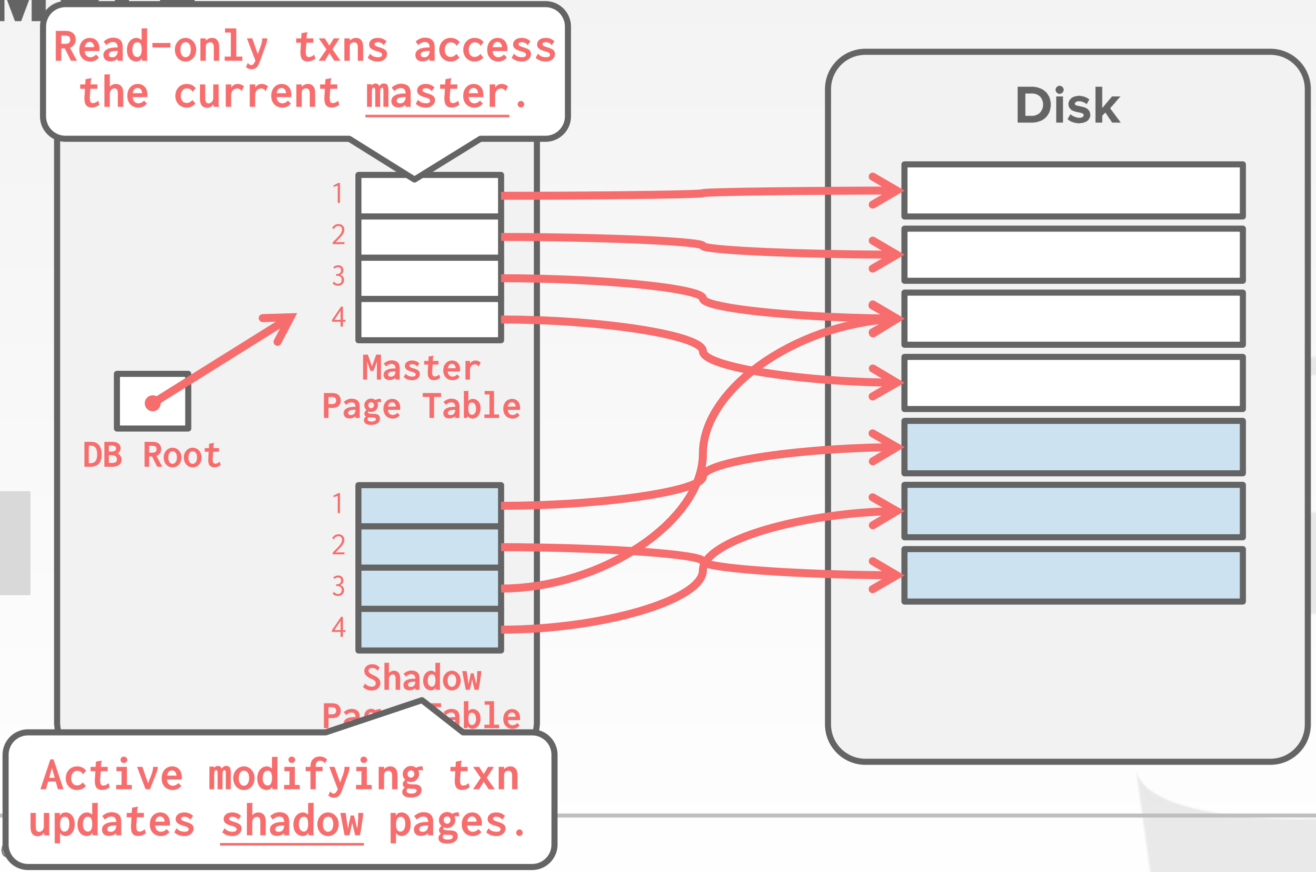

Txn $T_1$

# SHADOW PAGING – EXAMPLE

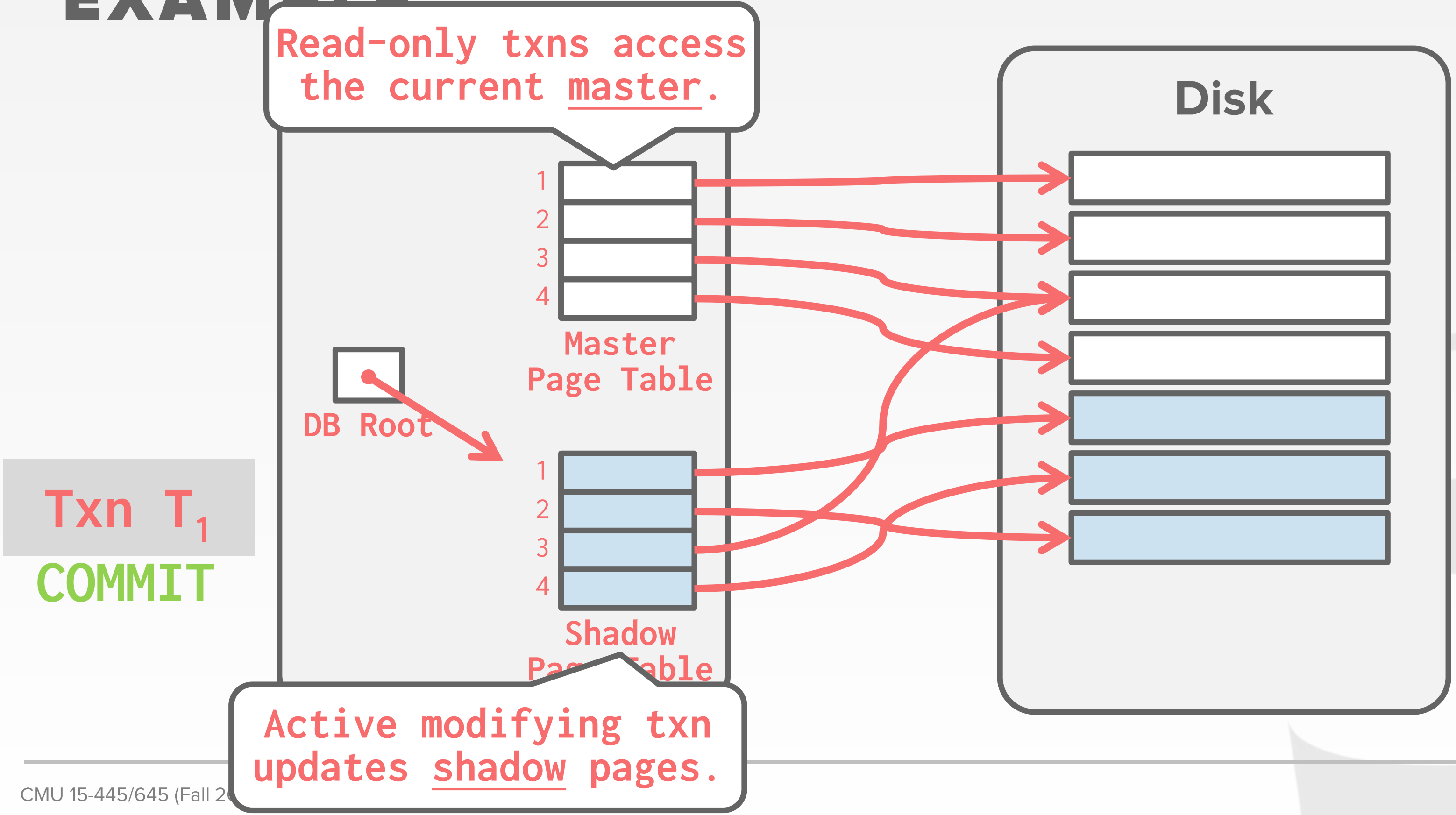


