THE UNIVERSITY *of* EDINBURGH

Advanced Database Systems

Spring 2024

Lecture #26:

Recovery

R&G: Chapters 16 & 18

Review: The ACID Properties

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

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

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

Durability: If a txn commits, its effects persist

The recovery manager ensures atomicity, DB consistency, and durability

Motivation

Atomicity:

Transactions may abort (“rollback”)

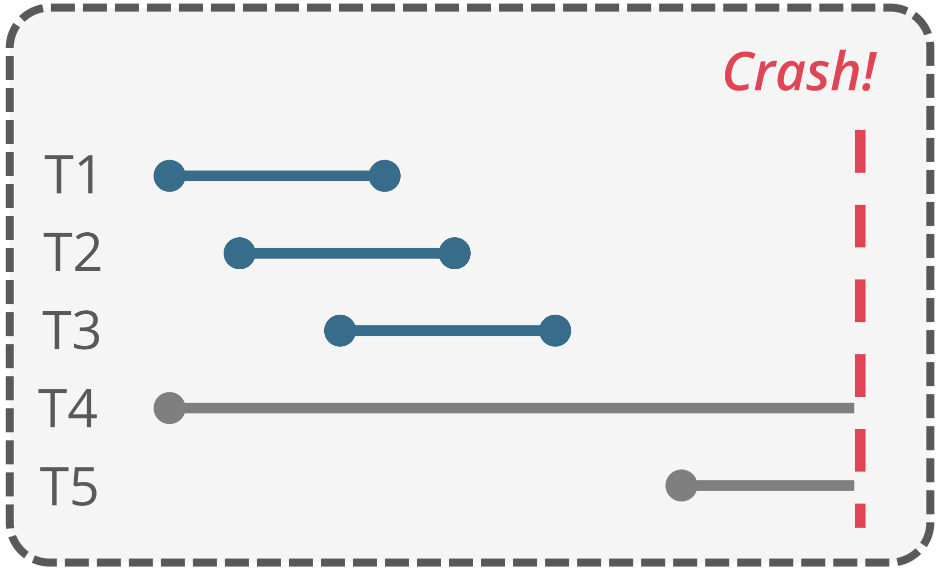
Durability:

What if the DBMS stops running?

Desired behaviour after system restarts:

T1, T2 & T3 should be durable

T4 & T5 should be aborted (effects not seen)



Types of Failures

Logical Errors

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

Internal State Errors

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

Transaction Failures

Software Failures

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

Hardware Failures

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

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

System Failures

Non-Repairable Hardware Failure

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

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

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 Database must be restored from an archived version (replica).

Storage Media Failures

Crash Recovery

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

Recovery algorithms have two parts:

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

Observation

The primary storage location of the database is on non-volatile storage (disk), but this is much slower than volatile storage (main memory)

Use volatile memory for faster access:

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

The DBMS needs to guarantee that:

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

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

Handling the Buffer Pool

Steal Policy

Whether the DBMS allows buffer pool frames with uncommitted updates to be replaced (i.e., the corresponding dirty pages flushed to non-volatile storage)

