CS-300: Data-Intensive Systems

Logging and Recovery

Prof. Anastasia Ailamaki, ***Prof. Sanidhya Kashyap***

EPFL

Today's focus

- Logging
  - Buffer pool policies
  - WAL
  - Logging schemes
- Recovery
  - LSN
  - Normal checkpoint and abort operations
  - Checkpoint
  - Recovery algorithm

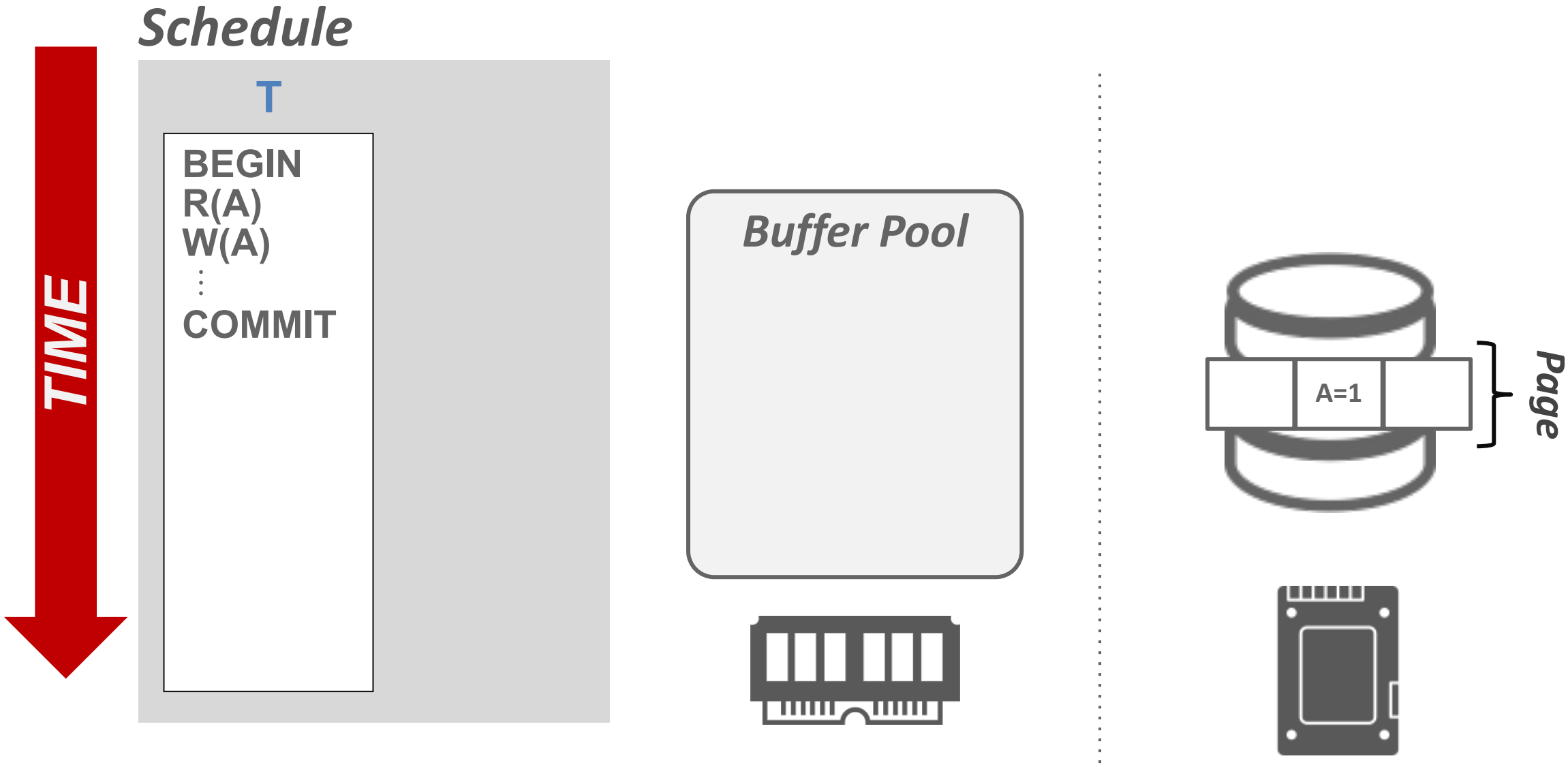
Until now …

- Concurrency control protocol provides:

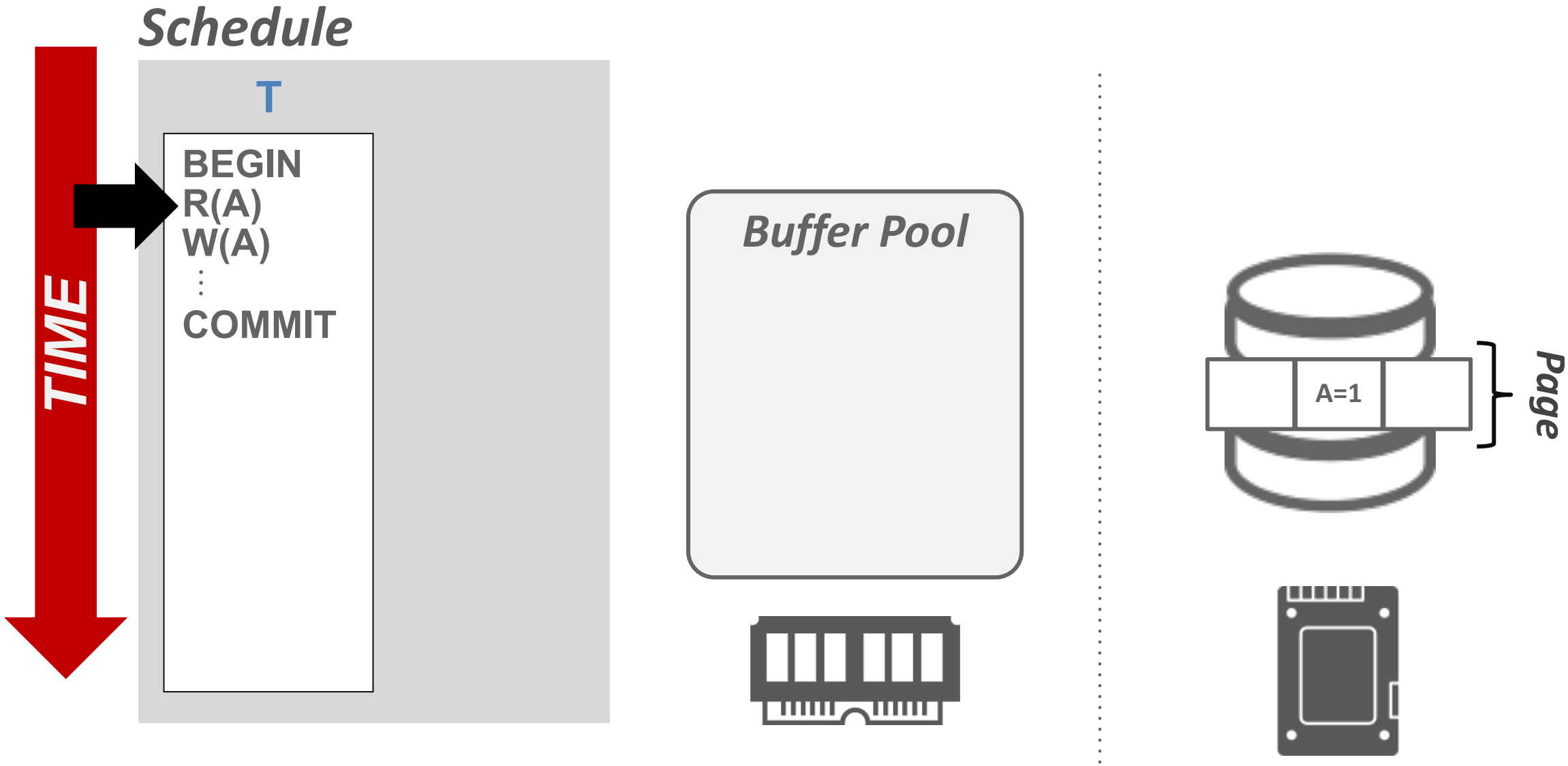
  Atomicity + Consistency + Isolation

- We now need to ensure **Atomicity + Durability**

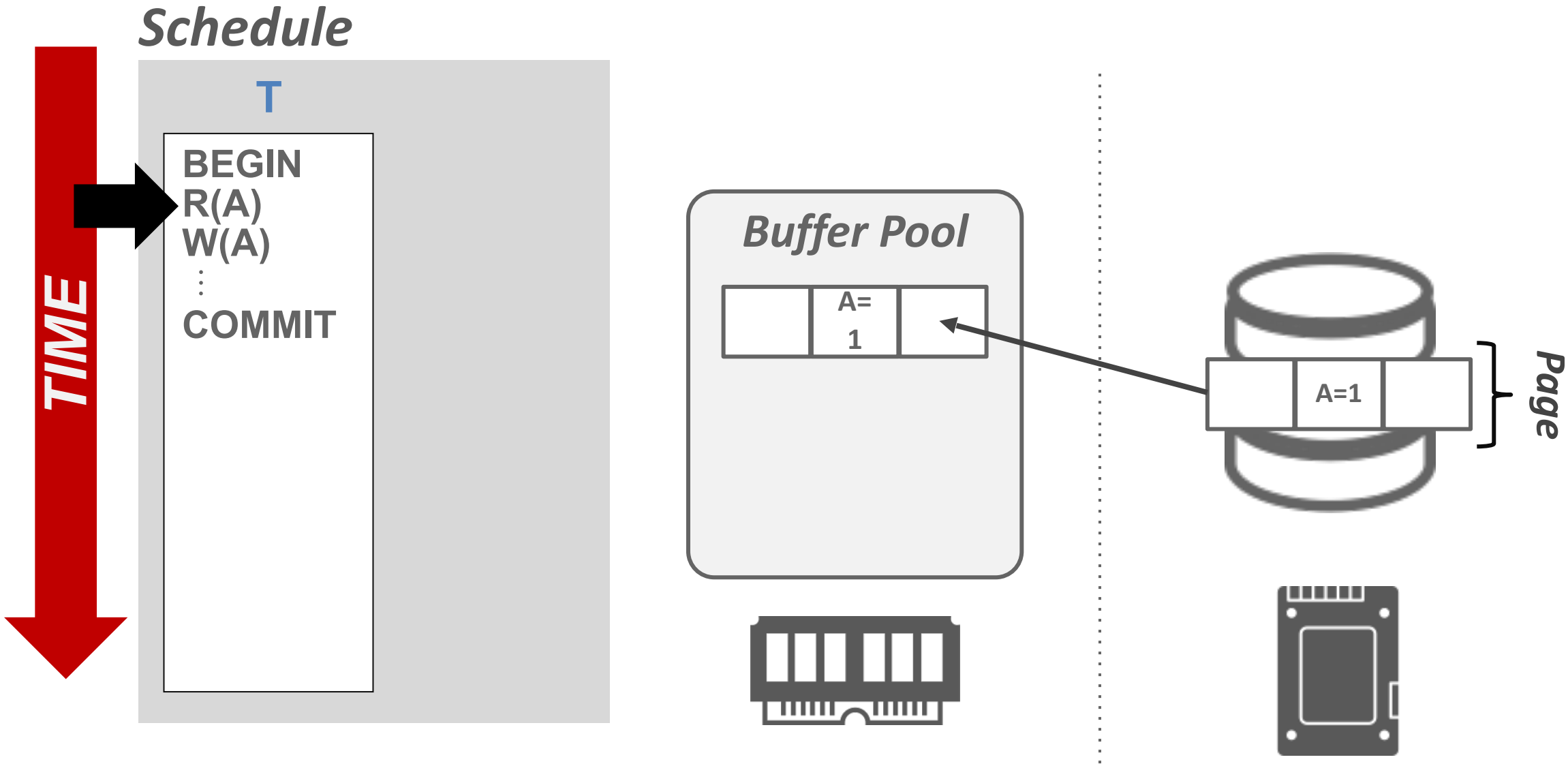
Motivation



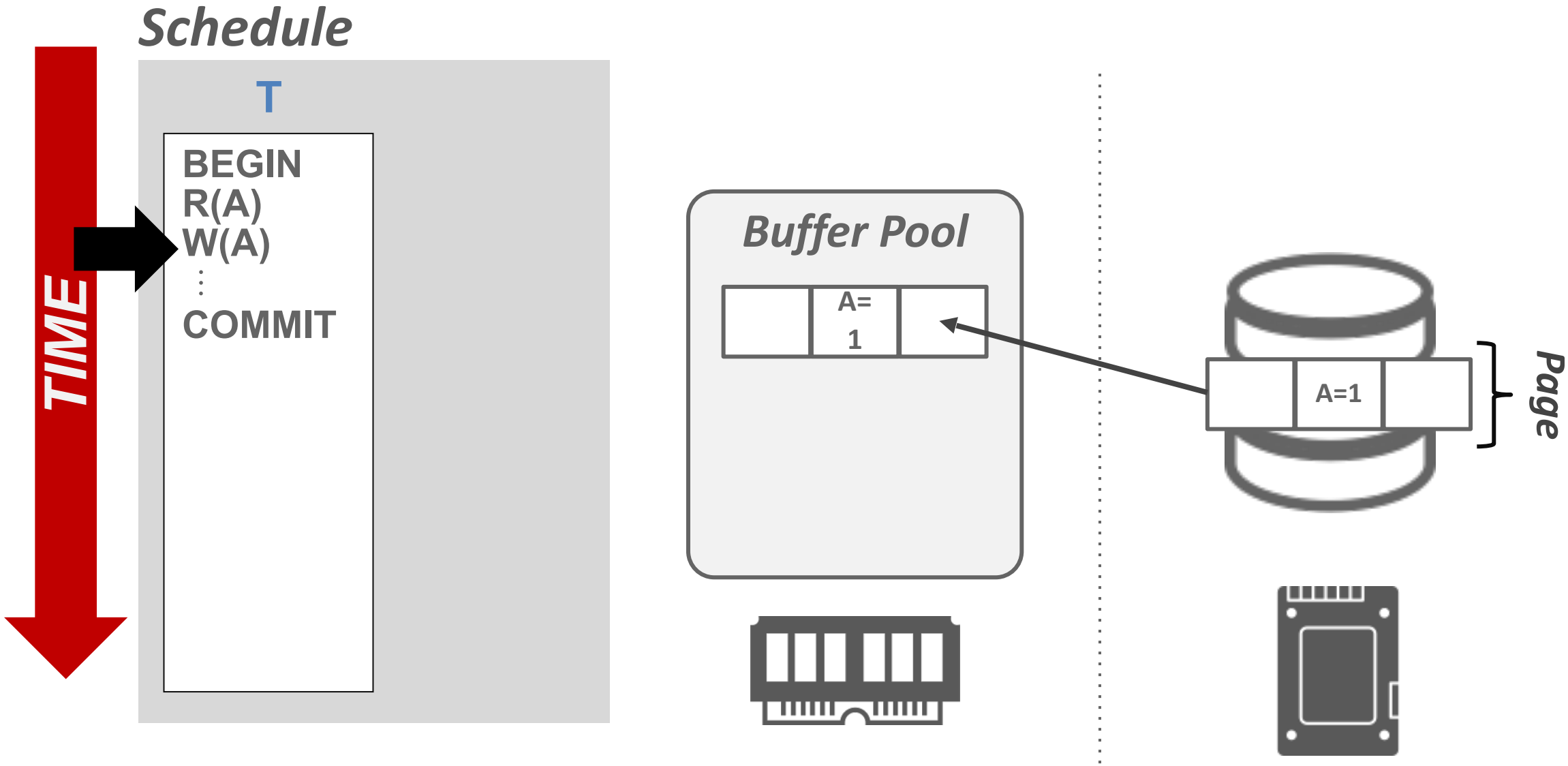
Motivation



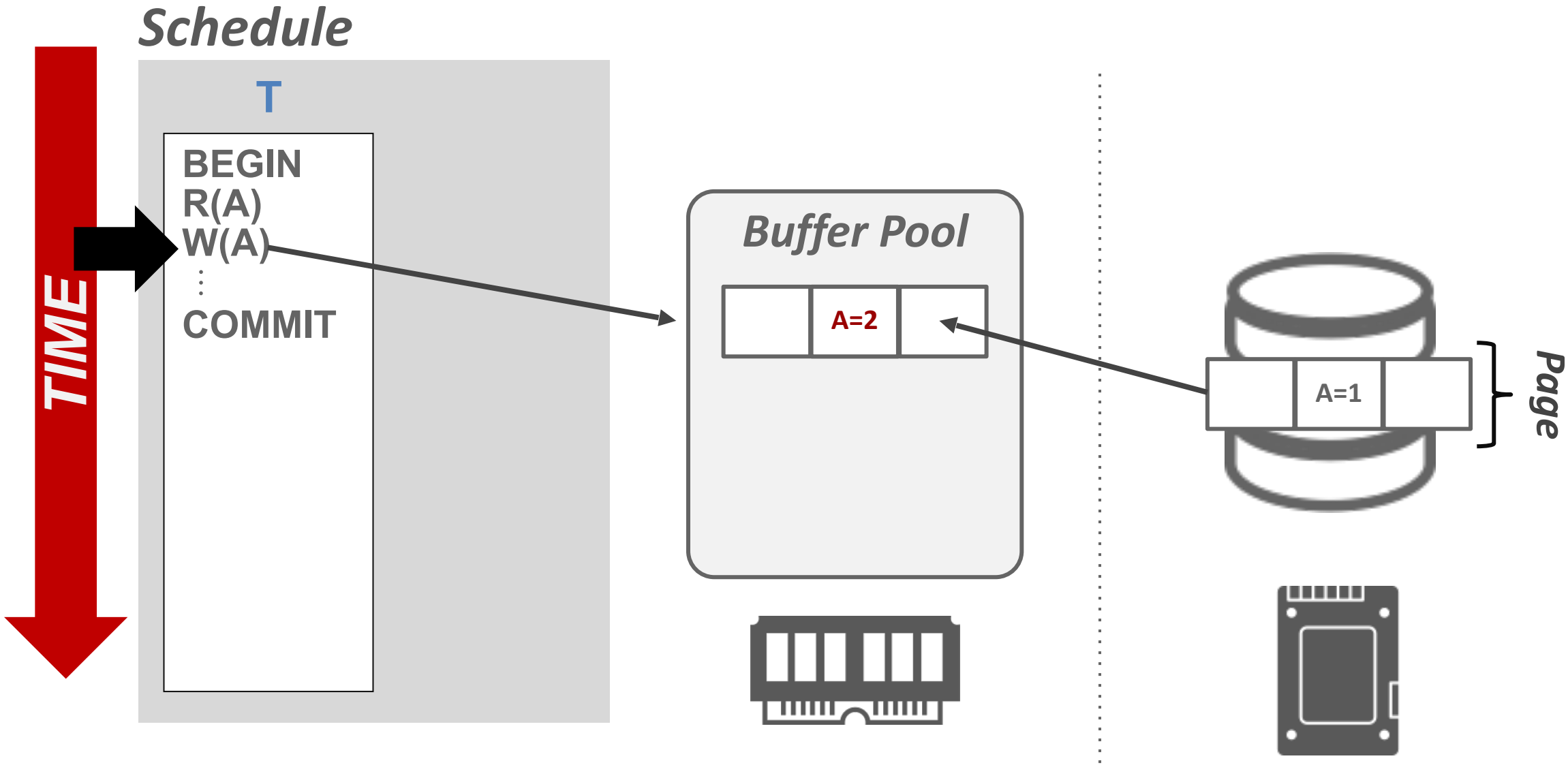
Motivation



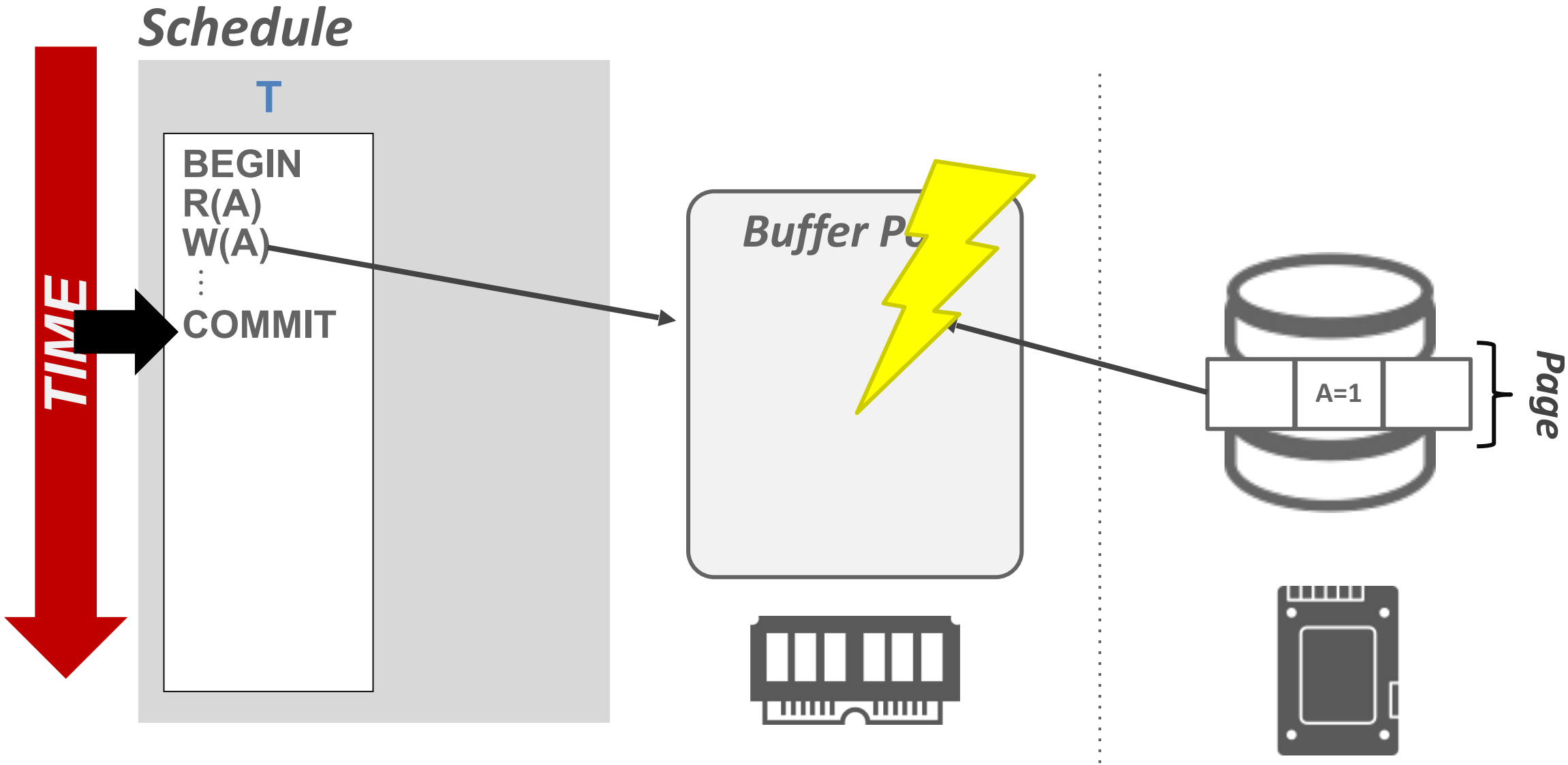
Motivation



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 → **preparing for the failure**
  - Actions after a failure to recover the database to a state that ensures atomicity, consistency, and durability → **handling the failure**

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 → **preparing for the failure**
  - Actions after a failure to recover the database to a state that ensures atomicity, consistency, and durability → **handling the failure**

Observation

- DB'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 write operations 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 committed it
  - No partial changes are durable if the txn is aborted

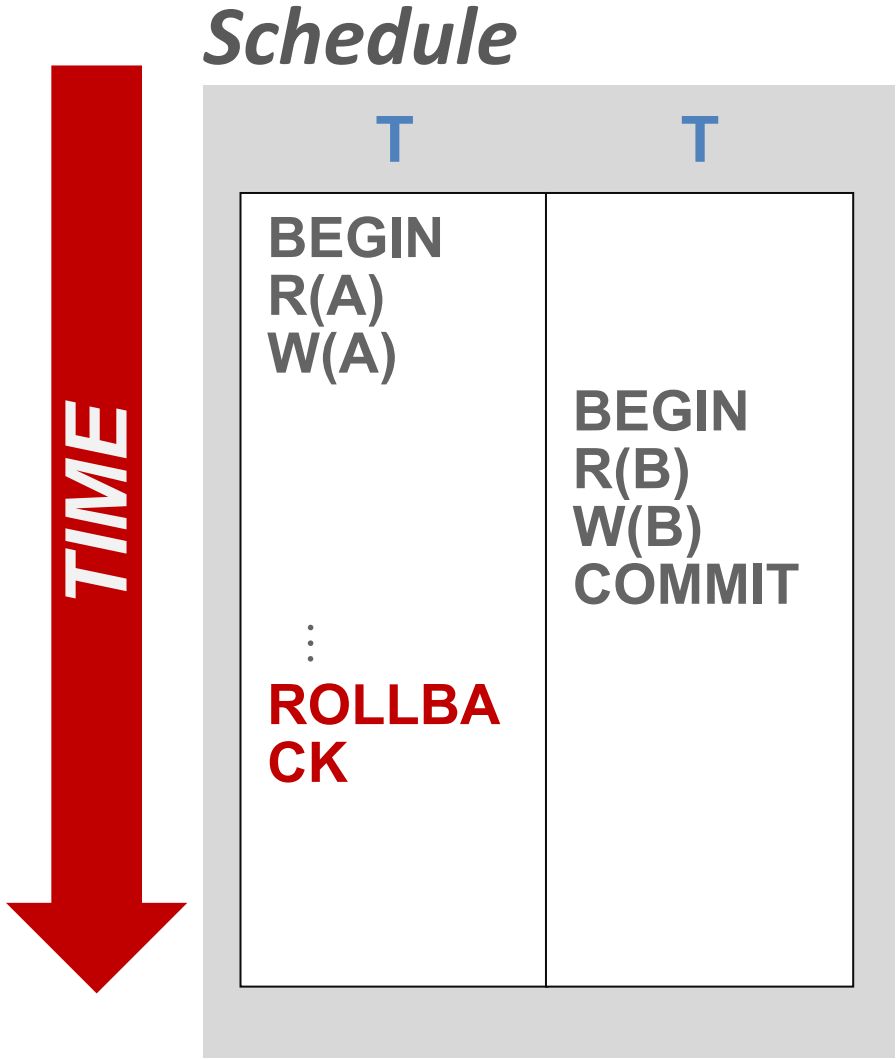
Two key primitives: Undo vs. Redo

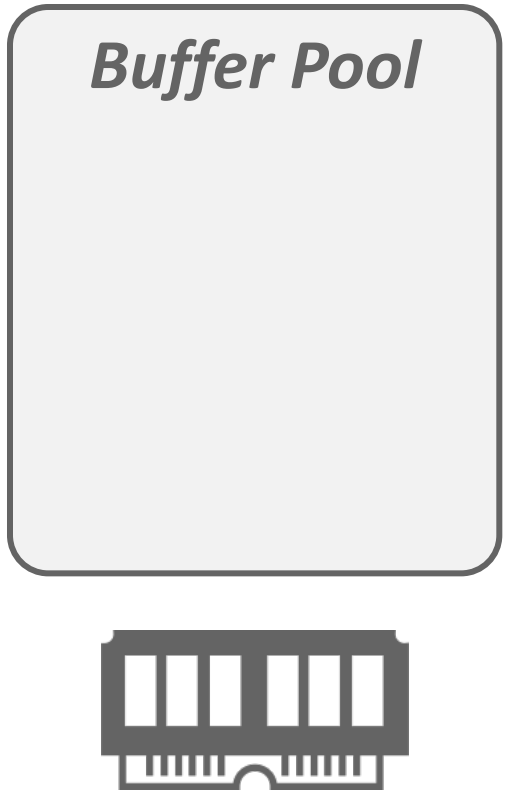
Undo: The process of removing the effects of an incomplete or aborted txn

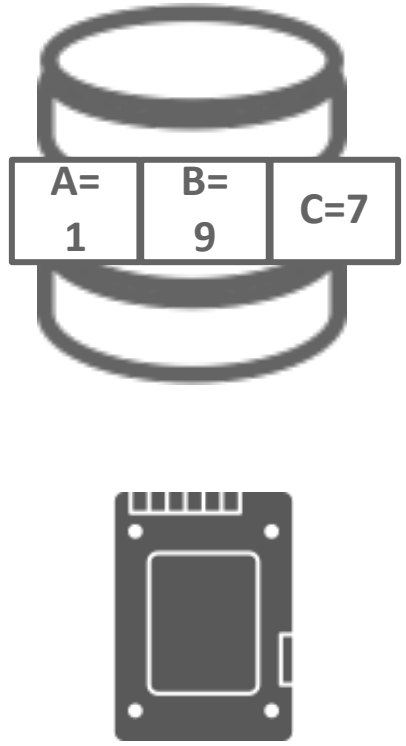
Redo: The process of re-applying the effects of a committed txn for durability

→ This functionality depends on how DBMS manages the buffer pool

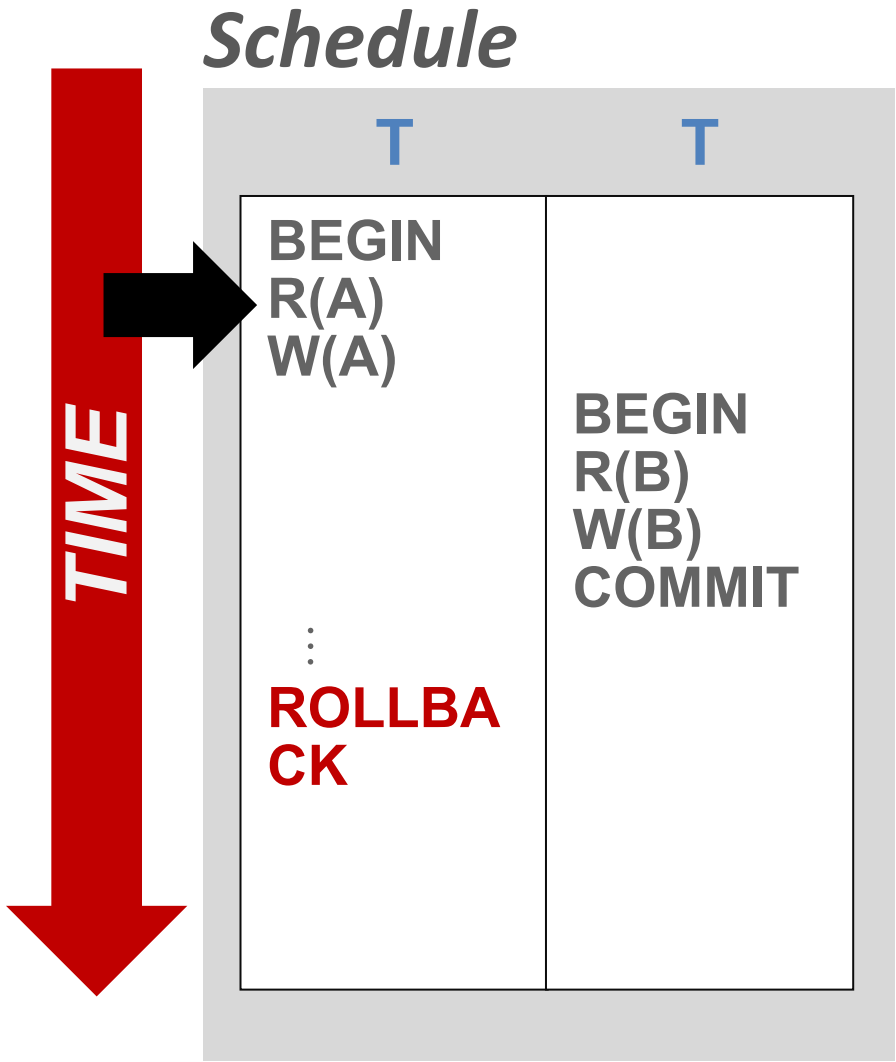
Buffer pool


Schedule
TIME
T
T
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
ROLLBA
CK

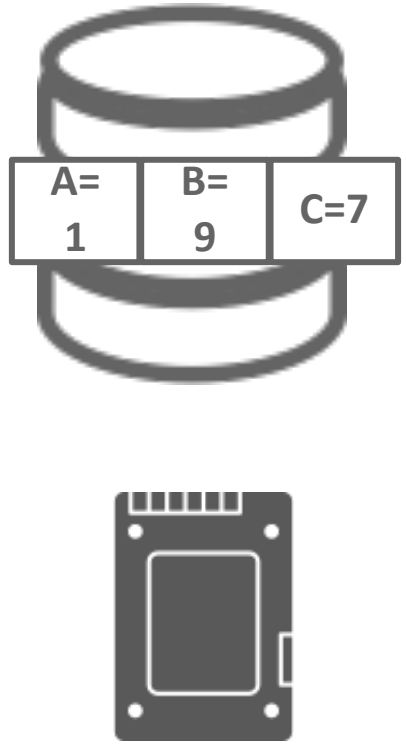

Buffer Pool


A=
1
B=
9
C=7

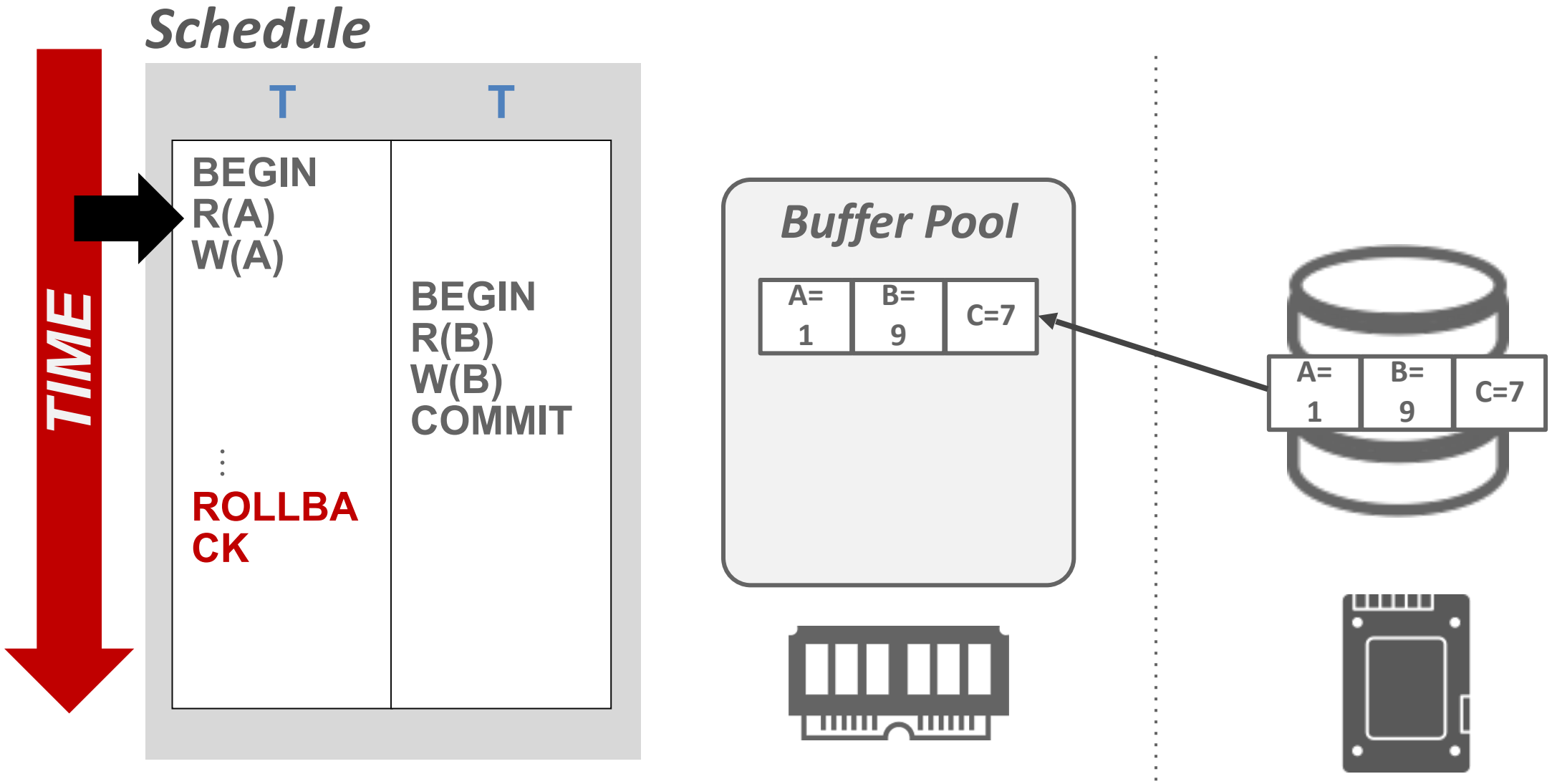
Buffer pool


Schedule
TIME
T
T
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
ROLLBA
CK