# SHADOW PAGING – EXAMPLE

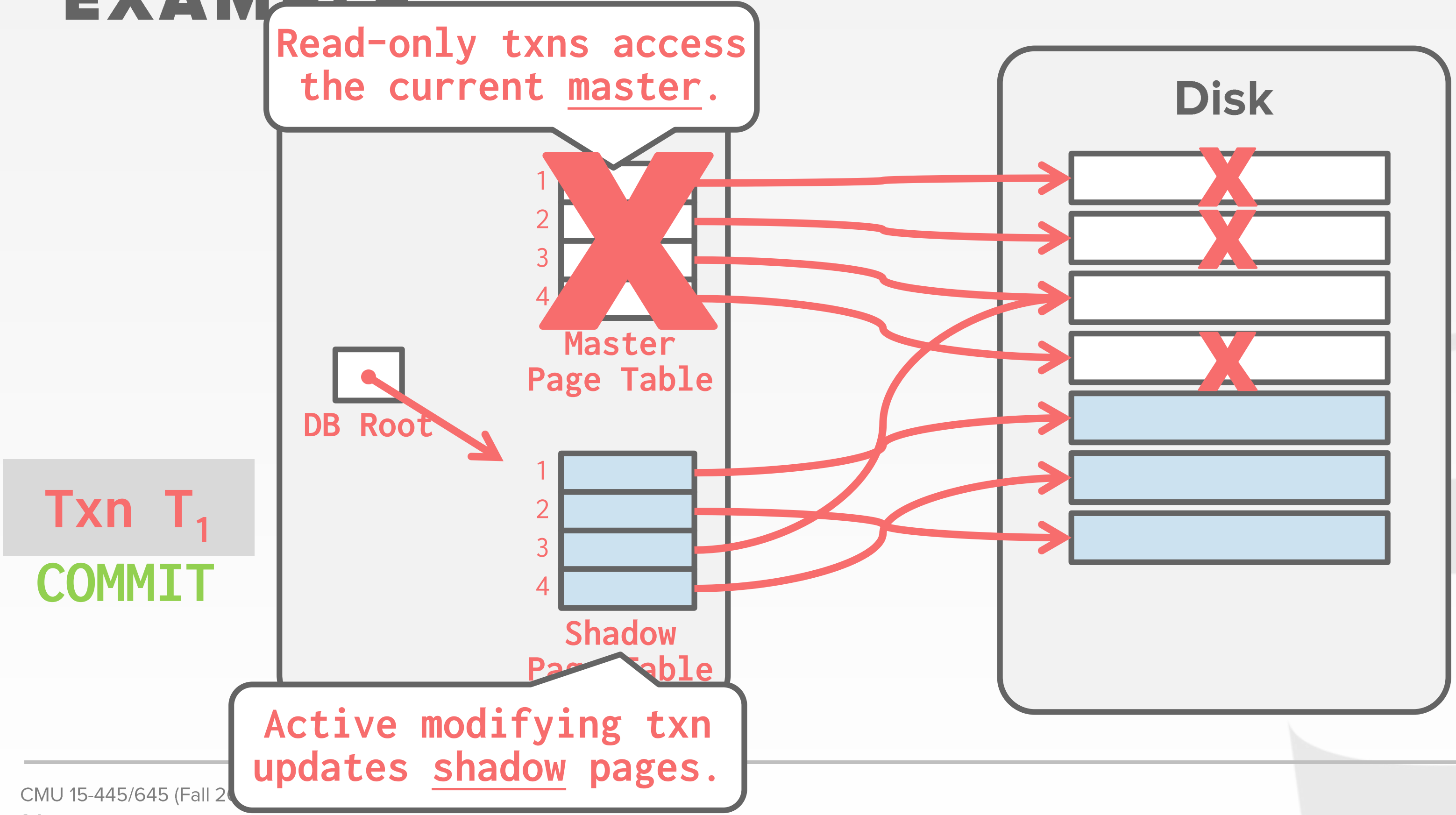
Txn $T_1$



# SHADOW PAGING – EXAMPLE



# SHADOW PAGING – EXAMPLE



# 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.

# OBSERVATION

Flushing non-contiguous pages on disk is slow (aka random writes).