STEAL: Is allowed

NO-STEAL: Is **not** allowed

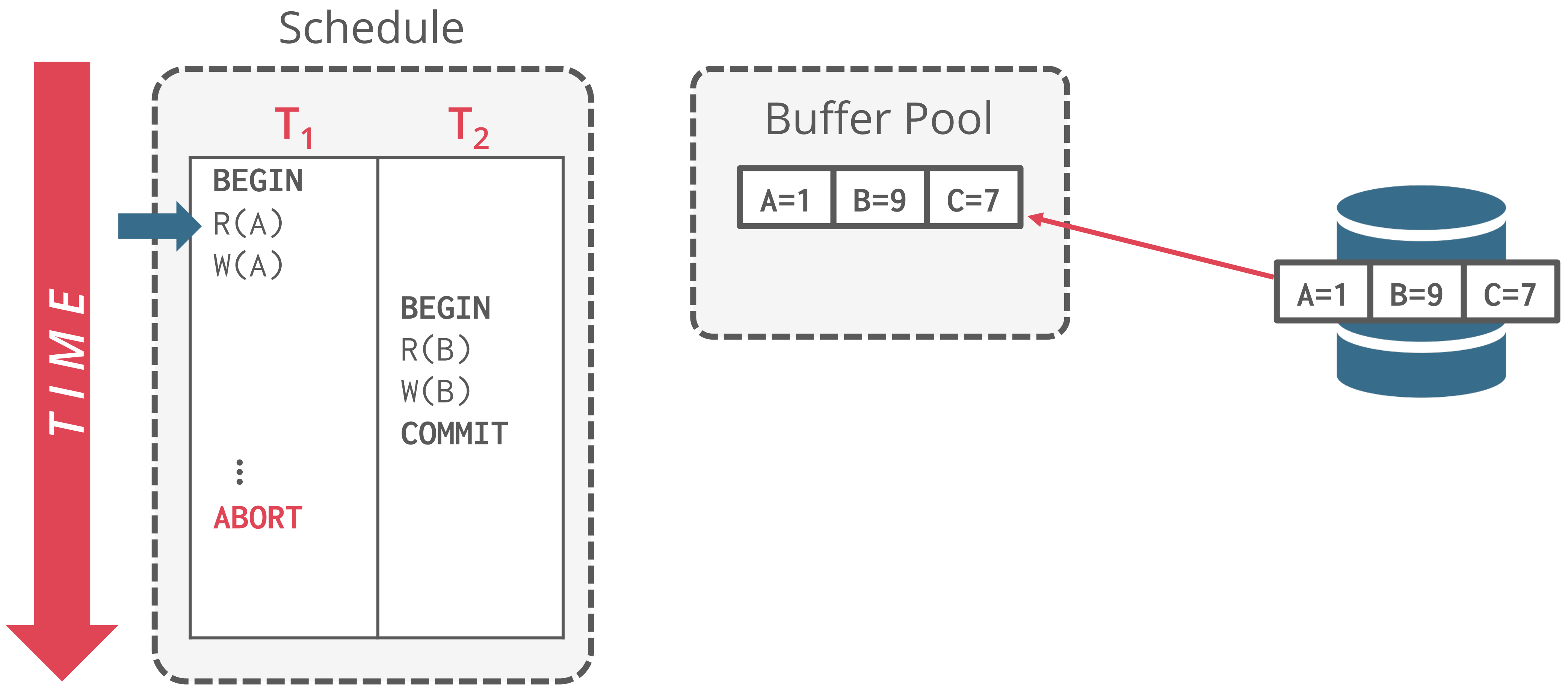
Force Policy

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

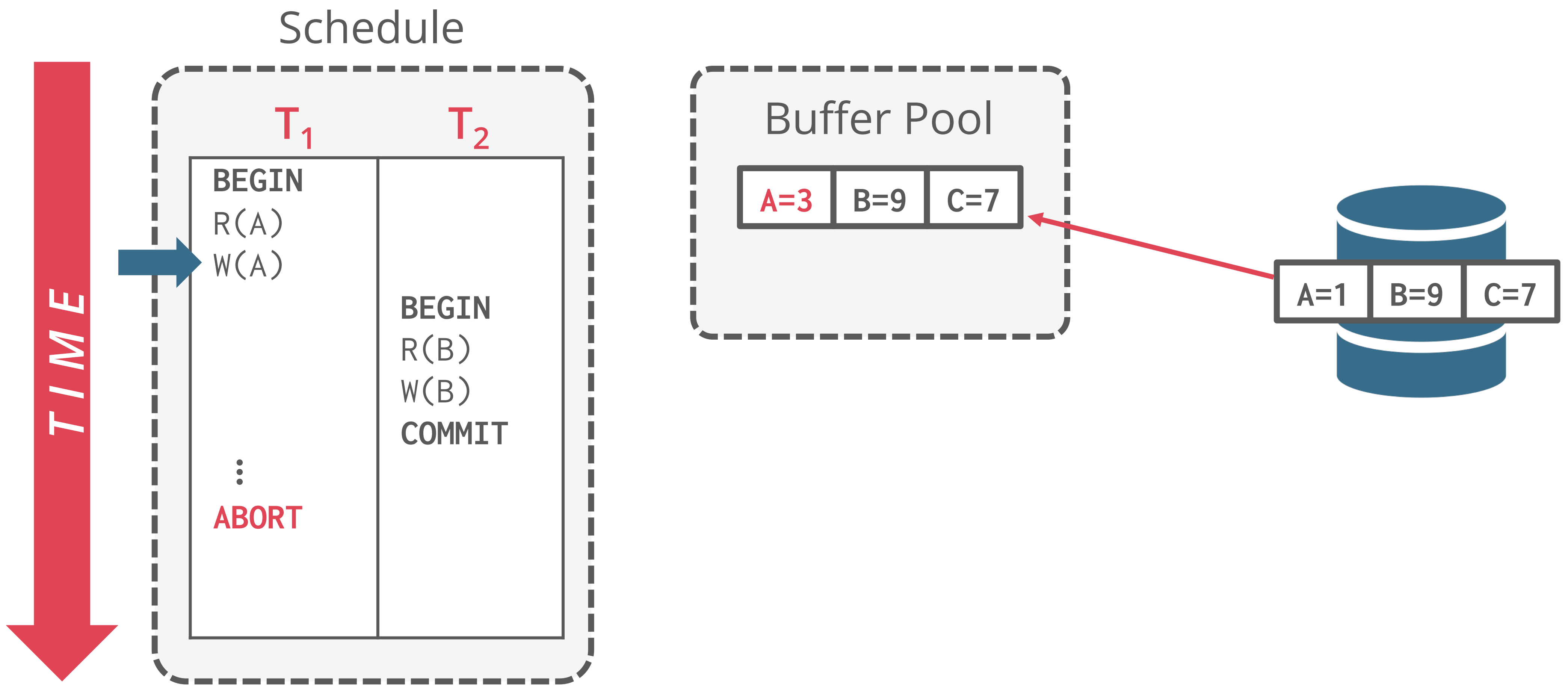
FORCE: Is enforced

NO-FORCE: Is **not** enforced

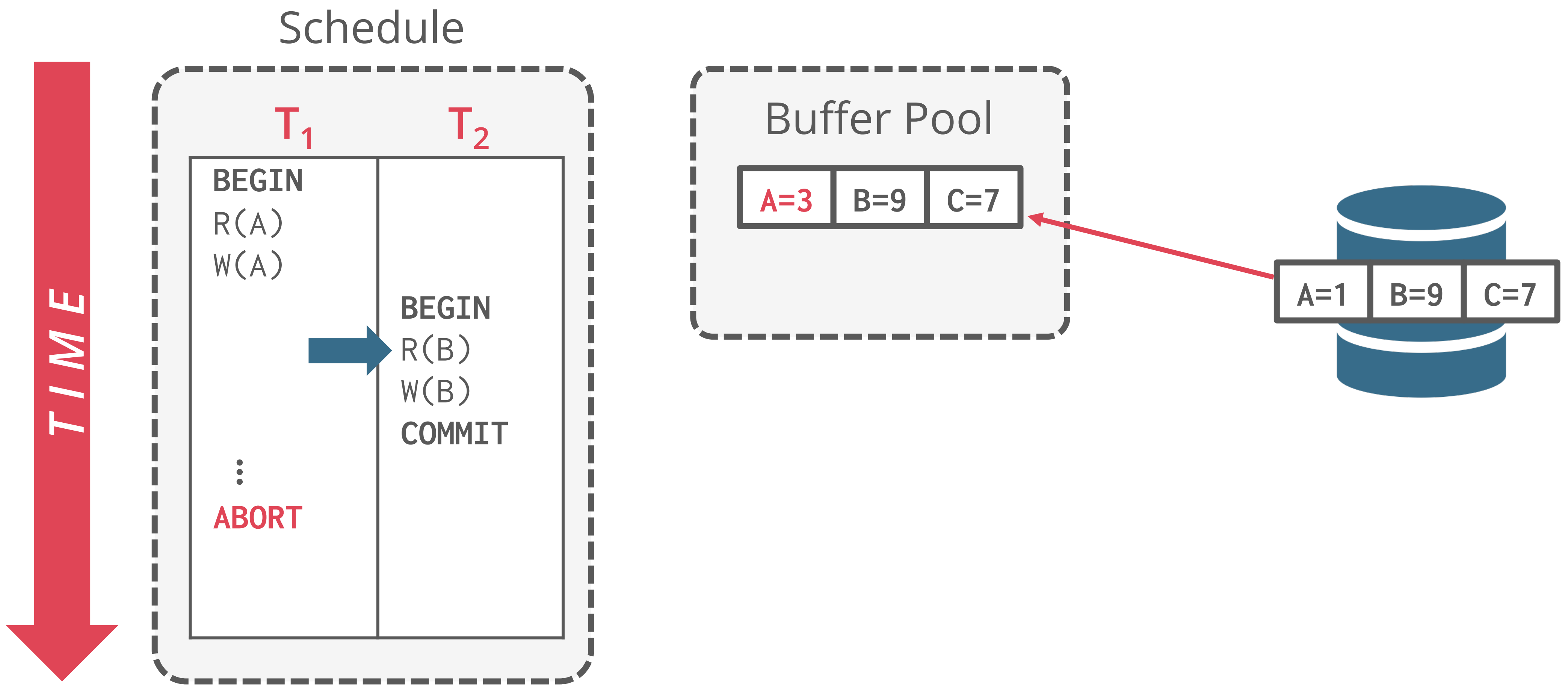
No-Steal + Force



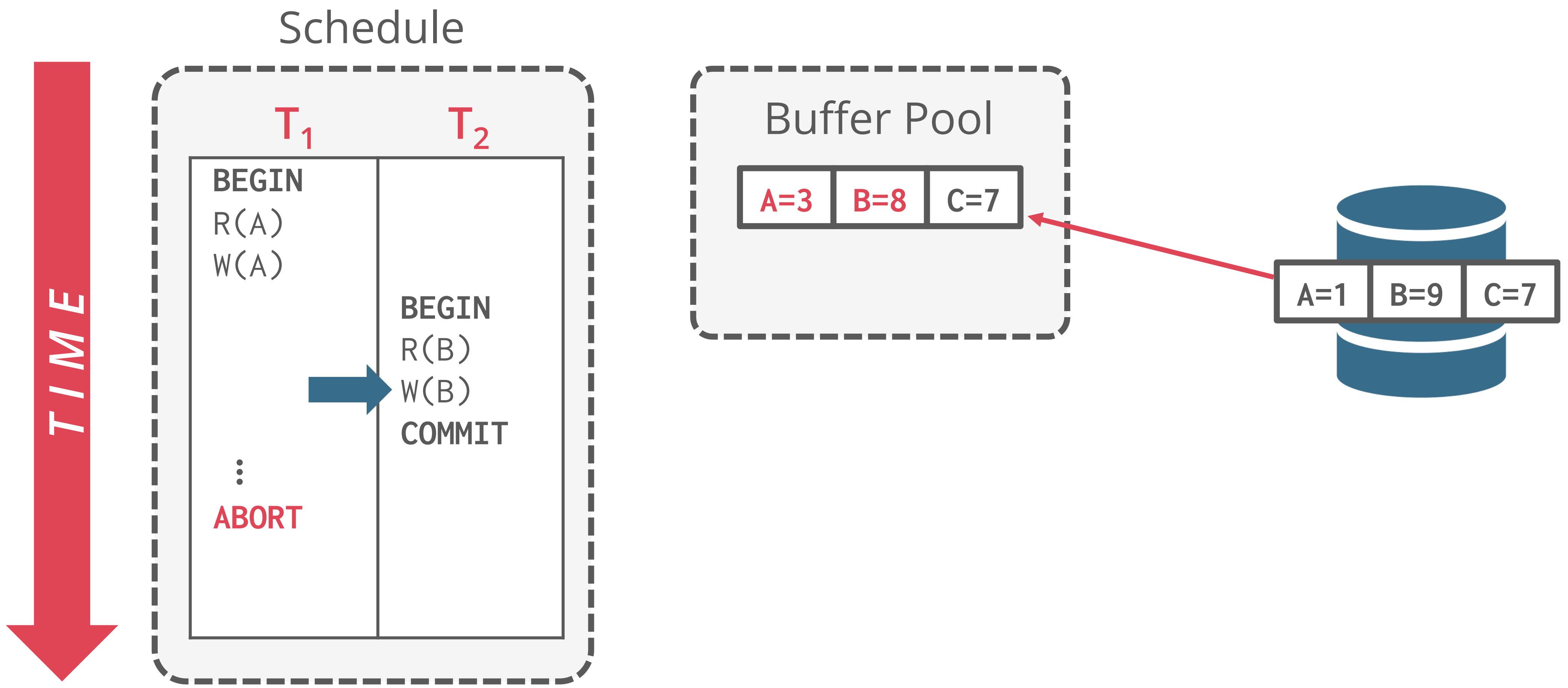
No-Steal + Force



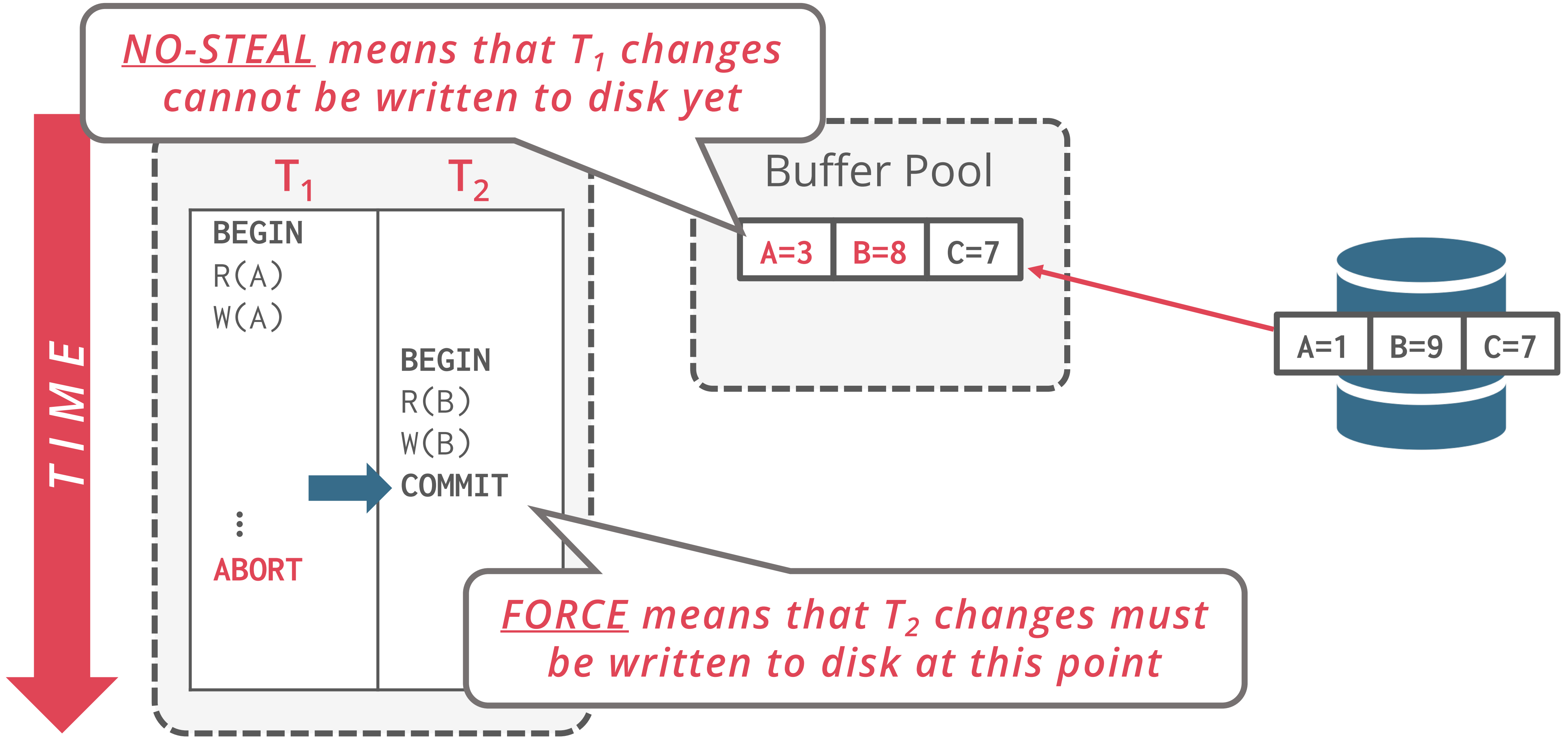