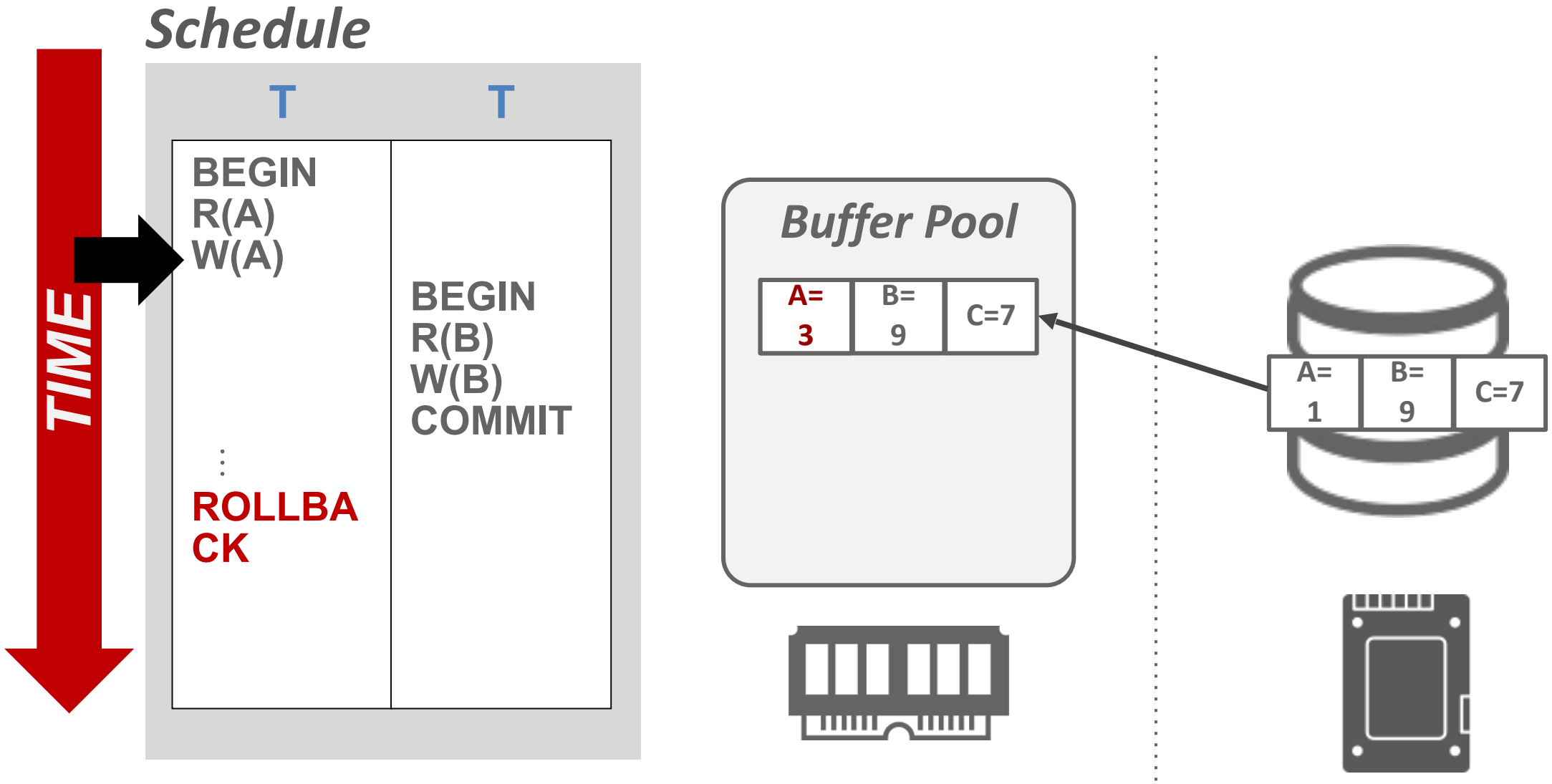

Buffer Pool


A=
1
B=
9
C=7

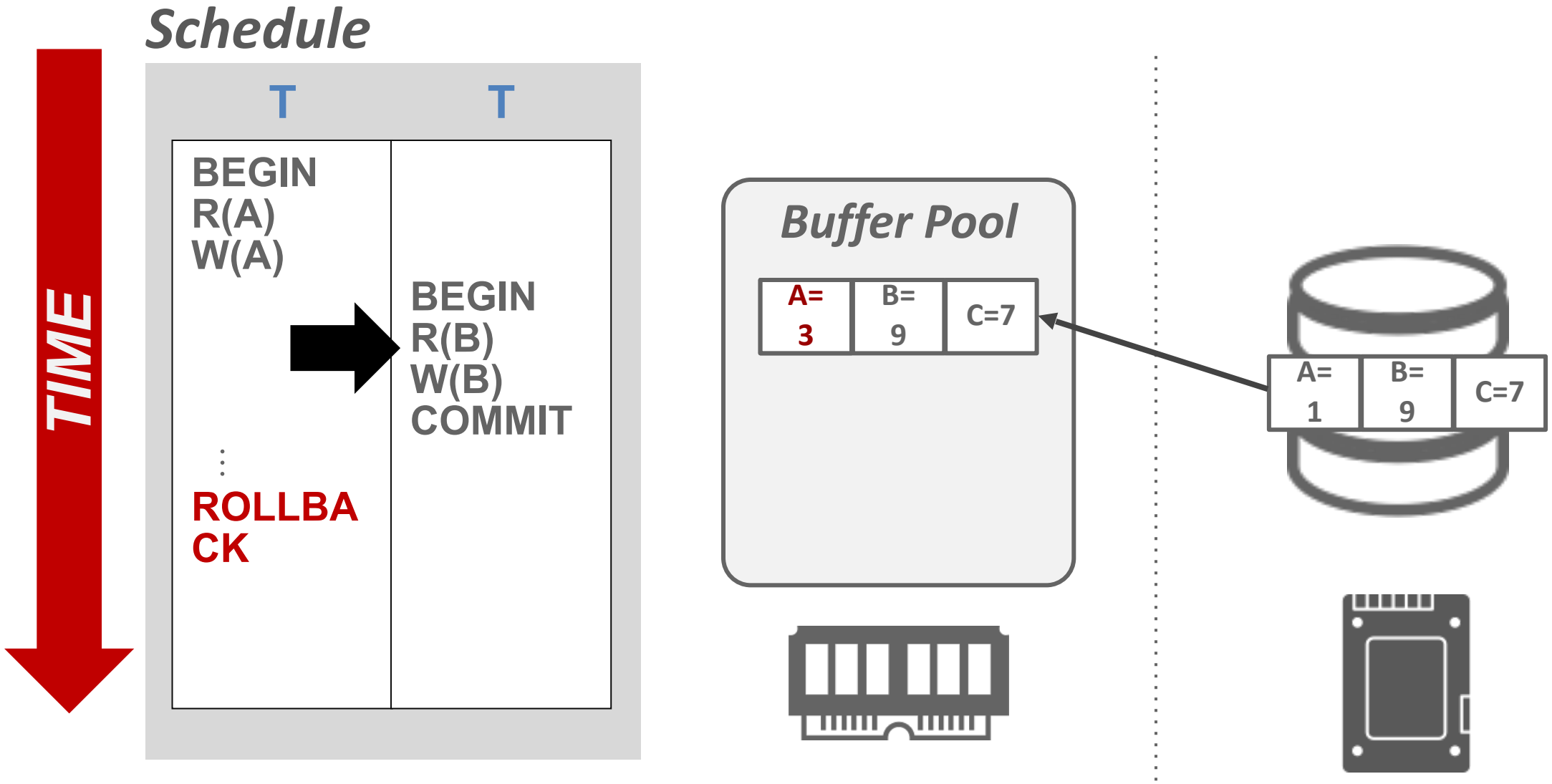
Buffer pool


Schedule
TIME
T
T
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
ROLLBA
CK
Buffer Pool
A=
1
B=
9
C=7
A=
1
B=
9
C=7

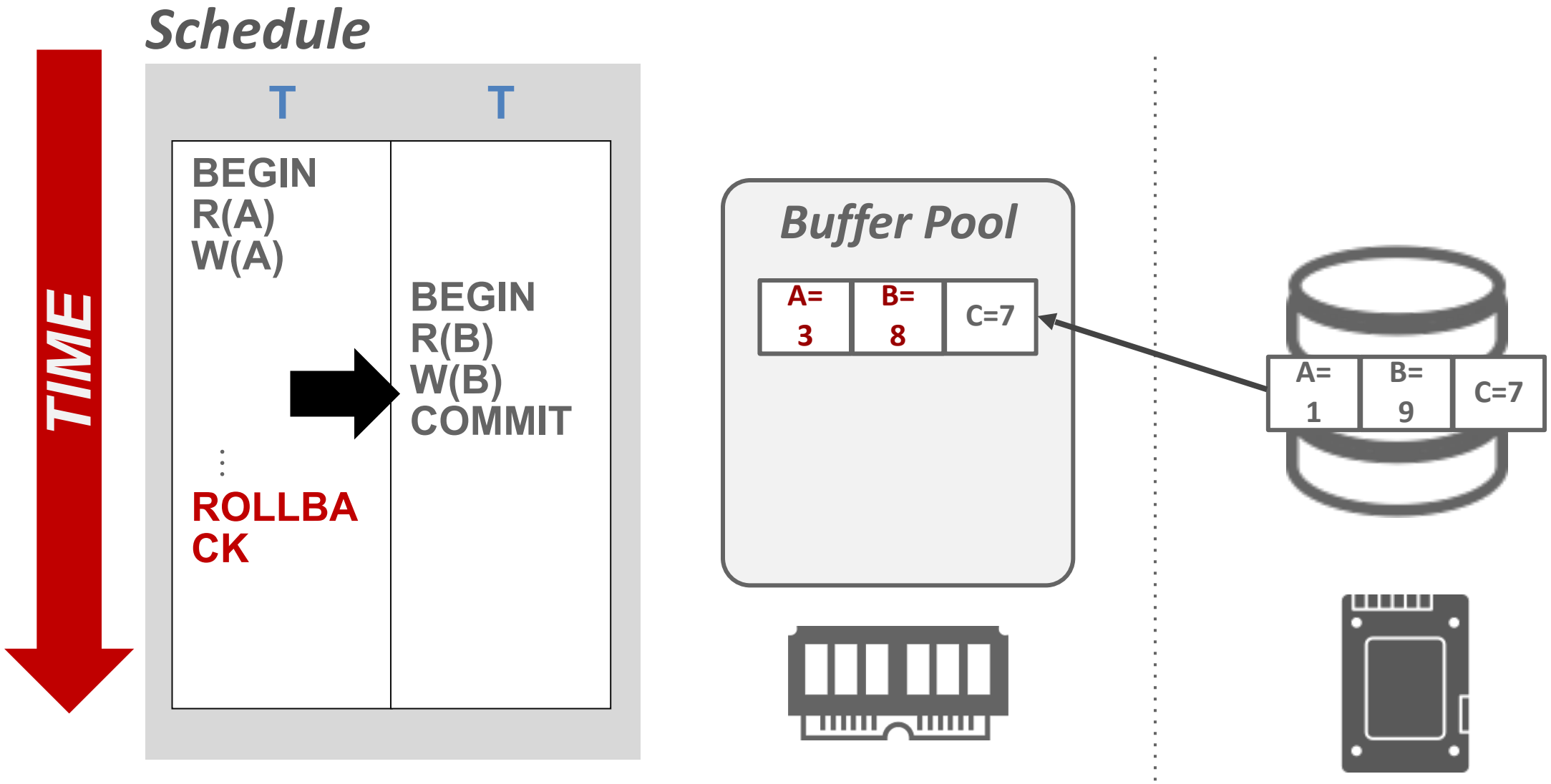
Buffer pool


Schedule
TIME
T
T
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
ROLLBA
CK
Buffer Pool
A=
3
B=
9
C=7
A=
1
B=
9
C=7

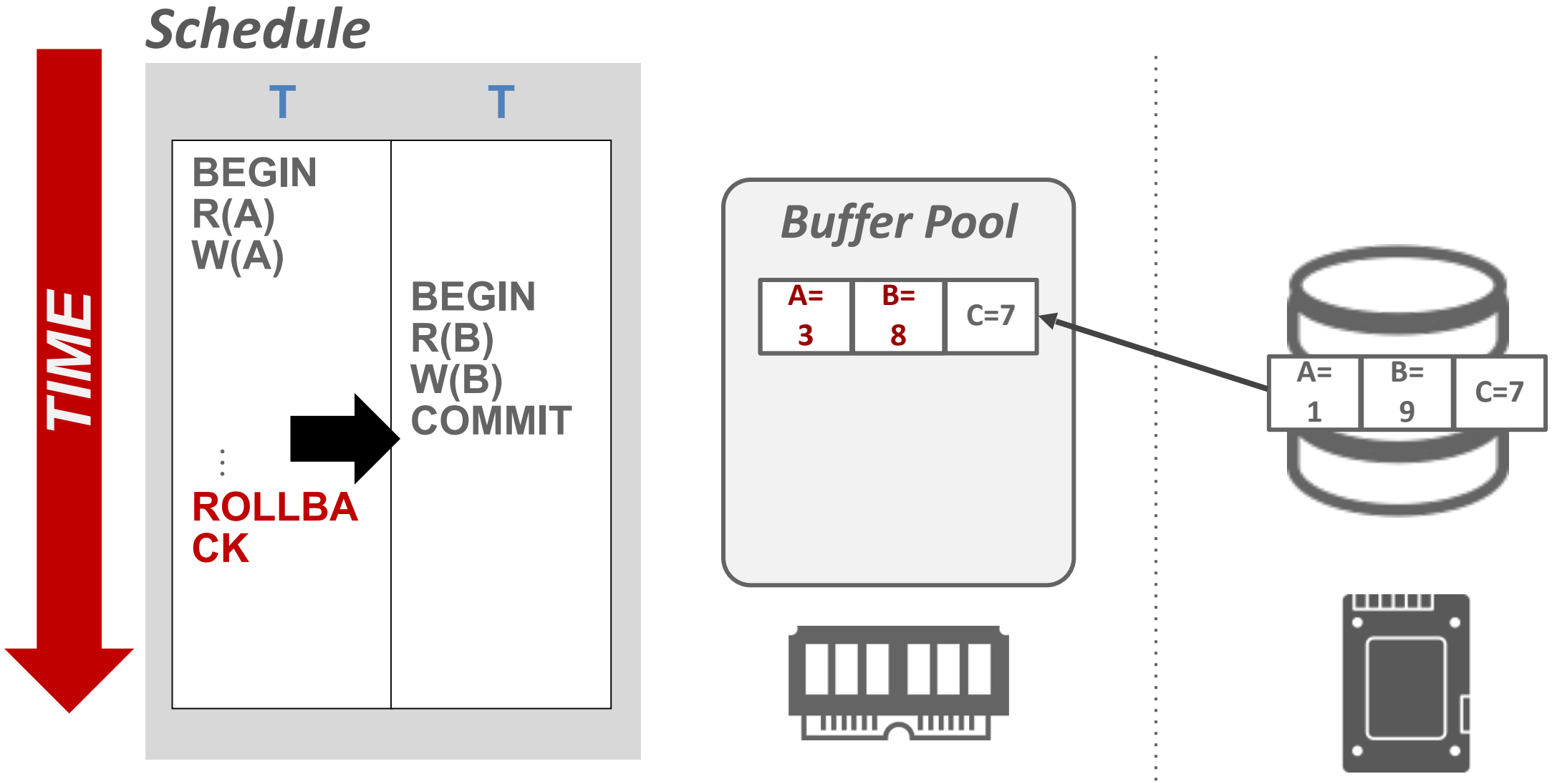
Buffer pool


Schedule
TIME
T
T
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
ROLLBA
CK
Buffer Pool
A=
3
B=
9
C=7
A=
1
B=
9
C=7

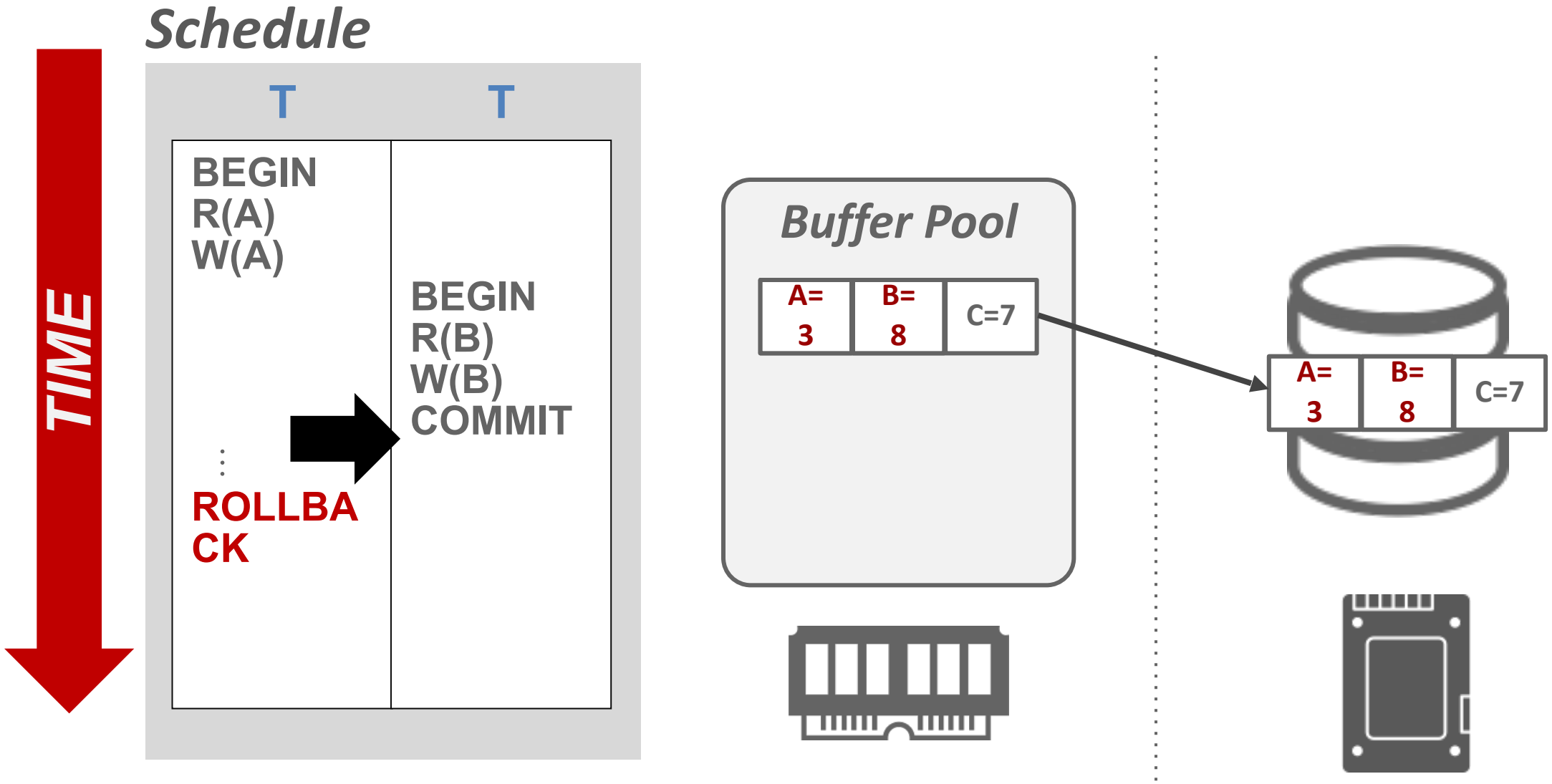
Buffer pool


Schedule
TIME
T
T
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
ROLLBA
CK
Buffer Pool
A=
3
B=
8
C=7
A=
1
B=
9
C=7

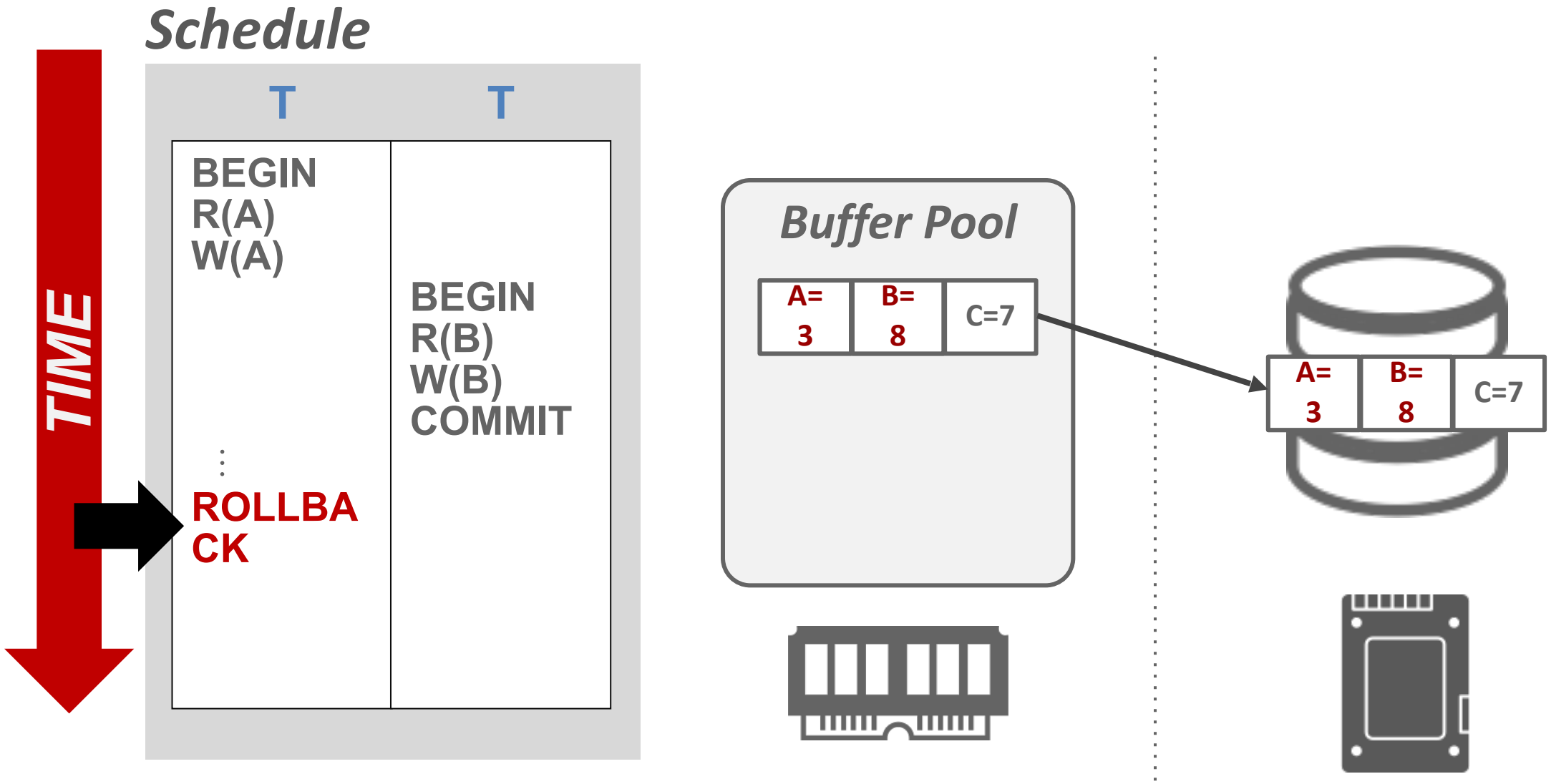
Buffer pool


Schedule
TIME
T
T
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
ROLLBA
CK
Buffer Pool
A=
3
B=
8
C=7
A=
1
B=
9
C=7

Buffer pool


Schedule
TIME
T
T
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
ROLLBA
CK
Buffer Pool
A=
3
B=
8
C=7
A=
3
B=
8
C=7

Buffer pool


Schedule
TIME
T
T
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
ROLLBA
CK
Buffer Pool
A=
3
B=
8
C=7
A=
3
B=
8
C=7

Steal policy

- A DBMS can **evict** dirty objects in the buffer pool modified by an uncommitted txn
- DBMS then **overwrite** the most recent committed version of that object in non-volatile storage

→ Buffer manager steals the page from the uncommitted transaction

Steal: Eviction + overwriting is allowed

No-steal: Eviction + overwriting is not allowed

- Only committed data is written to non-volatile storage

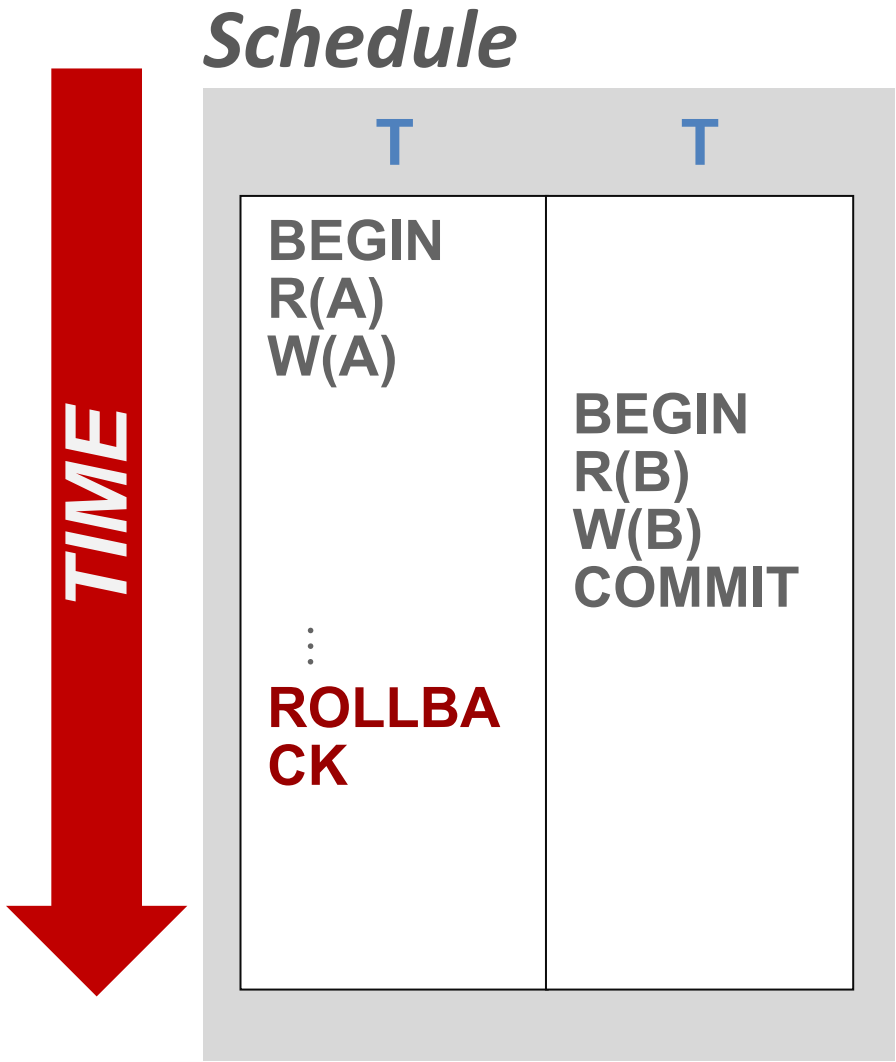
Force policy

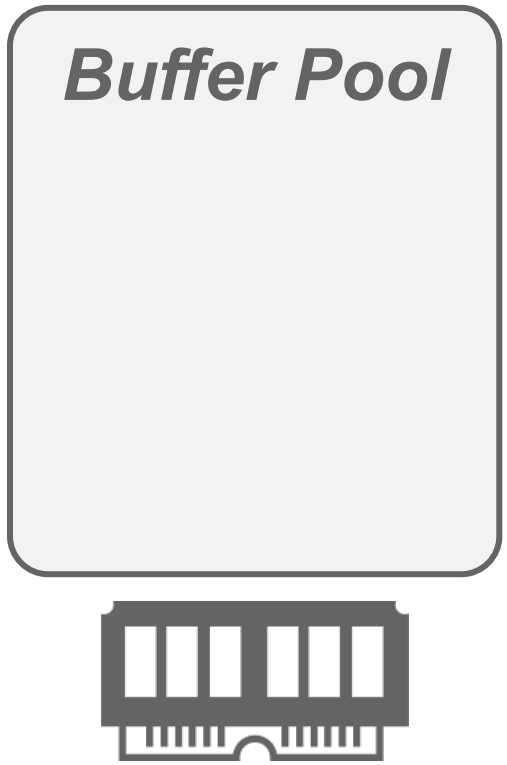
- Dictates whether a DBMS requires all updates made by a txn are written back to non-volatile storage **before** a txn can commit