We need a way to convert random writes into sequential writes.

# WRITE-AHEAD LOG

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

→ Assume that the log is on stable storage.

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

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

# WAL PROTOCOL

All log records pertaining to an updated page are written to non-volatile storage before the page itself is allowed to be 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

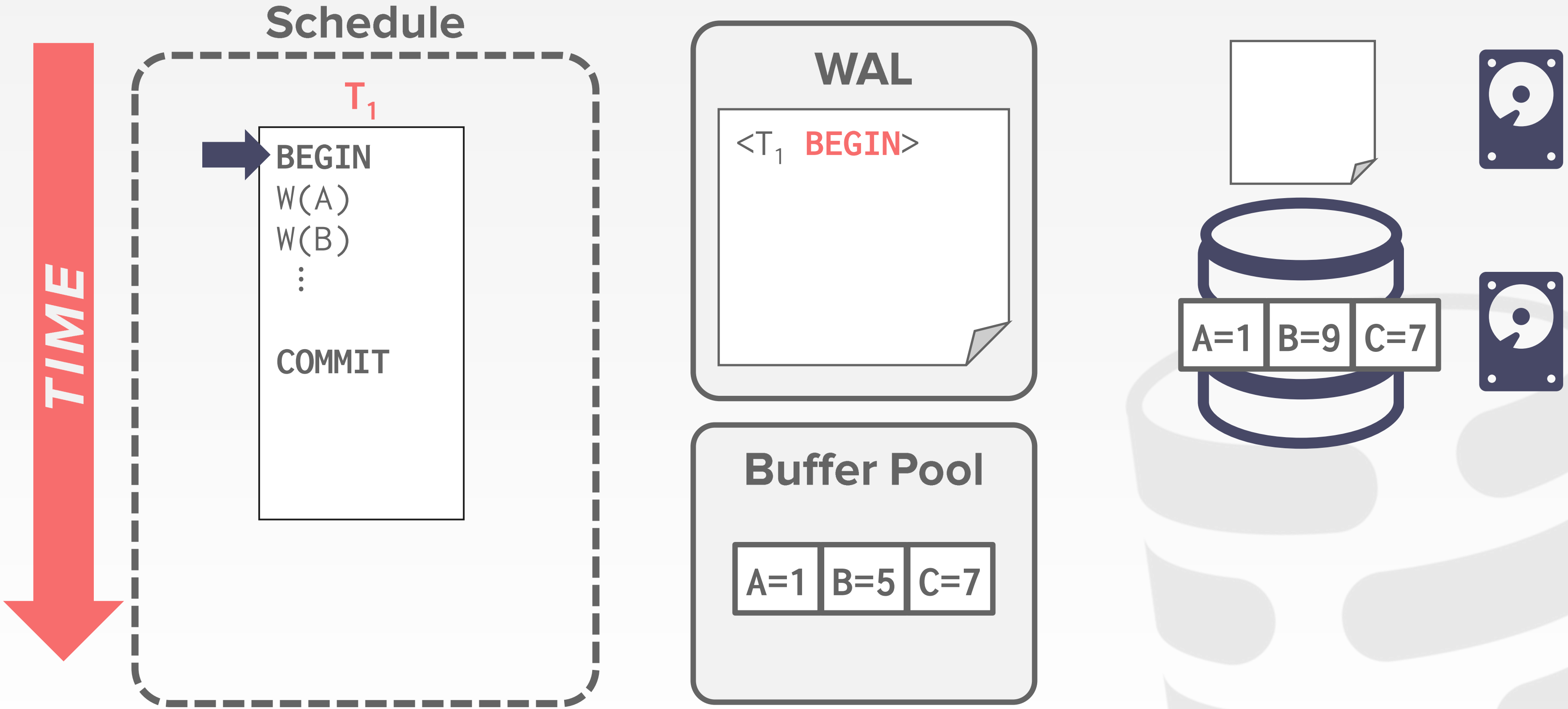
Each log entry contains information about the change to a single object:

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



CARNEGIE MELLON
DATABASE GROUP

# WAL – EXAMPLE


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

# WAL – EXAMPLE

## Schedule

*TIME*

$T_1$

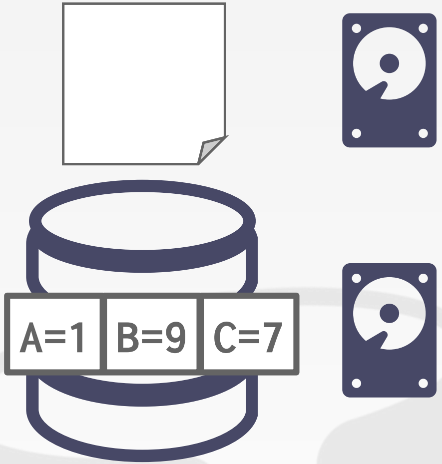
```
BEGIN
W(A)
W(B)
 ⋮

COMMIT
```

## WAL

<$T_1$ BEGIN>
<$T_1$, A, 1, 8>

## Buffer Pool

| | | |
|---|---|---|
| A=8 | B=5 | C=7 |






CARNEGIE MELLON DATABASE GROUP

# WAL – EXAMPLE

## Schedule

TIME

$T_1$

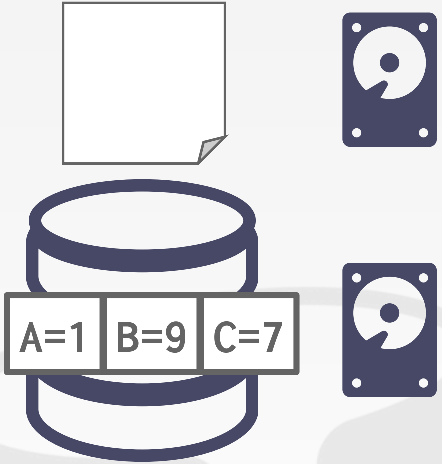
```
BEGIN
W(A)
W(B)
 ⋮

COMMIT
```

### WAL

<$T_1$ BEGIN>
<$T_1$, A, 1, 8>
<$T_1$, B, 5, 9>

### Buffer Pool

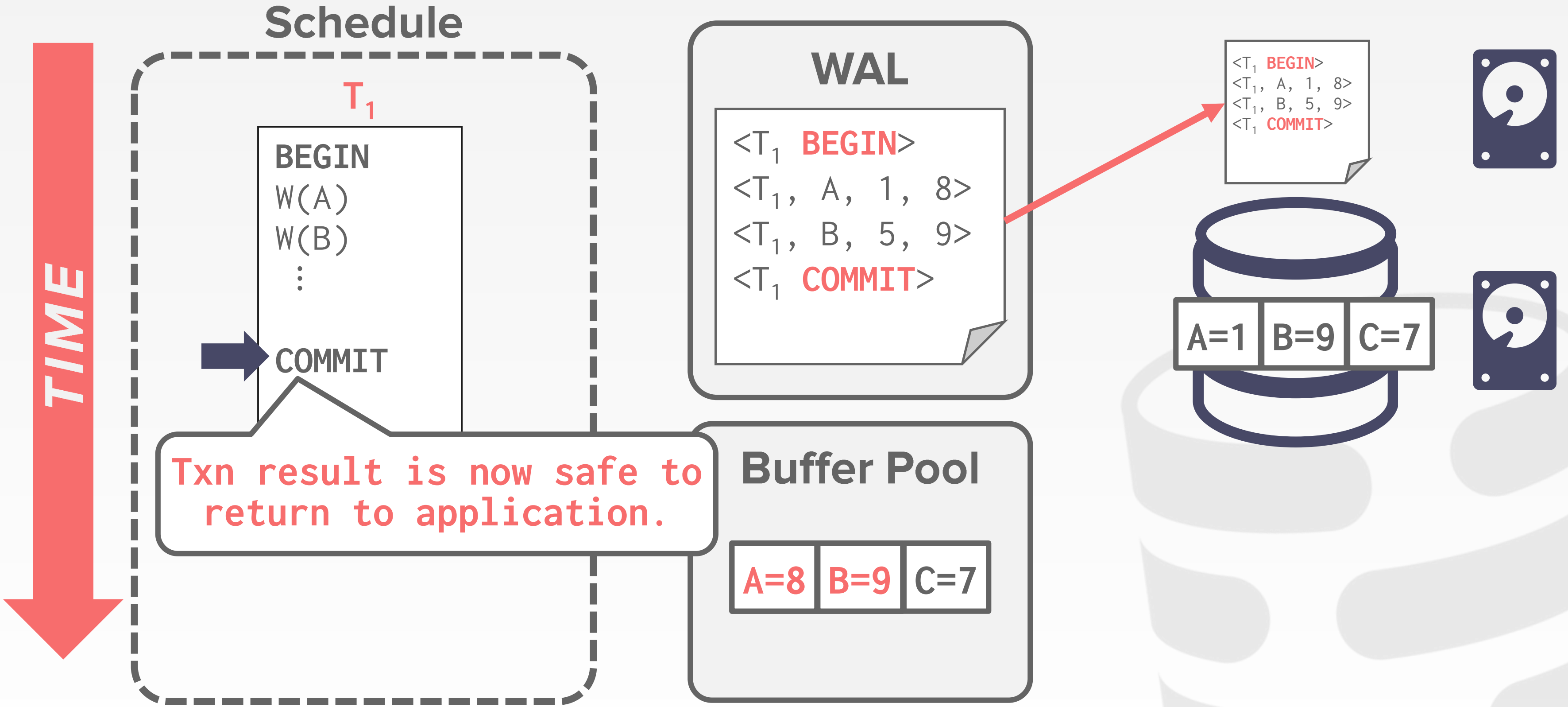
| | | |
|---|---|---|
| A=8 | B=9 | C=7 |