No-Steal + Force



NO-STEAL + FORCE

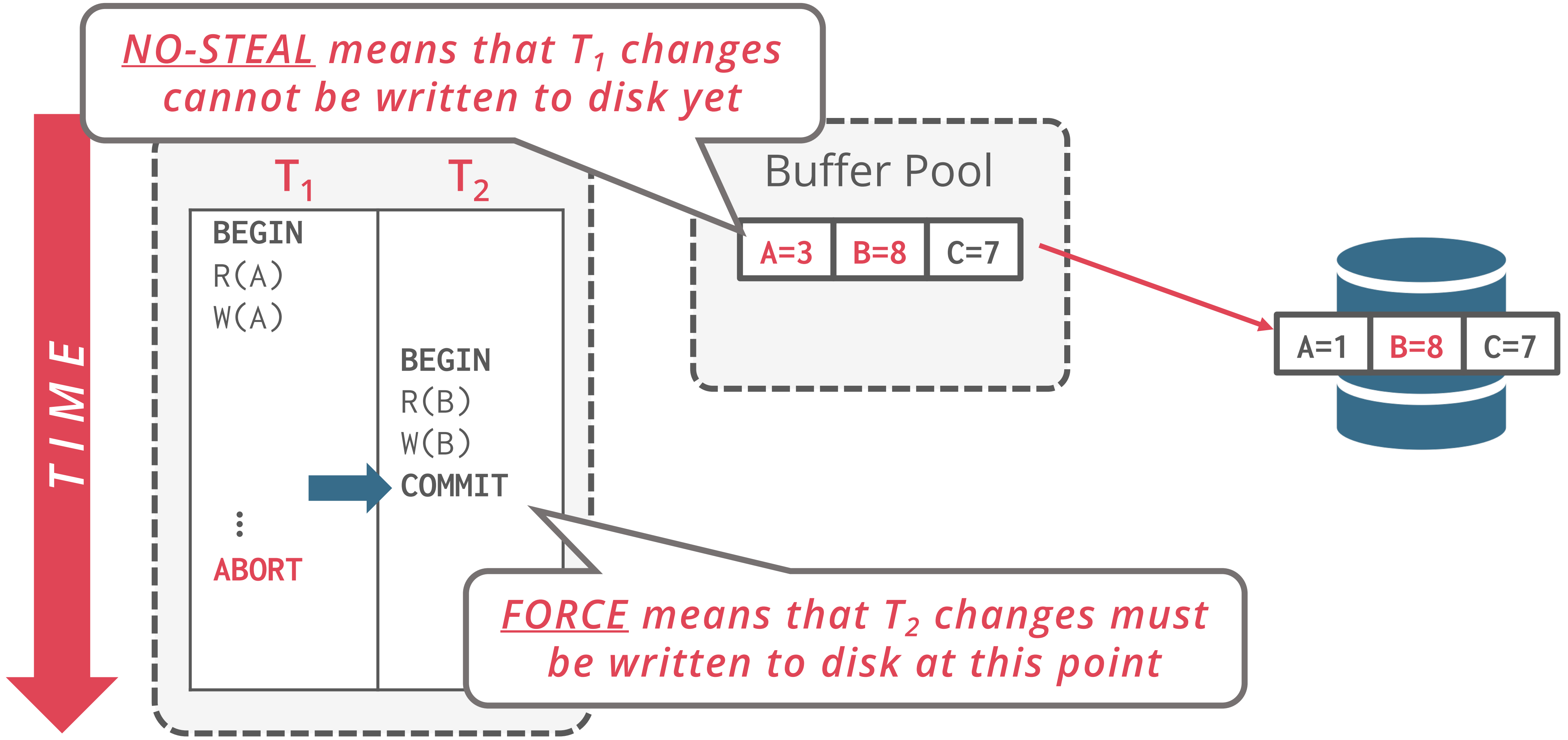


NO-STEAL + FORCE

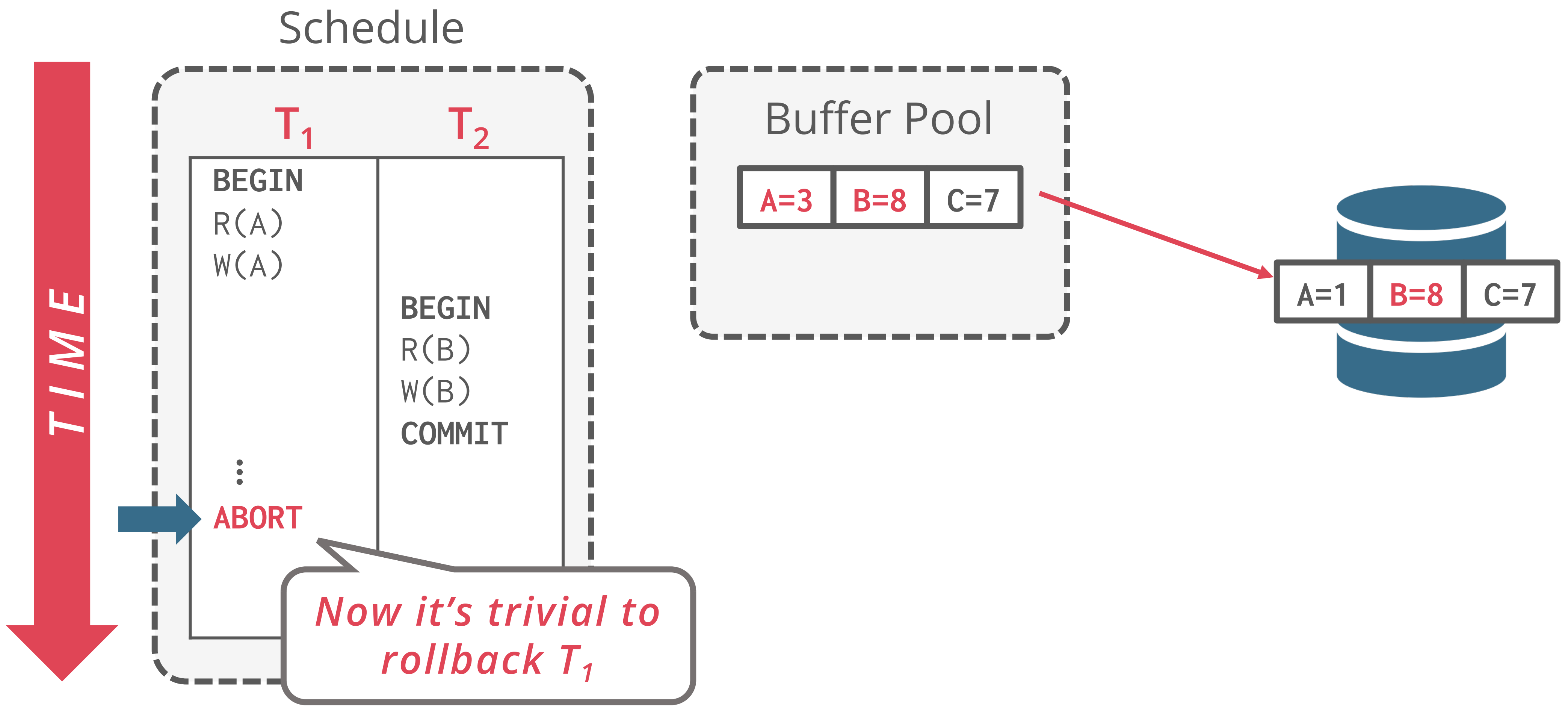


NO-STEAL means that T_1 changes cannot be written to disk yet
TIME
T_1
T_2
BEGIN
R(A)
W(A)
⋮
ABORT
BEGIN
R(B)
W(B)
COMMIT
Buffer Pool
A=3
B=8
C=7
A=1
B=9
C=7
FORCE means that T_2 changes must be written to disk at this point

No-Steal + Force


NO-STEAL means that T_1 changes cannot be written to disk yet
TIME
T_1
T_2
BEGIN
R(A)
W(A)
⋮
ABORT
BEGIN
R(B)
W(B)
COMMIT
Buffer Pool
A=3
B=8
C=7
A=1
B=8
C=7
FORCE means that T_2 changes must be written to disk at this point

NO-STEAL + FORCE



No-Steal + Force

This approach is the **easiest to implement**

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

But has **important drawbacks**

- **Poor performance**: flushing non-contiguous pages (random writes) is slow
  - Plus, what if DBMS crashes halfway through flushing? Not atomic
- **Memory requirements**: NO-STEAL assumes that all pages modified by uncommitted transactions can be accommodated in the buffer pool

More on Steal and Force

STEAL: Why enforcing atomicity is hard?

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

NO-FORCE: Why enforcing durability is hard?