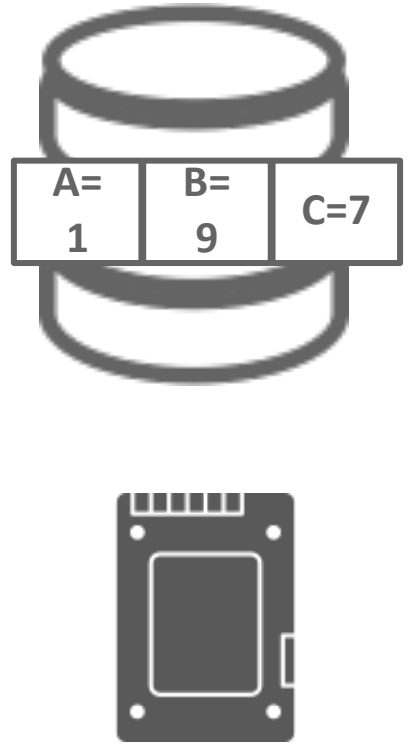
Force: Write-back is required

No-force: Write-back is not required

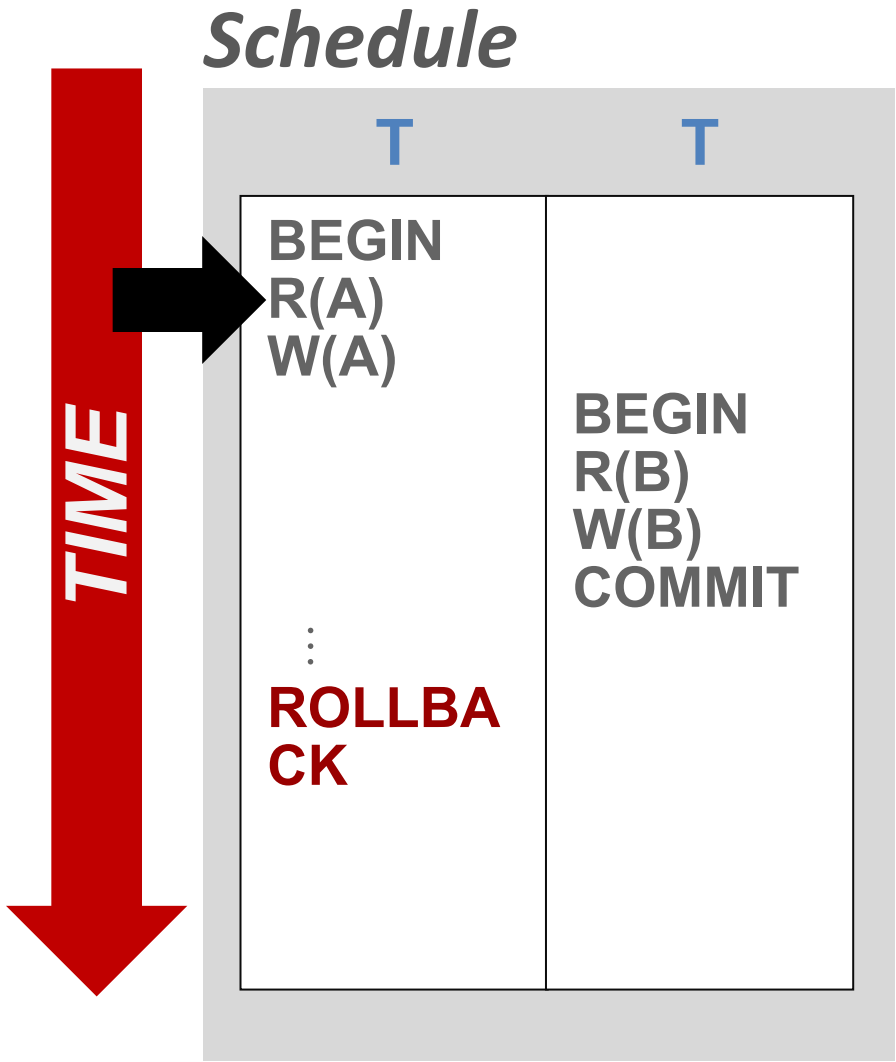
No-steal + force

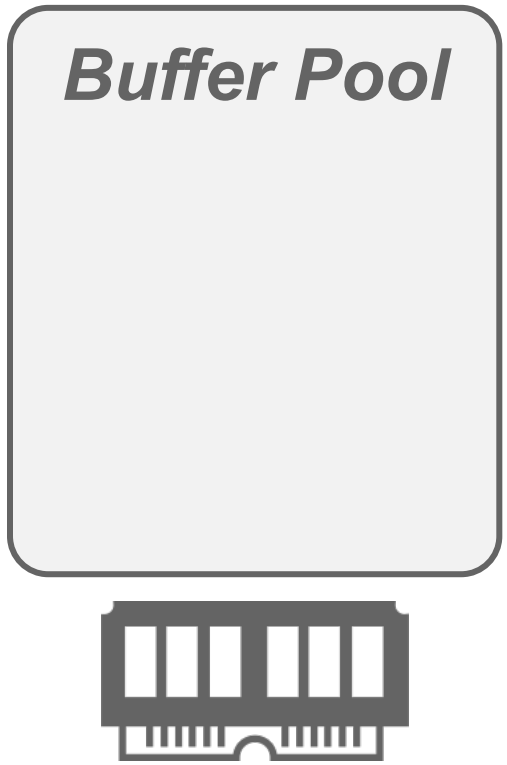


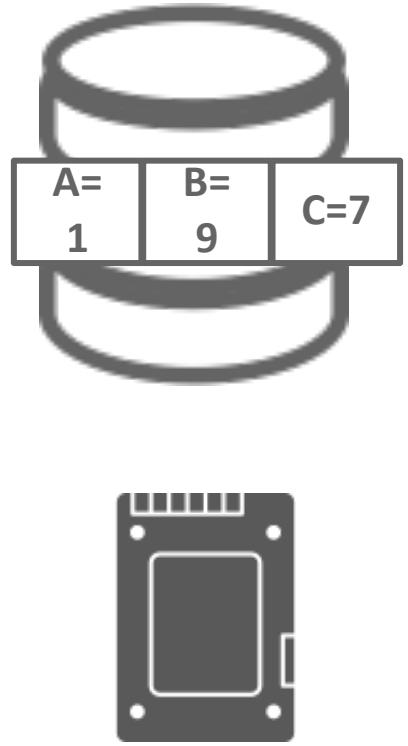




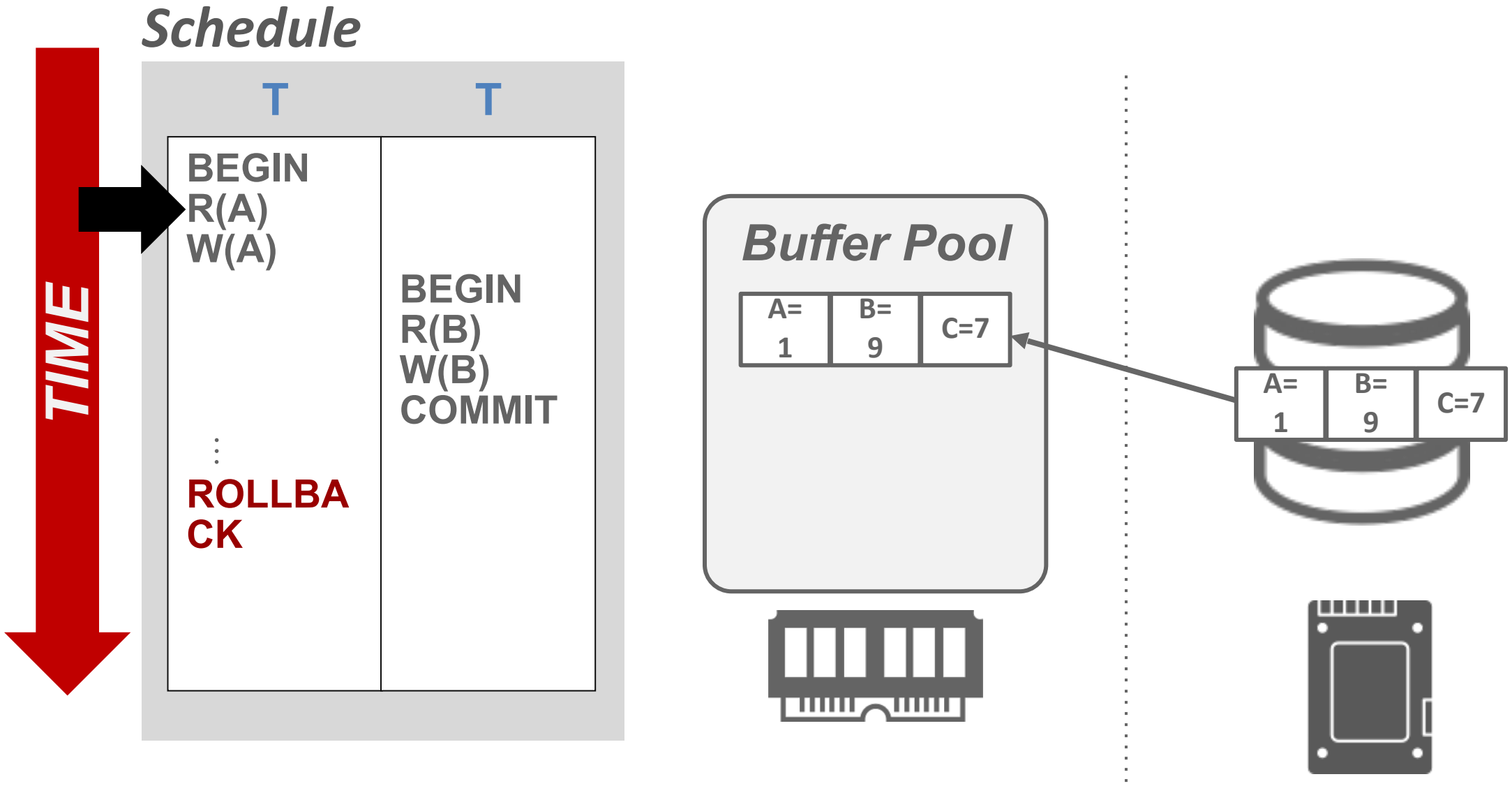
No-steal + force



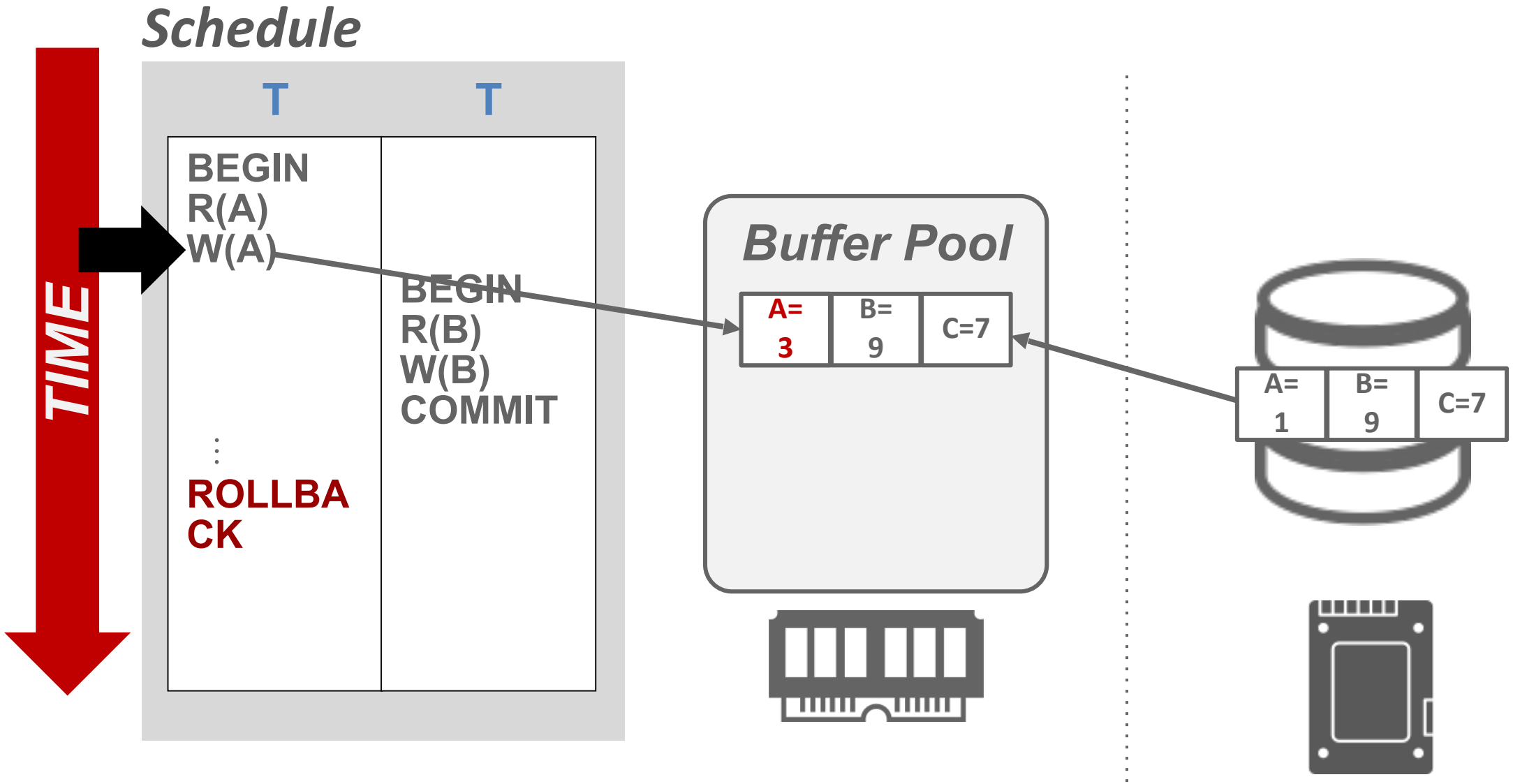




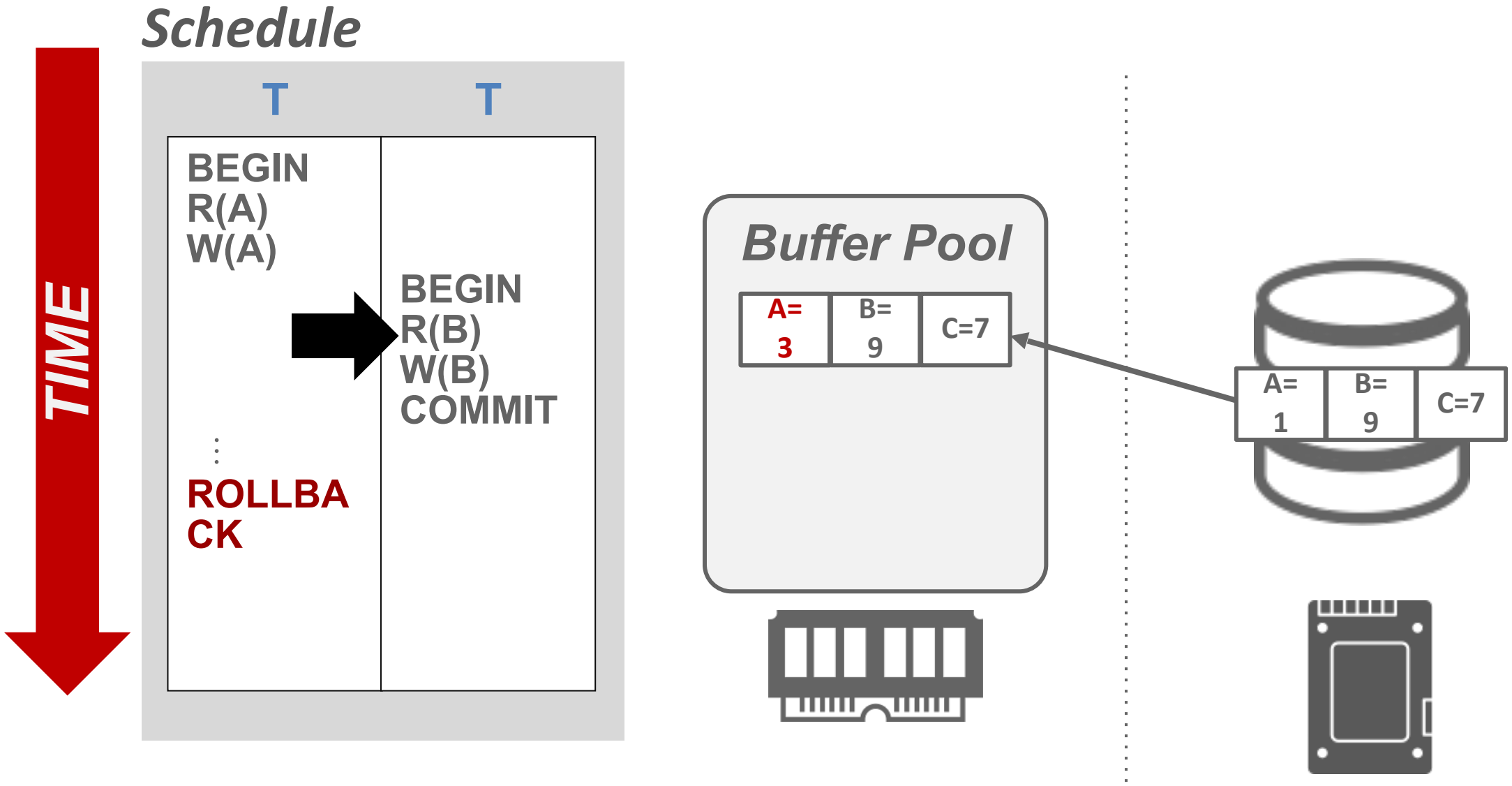
No-steal + force



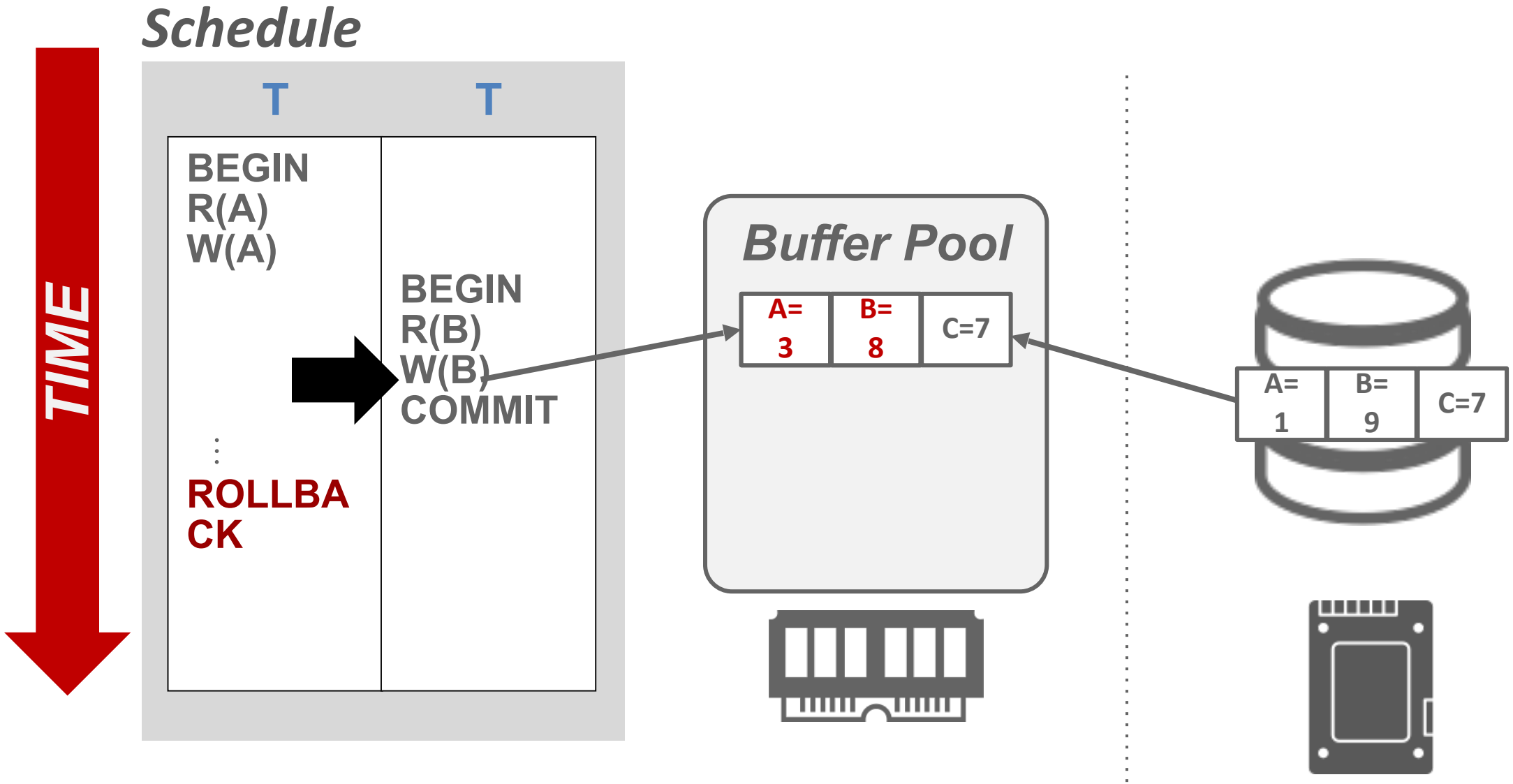
No-steal + force



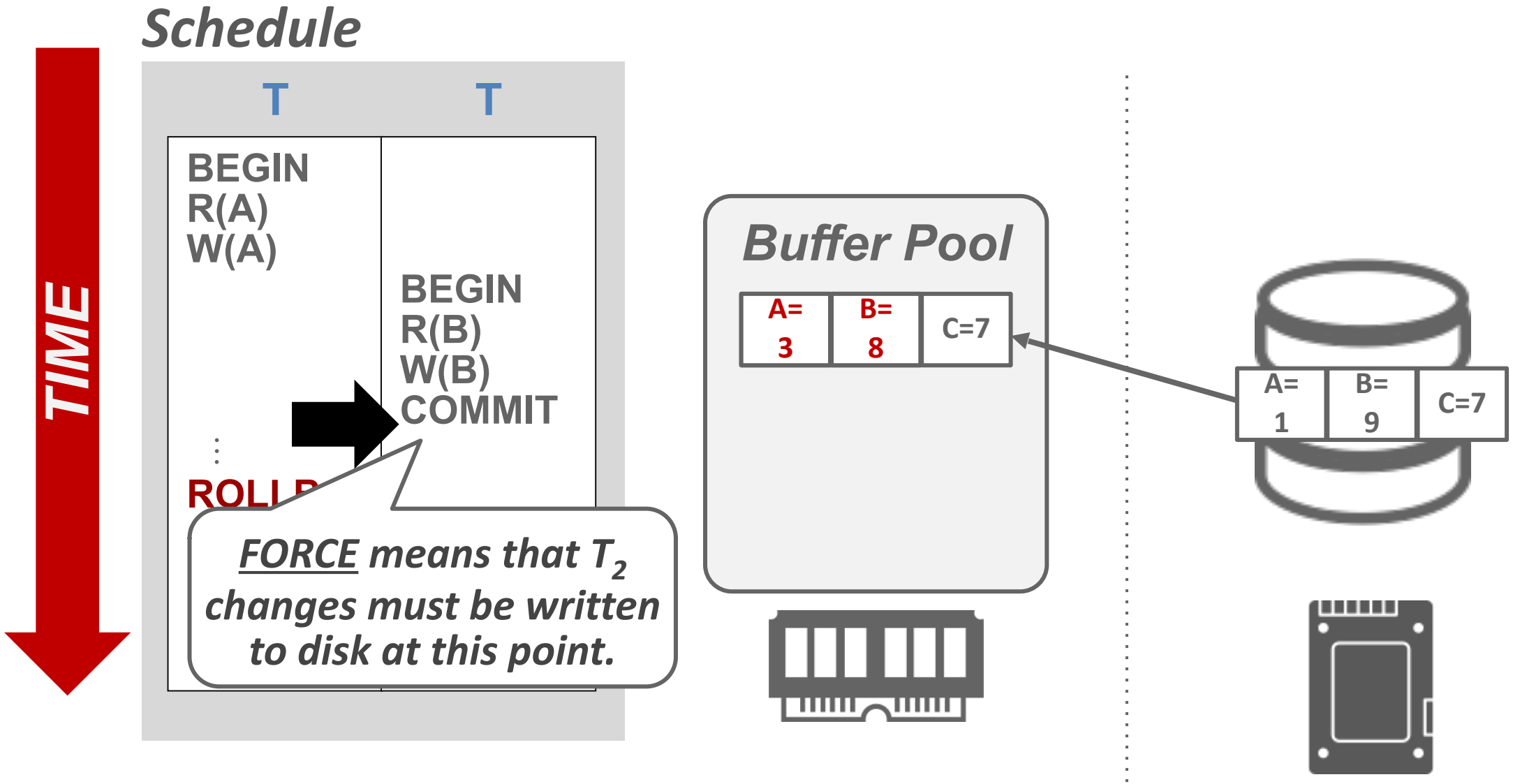
No-steal + force



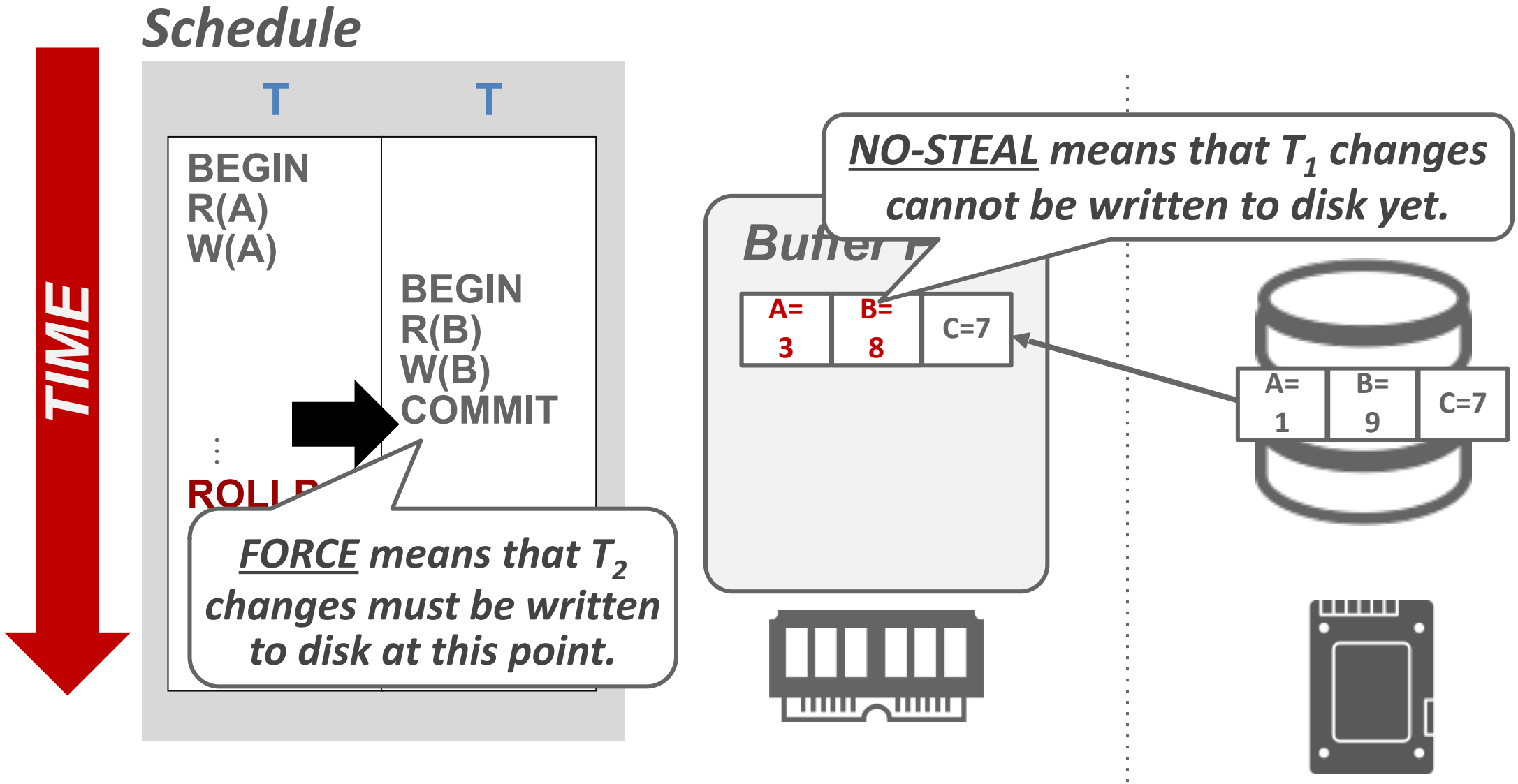
No-steal + force



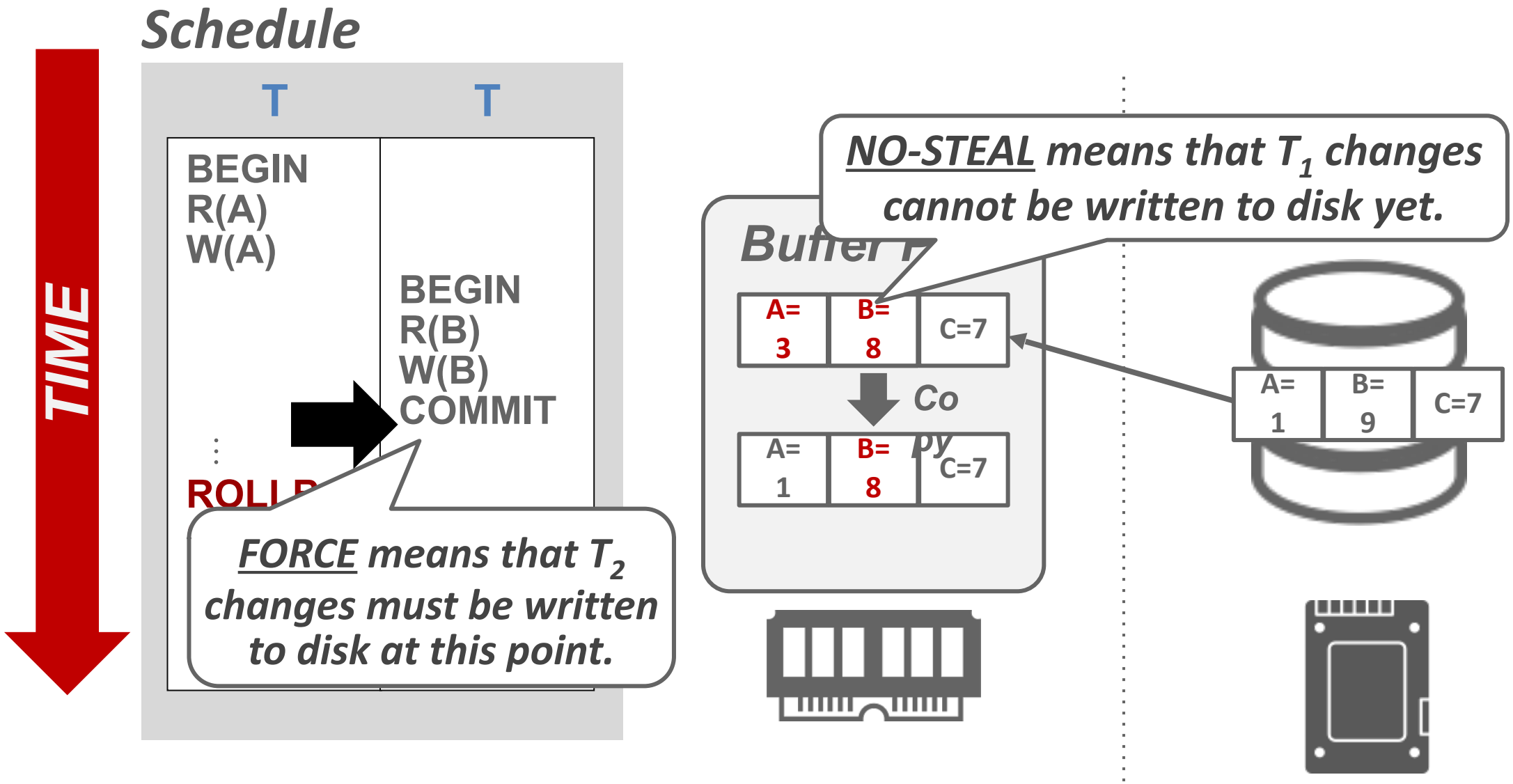
No-steal + force



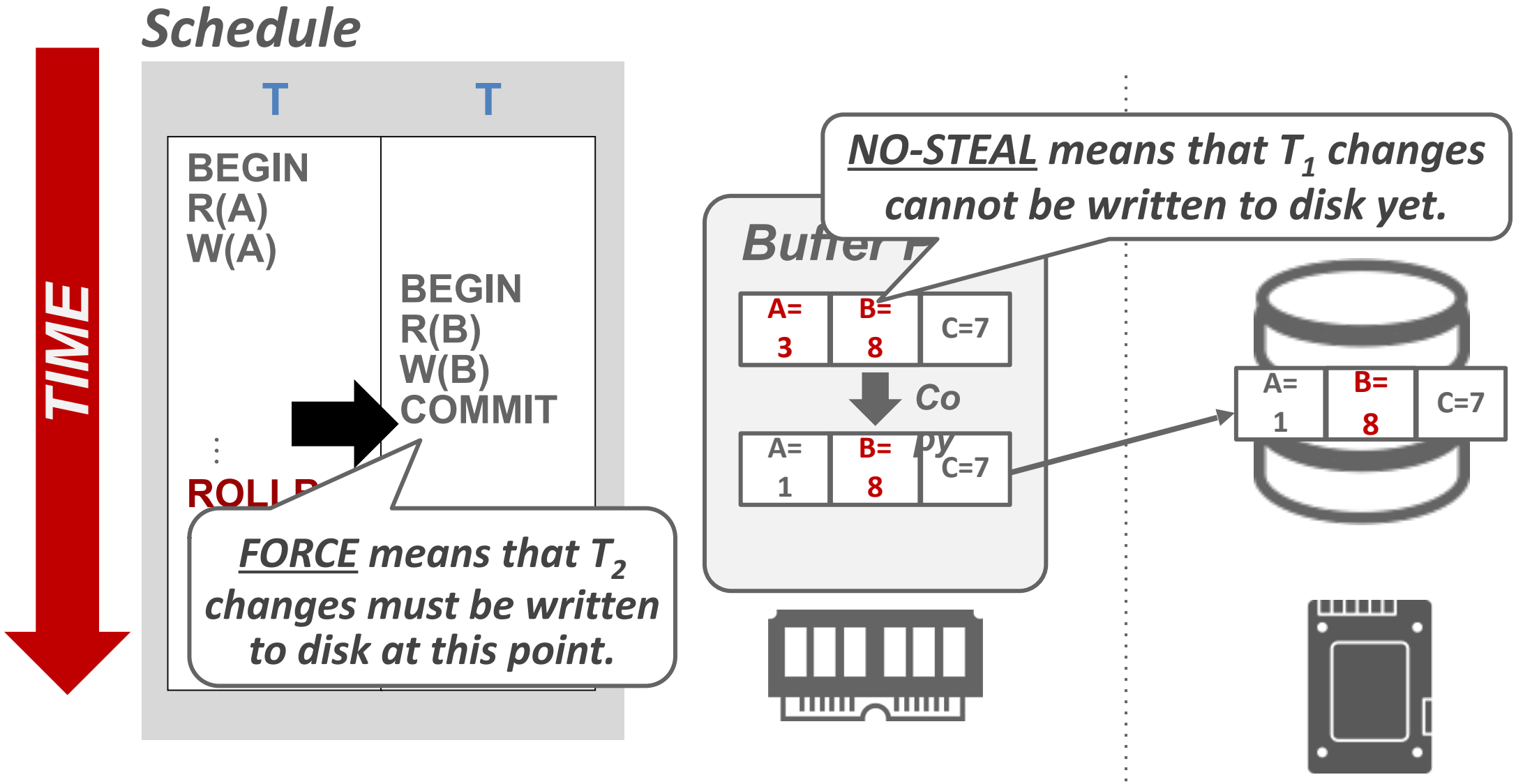
No-steal + force



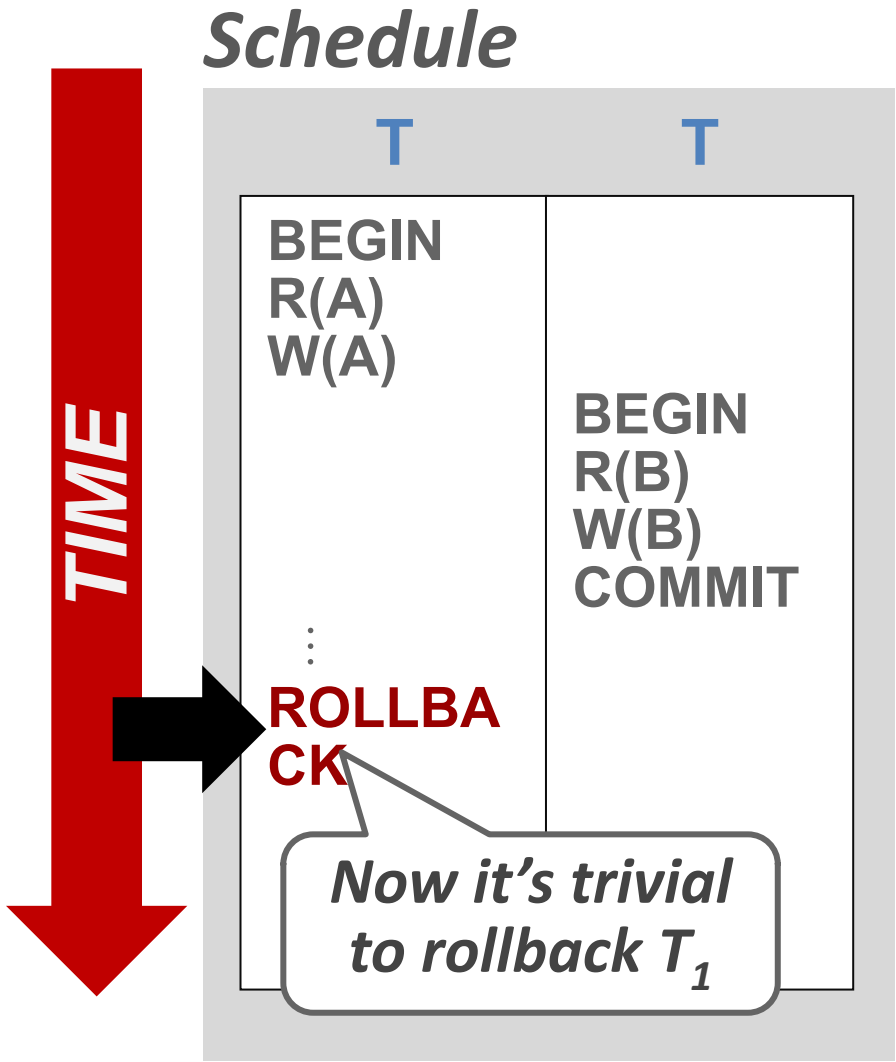
No-steal + force

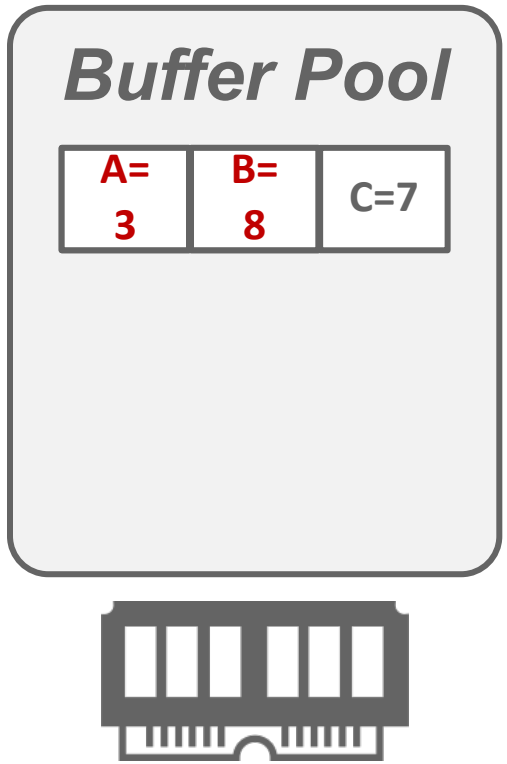


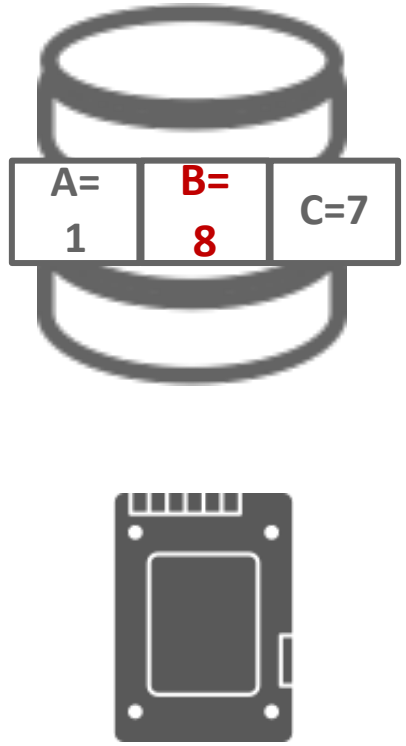
No-steal + force



No-steal + force







No-steal + force

This approach 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 committee commit time

Issues with no-steal + force?

No-steal + force

This approach 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 committee commit time

Issues with no-steal + force?

- High memory pressure: requires all pages to be buffered in memory until commit
- Slow commit: Force flushes all dirty pages at commit time, increasing latency
- Inefficient for concurrent updates: excessive I/O per commit for multiple txns

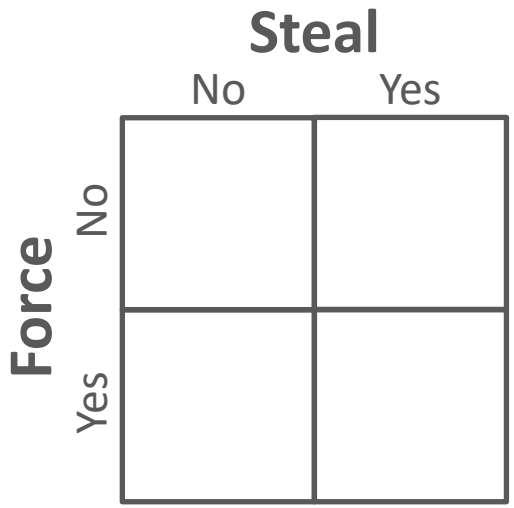
Write-ahead log (WAL)

- Maintains 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 necessary information to perform undo and redo actions to restore DB

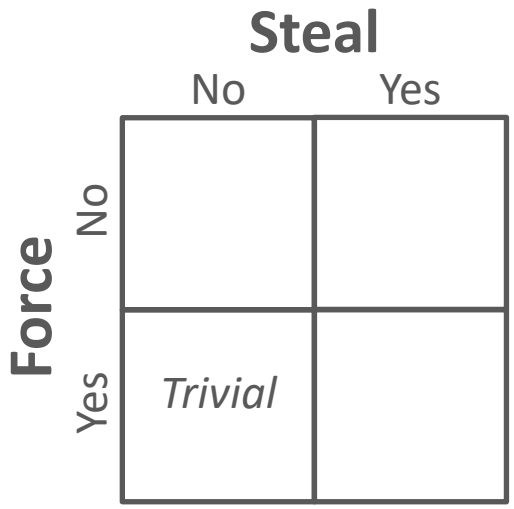
- DBMS must write to disk the log file records that correspond to changes made to a database object before it can flush that object to disk

Buffer pool policy: **Steal + No-force**

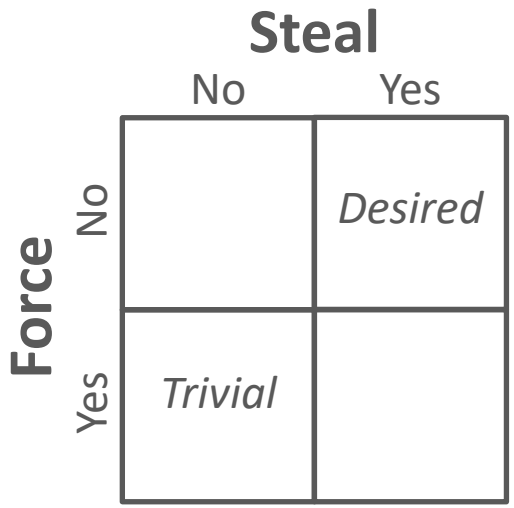
Buffer pool + WAL



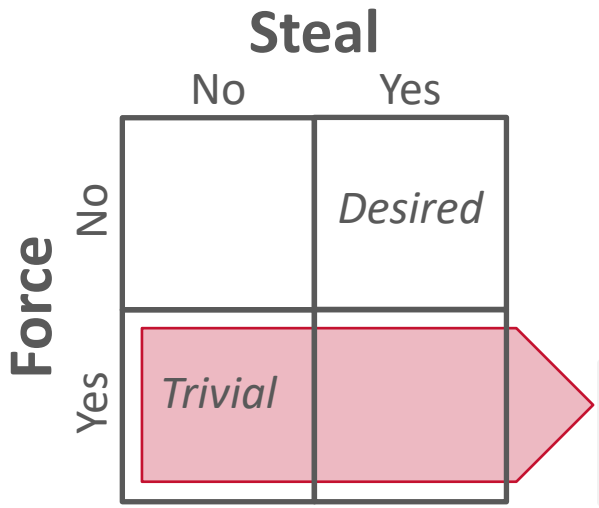
Buffer pool + WAL



Buffer pool + WAL



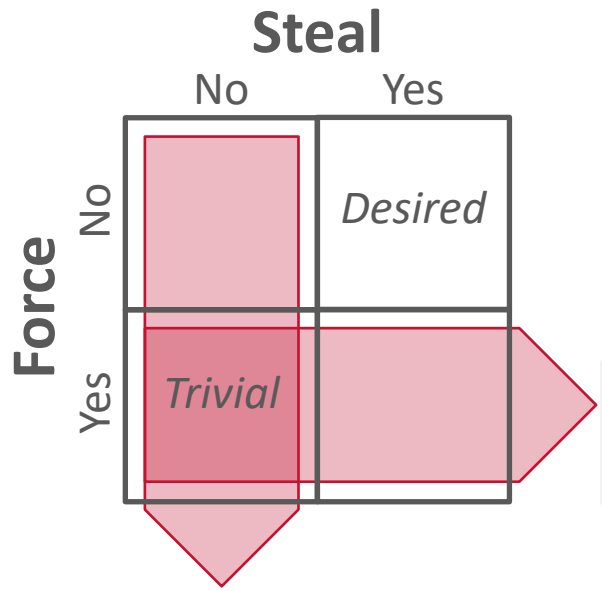
Buffer pool + WAL



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

Poor response time, but enforces durability of committed txns.

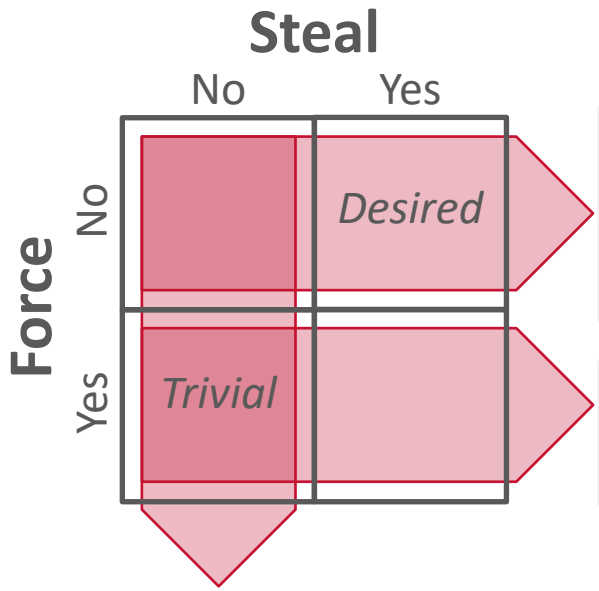
Buffer pool + WAL



Force (on every update, flush the updated page to disk)
Poor response time, but enforces durability of committed txns.

No-Steal
Low throughput, but works for aborted txns

Buffer pool + WAL

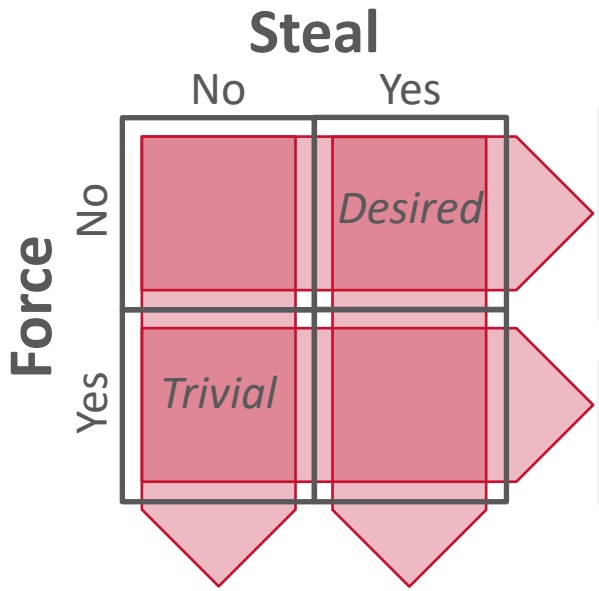


No-Steal
Low throughput, but works for aborted txns

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.