CARNEGIE MELLON DATABASE GROUP

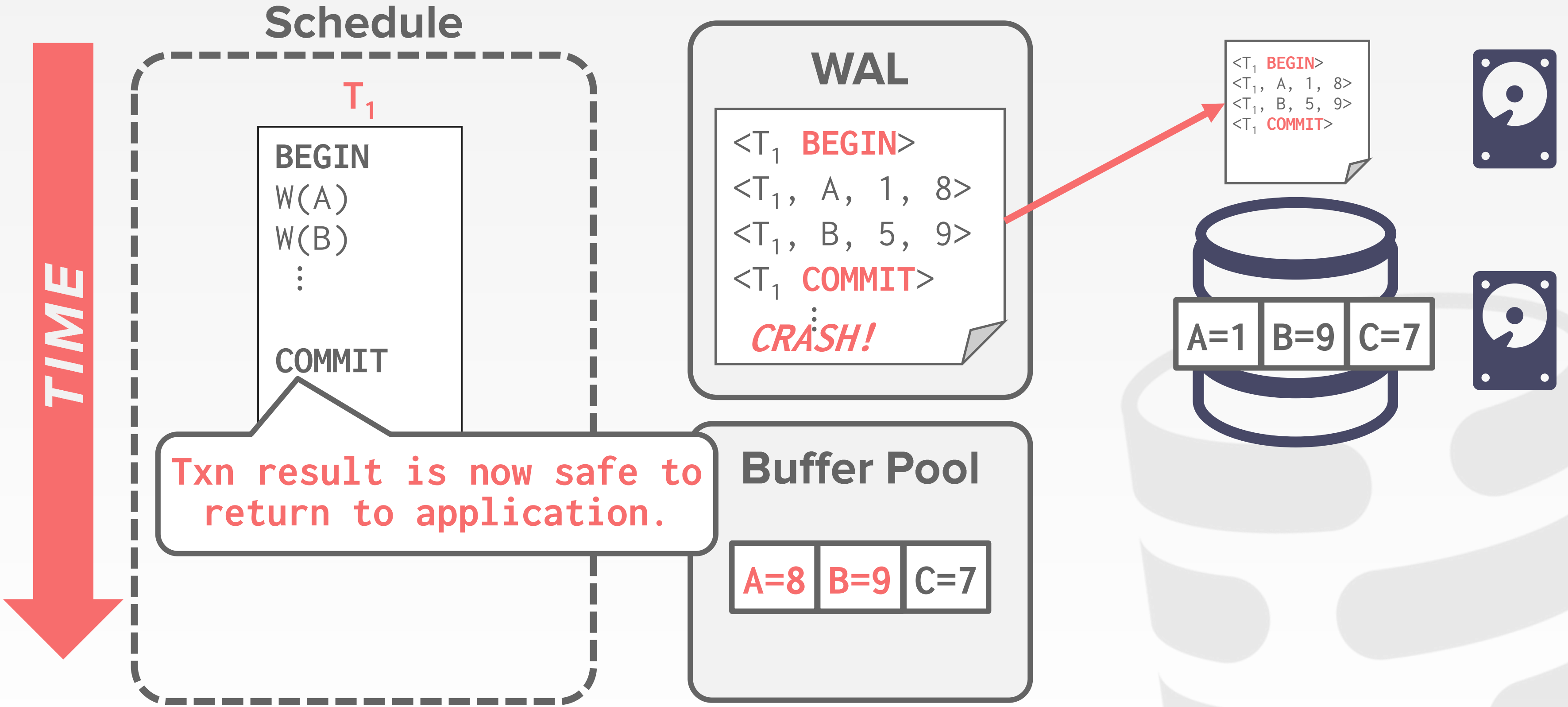
# WAL – EXAMPLE






CARNEGIE MELLON DATABASE GROUP

# WAL – EXAMPLE


Schedule
TIME
$T_1$
BEGIN
W(A)
W(B)
⋮
COMMIT
Txn result is now safe to return to application.
WAL
<$T_1$ BEGIN>
<$T_1$, A, 1, 8>
<$T_1$, B, 5, 9>
<$T_1$ COMMIT>
⋮
CRASH!
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=9
C=7

# WAL – IMPLEMENTATION

**When should we write log entries to disk?**

→ When the transaction commits.

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

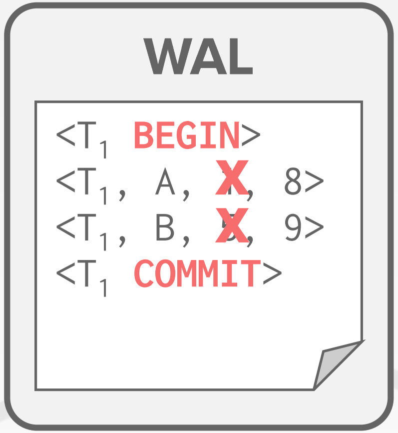
**When should we write dirty records to disk?**

→ Every time the txn executes an update?

→ Once when the txn commits?

# WAL – DEFERRED UPDATES

If we prevent the DBMS from writing dirty records to disk until the txn commits, then we don't need to store their original values.






CARNEGIE MELLON DATABASE GROUP

# WAL – DEFERRED UPDATES

If we prevent the DBMS from writing dirty records to disk until the txn commits, then we don't need to store their original values.