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

Buffer Pool Policies

Runtime Performance

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

Recovery Performance

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

Undo + Redo

No Undo + No Redo

Undo: removing the effects of an incomplete or aborted txn

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

Almost every DBMS uses STEAL + NO-FORCE

Basic Idea: Logging

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

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

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

Write-Ahead Logging (WAL)

Before making a change in the database, record the change in a log file

- The DBMS stages all log records of a txn in memory (usually backed by buffer pool)

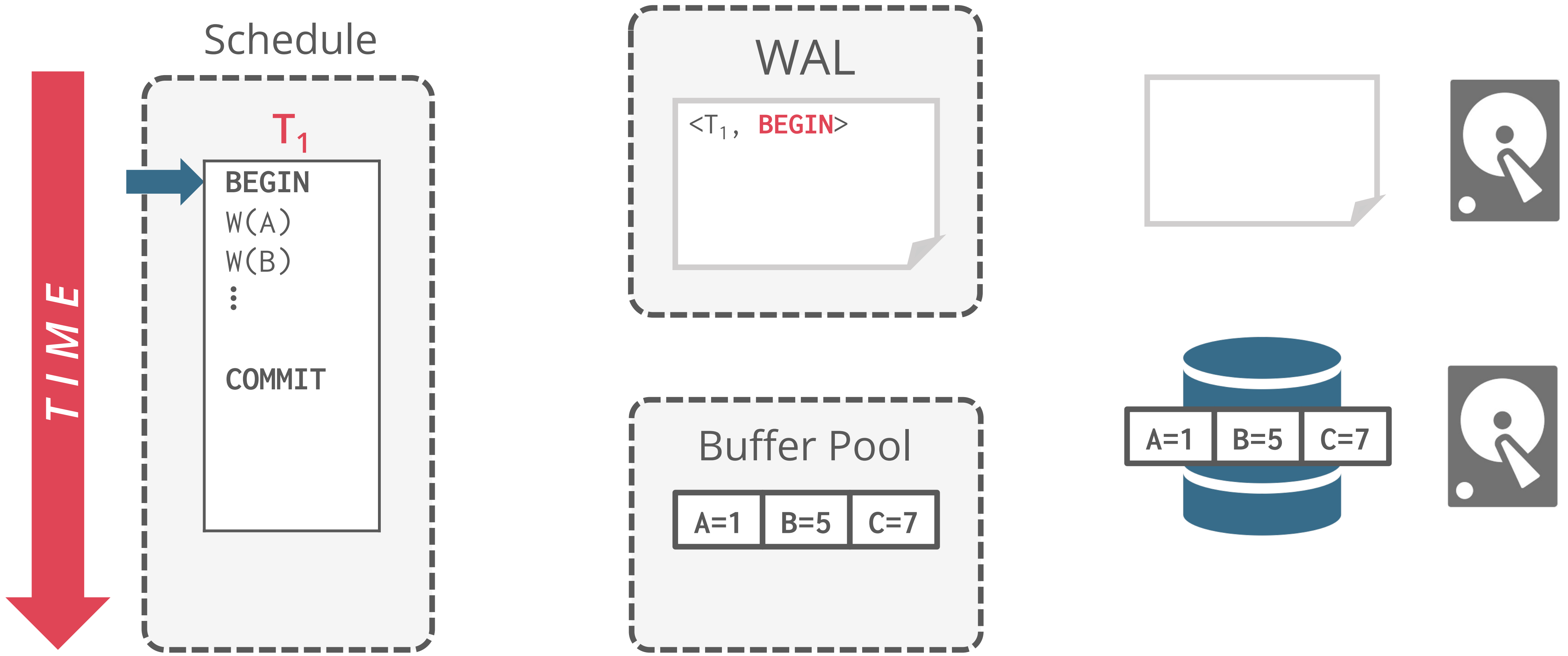
All log records pertaining to an updated page must be written to non-volatile storage **before** the page itself is overwritten to non-volatile storage

- The log records contain UNDO info ⇒ can exploit to guarantee Atomicity

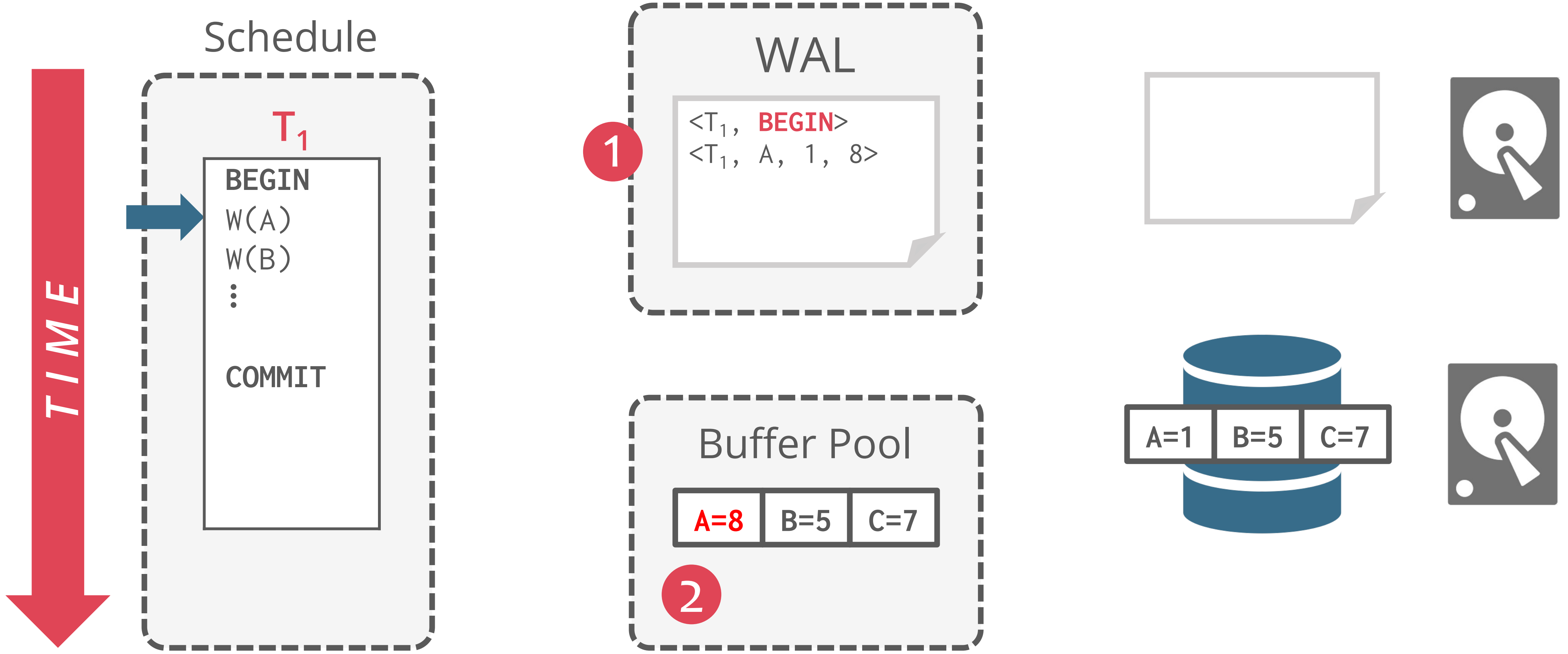
A txn is not considered committed until **all** of its log records including its “commit” record are written to non-volatile storage