Buffer pool + 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 throughput, 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 keeps a dedicated region for txn's log records in volatile storage (usually backed by buffer pool)

- **First** write the log records wrt an updated page **and then** page itself in non-volatile storage

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

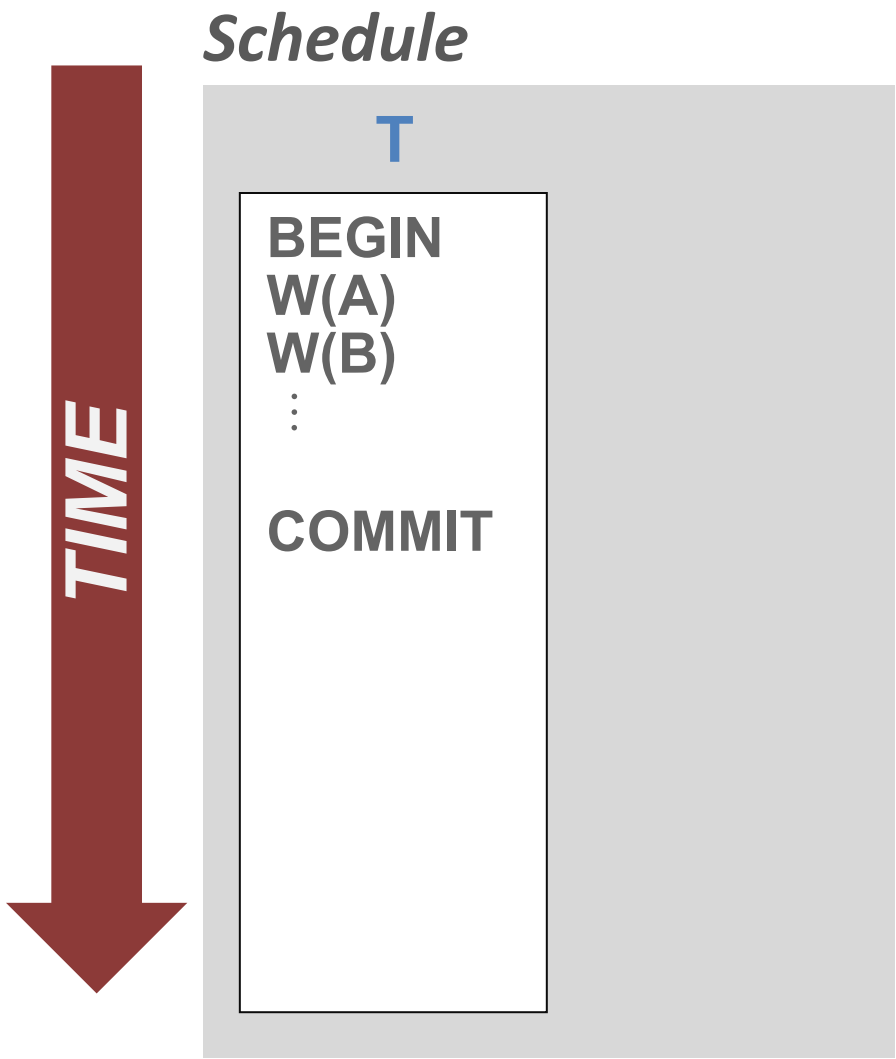
WAL protocol

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

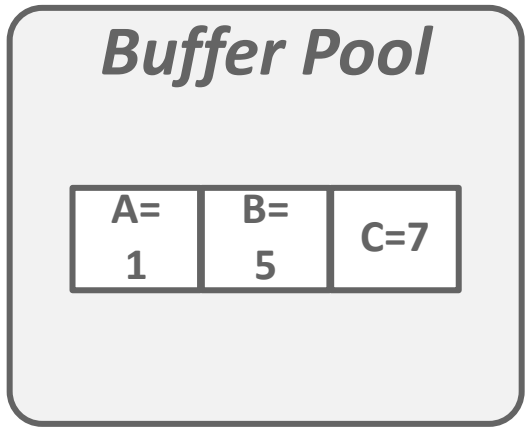
- Append a record every time a txn changes an object:
  - Transaction ID
  - Object ID
  - Before value (**undo)**
  - After value (**redo)**

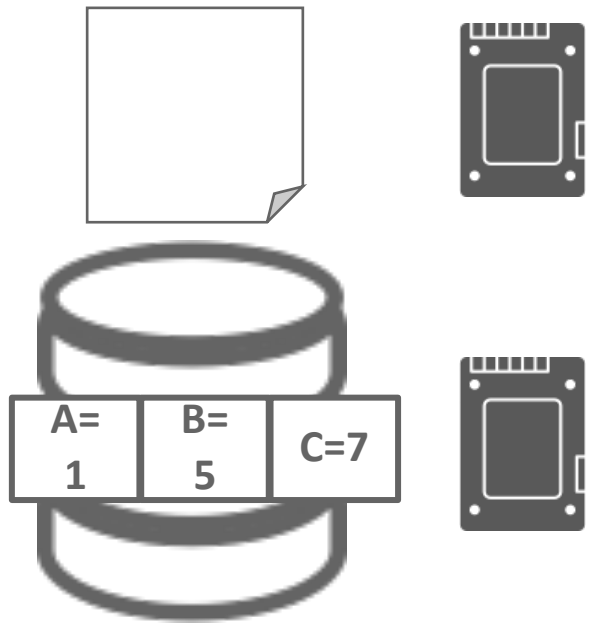
- When a txn finishes, the DBMS appends a **<COMMIT>** record to the log
  - Make sure that all log records are flushed before it returns an acknowledgement to application