**WAL**

<$T_1$ BEGIN>
<$T_1$, A, ~~1~~, 8>
<$T_1$, B, ~~2~~, 9>
<$T_1$ COMMIT>

Replay the log and redo each update.

<$T_1$ BEGIN>
<$T_1$, A, 8>
<$T_1$, B, 9>
<$T_1$ COMMIT>
*CRASH!*

Simply ignore all of $T_1$'s updates.

<$T_1$ BEGIN>
<$T_1$, A, 8>
<$T_1$, B, 9>
*CRASH!*

# WAL – DEFERRED UPDATES

This won't work if the change set of a txn is larger than the amount of memory available.

The DBMS cannot undo changes for an aborted txn if it doesn't have the original values in the log.

We need to use the **STEAL** policy.

# BUFFER POOL POLICIES

## 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

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

# CHECKPOINTS

The WAL will grow forever.

After a crash, the DBMS has to 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

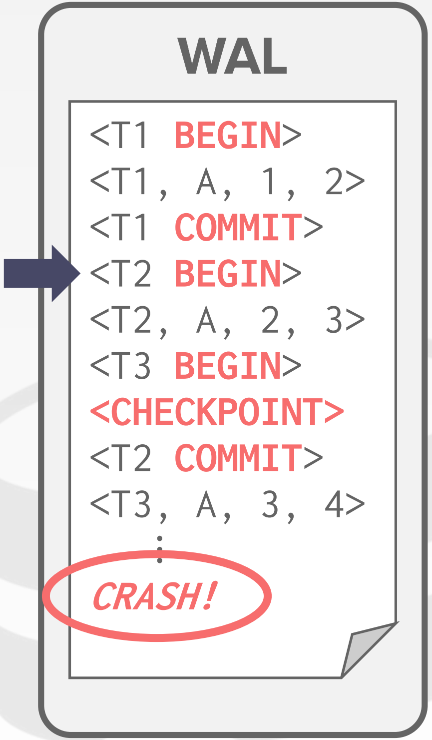
Output onto stable storage all log records currently residing in main memory.

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 ($T_1$).






CARNEGIE MELLON
DATABASE GROUP

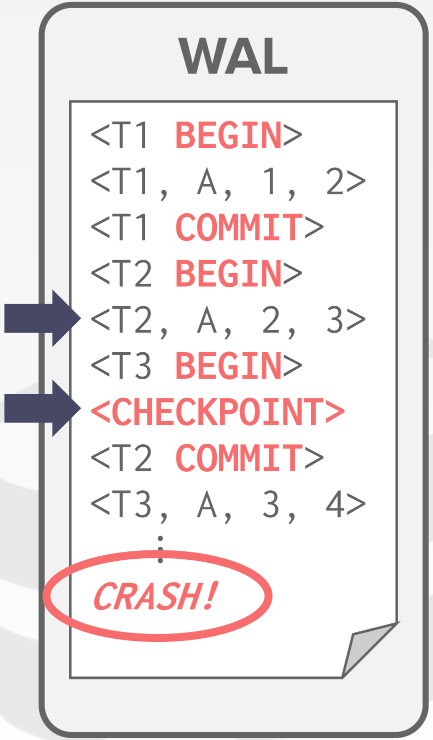
# CHECKPOINTS

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.





CARNEGIE MELLON DATABASE GROUP

# CHECKPOINTS – CHALLENGES

We have to stall all txns when take 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.

# LOGGING SCHEMES

## Physical Logging

→ Record the changes made to a specific location in the database.
→ Example: Position of a record in a page.

## 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 data organization of the page.

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,
 ObjectId=1,
 Before=ABC,
 After=XYZ>
<T1,
 Index=X_PKEY,
 IndexPage=45,
 Key=(1,Record1)>
```

# CONCLUSION

Write-Ahead Log to handle loss of volatile storage.

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

On recovery: undo uncommitted txns + redo committed txns.

# NEXT CLASS

Recovery with ARIES.