- The log records contain REDO info ⇒ can exploit to guarantee Durability

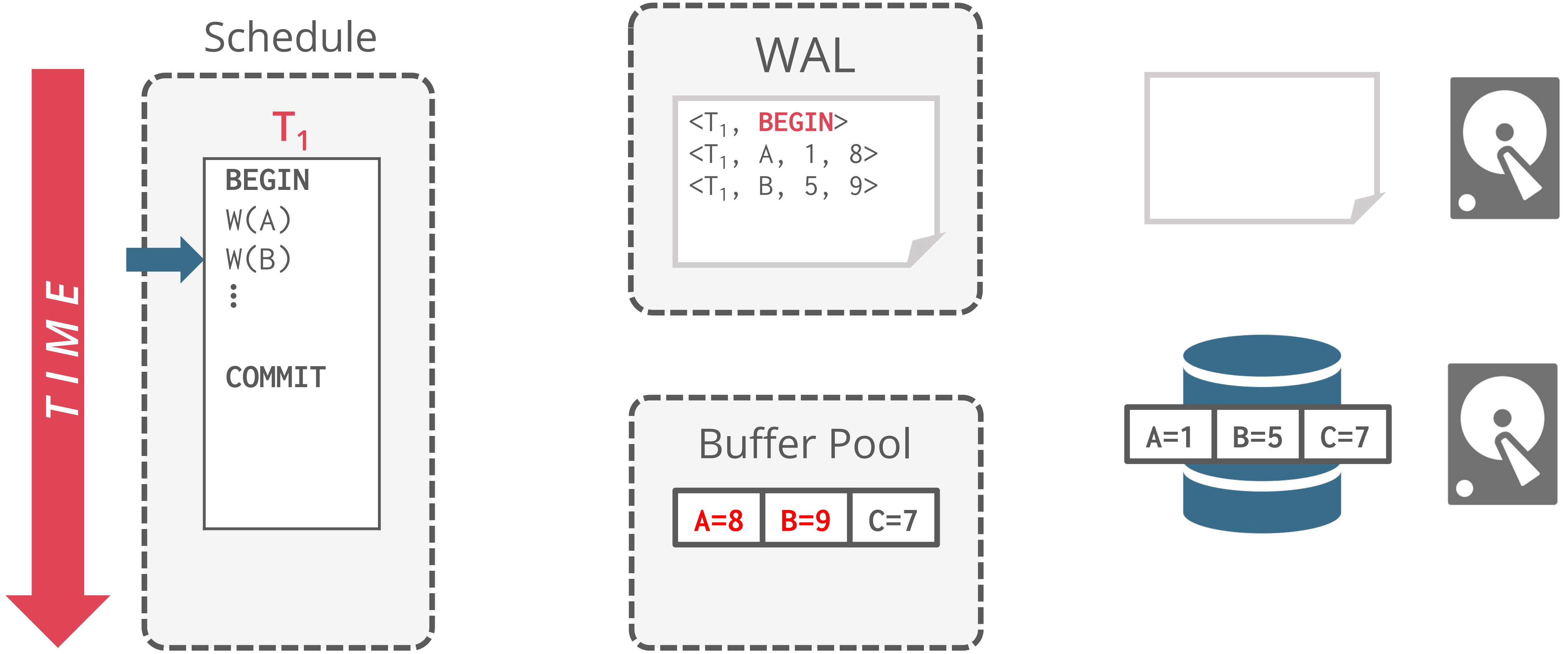
WAL – Example


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=5
C=7

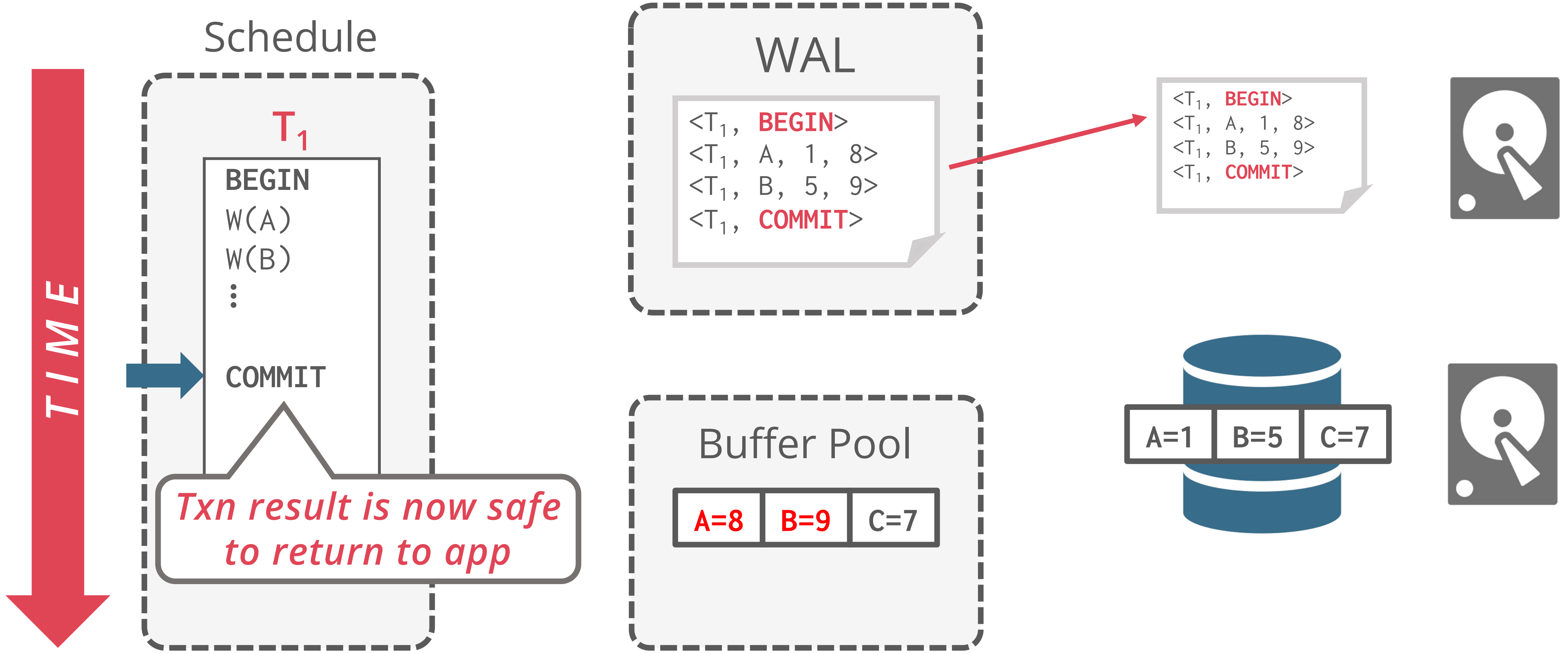
WAL – Example


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

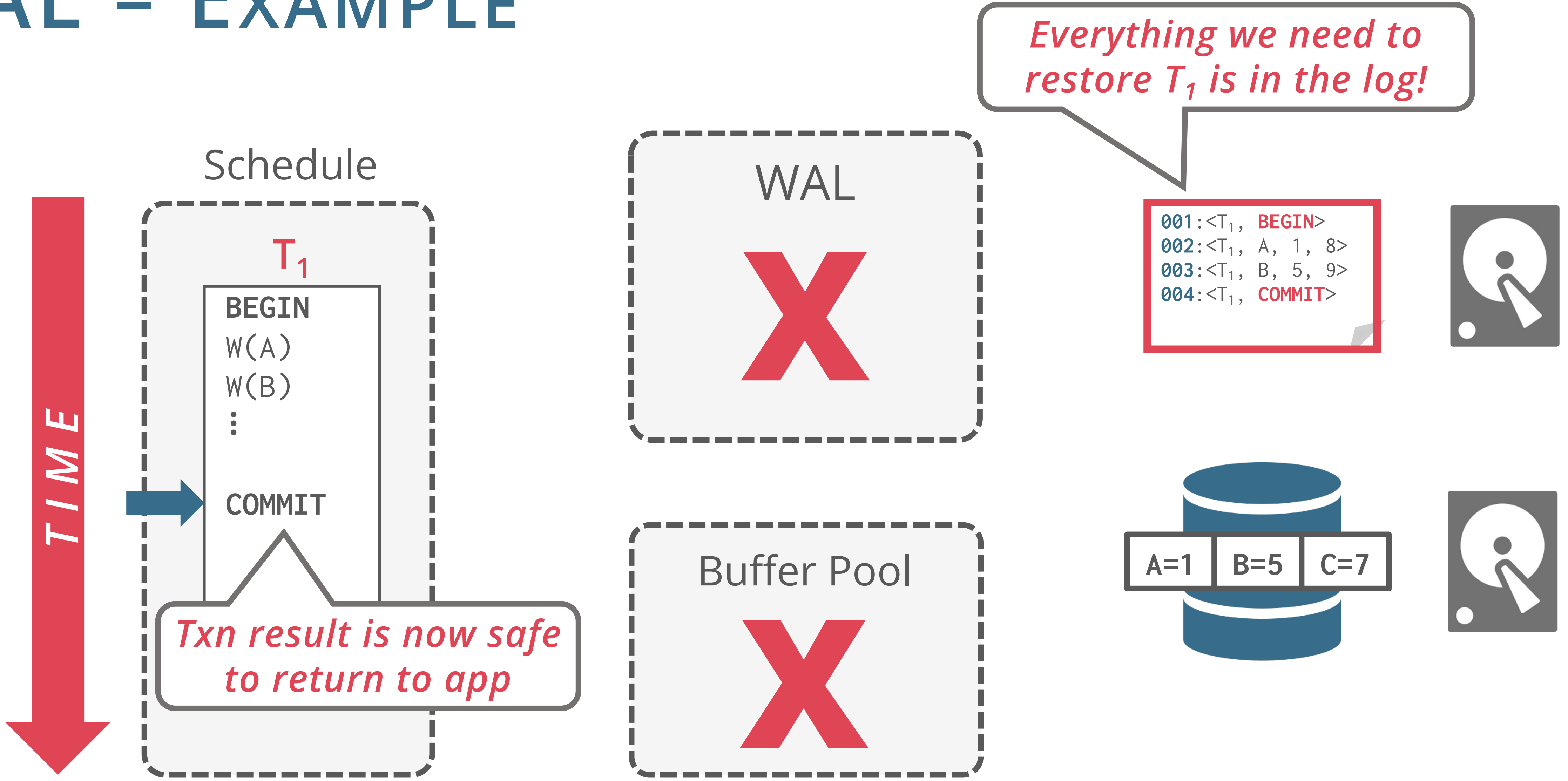
WAL – EXAMPLE


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

WAL – EXAMPLE


Everything we need to restore T_1 is in the log!
Schedule
TIME
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
Txn result is now safe to return to app
WAL
Buffer Pool
A=1
B=5
C=7

```
001:<T1, BEGIN>
002:<T1, A, 1, 8>
003:<T1, B, 5, 9>
004:<T1, COMMIT>
```

ARIES

Recovery algorithm developed at IBM Research in early 1990s

Write-Ahead Logging

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

Recovery in three phases:

- **Analyse:** identify active txns and dirty pages at the time of crash
- **Redo:** repeat history to restore exact state just before the crash
- **Undo**: rollback all uncommitted txns

ARIES – Recovery Phases

Phase #1 – Analysis

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

Phase #2 – Redo

Repeat <u>all</u> actions starting from an appropriate point in the log (even txns that will abort)

Phase #3 – Undo

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

SUMMARY

Recovery Manager guarantees Atomicity & Durability

- Supports rollback to guarantee consistency

Use WAL to allow **STEAL** + **NO-FORCE** w/o sacrificing correctness

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