WAL example

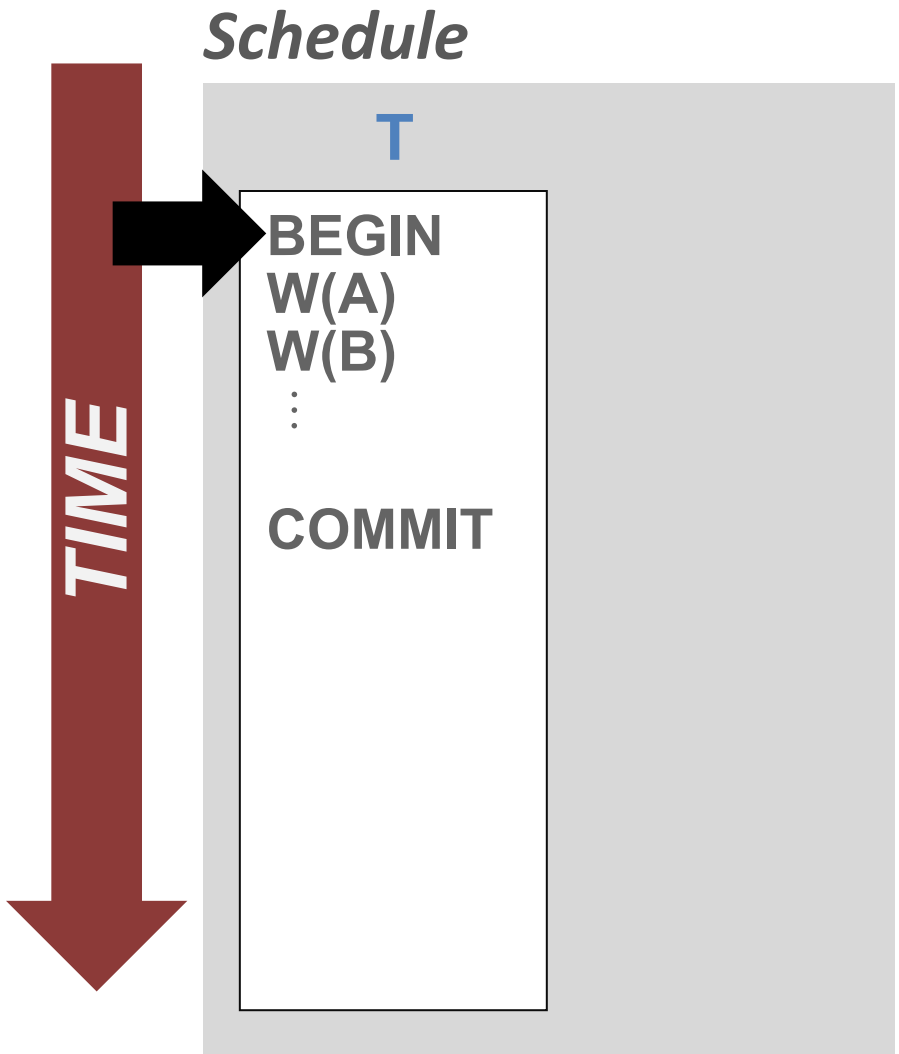

Schedule
TIME
T
BEGIN
W(A)
W(B)
COMMIT

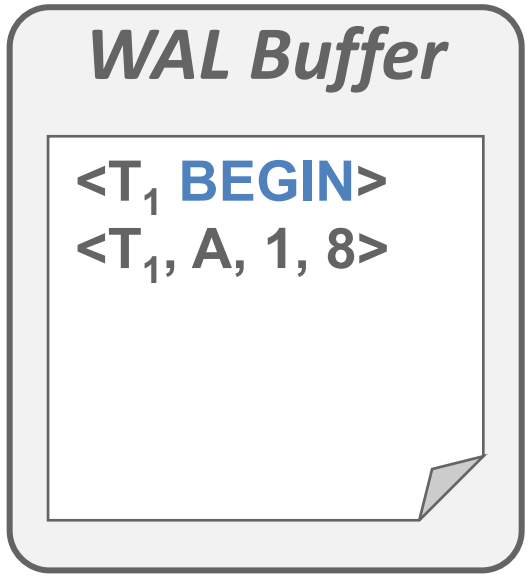

WAL Buffer

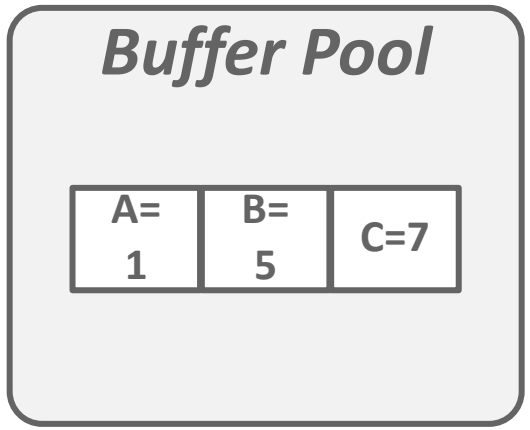

Buffer Pool
A= 1
B= 5
C=7

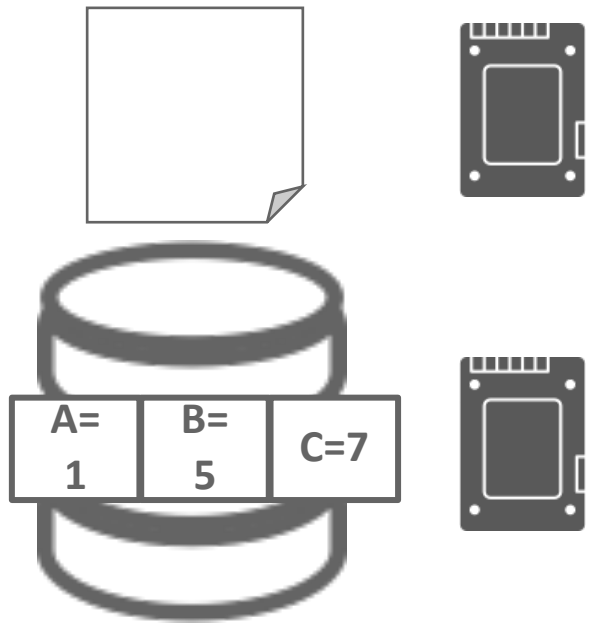

A= 1
B= 5
C=7

WAL example

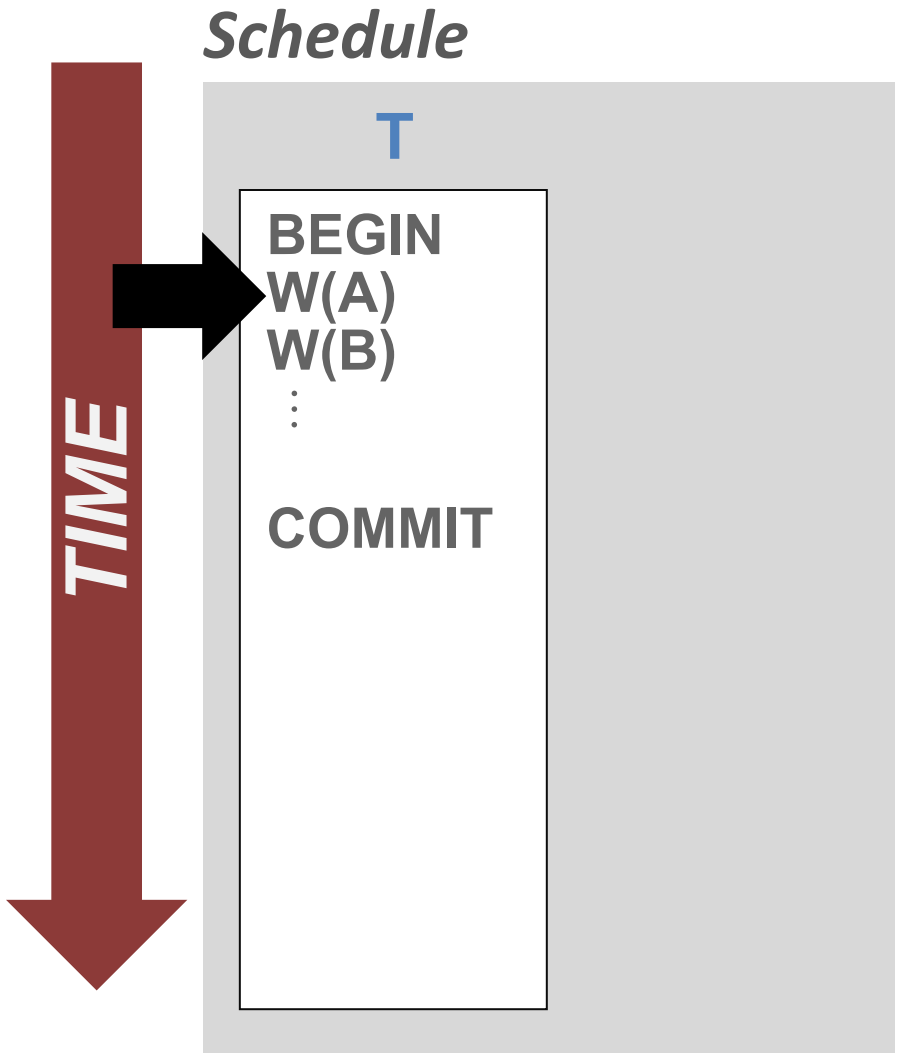

Schedule
T
BEGIN
W(A)
W(B)
⋮
COMMIT
TIME

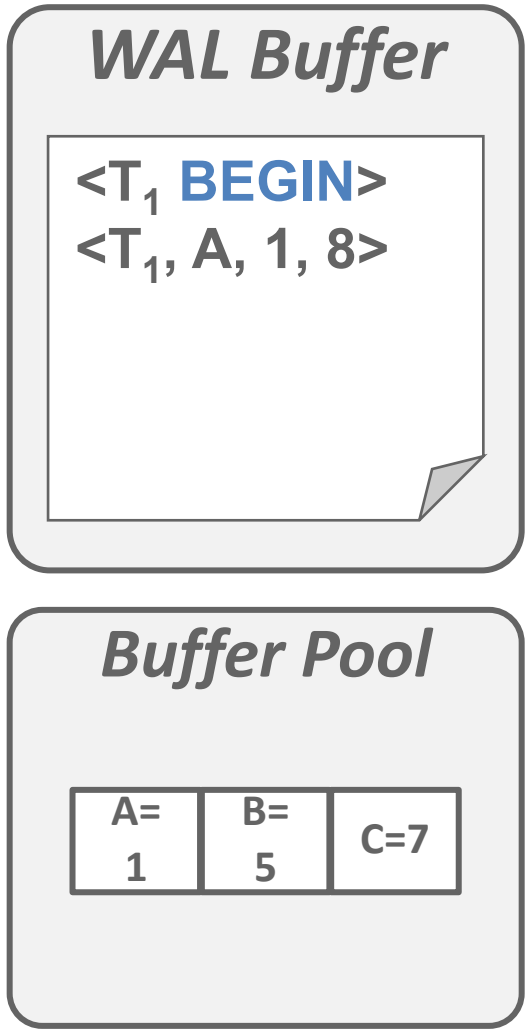

WAL Buffer
<T1 BEGIN>
<T1, A, 1, 8>

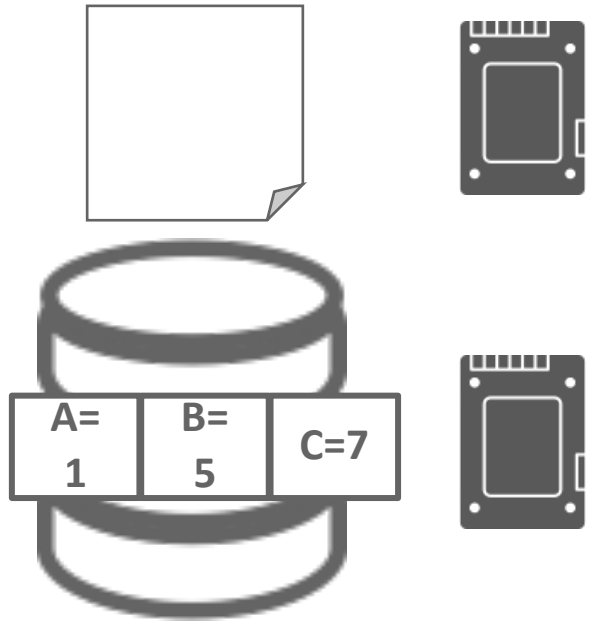

Buffer Pool
A= 1
B= 5
C=7


A= 1
B= 5
C=7

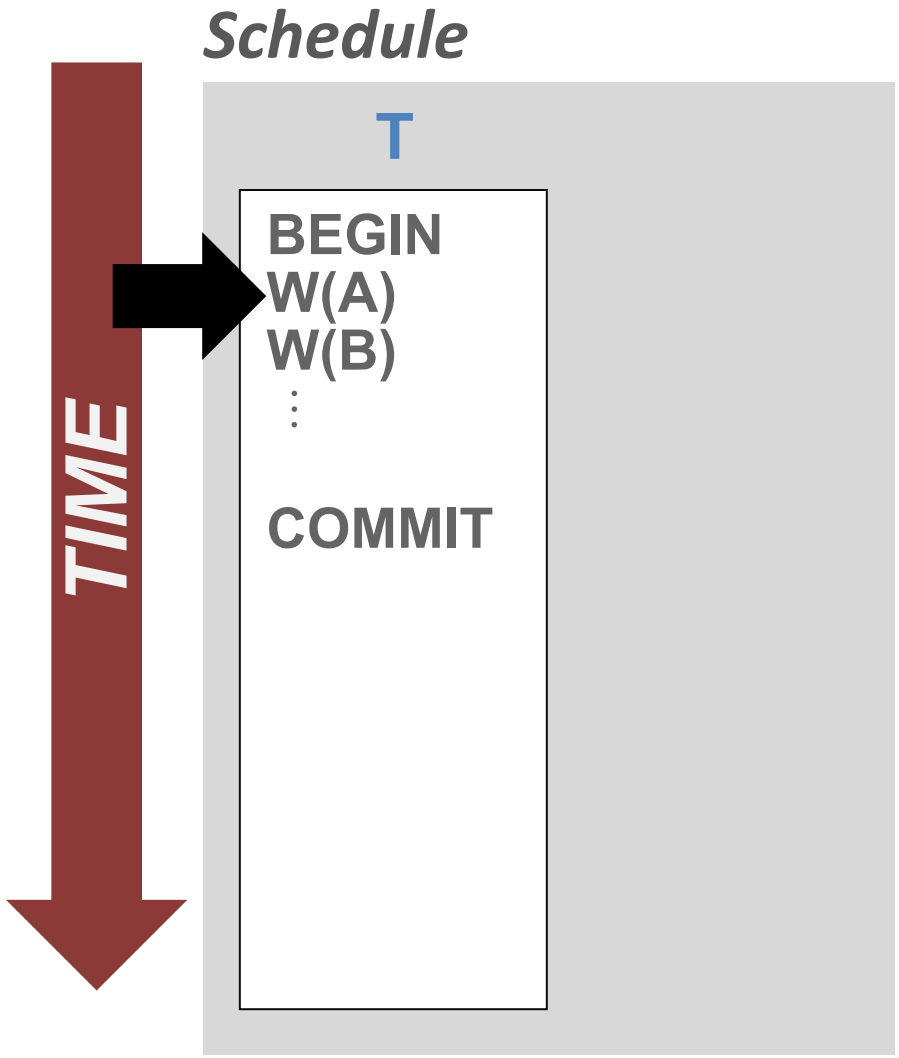
WAL example

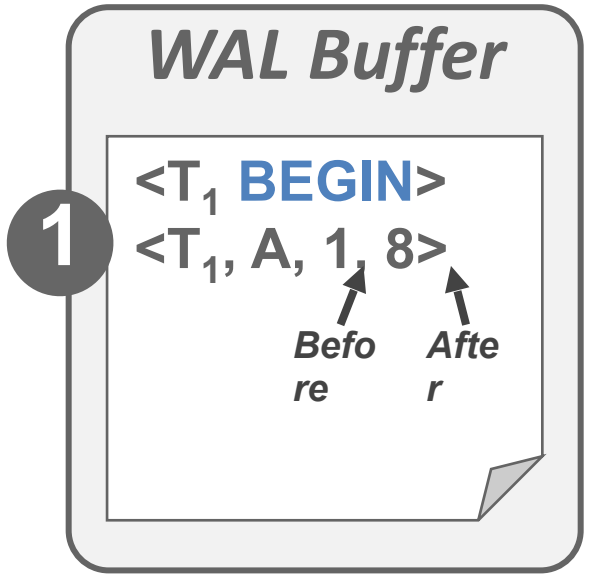

Schedule
T
BEGIN
W(A)
W(B)
⋮
COMMIT
TIME

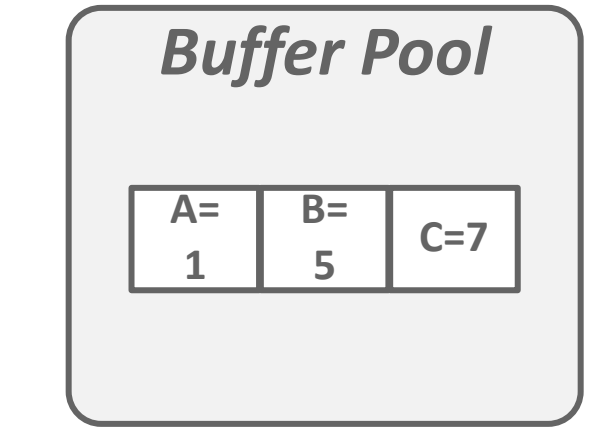

WAL Buffer
<T1 BEGIN>
<T1, A, 1, 8>
Buffer Pool
A= 1
B= 5
C=7

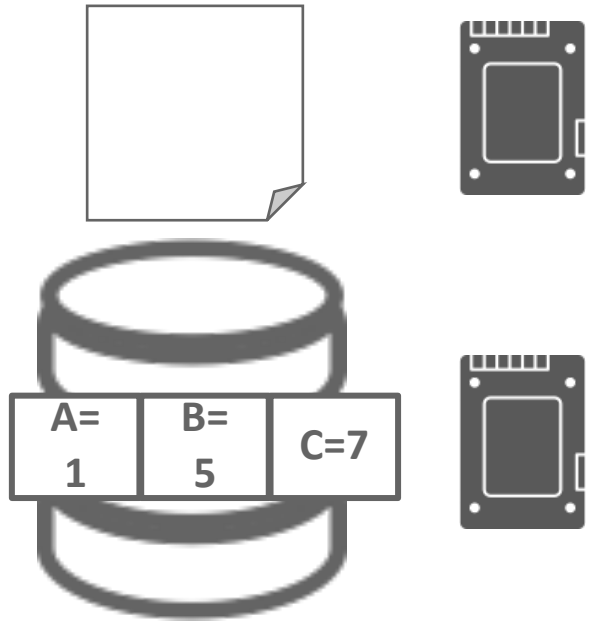

A= 1
B= 5
C=7

WAL example

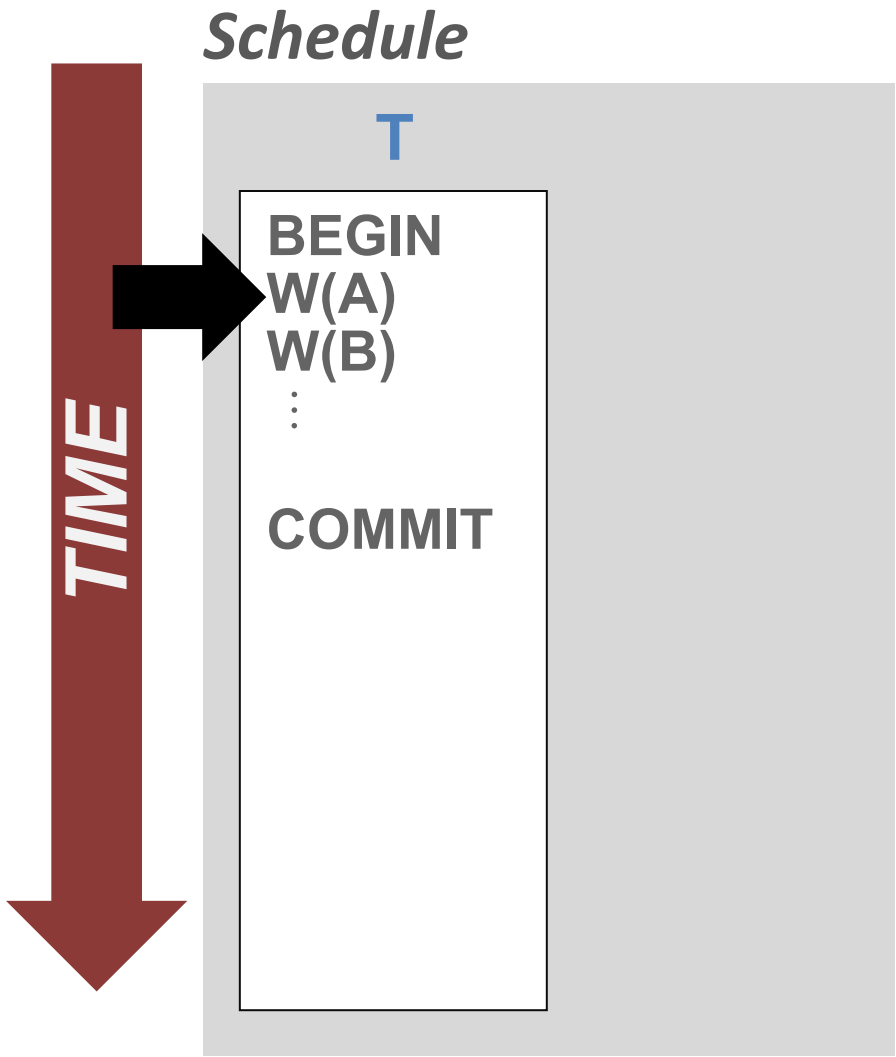

Schedule
T
BEGIN
W(A)
W(B)
⋮
COMMIT
TIME

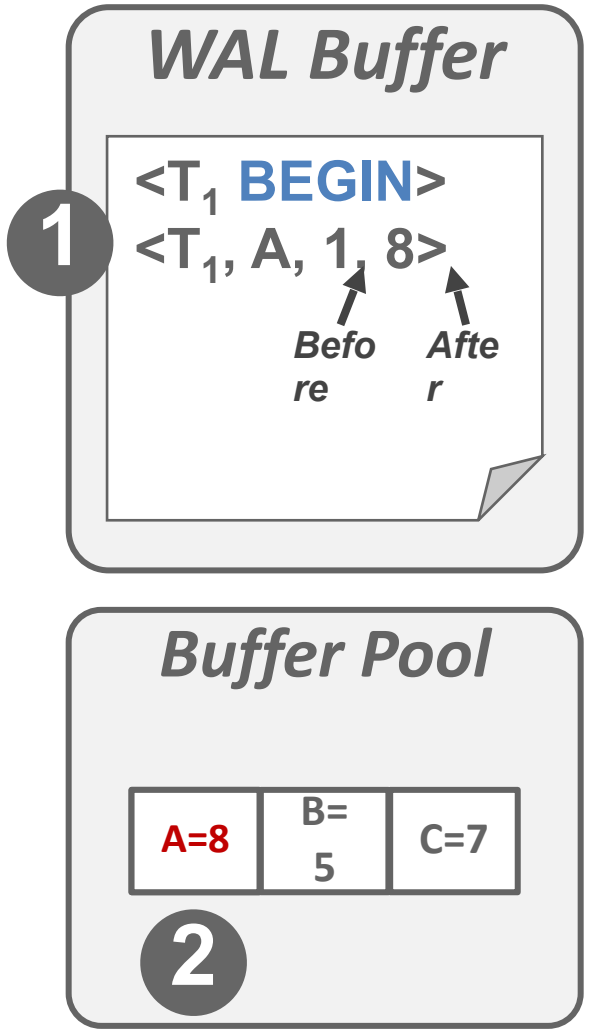

WAL Buffer
1
<T1 BEGIN>
<T1, A, 1, 8>
Before
After

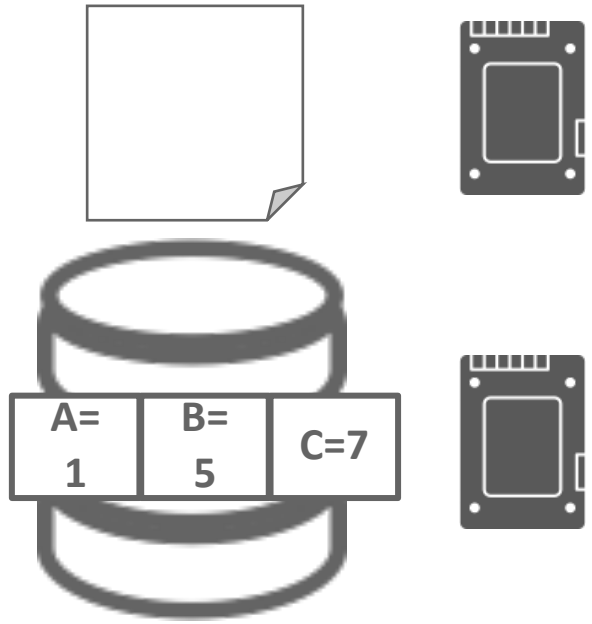

Buffer Pool
A= 1
B= 5
C=7


A= 1
B= 5
C=7

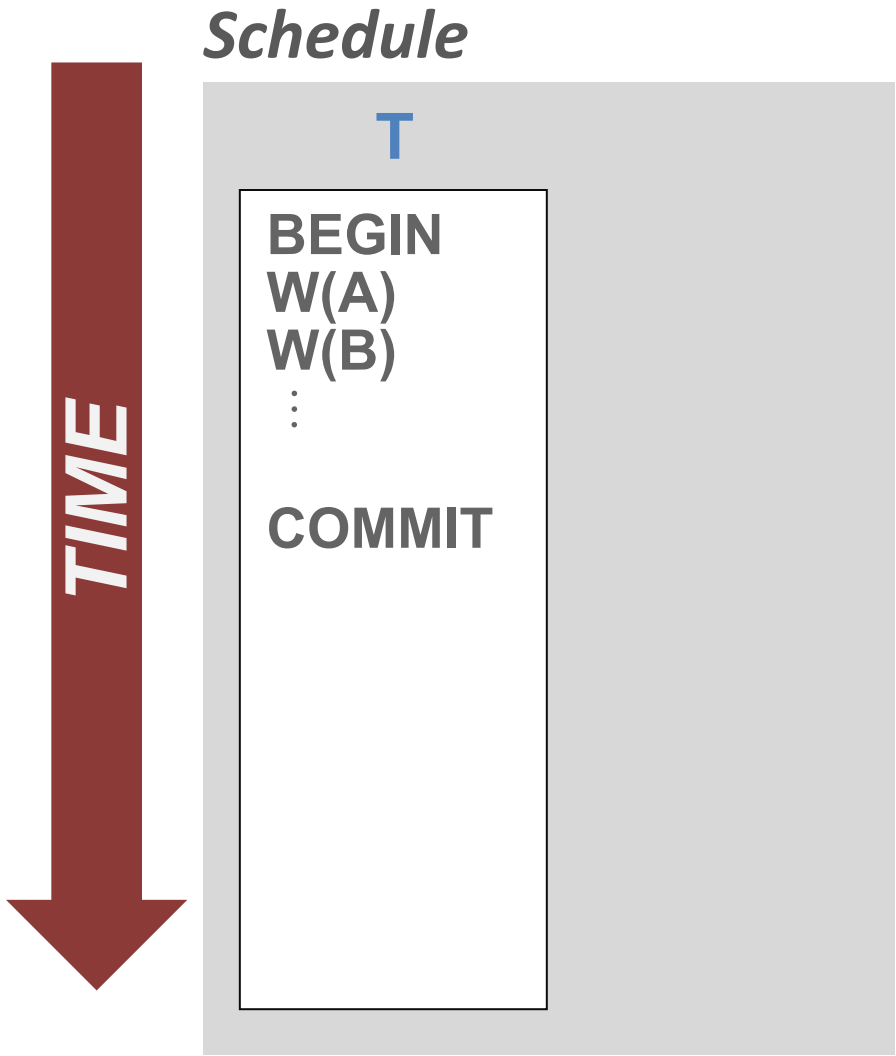
WAL example

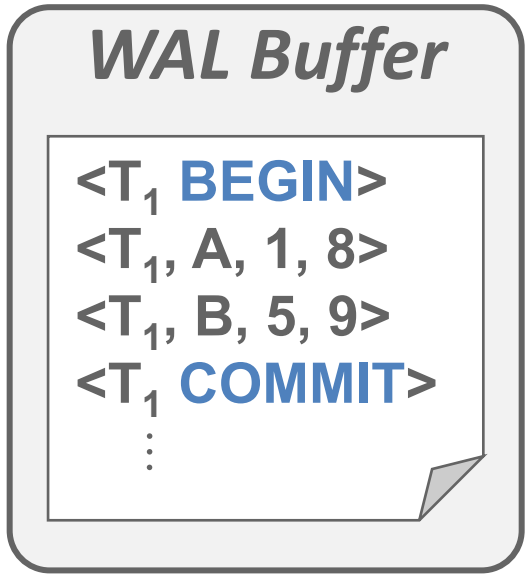

Schedule
T
BEGIN
W(A)
W(B)
⋮
COMMIT
TIME

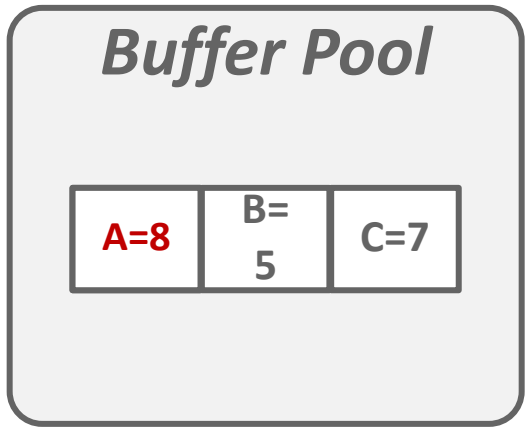

WAL Buffer
1
<T1 BEGIN>
<T1, A, 1, 8>
Before
After
Buffer Pool
A=8
B=5
C=7
2

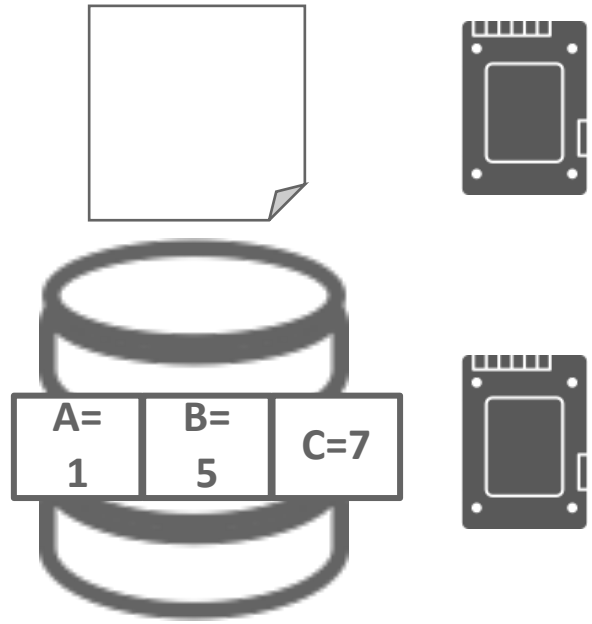

A=1
B=5
C=7

WAL example

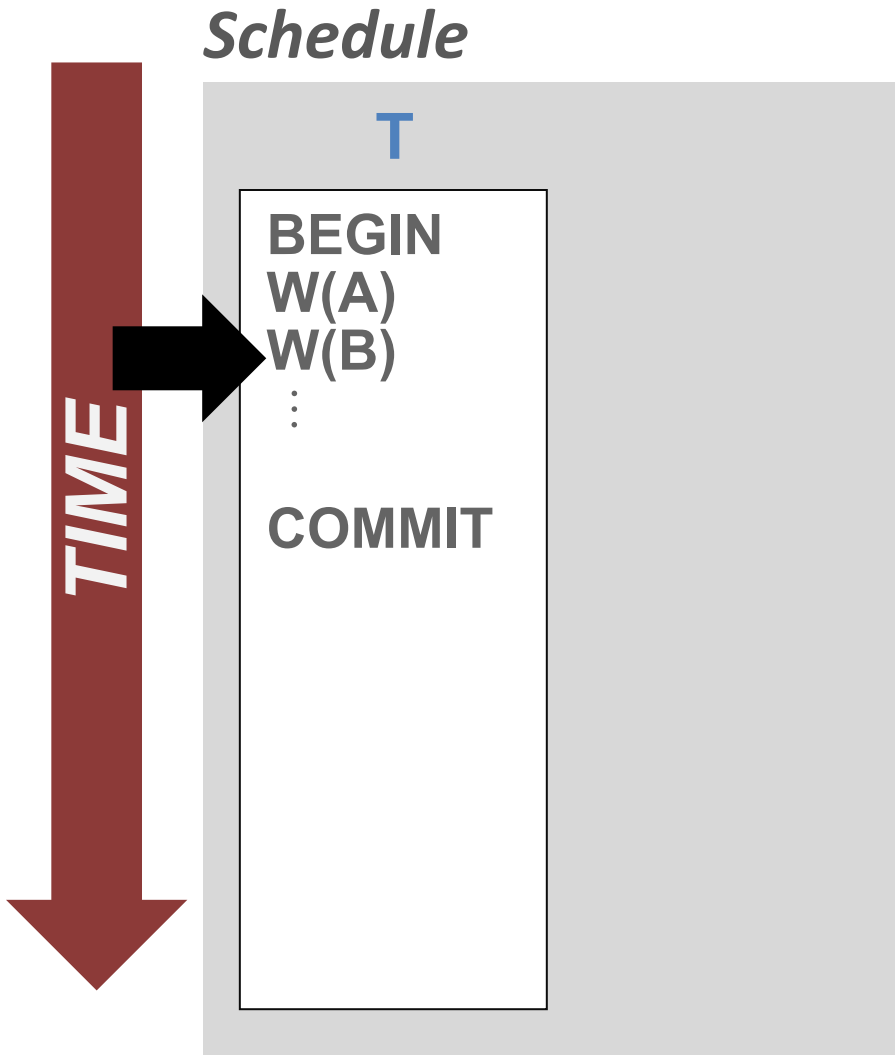

Schedule
TIME
T
BEGIN
W(A)
W(B)
⋮
COMMIT

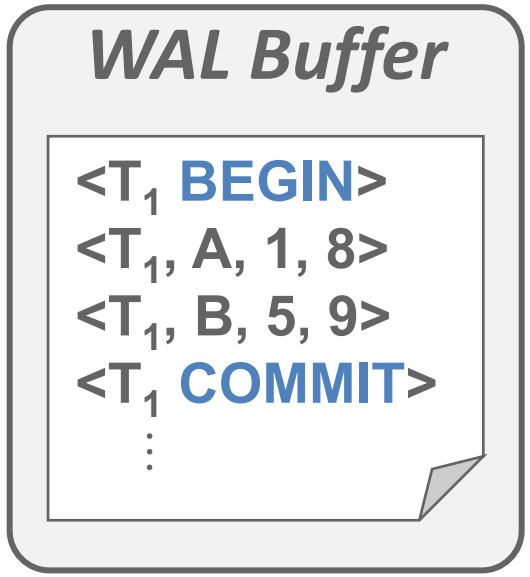

WAL Buffer
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
<T1 COMMIT>
⋮

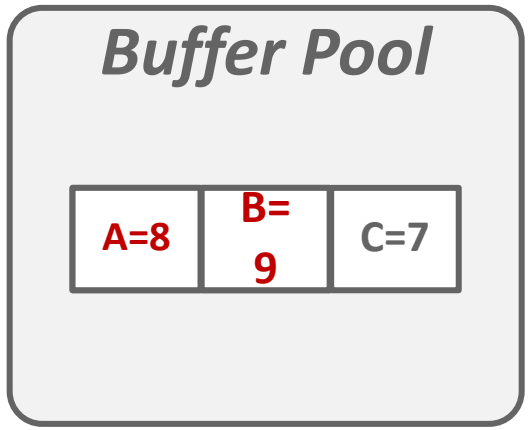

Buffer Pool
A=8
B=
5
C=7

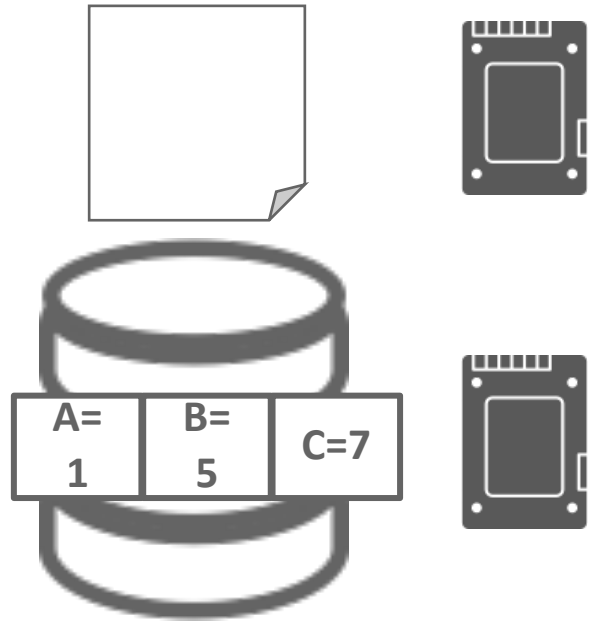

A=
1
B=
5
C=7

WAL example

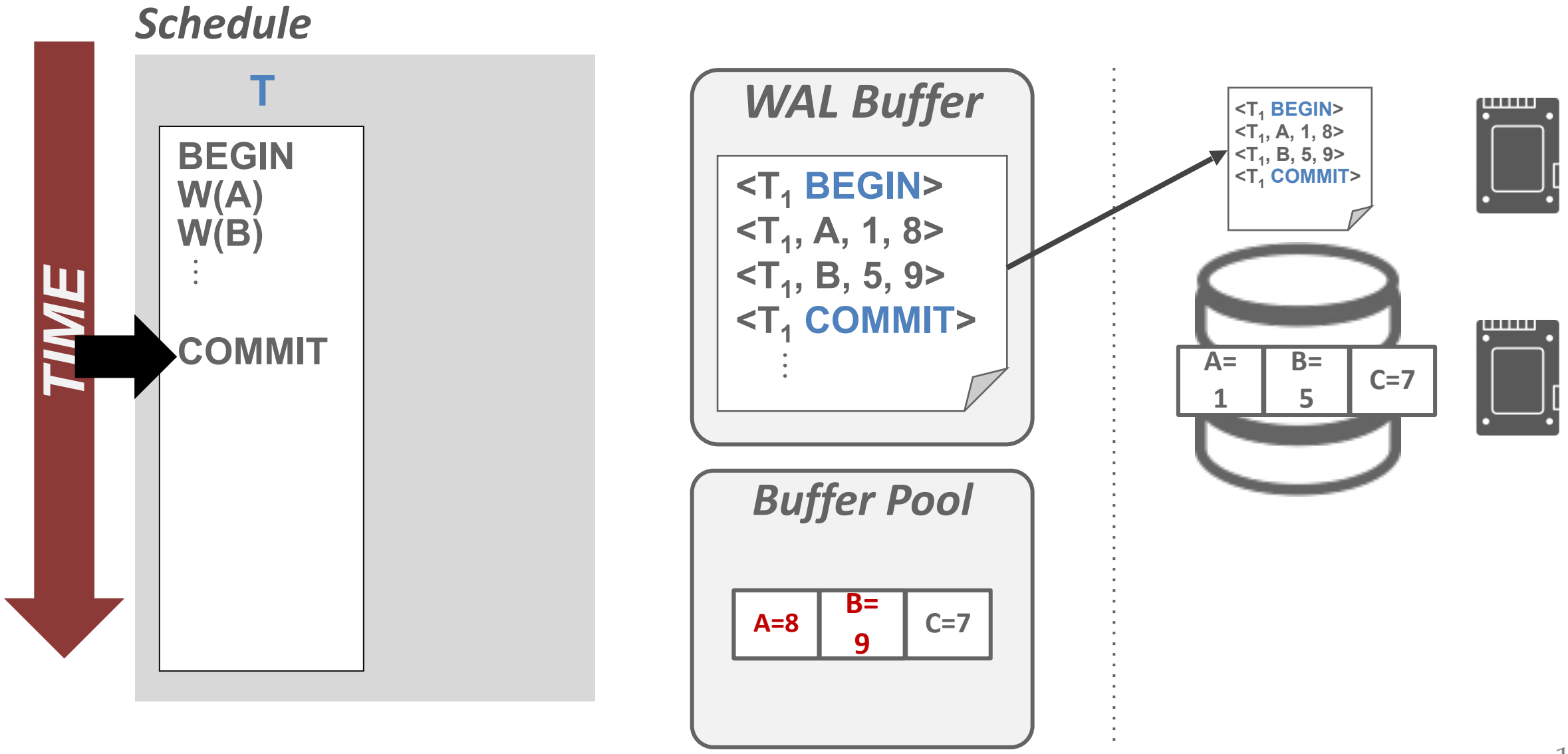

Schedule
TIME
T
BEGIN
W(A)
W(B)
⋮
COMMIT


WAL Buffer
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
<T1 COMMIT>
⋮

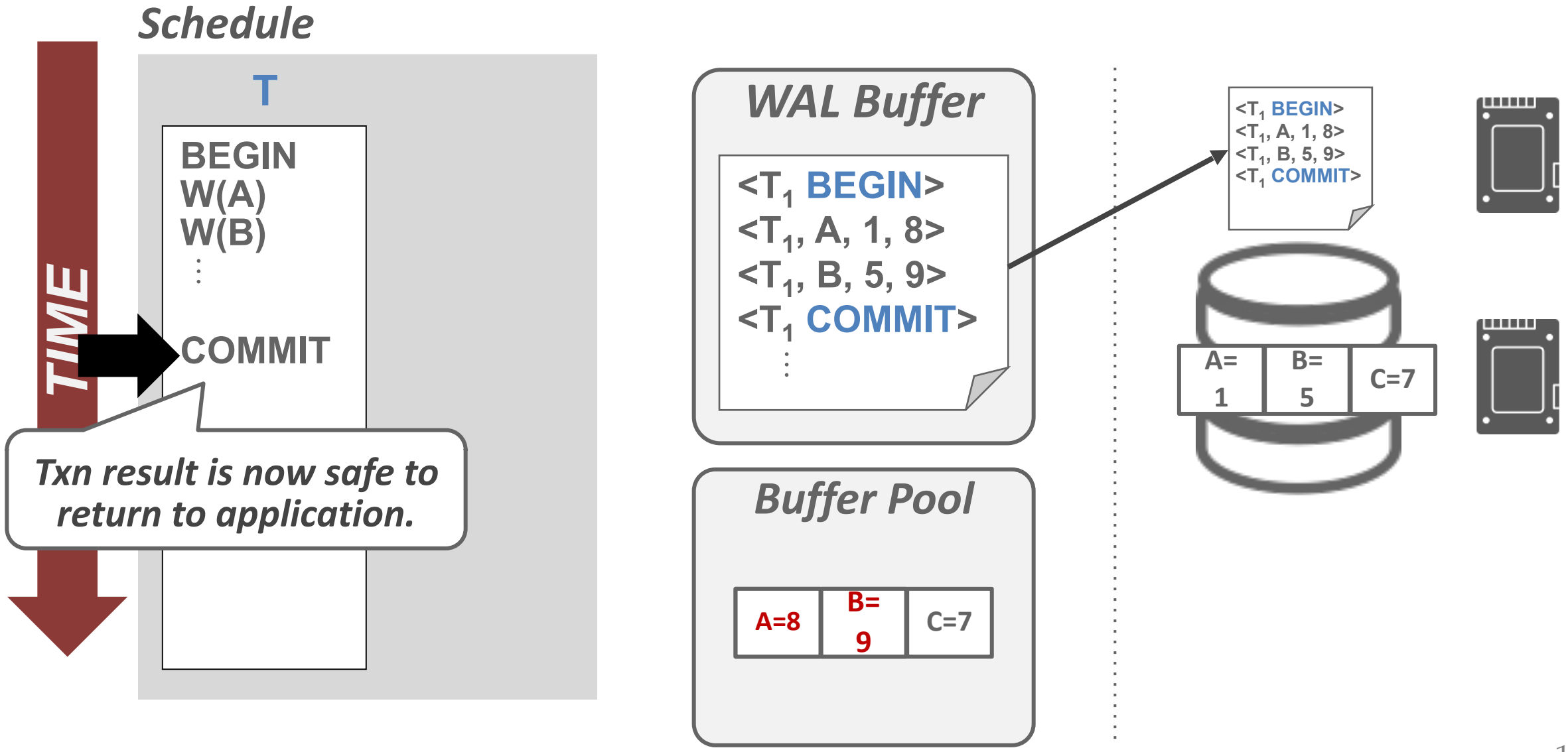

Buffer Pool
A=8
B=
9
C=7


A=
1
B=
5
C=7

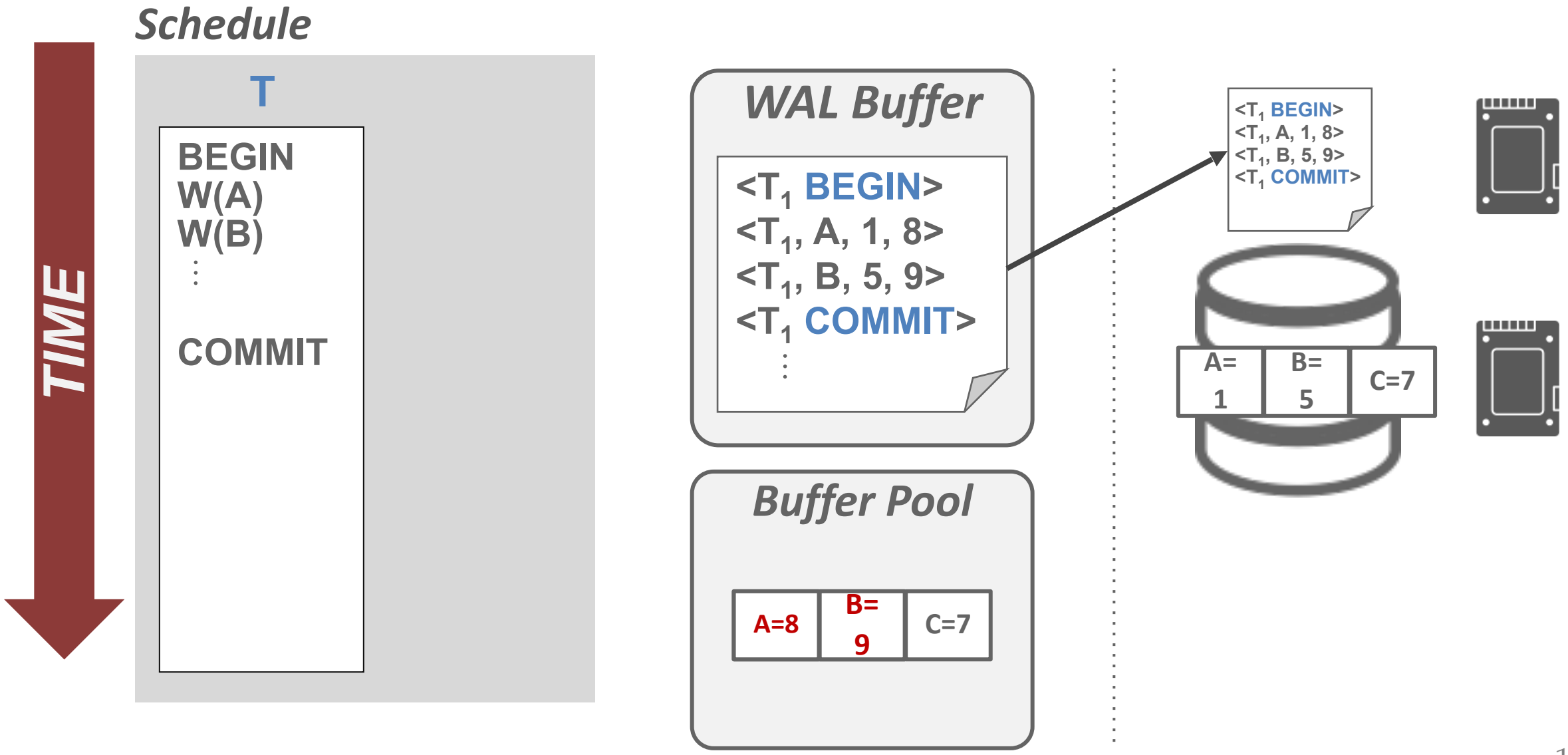
WAL example


Schedule
T
BEGIN
W(A)
W(B)
COMMIT
TIME
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

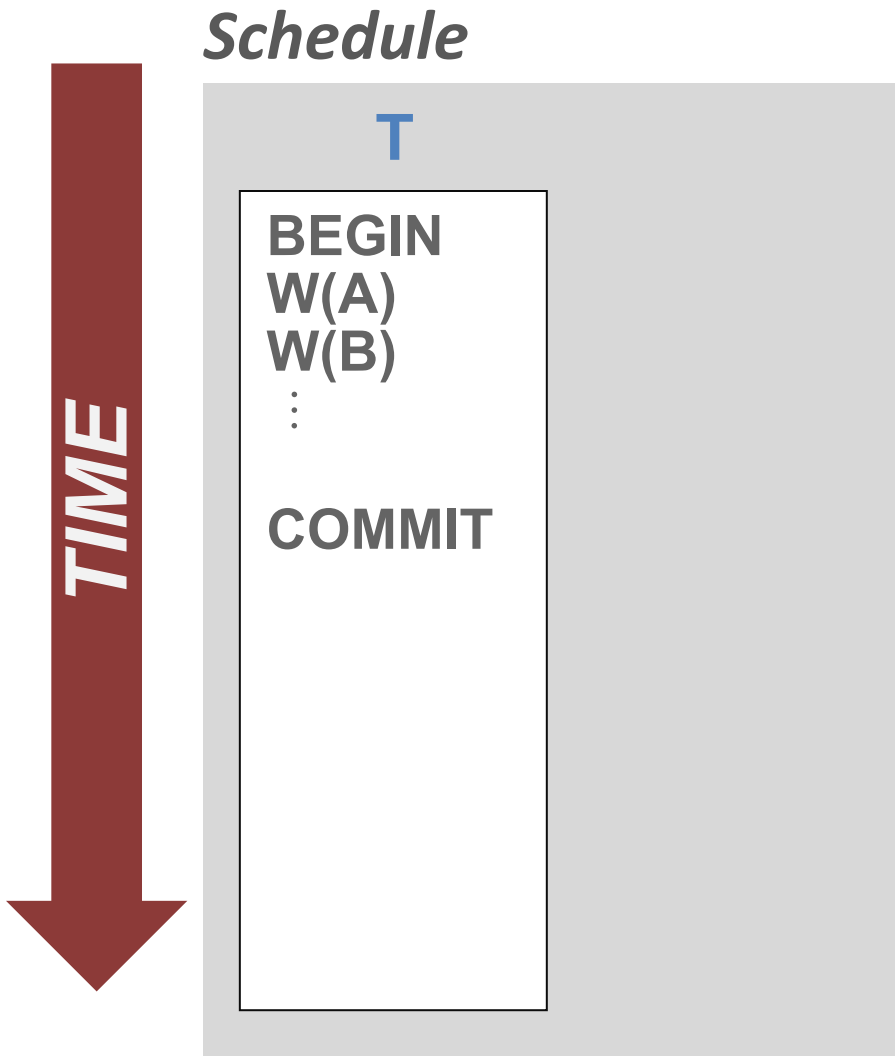
WAL example

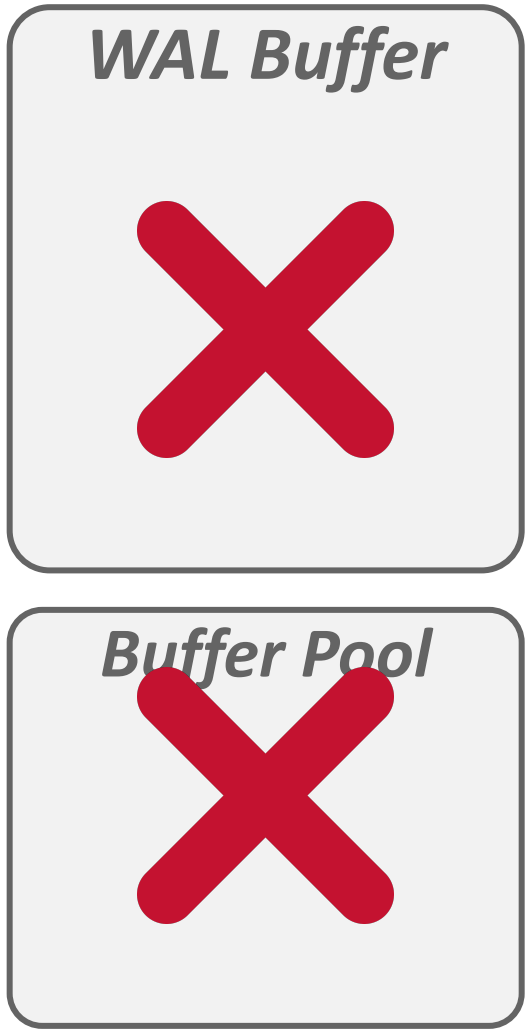

Schedule
T
BEGIN
W(A)
W(B)
COMMIT
TIME
Txn result is now safe to return to application.
WAL Buffer
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
<T1 COMMIT>
Buffer Pool
A=8
B= 9
C=7
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
<T1 COMMIT>
A= 1
B= 5
C=7

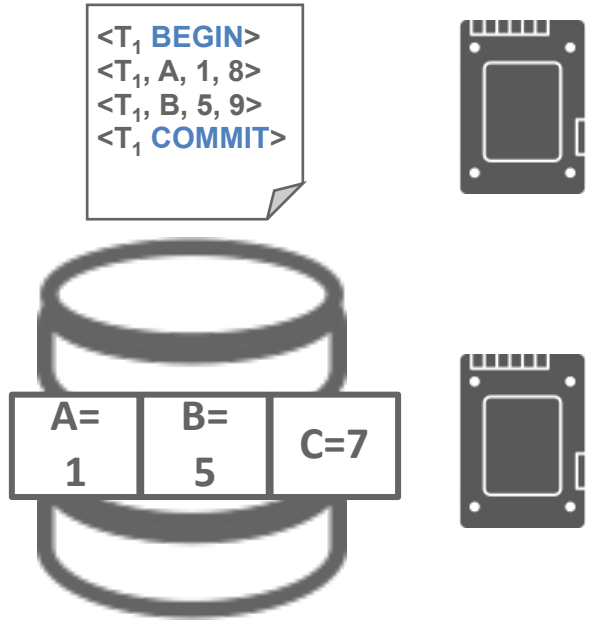
WAL example


Schedule
TIME
T
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


Schedule
TIME
T
BEGIN
W(A)
W(B)
COMMIT


WAL Buffer
Buffer Pool


<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
<T1 COMMIT>
A= 1
B= 5
C=7

WAL example

Schedule

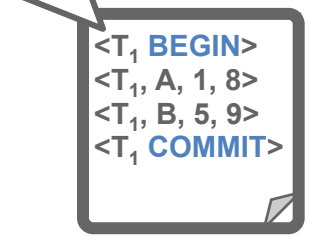
TIME

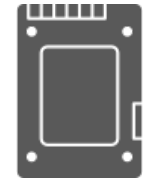
T
BEGIN W(A) W(B) ⋮ COMMIT

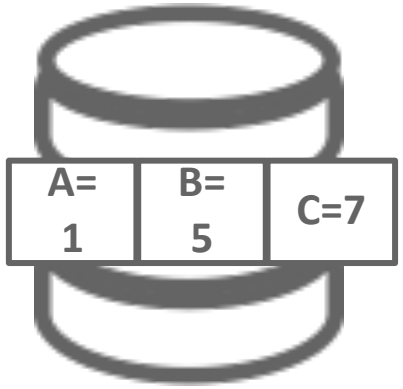
Everything we need to restore T_1 is in the log!

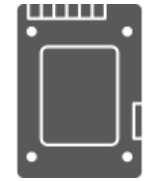
WAL Buffer

Buffer Pool









WAL implementation

- Flushing the log buffer to disk every time a txn commits becomes 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)

Buffer pool policies

- Almost every DBMS uses **No-force** + **steal**

Runtime Performance

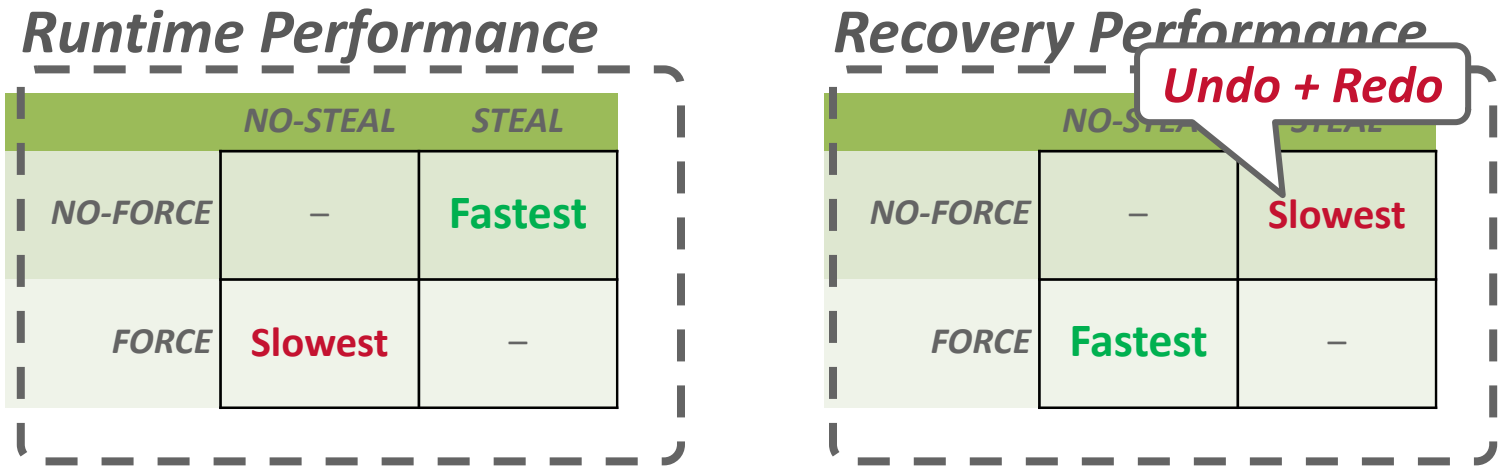
	NO-STEAL	STEAL
NO-FORCE	–	Fastest
FORCE	Slowest	–

Recovery Performance

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

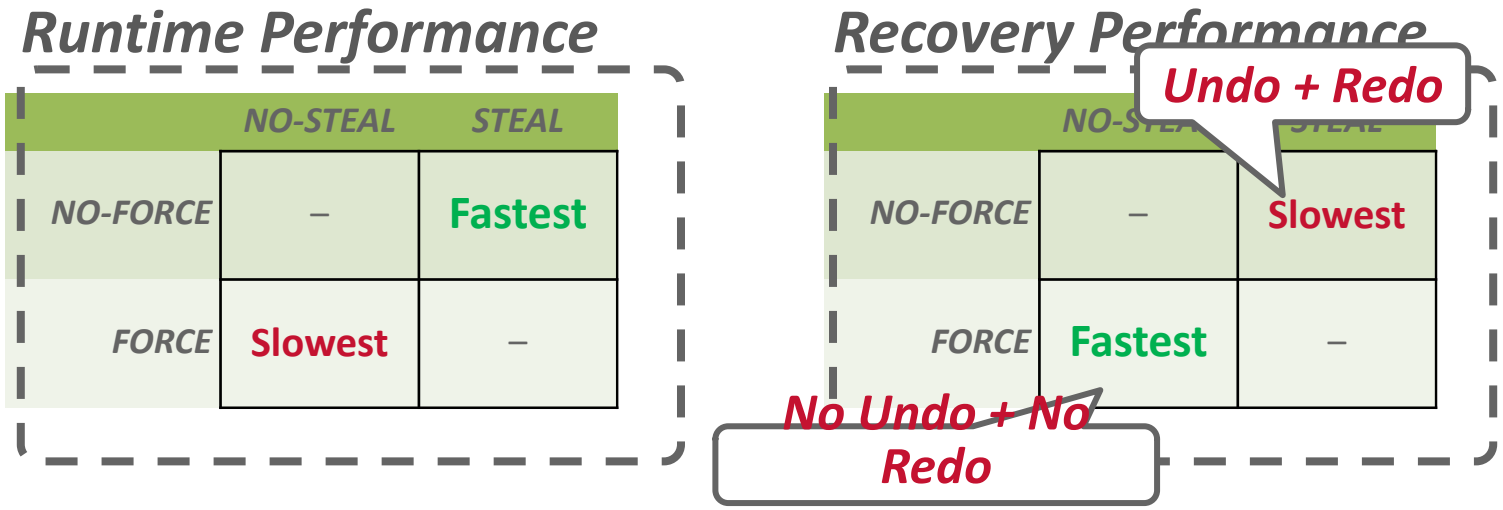
Buffer pool policies

- Almost every DBMS uses **No-force** + **steal**



Buffer pool policies

- Almost every DBMS uses **No-force** + **steal**



Logging schemes

Physical logging

- Record the byte-level changes made to a specific page (e.g., 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: Byte-level changes for a single tuple identified by page ID + slot number
- Does not specify page organization

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 *concurrent txns running at lower isolation levels*
  - Difficult to determine which parts of the database may have been modified by a query before crash
  - Recovery takes longer because DBMS re-executes every query in the log again

Summary: logging

- WAL 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

Crash recovery

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

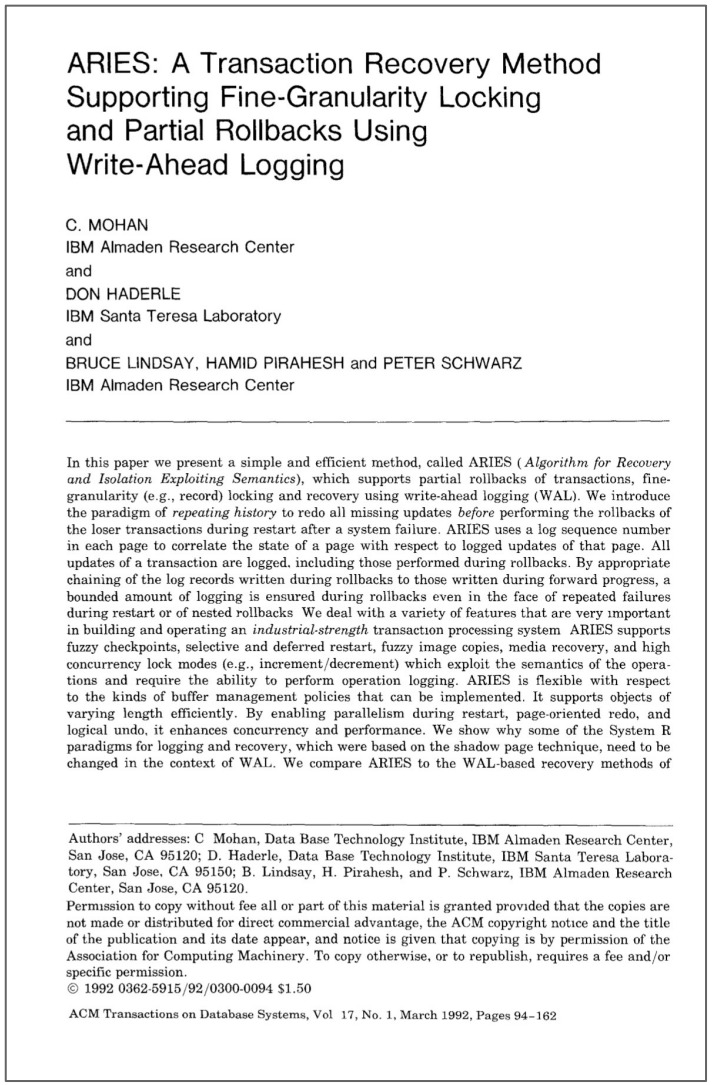
- Recovery algorithms have two parts:
  - Actions during normal txn processing to ensure that the DBMS can recover from a failure → **preparing for the failure**
  - Actions after a failure to recover the database to a state that ensures atomicity, consistency, and durability → **handling the failure**

ARIES

Algorithms for Recovery and Isolation Exploitation Semantics

- Developed at IBM Research in early 1990s for the DB2 DBMS

- Not all systems implement ARIES exactly as defined in this paper but they follow it closely → canonical representation of today's DB recovery protocol



ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging

C. MOHAN
IBM Almaden Research Center
and
DON HADERLE
IBM Santa Teresa Laboratory
and
BRUCE LINDSAY, HAMID PIRAHESH and PETER SCHWARZ
IBM Almaden Research Center

In this paper we present a simple and efficient method, called ARIES (*Algorithm for Recovery and Isolation Exploiting Semantics*), which supports partial rollbacks of transactions, fine-granularity (e.g., record) locking and recovery using write-ahead logging (WAL). We introduce the paradigm of *repeating history* to redo all missing updates *before* performing the rollbacks of the loser transactions during restart after a system failure. ARIES uses a log sequence number in each page to correlate the state of a page with respect to logged updates of that page. All updates of a transaction are logged, including those performed during rollbacks. By appropriate chaining of the log records written during rollbacks to those written during forward progress, a bounded amount of logging is ensured during rollbacks even in the face of repeated failures during restart or of nested rollbacks We deal with a variety of features that are very important in building and operating an *industrial-strength* transaction processing system ARIES supports fuzzy checkpoints, selective and deferred restart, fuzzy image copies, media recovery, and high concurrency lock modes (e.g., increment/decrement) which exploit the semantics of the operations and require the ability to perform operation logging. ARIES is flexible with respect to the kinds of buffer management policies that can be implemented. It supports objects of varying length efficiently. By enabling parallelism during restart, page-oriented redo, and logical undo, it enhances concurrency and performance. We show why some of the System R paradigms for logging and recovery, which were based on the shadow page technique, need to be changed in the context of WAL. We compare ARIES to the WAL-based recovery methods of

Authors' addresses: C Mohan, Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120; D. Haderle, Data Base Technology Institute, IBM Santa Teresa Laboratory, San Jose, CA 95150; B. Lindsay, H. Pirahesh, and P. Schwarz, IBM Almaden Research Center, San Jose, CA 95120.

ACM Transactions on Database Systems, Vol 17, No. 1, March 1992, Pages 94–162

ARIES: main ideas

Write-ahead logging:

- Flush WAL record(s) changes to disk before a database object is written to disk
- Must use **Steal** + **No-force** buffer pool policies

Repeating history during redo:

- On DBMS restart, retrace actions and restore DB to exact state before crash

Logging changes during undo:

- Record undo actions to log to ensure action is not repeated in the event of repeated failures

Today's focus

- Logging
  - Buffer pool policies
  - WAL
  - Logging schemes
- Recovery
  - LSN
  - Normal checkpoint and abort operations
  - Checkpoint
  - Recovery algorithm

WAL records

- Need to extend log record format to include additional info:

- Every log record includes a globally unique **log sequence number** (LSN)
  - LSNs represent the physical order that txns make changes to the DB

- Various components in the system keep track of **LSN**s that pertain to them …

Log sequence numbers (LSNs)

Name	Location	Definition
flushedLSN	Memory	Last LSN in log on disk
pageLSN	$page_x$	Newest update to $page_x$
recLSN	DPT†	Oldest update to $page_x$ since it was last flushed
lastLSN	ATT*	Latest record of txn T_i
MasterRecord	Disk	LSN of latest checkpoint

† DPT = Dirty Page Table

* ATT = Active Transaction Table

WAL bookkeeping

LSN: Unique and monotonically increasing

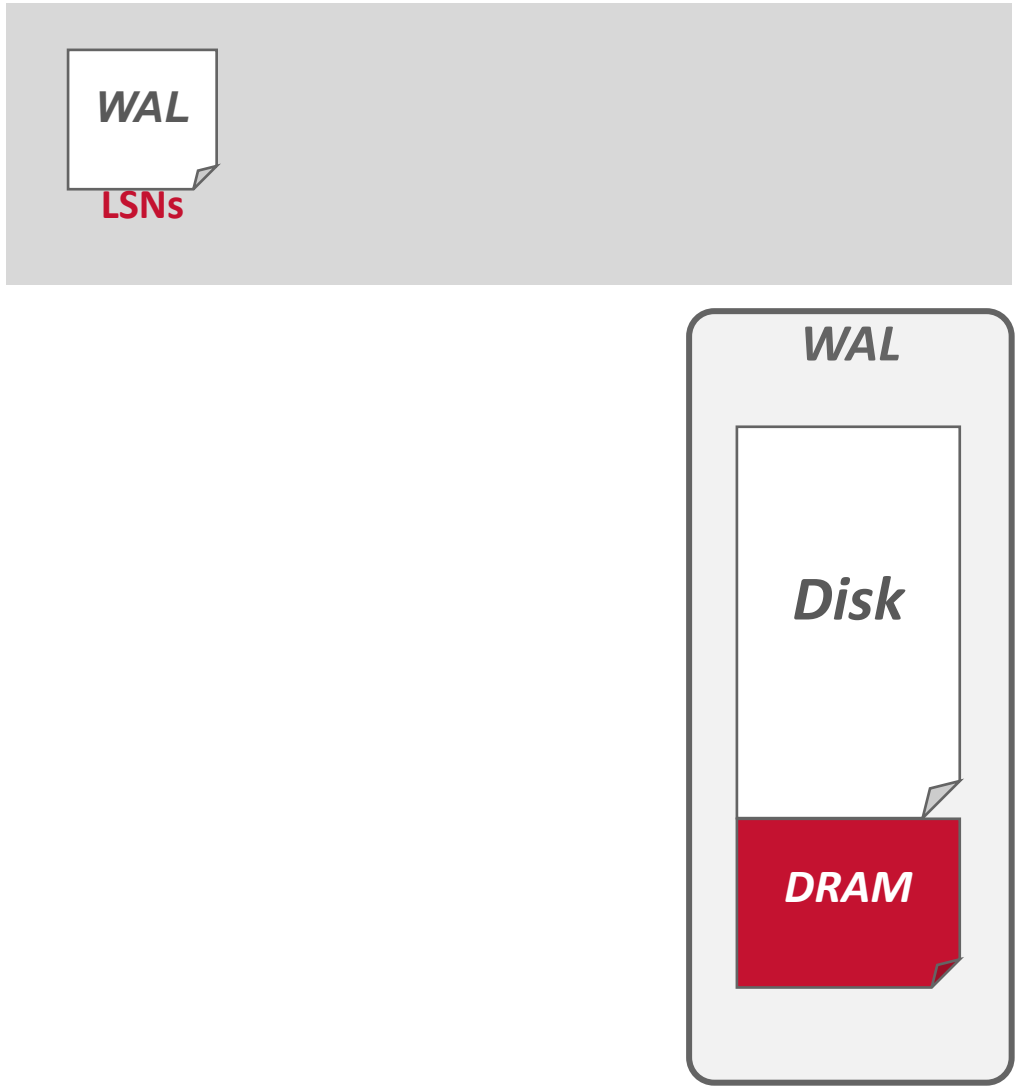
Each data page contains a **pageLSN**

→The LSN of the most recent log record that updated the page

Systems keep track of **flushedLSN**

→ the max LSN flushed so far

WAL: Before a page$_x$ is written, **pageLSN$_x$ <= flushedLSN**



WAL bookkeeping

LSN: Unique and monotonically increasing

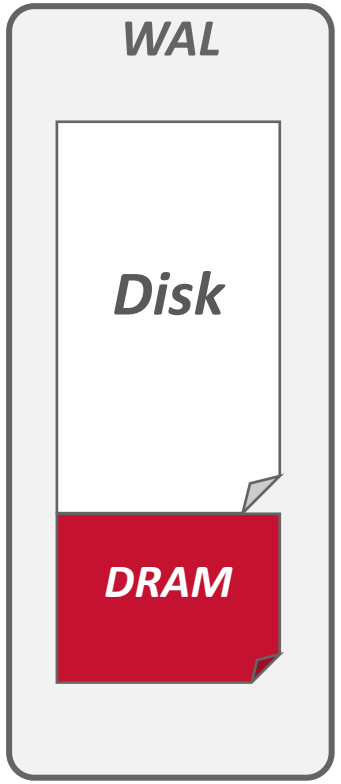
Each data page contains a **pageLSN**

→The LSN of the most recent log record that updated the page

Systems keep track of **flushedLSN**

→ the max LSN flushed so far

WAL: Before a page$_x$ is written, **pageLSN$_x$ <= flushedLSN**



WAL bookkeeping

LSN: Unique and monotonically increasing

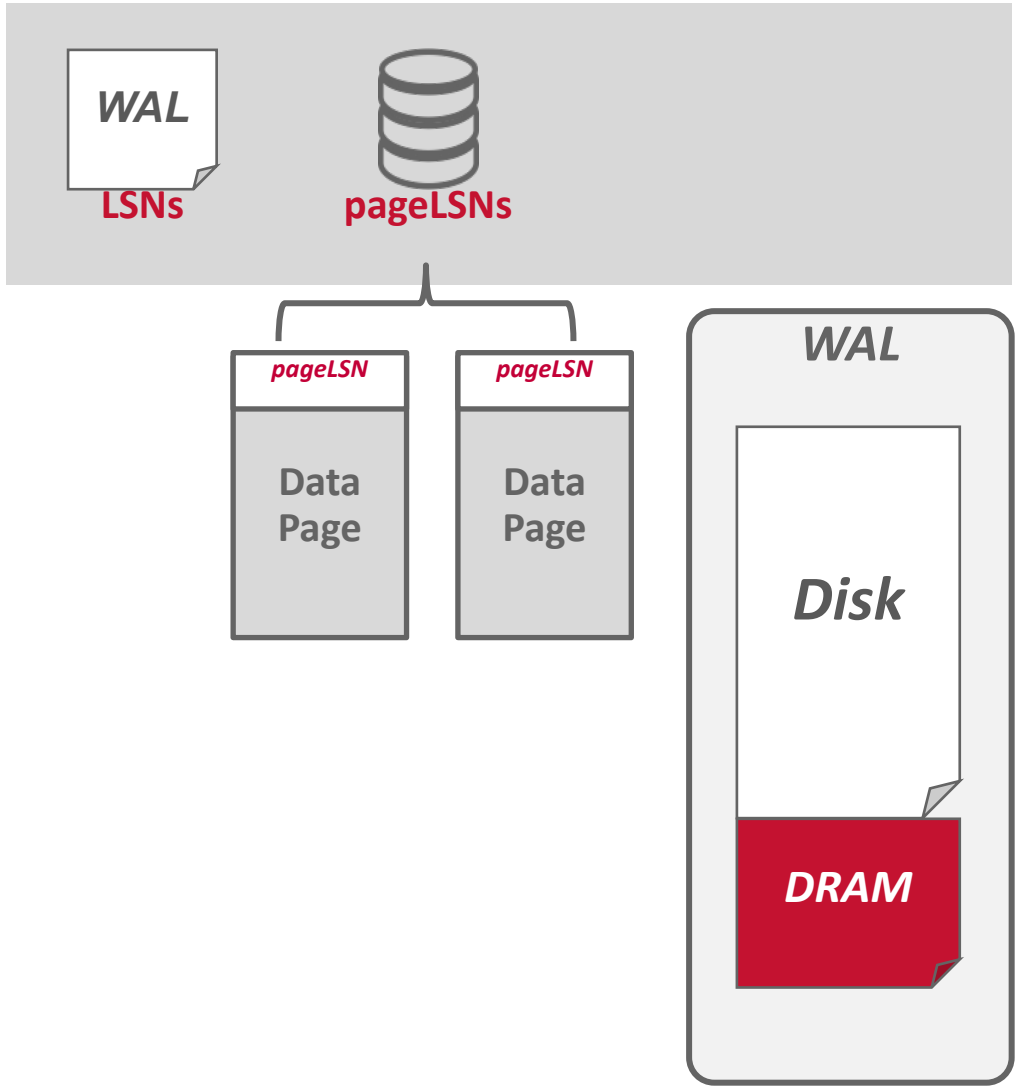
Each data page contains a **pageLSN**

→The LSN of the most recent log record that updated the page

Systems keep track of **flushedLSN**

→ the max LSN flushed so far

WAL: Before a page$_x$ is written, **pageLSN$_x$ <= flushedLSN**



WAL bookkeeping

LSN: Unique and monotonically increasing

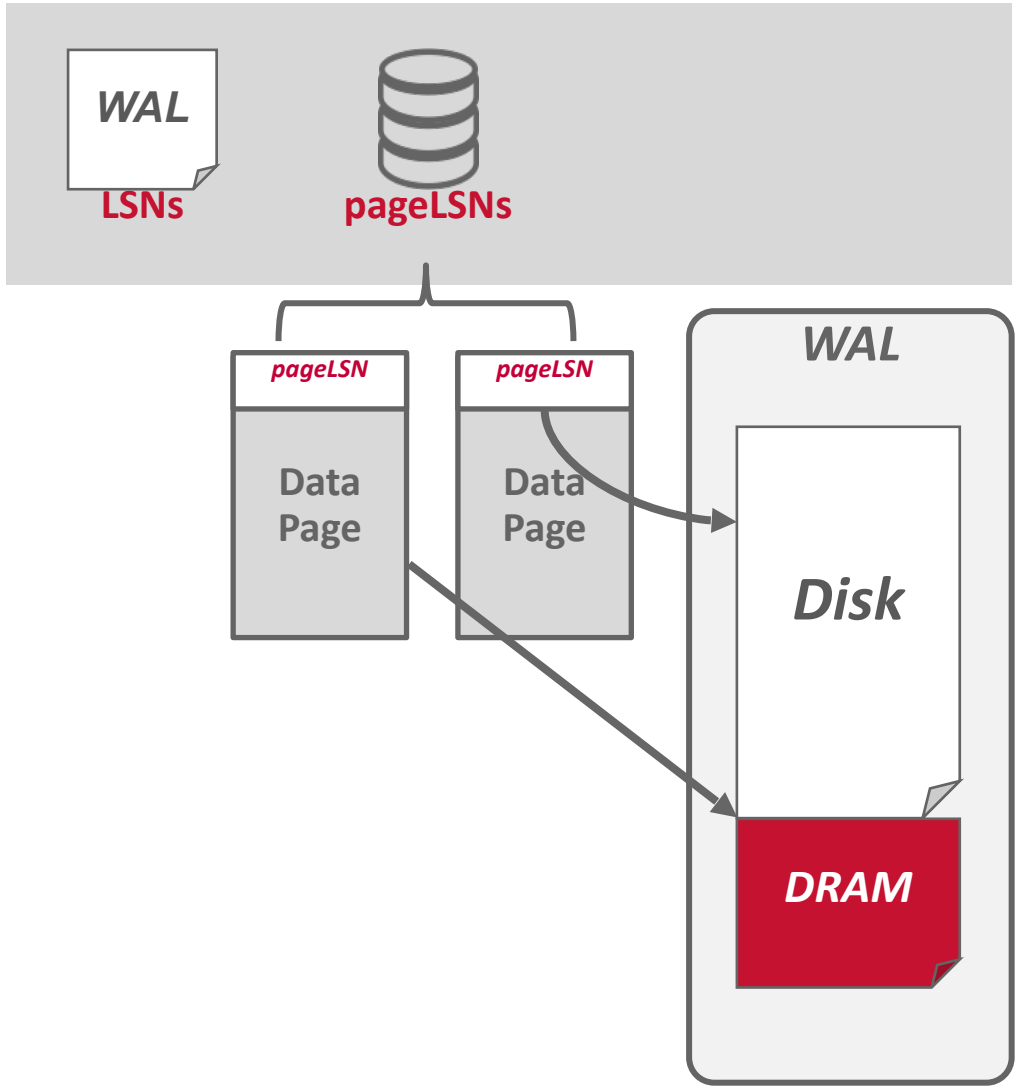
Each data page contains a **pageLSN**

→The LSN of the most recent log record that updated the page

Systems keep track of **flushedLSN**

→ the max LSN flushed so far

WAL: Before a page$_x$ is written, **pageLSN$_x$ <= flushedLSN**



WAL bookkeeping

LSN: Unique and monotonically increasing

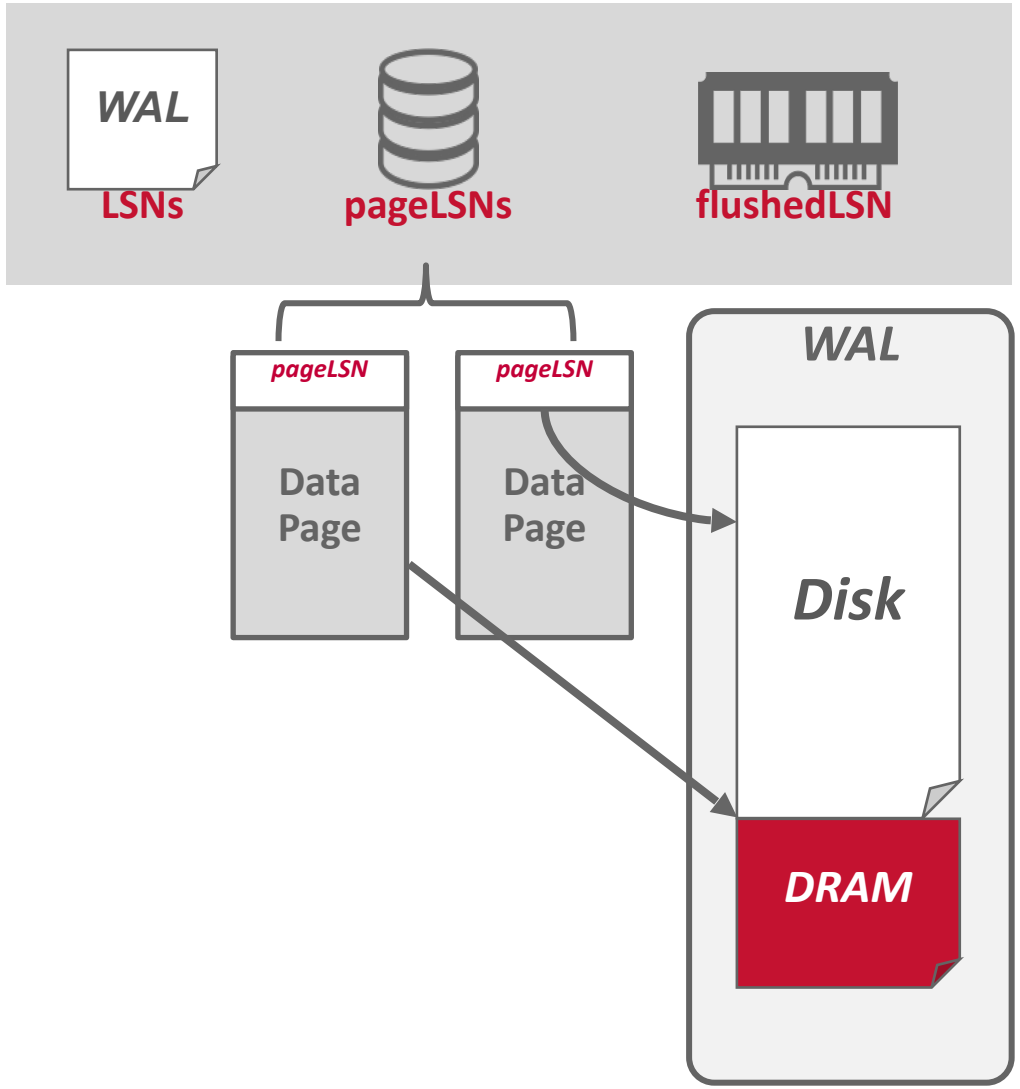
Each data page contains a **pageLSN**

→The LSN of the most recent log record that updated the page

Systems keep track of **flushedLSN**

→ the max LSN flushed so far

WAL: Before a $page_x$ is written, **$pageLSN_x$ <= flushedLSN**



WAL bookkeeping

LSN: Unique and monotonically increasing

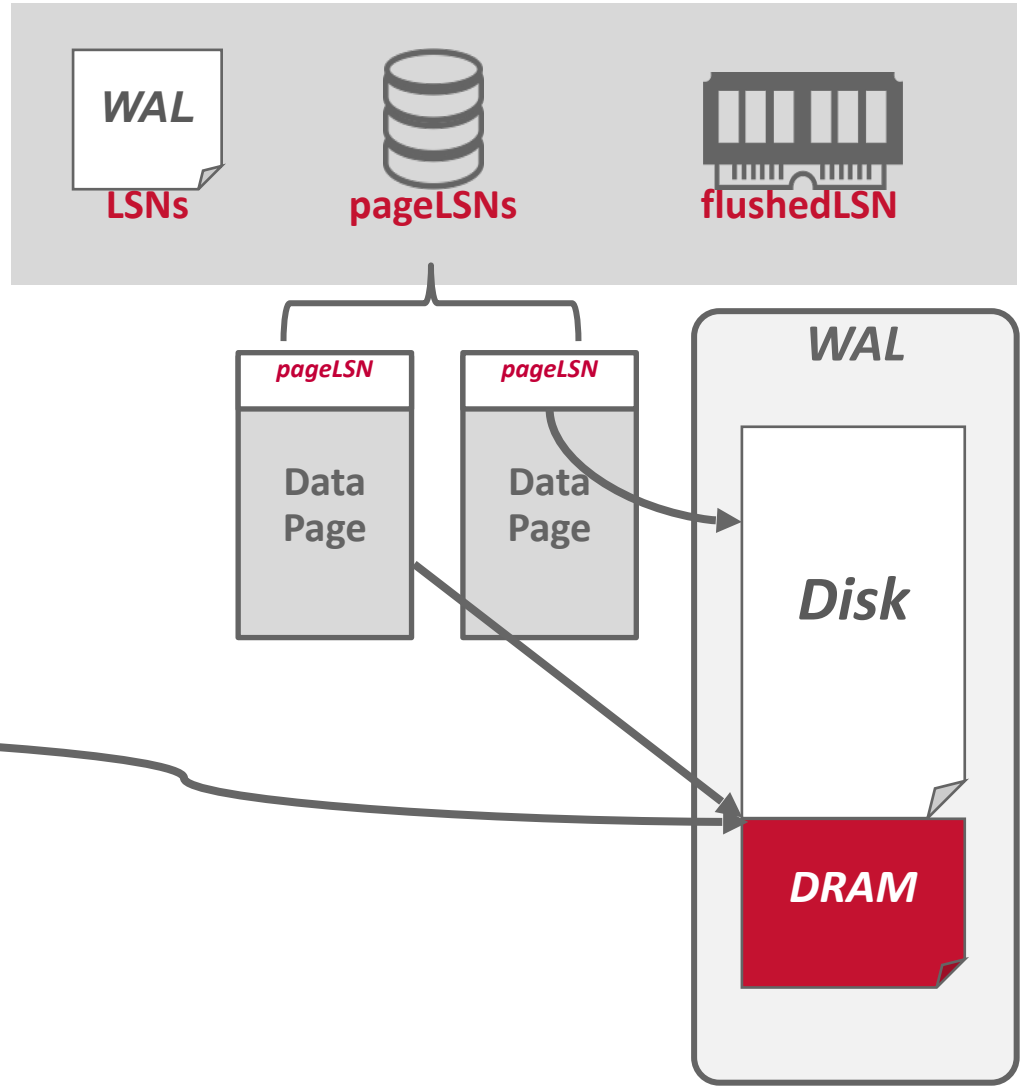
Each data page contains a **pageLSN**

→The LSN of the most recent log record that updated the page

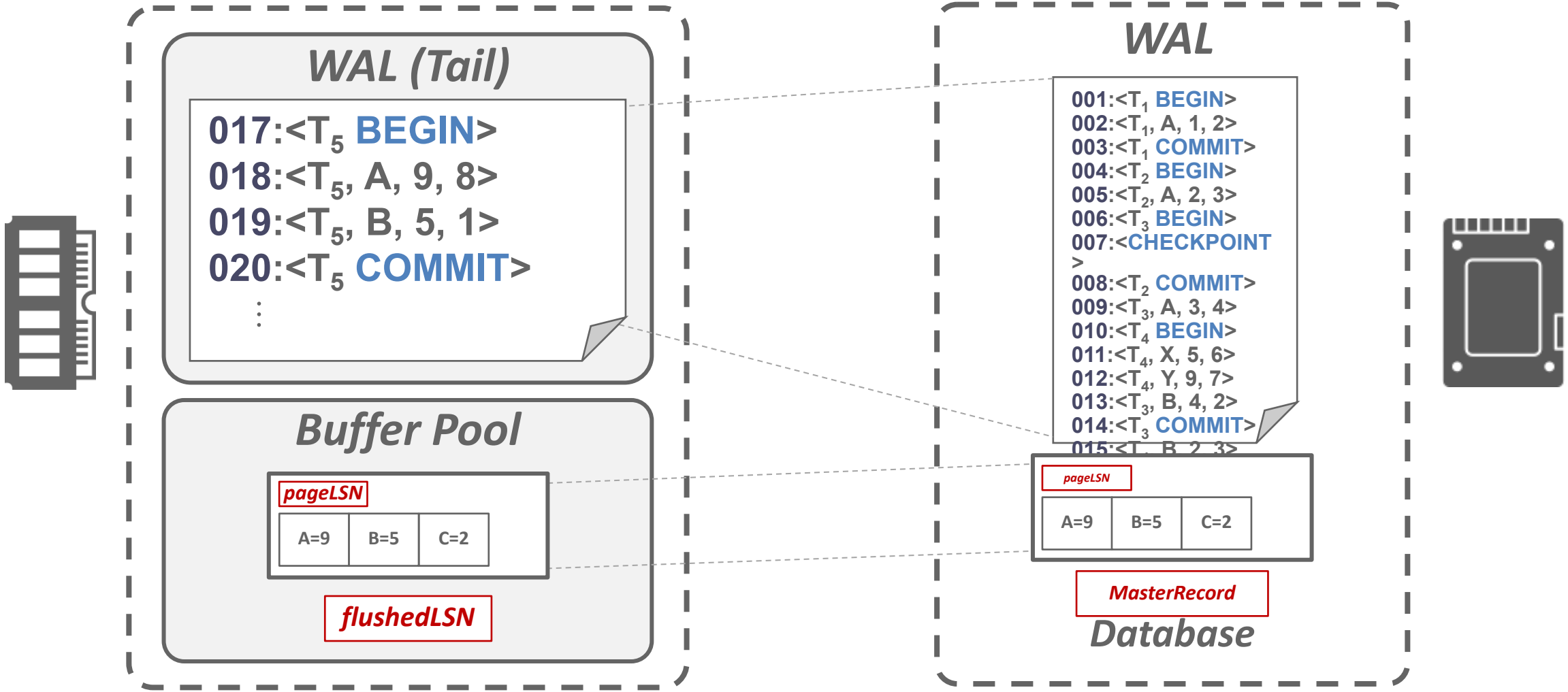
Systems keep track of **flushedLSN**

→ the max LSN flushed so far

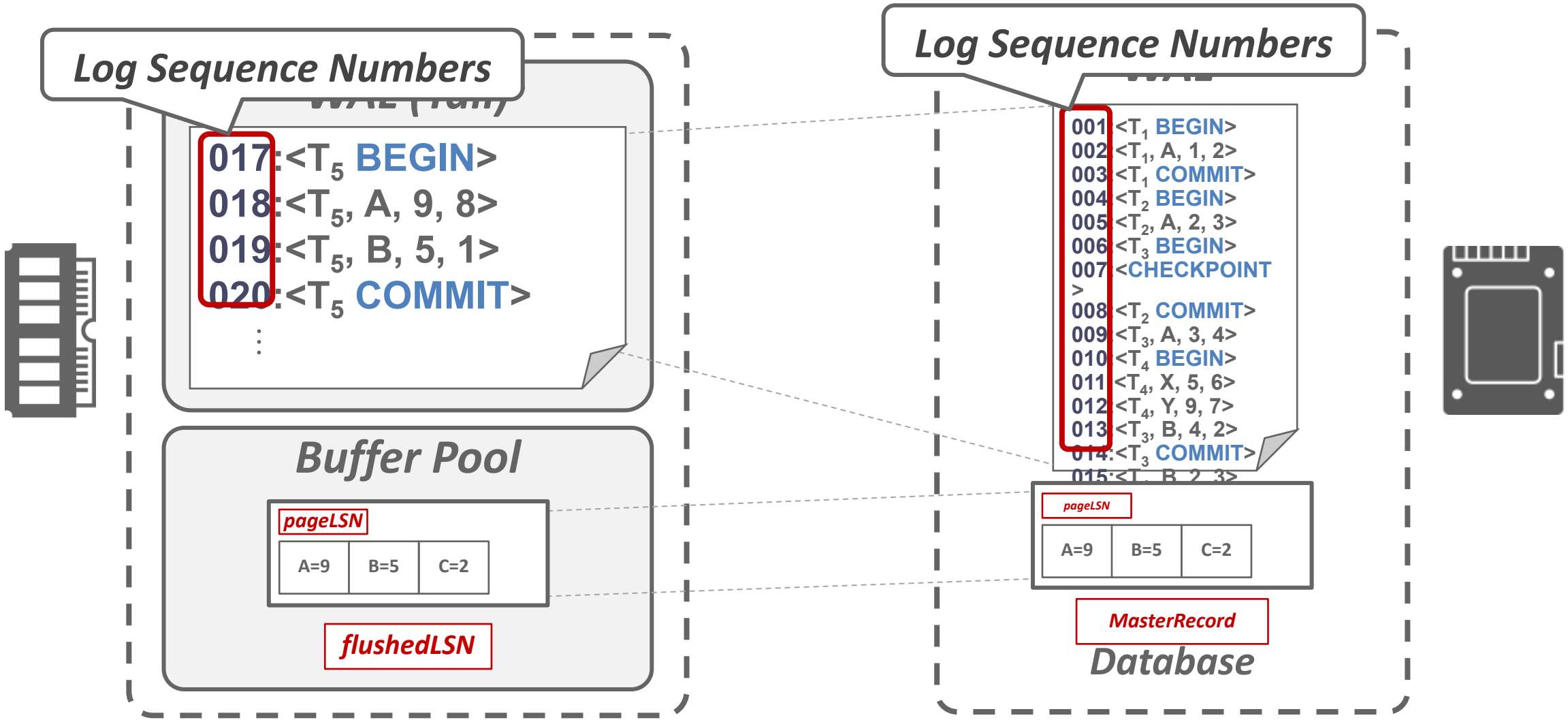
WAL: Before a page$_x$ is written, **pageLSN$_x$ <= flushedLSN**



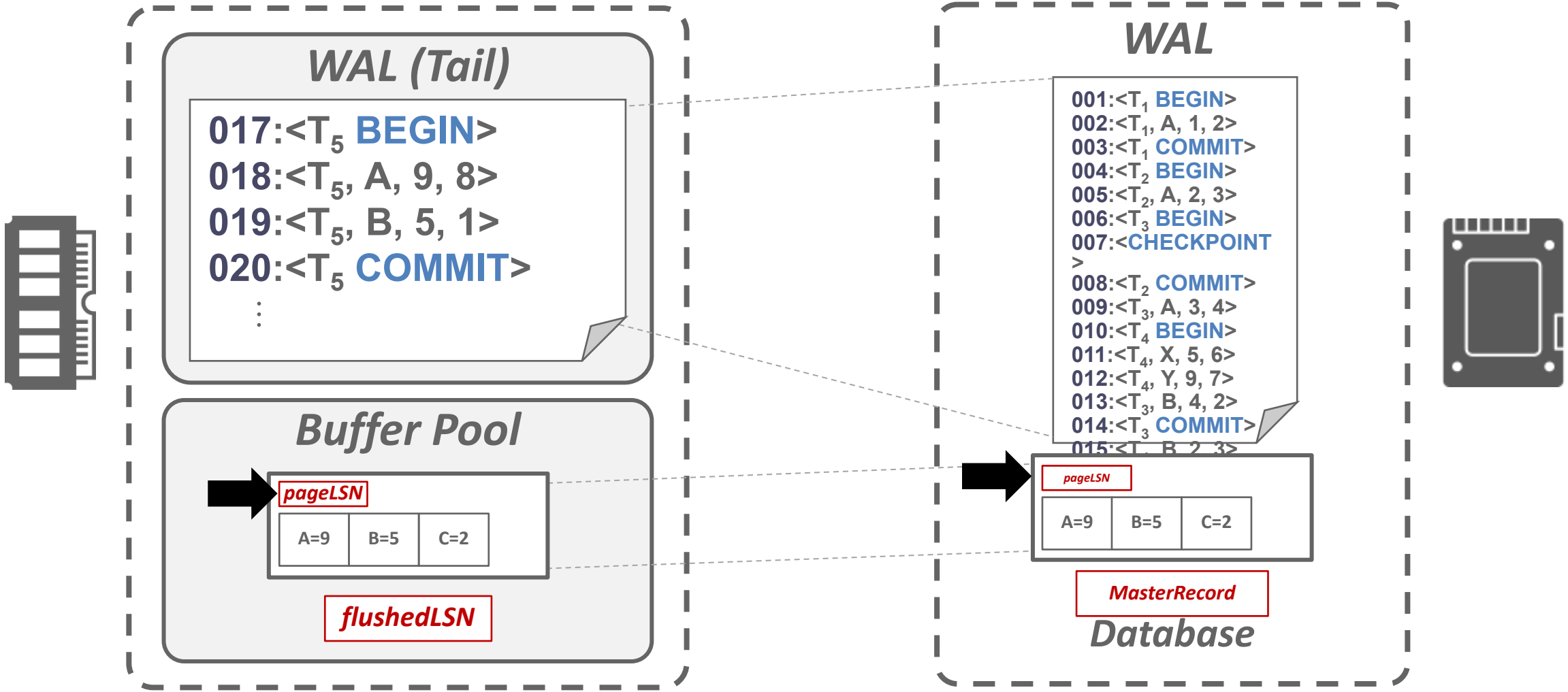
Writing log records



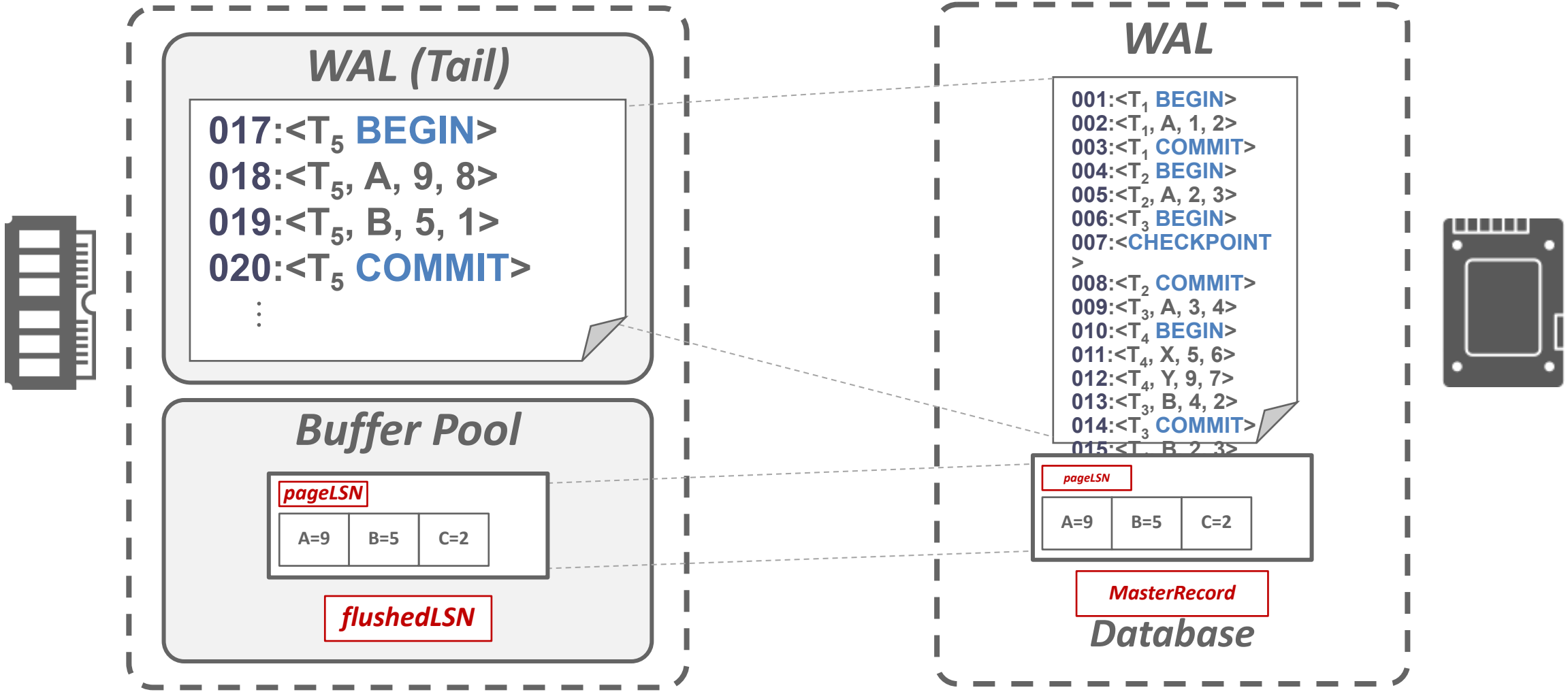
Writing log records



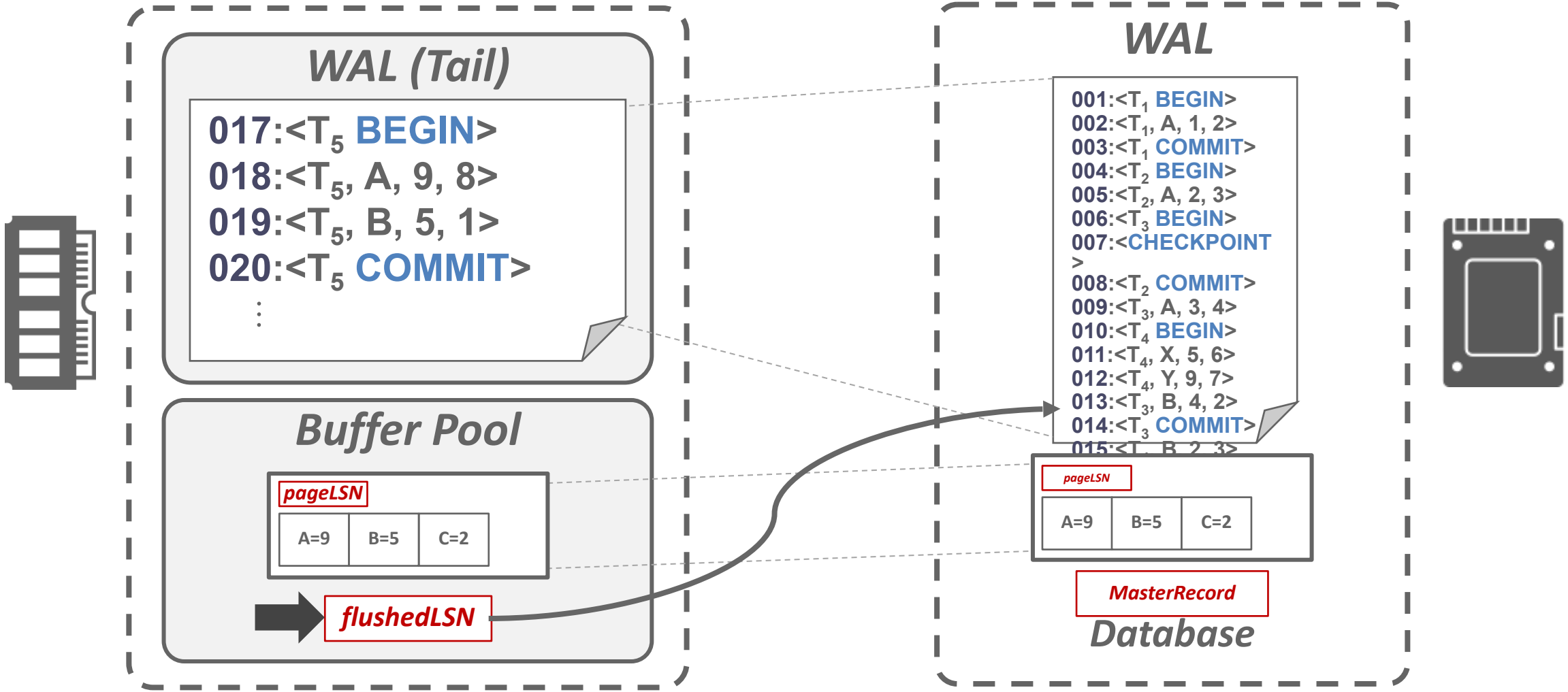
Writing log records



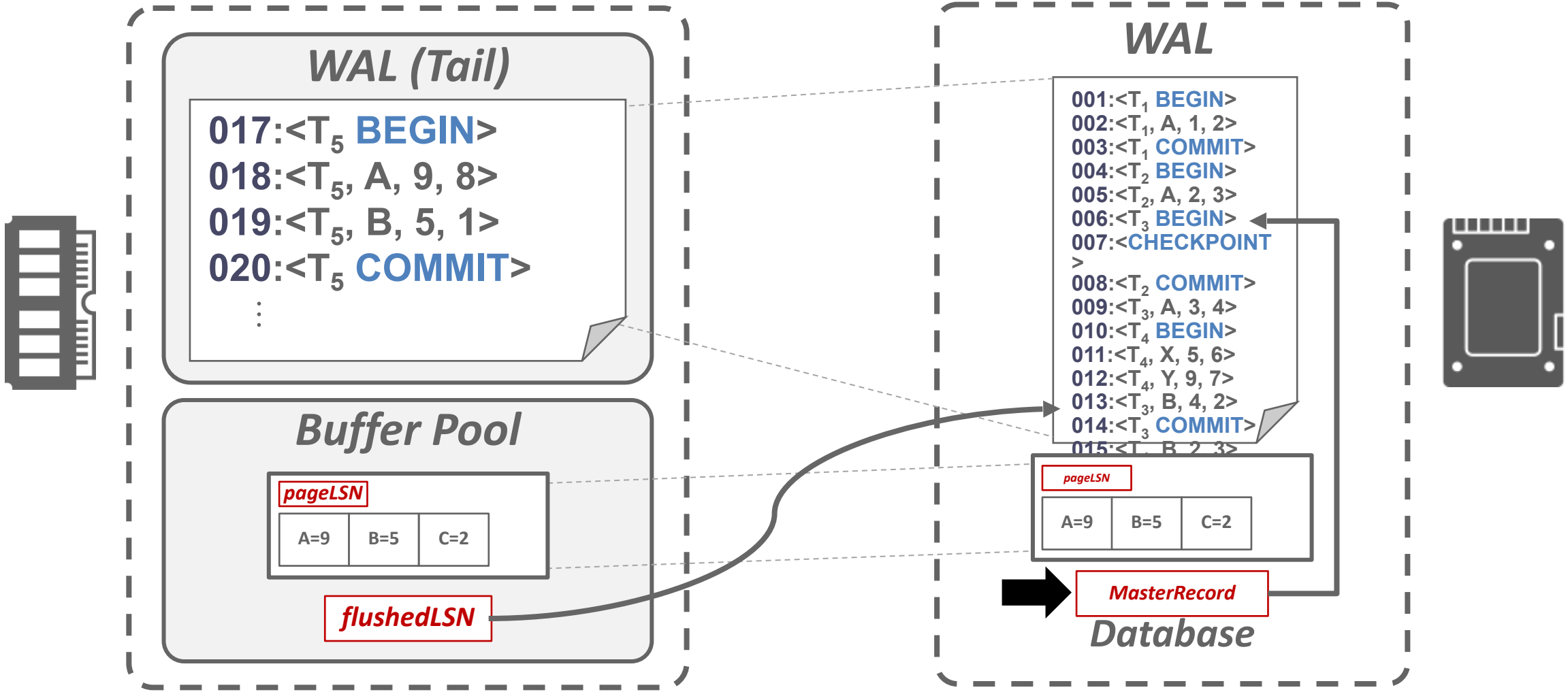
Writing log records



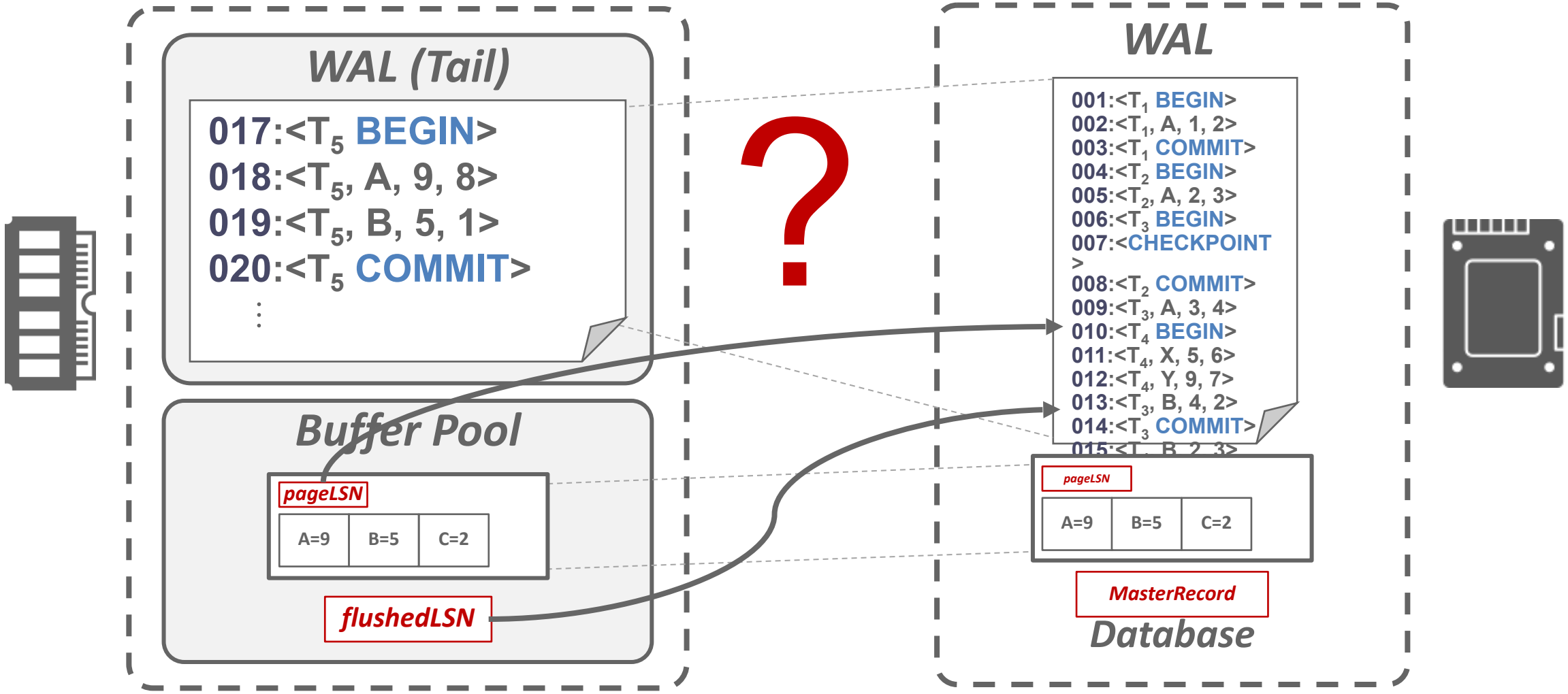
Writing log records



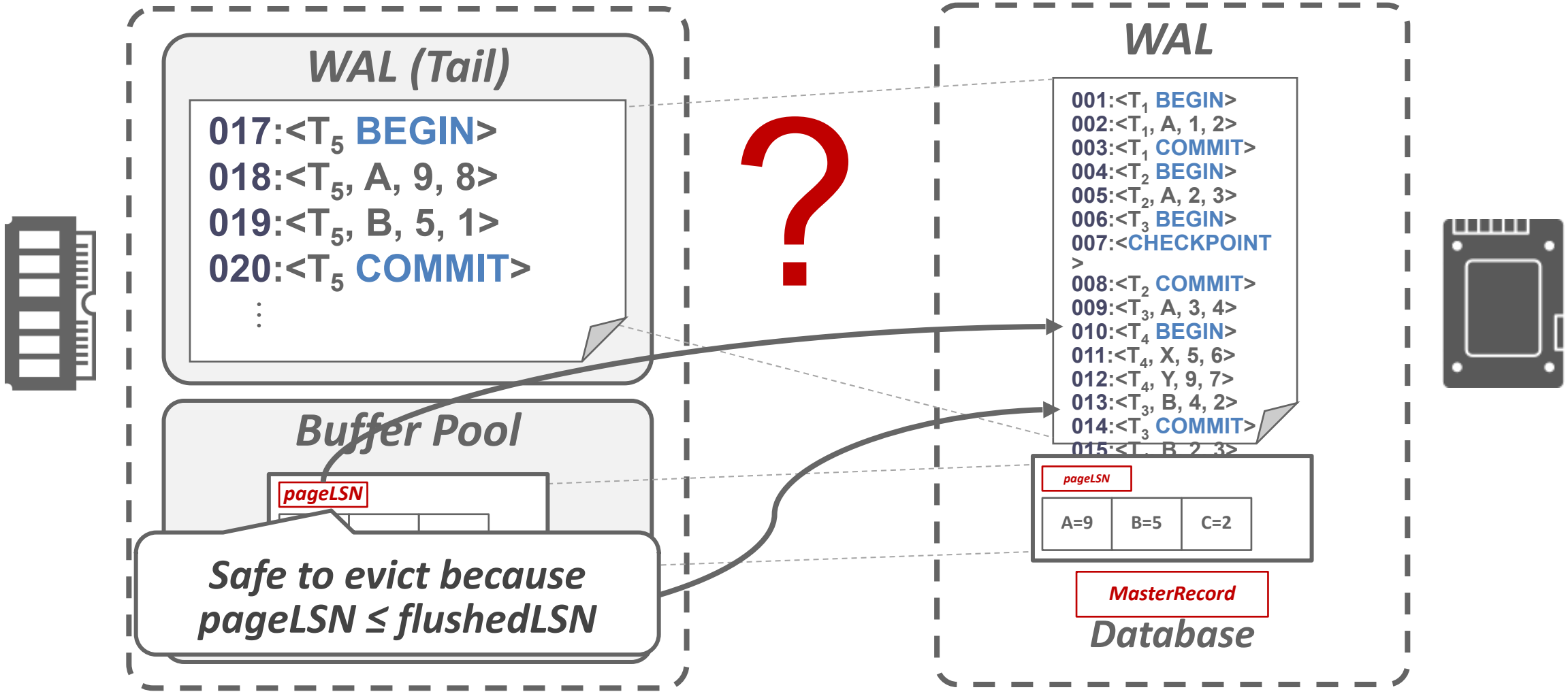
Writing log records



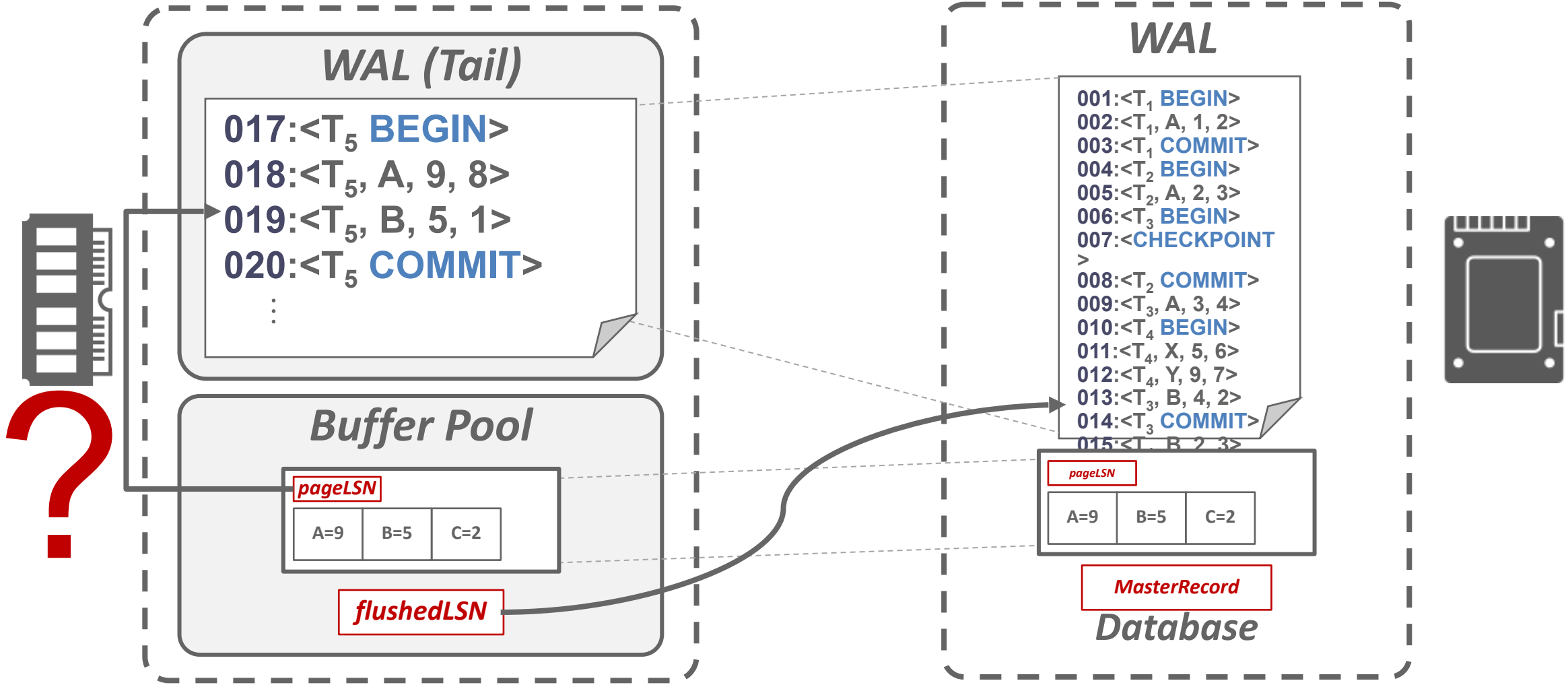
Writing log records



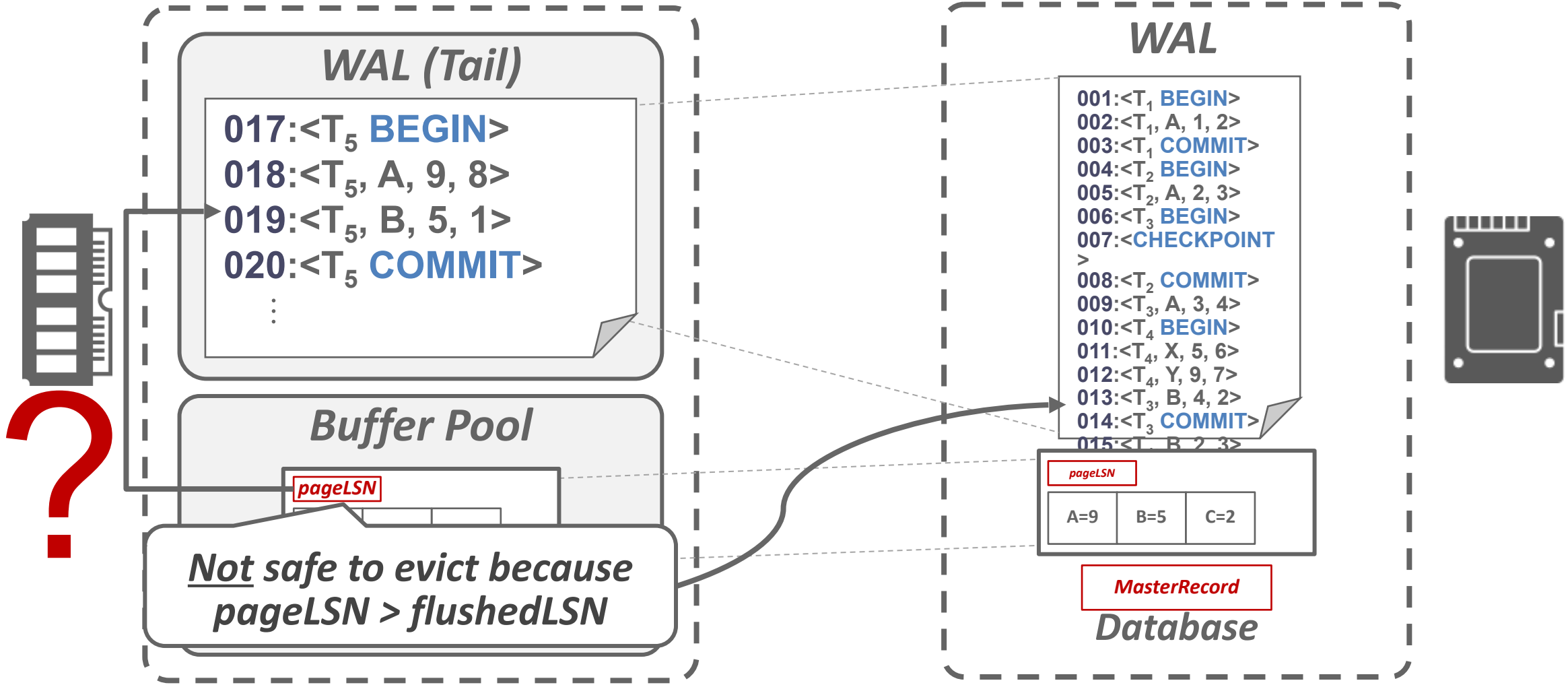
Writing log records



Writing log records



Writing log records



Writing LOG records

- All log records have an LSN
- Update the pageLSN every time a txn modifies a record in the page
- Update the flushedLSN in memory every time the DBMS writes the WAL buffer to disk

Normal execution

- Each txn invokes a sequence of reads and writes, followed by a commit or rollback

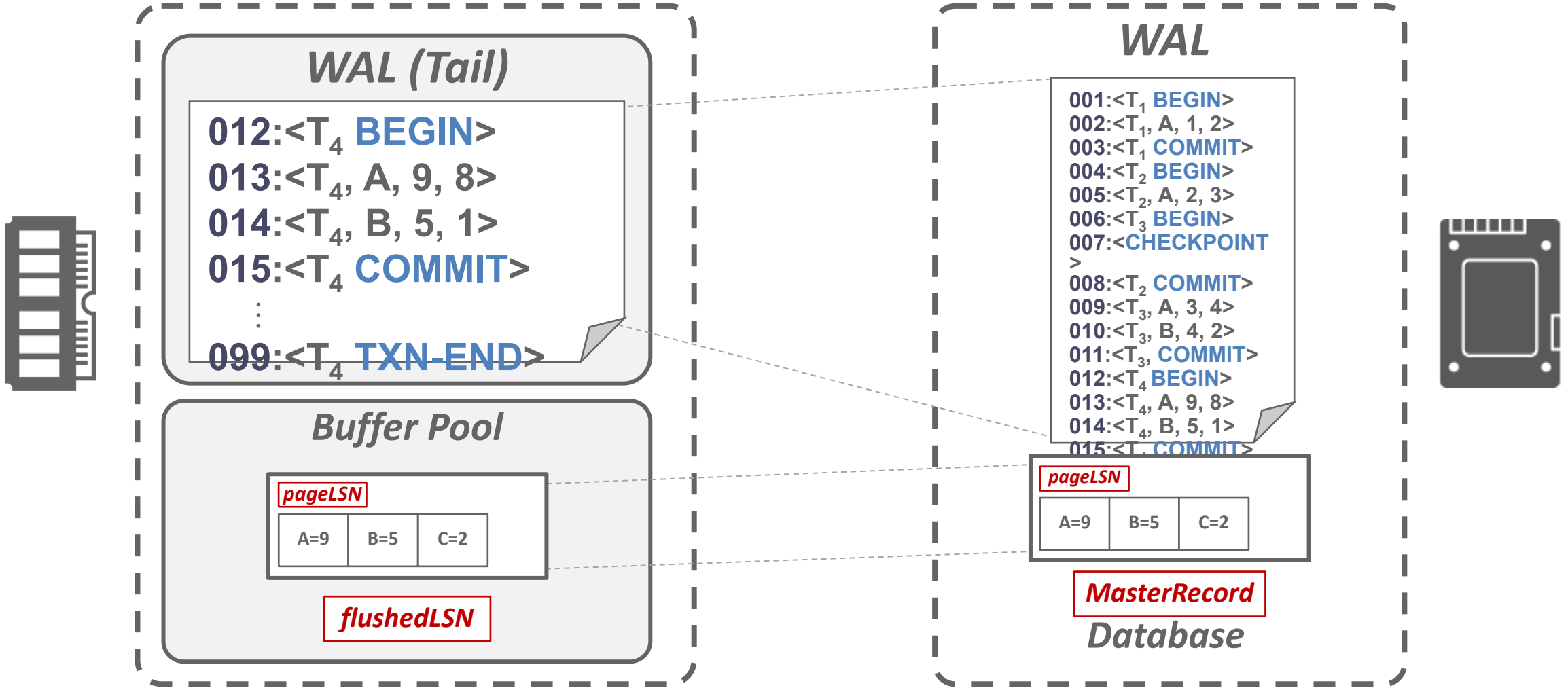
- Assumptions made right now:
  - All log records fit within a single page
  - Disk writes are atomic
  - Single-versions tuples with Strong Strict 2PL
  - **Steal** + **No-force** buffer management with WAL

Transaction commit

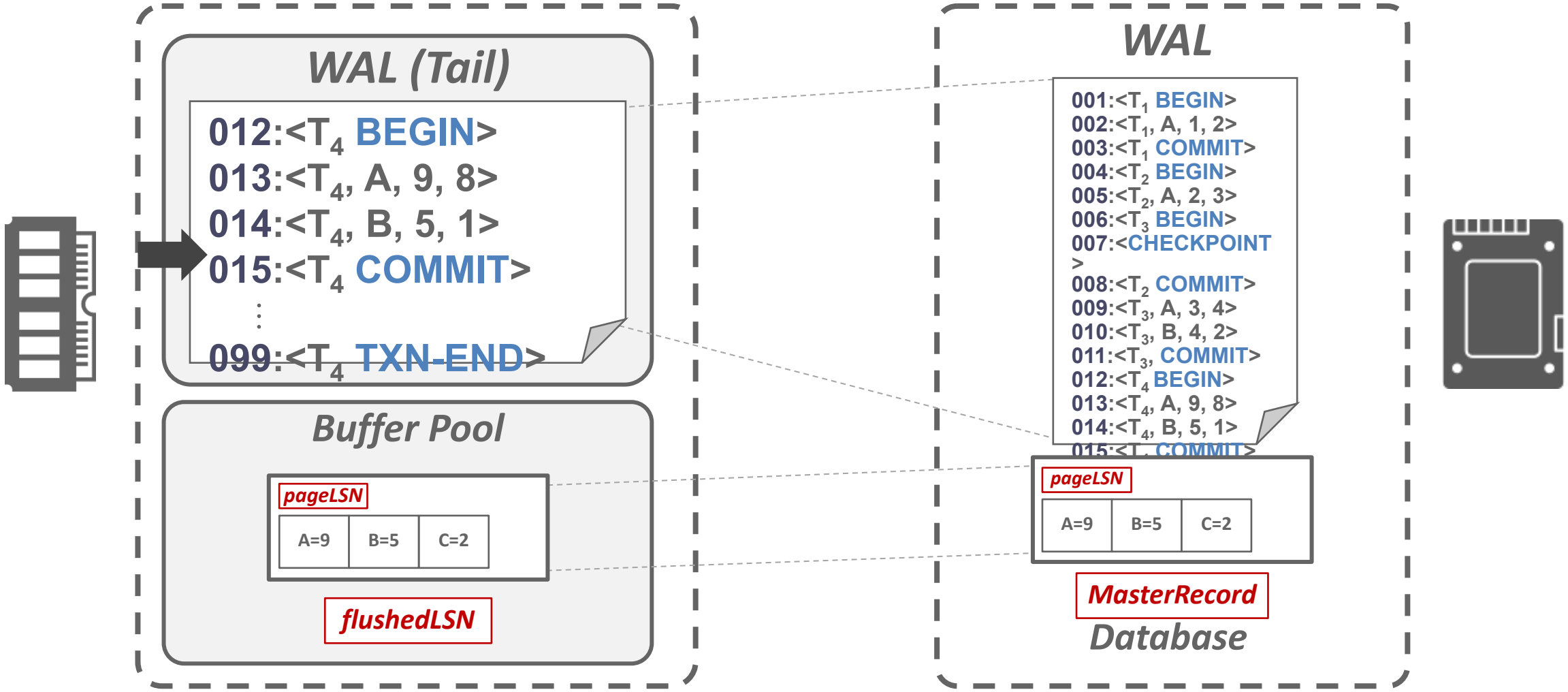
- When a txn commits, the DBMS writes a **COMMIT** record to a log and guarantees all log records up to txn's **COMMIT** record are flushed to disk
  - Log flushes are sequential, synchronous writes to disk
  - Many log records per log page

- When the commit succeeds, write a **TXN-END** record to log
  - Indicates that no new log record for a txn will appear in the log ever again
  - This does not need to be flushed immediately

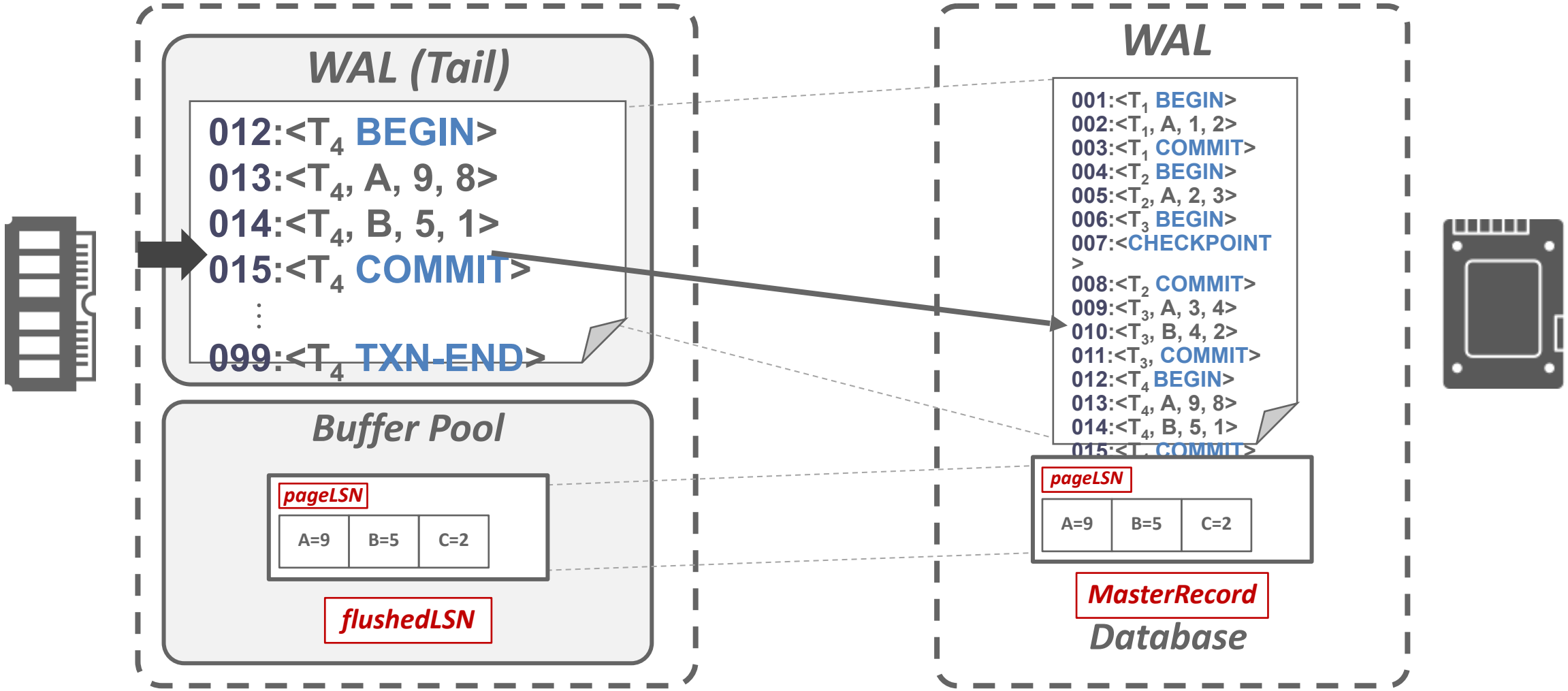
Transaction commit



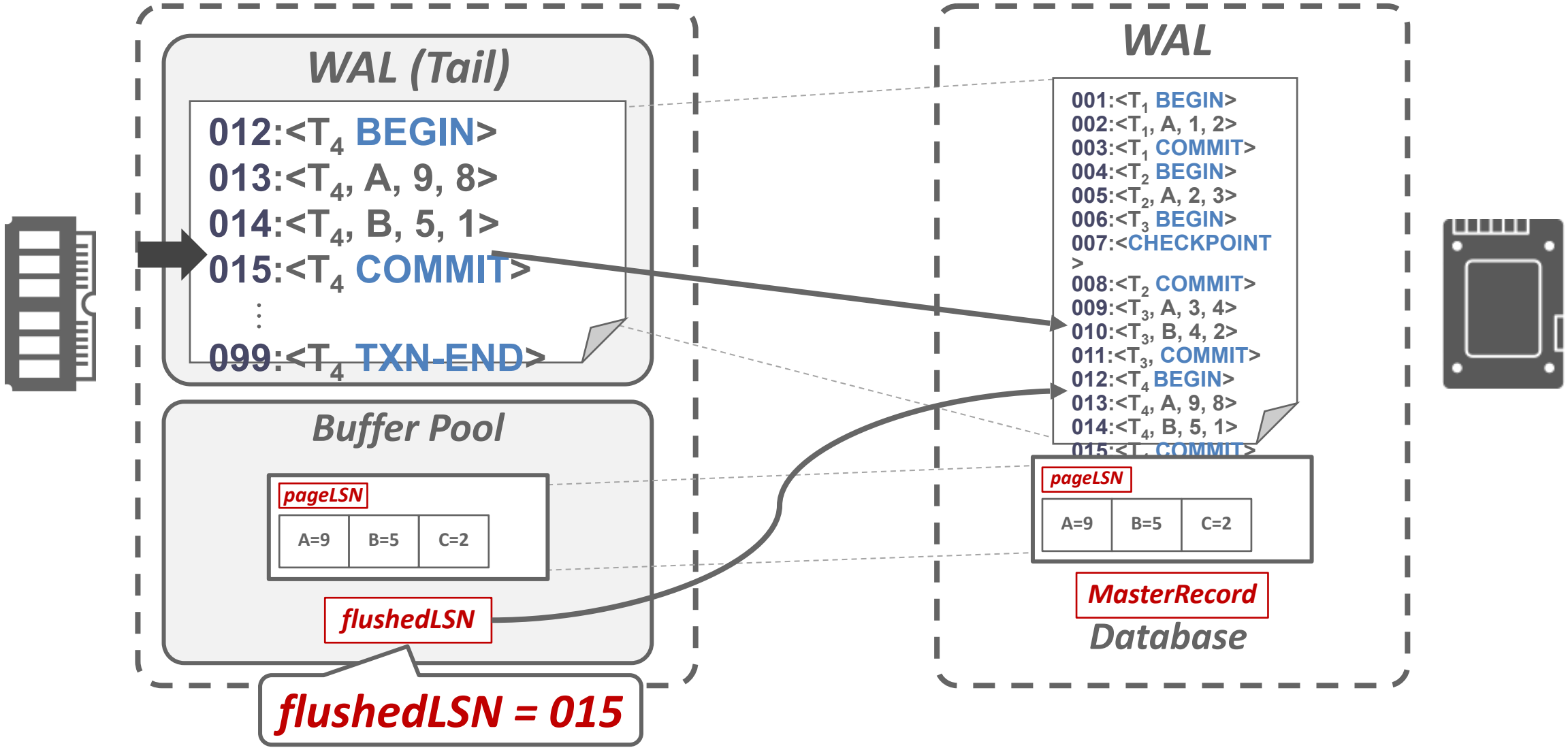
Transaction commit



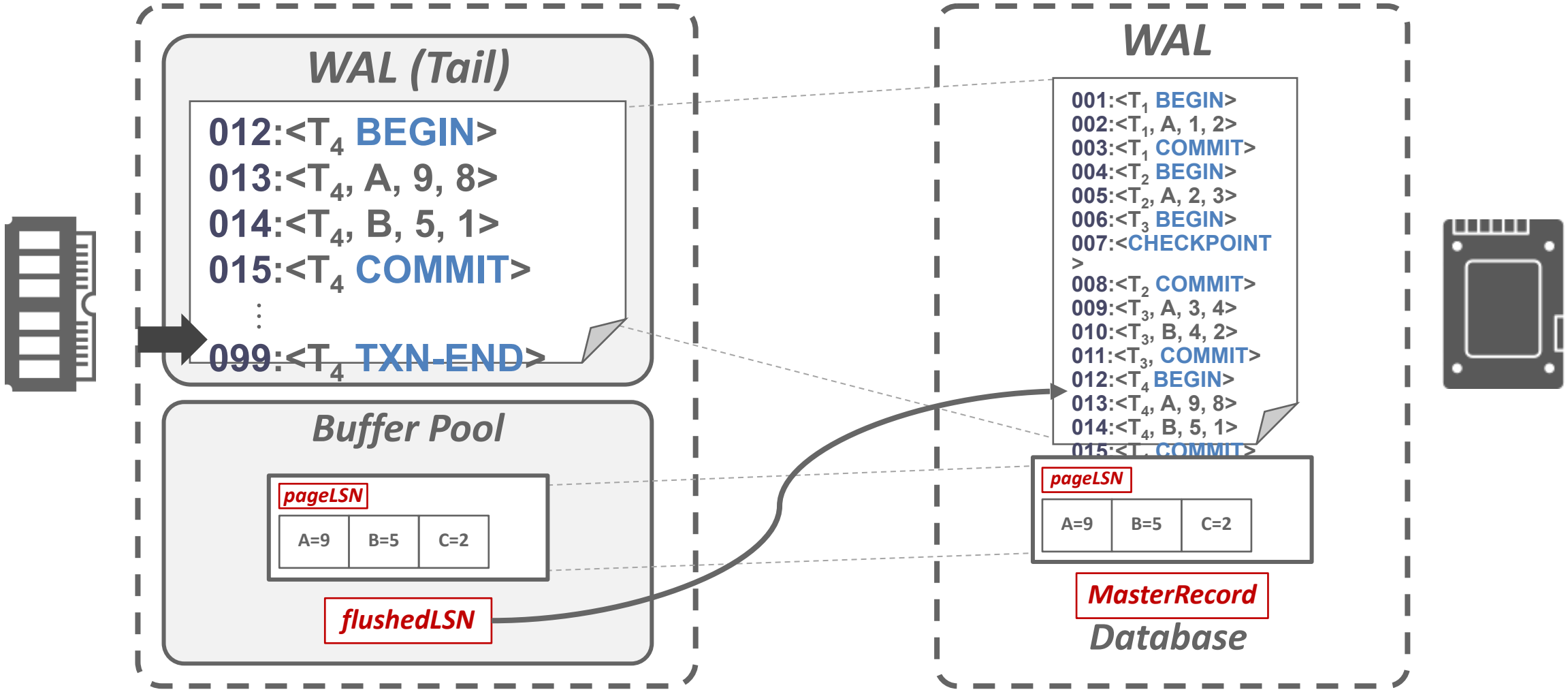
Transaction commit



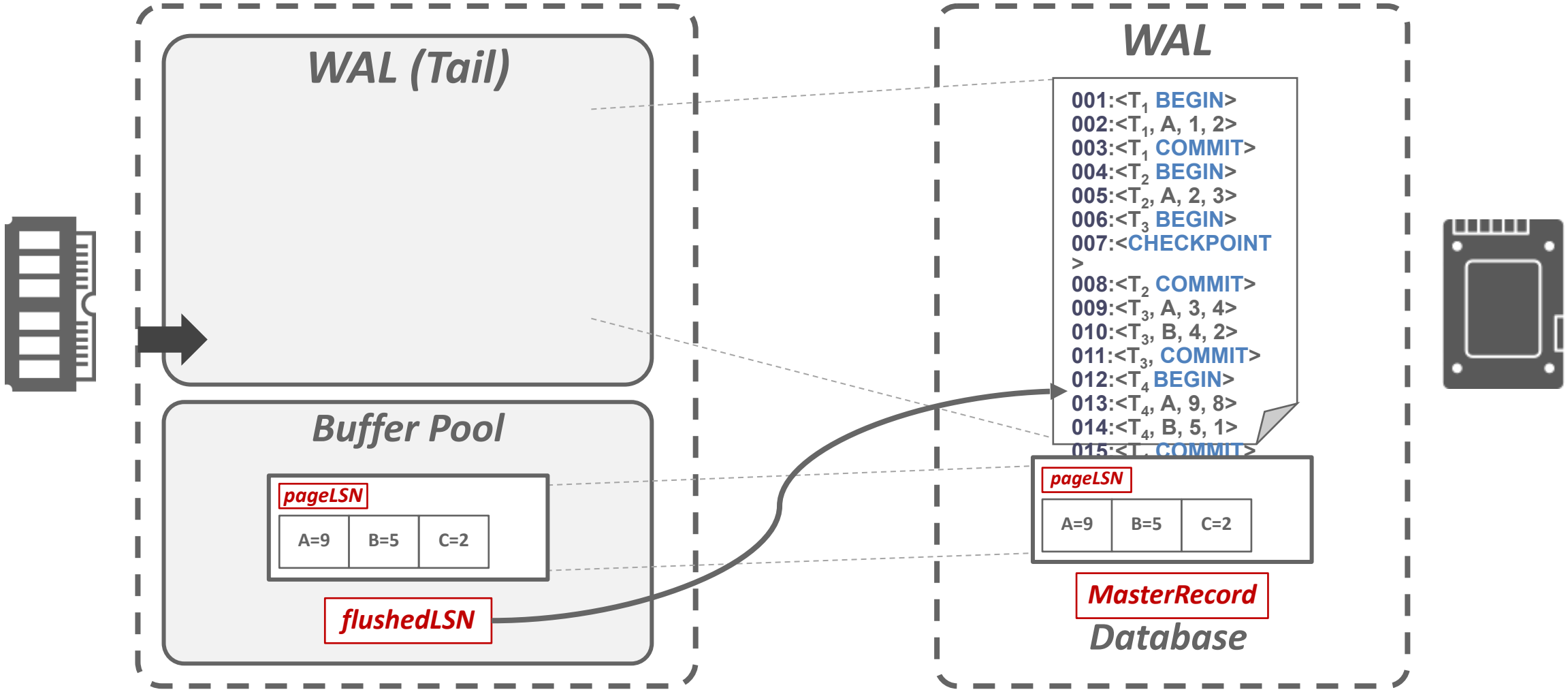
Transaction commit



Transaction commit



Transaction commit



Transaction abort

- Aborting a txn is a special case of the ARIES undo operation applied to only one txn

- Need to add another field to log records:
  - **prevLSN**: The previous **LSN** for the txn
  - This maintains a linked-list for each txn that makes it easy to walk through its records

Compensation log records (CLR)

- Ensures that undo actions during recovery are **logged and redoable**
  - Prevents the same undo if a crash happens again

- A **CLR** describes the actions taken to undo the actions of a previous update record

- It has all the fields of an updated log record plus the **undoNextLSN** pointer (i.e, the next-to-be-undone LSN)

- **CLRs** are added to log records but the DBMS does **not** wait for them to be flushed before notifying the application that the txn aborted

Abort algorithm

- First write an **ABORT** record to log for the txn
- Then analyze the txn's updates in reverse order
- For each updated record:
  - Write a **CLR** entry to the log
  - Restore old value
- Lastly, write a **TXN-END** record and release locks

Note: CLRs never need to be undone

Today's focus

- Logging
  - Buffer pool policies
  - WAL
  - Logging schemes
- Recovery
  - LSN
  - Normal checkpoint and abort operations
  - Checkpoint
  - Recovery algorithm

Checkpoints

WAL can grow infinite → need an approach to have faster recovery

- DBMS checkpoints WAL to truncate it
- Works by halting everything when DBMS takes a checkpoint to ensure a consistent snapshot:
  - Halt the start of any new txns
  - Wait until all active txns finish executing
  - Flushes dirty pages on disk

- Also known as **non-fuzzy checkpoint**
  - **Bad** for runtime performance but makes recovery **easy**

Slightly better checkpoints

Pause modifying txns while the DBMS takes the checkpoint

- Flushes dirty pages on disk
- Prevent queries from acquiring write latch on table/index pages
- Don’t have to wait until all txns finish before taking the checkpoint

Record internal state as of the beginning of the checkpoint to handle partial updates

- **Active Transaction Table (ATT)**
- **Dirty Page Table (DPT)**

Active transaction table (ATT)

One entry per currently active txn

- **txnID:** Unique txn identifier
- **status:** The current “mode” of the txn
- **lastLSN:** Most recent LSN created by the txn

Txn status codes:

- Running (R)
- Committing (C)
- Candidate for undo (U)

Dirty page table (DPT)

- Keep track of which pages in the buffer pool contain changes that have not been flushed to disk

- One entry per dirty page in the buffer pool
  - **recLSN:** The LSN of the log record that first caused the page to be dirty

Slightly better checkpoints

At the first checkpoint, assuming P_{11} was flushed, T_2 is still running and there is only one dirty page (P_{22})

At the second checkpoint, assuming P_{22} was flushed, T_2 and T_3 are active and the dirty pages are (P_{11}, P_{33})

This still is not ideal because the DBMS must stall txns during checkpoint …

001:<T_1 BEGIN>
002:<T_2 BEGIN>
003:<T_1, A→P_{11}, 100, 120>
004:<T_1 COMMIT>
005:<T_2, C→P_{22}, 100, 120>
006:<T_1 TXN-END >
007:<CHECKPOINT
- ATT={T_2},
- DPT={P_{22}}>

008:<T_3 BEGIN>
009:<T_2, A→P_{11}, 120, 130>
010:<T_2 COMMIT>
011:<T_3, B→P_{33}, 200, 400>
012:<CHECKPOINT
- ATT={T_2,T_3},
- DPT={P_{11},P_{33}}>

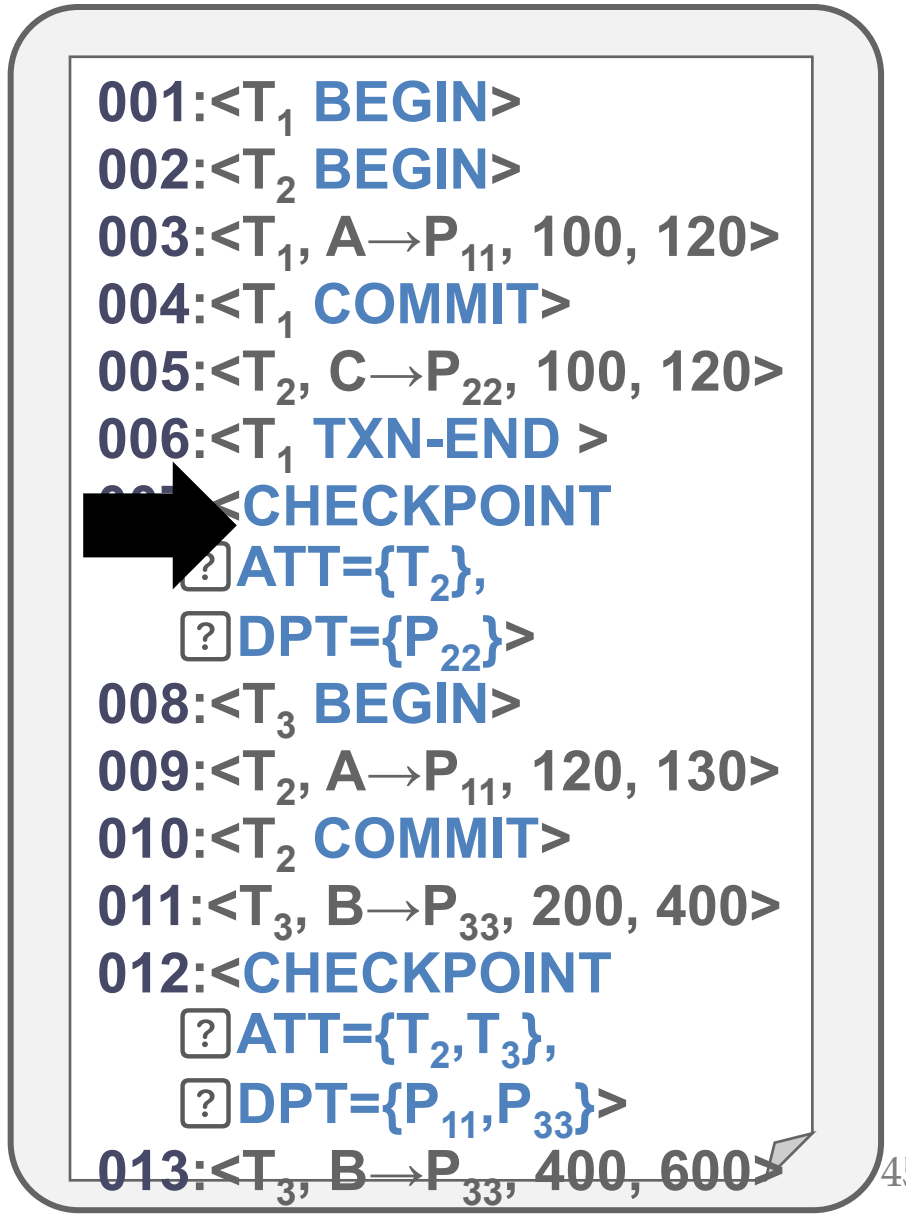
013:<T_3, B→P_{33}, 400, 600>

Slightly better checkpoints

At the first checkpoint, assuming P_{11} was flushed, T_2 is still running and there is only one dirty page (P_{22})

At the second checkpoint, assuming P_{22} was flushed, T_2 and T_3 are active and the dirty pages are (P_{11}, P_{33})

This still is not ideal because the DBMS must stall txns during checkpoint …

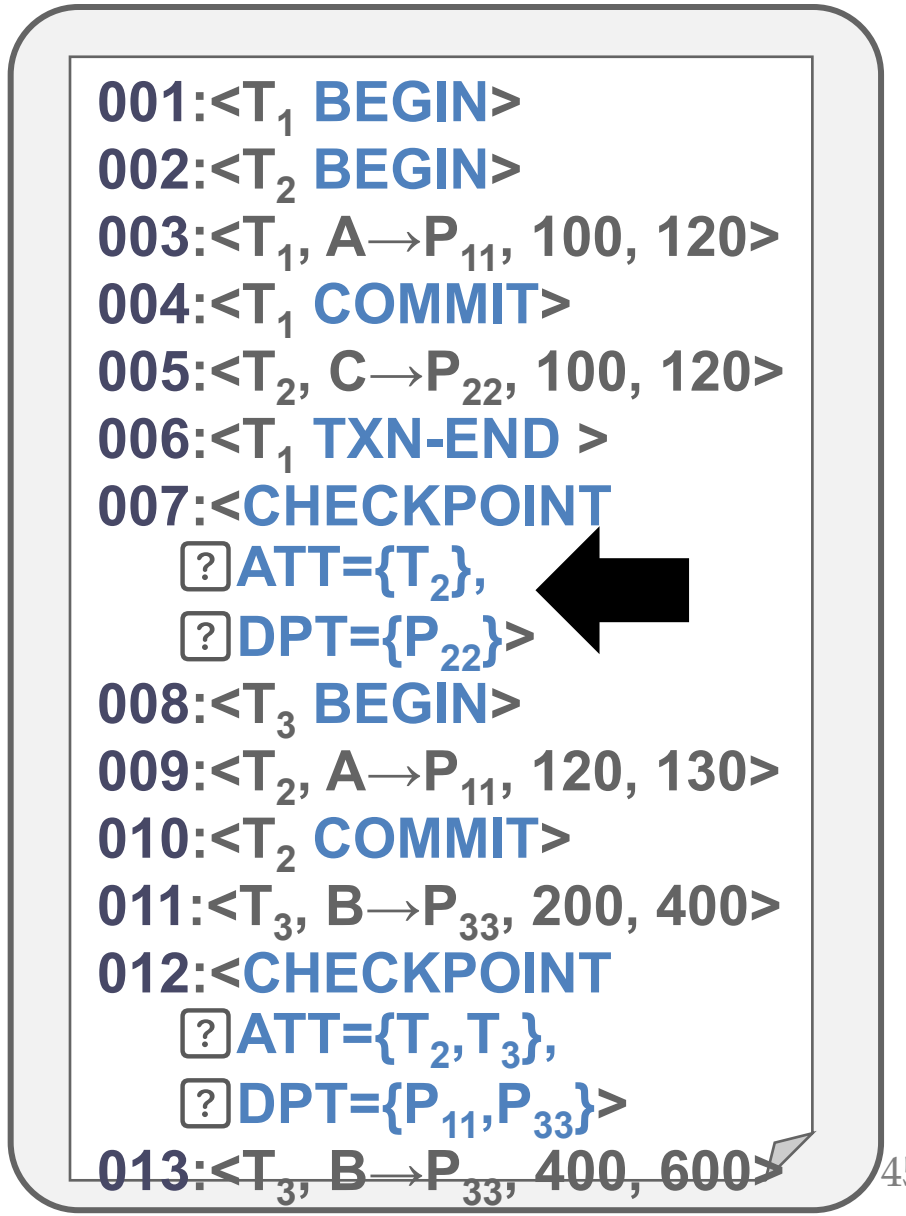


Slightly better checkpoints

At the first checkpoint, assuming P_{11} was flushed, T_2 is still running and there is only one dirty page (P_{22})

At the second checkpoint, assuming P_{22} was flushed, T_2 and T_3 are active and the dirty pages are (P_{11}, P_{33})

This still is not ideal because the DBMS must stall txns during checkpoint …



Slightly better checkpoints

At the first checkpoint, assuming P_{11} was flushed, T_2 is still running and there is only one dirty page (P_{22})

At the second checkpoint, assuming P_{22} was flushed, T_2 and T_3 are active and the dirty pages are (P_{11}, P_{33})

This still is not ideal because the DBMS must stall txns during checkpoint …

001:<T_1 BEGIN>
002:<T_2 BEGIN>
003:<T_1, A→P_{11}, 100, 120>
004:<T_1 COMMIT>
005:<T_2, C→P_{22}, 100, 120>
006:<T_1 TXN-END >
007:<CHECKPOINT
- ATT={T_2},
- DPT={P_{22}}>

008:<T_3 BEGIN>
009:<T_2, A→P_{11}, 120, 130>
010:<T_2 COMMIT>
011:<T_3, B→P_{33}, 200, 400>
012:<CHECKPOINT
- ATT={T_2,T_3},
- DPT={P_{11},P_{33}}>

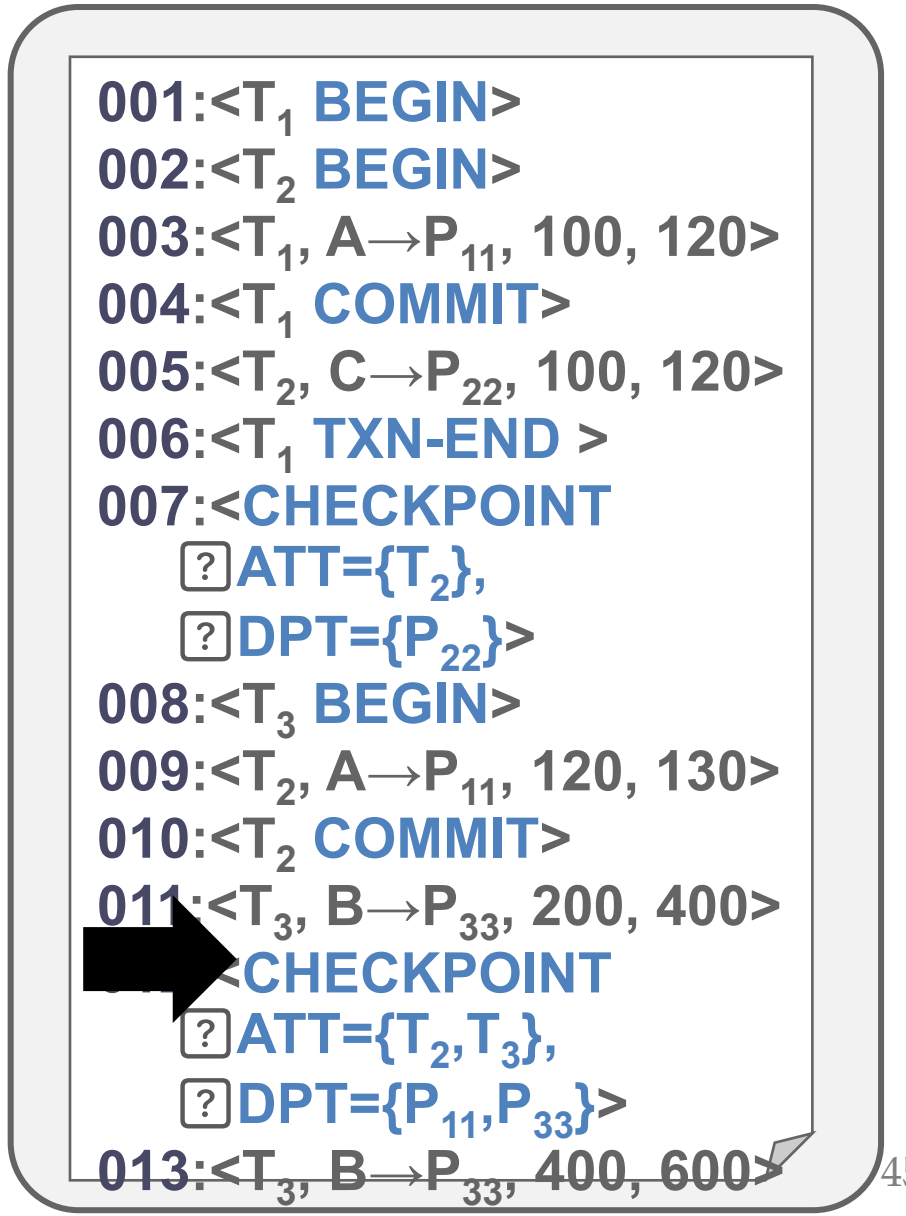
013:<T_3, B→P_{33}, 400, 600>

Slightly better checkpoints

At the first checkpoint, assuming P_{11} was flushed, T_2 is still running and there is only one dirty page (P_{22})

At the second checkpoint, assuming P_{22} was flushed, T_2 and T_3 are active and the dirty pages are (P_{11}, P_{33})

This still is not ideal because the DBMS must stall txns during checkpoint …

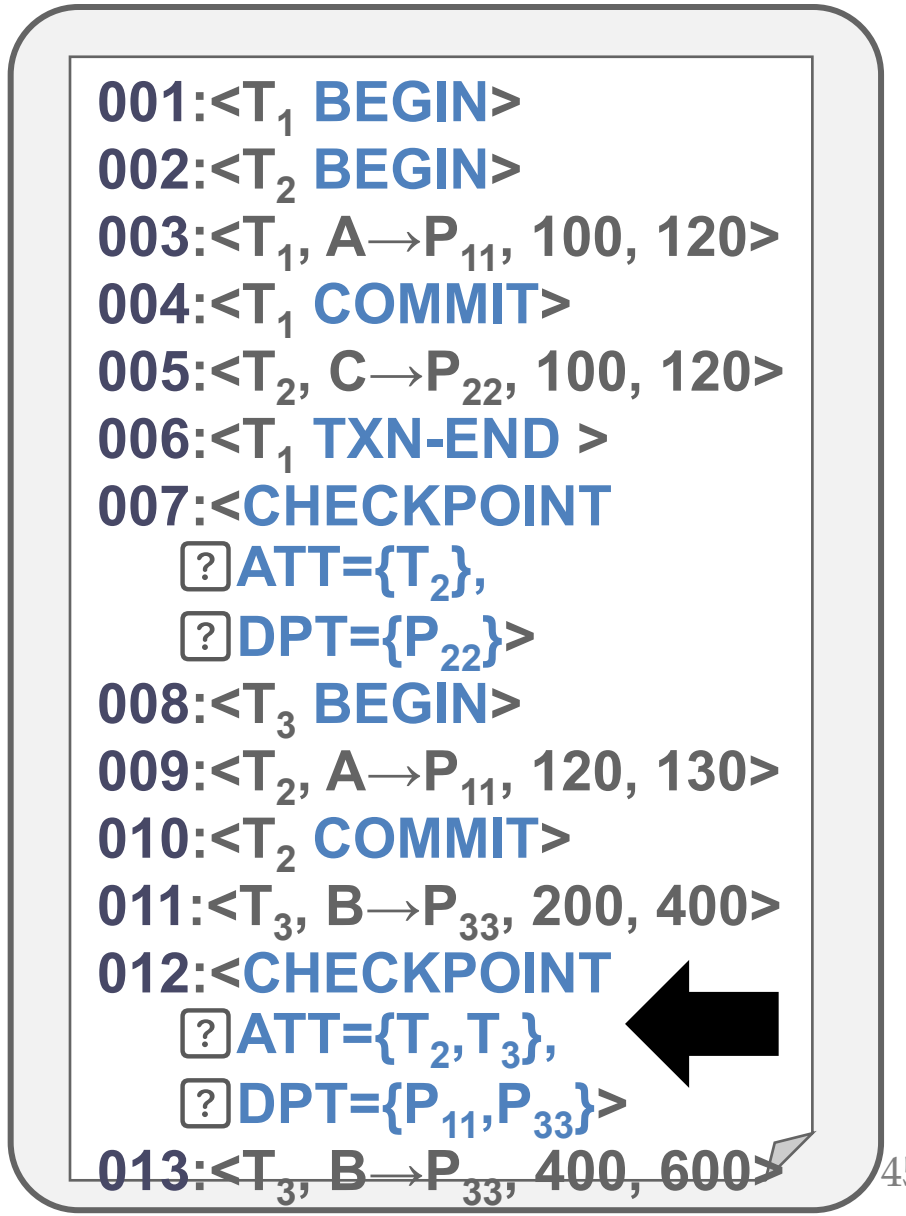


Slightly better checkpoints

At the first checkpoint, assuming P_{11} was flushed, T_2 is still running and there is only one dirty page (P_{22})

At the second checkpoint, assuming P_{22} was flushed, T_2 and T_3 are active and the dirty pages are (P_{11}, P_{33})

This still is not ideal because the DBMS must stall txns during checkpoint …



Fuzzy checkpoints

A **fuzzy checkpoint** is where the DBMS follows active txns to continue to run while the system writes the log records for checkpoint

- No attempt to force dirty pages to disk

New log records to track checkpoint boundaries:

- **CHECKPOINT-BEGIN:** Indicates the start of checkpoint
- **CHECKPOINT-END:** Contains ATT + DPT

Fuzzy checkpoints

Assume the DBMS flushes P_{11} before the first checkpoint starts

Any txn that begins after the checkpoint starts is excluded from the ATT in the **CHECKPOINT-END** record

The LSN of the **CHECKPOINT-BEGIN** record is written to the **MasterRecord** when it completes

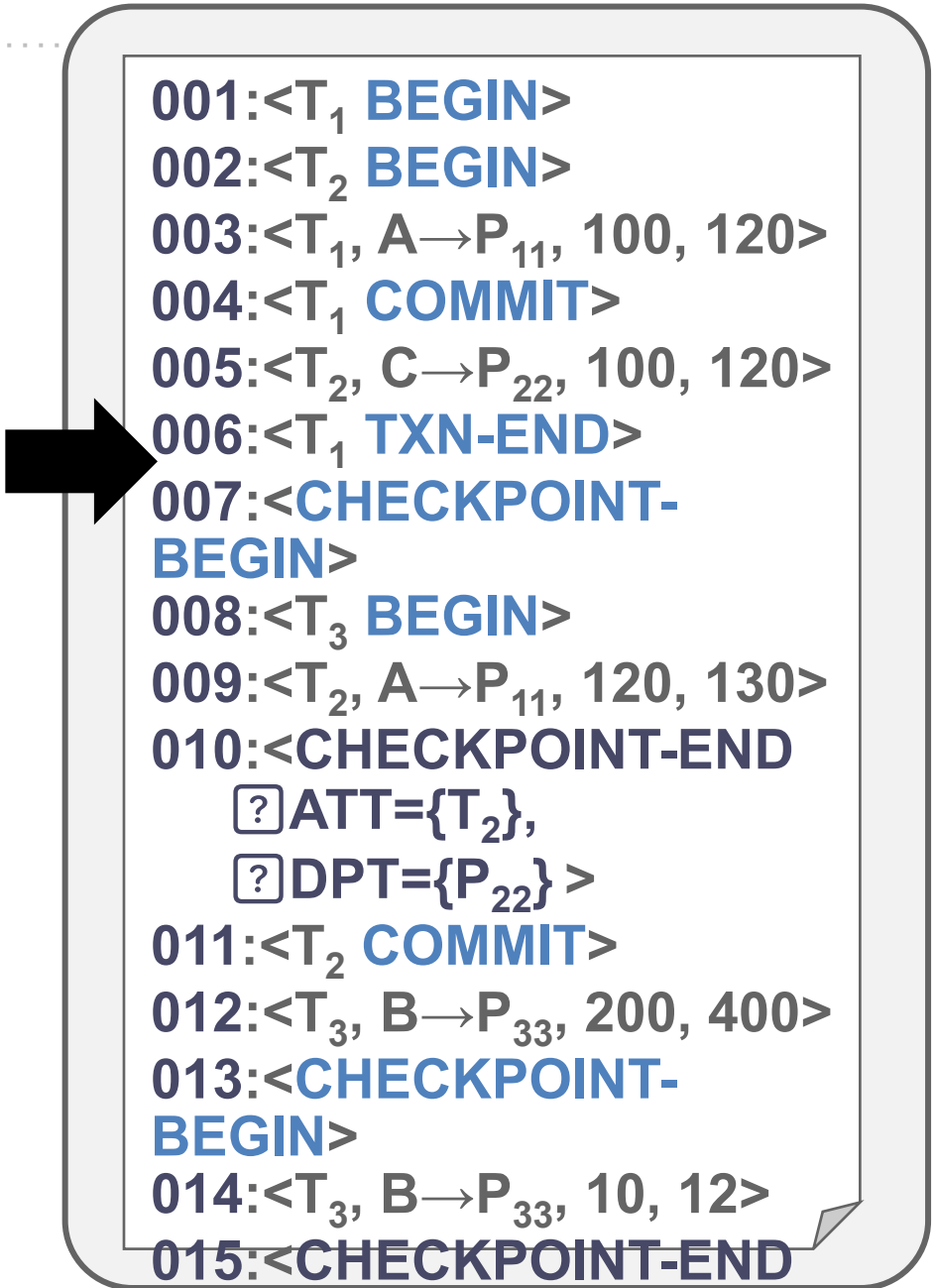
001:<T_1 BEGIN>
002:<T_2 BEGIN>
003:<T_1, A→P_{11}, 100, 120>
004:<T_1 COMMIT>
005:<T_2, C→P_{22}, 100, 120>
006:<T_1 TXN-END>
007:<CHECKPOINT-BEGIN>
008:<T_3 BEGIN>
009:<T_2, A→P_{11}, 120, 130>
010:<CHECKPOINT-END
ATT={T_2},
DPT={P_{22}}>
011:<T_2 COMMIT>
012:<T_3, B→P_{33}, 200, 400>
013:<CHECKPOINT-BEGIN>
014:<T_3, B→P_{33}, 10, 12>
015:<CHECKPOINT-END

Fuzzy checkpoints

Assume the DBMS flushes P_{11} before the first checkpoint starts

Any txn that begins after the checkpoint starts is excluded from the ATT in the **CHECKPOINT-END** record

The LSN of the **CHECKPOINT-BEGIN** record is written to the **MasterRecord** when it completes

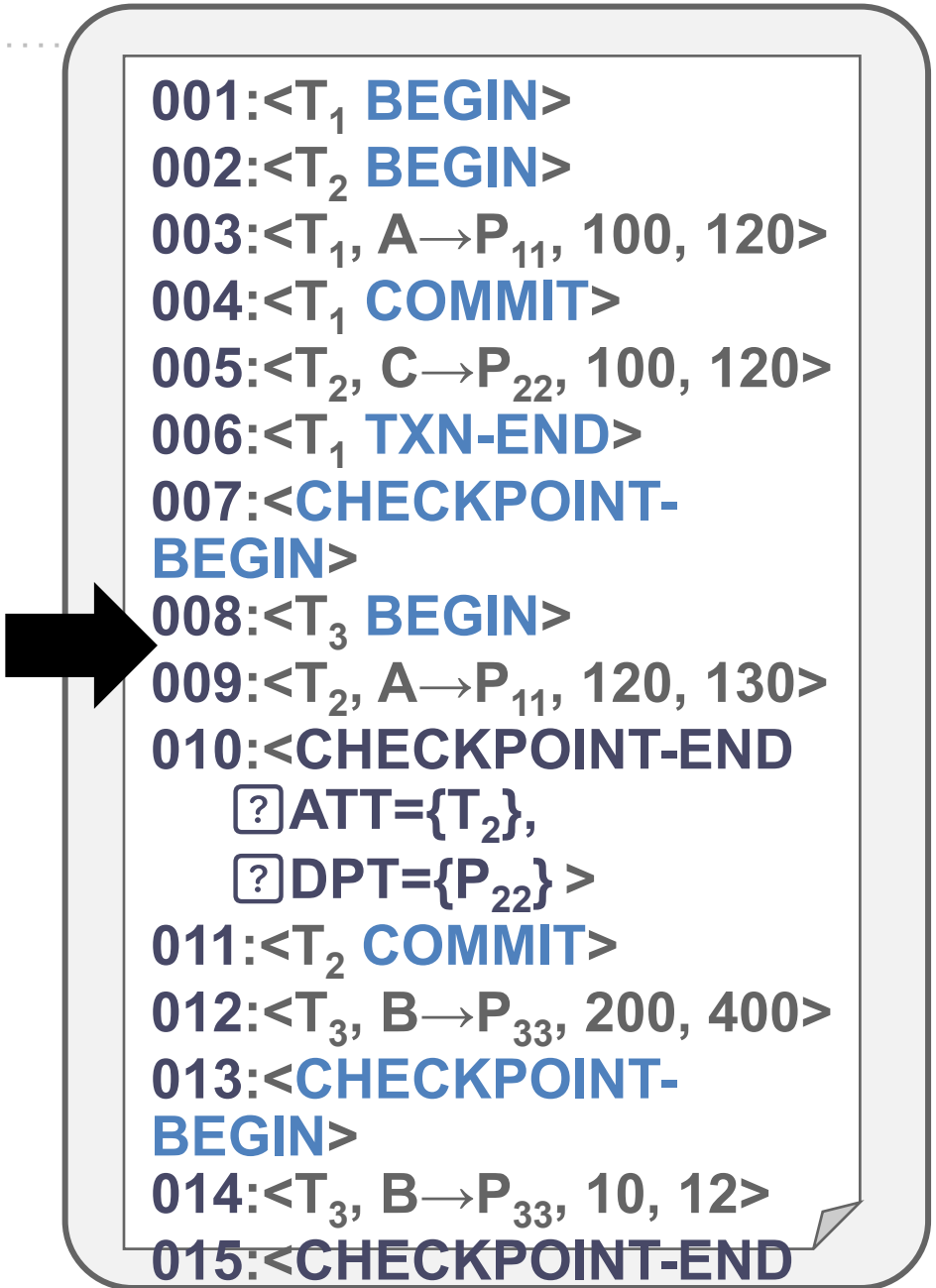


Fuzzy checkpoints

Assume the DBMS flushes P_{11} before the first checkpoint starts

Any txn that begins after the checkpoint starts is excluded from the ATT in the **CHECKPOINT-END** record

The LSN of the **CHECKPOINT-BEGIN** record is written to the **MasterRecord** when it completes

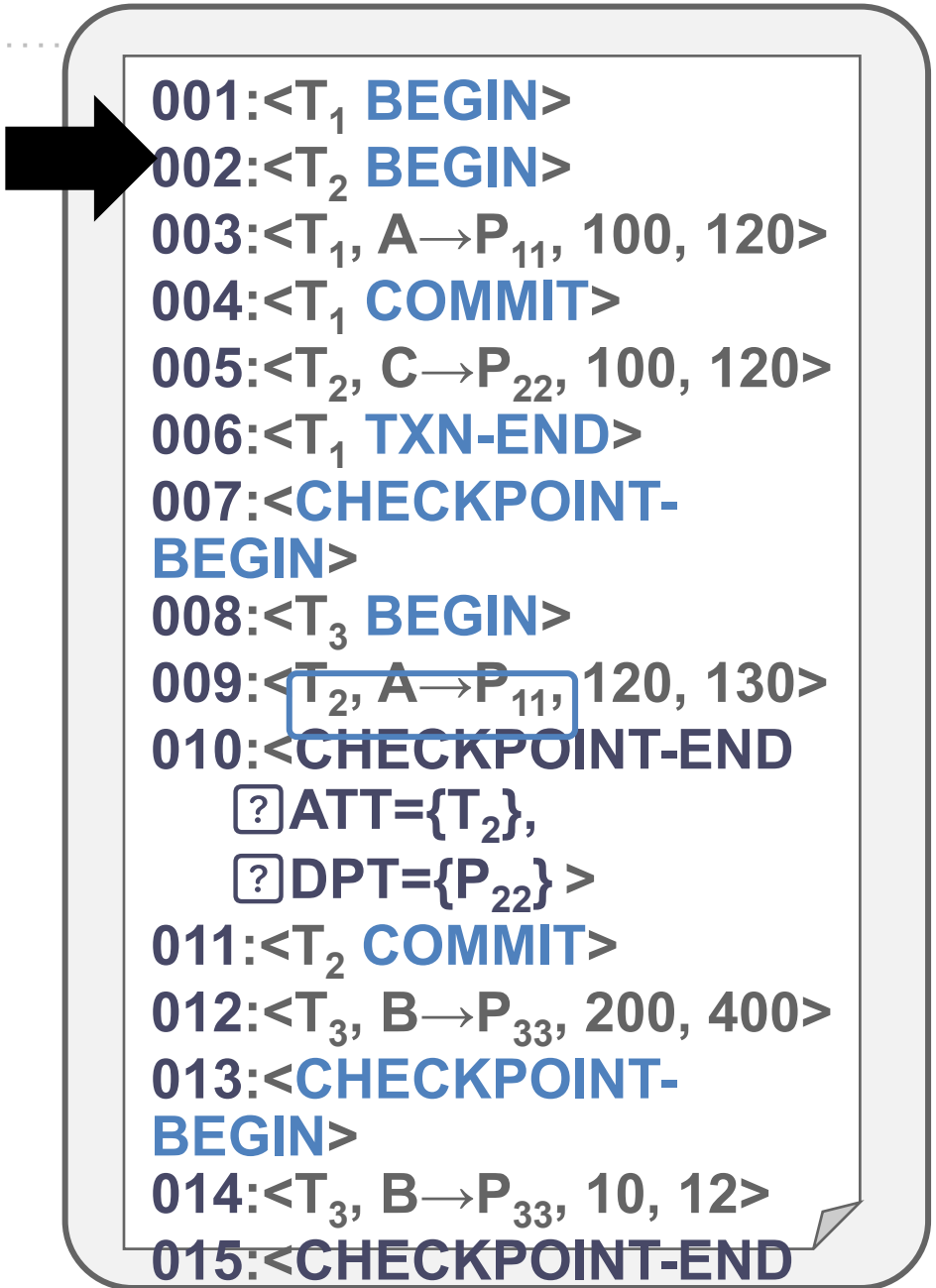


Fuzzy checkpoints

Assume the DBMS flushes P_{11} before the first checkpoint starts

Any txn that begins after the checkpoint starts is excluded from the ATT in the **CHECKPOINT-END** record

The LSN of the **CHECKPOINT-BEGIN** record is written to the **MasterRecord** when it completes

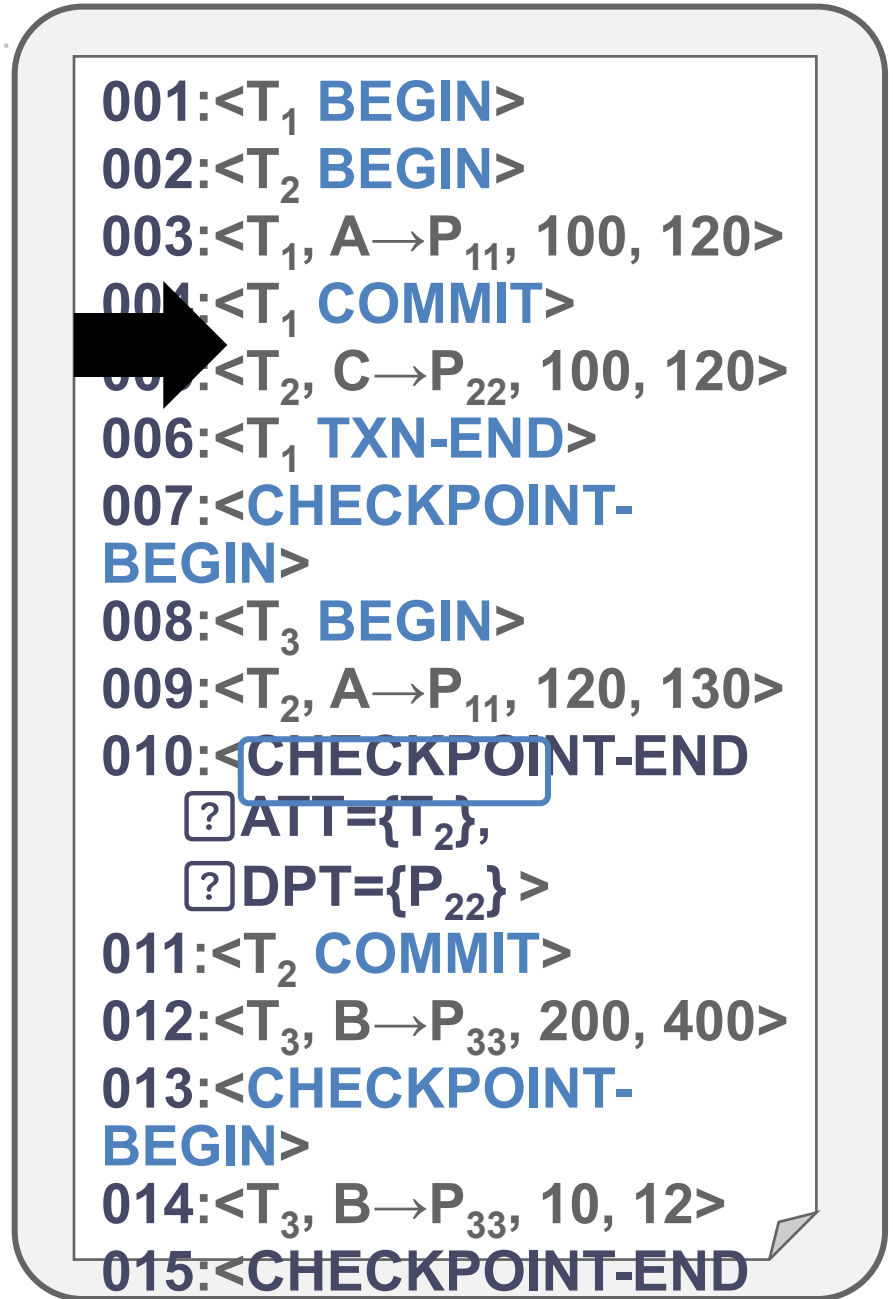


Fuzzy checkpoints

Assume the DBMS flushes P_{11} before the first checkpoint starts

Any txn that begins after the checkpoint starts is excluded from the ATT in the **CHECKPOINT-END** record

The LSN of the **CHECKPOINT-BEGIN** record is written to the **MasterRecord** when it completes



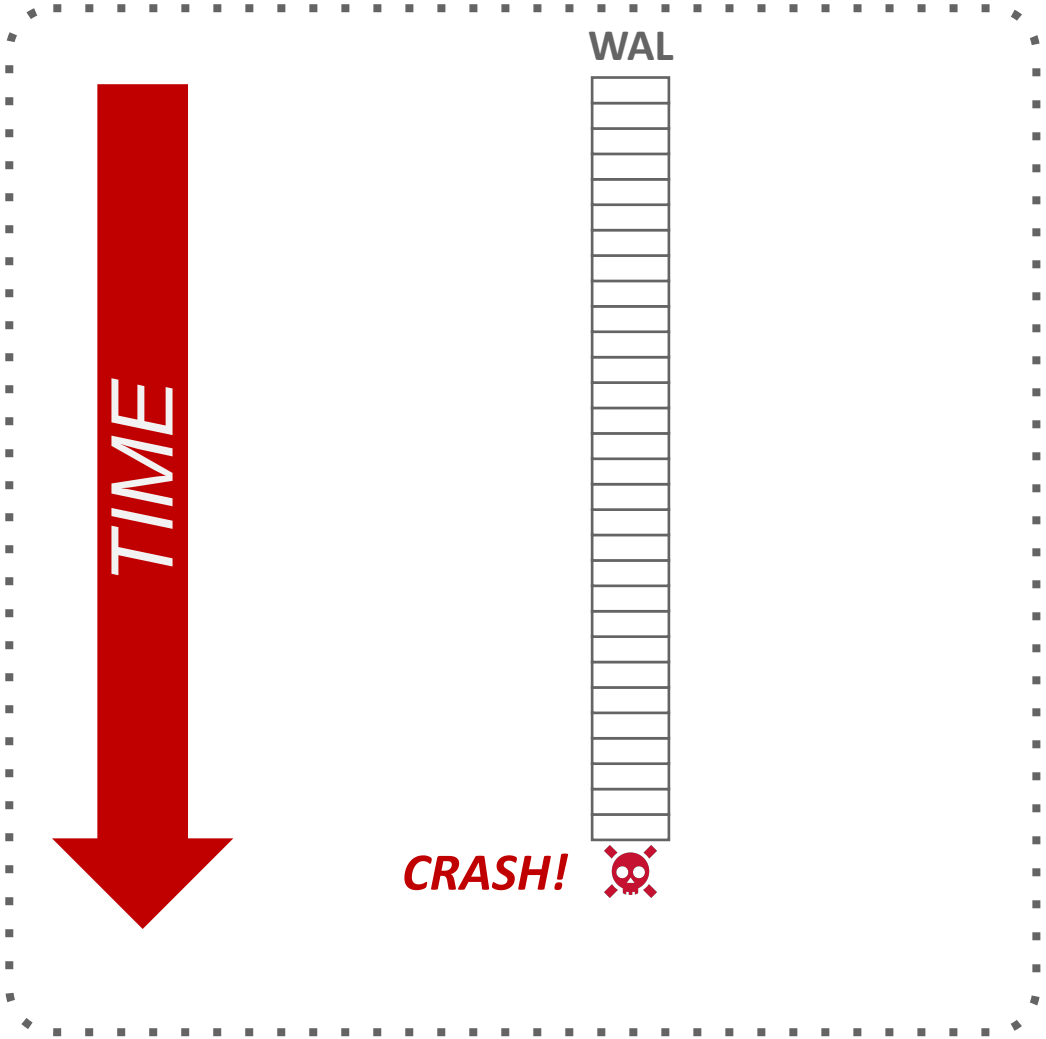
ARIES: recovery phase overview

Start from last **BEGIN-CHECKPOINT** found via **MasterRecord**

Analysis: Figure out which txns committed or failed since checkpoint

Redo: Repeat all actions

Undo: Reverse effects of failed txns



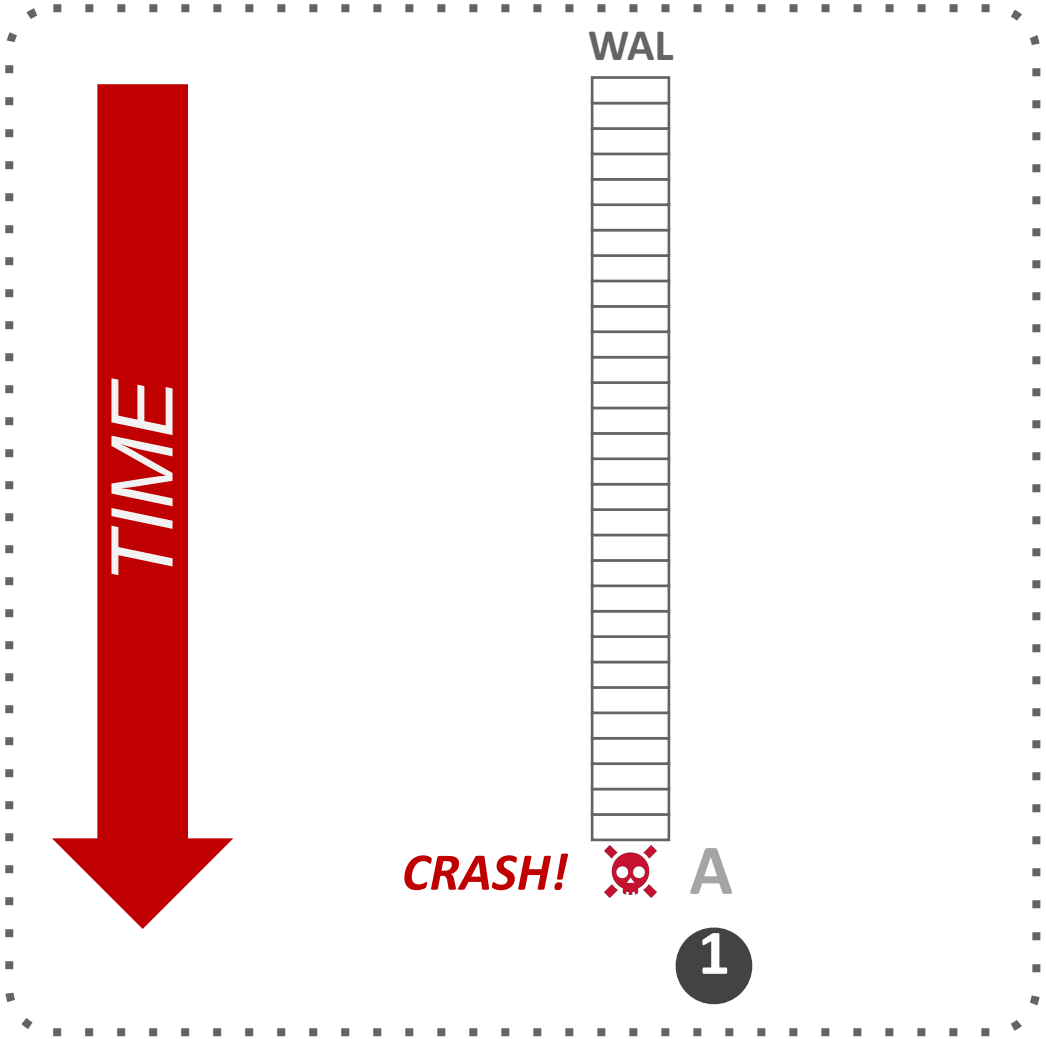
ARIES: recovery phase overview

Start from last **BEGIN-CHECKPOINT** found via **MasterRecord**

Analysis: Figure out which txns committed or failed since checkpoint

Redo: Repeat all actions

Undo: Reverse effects of failed txns



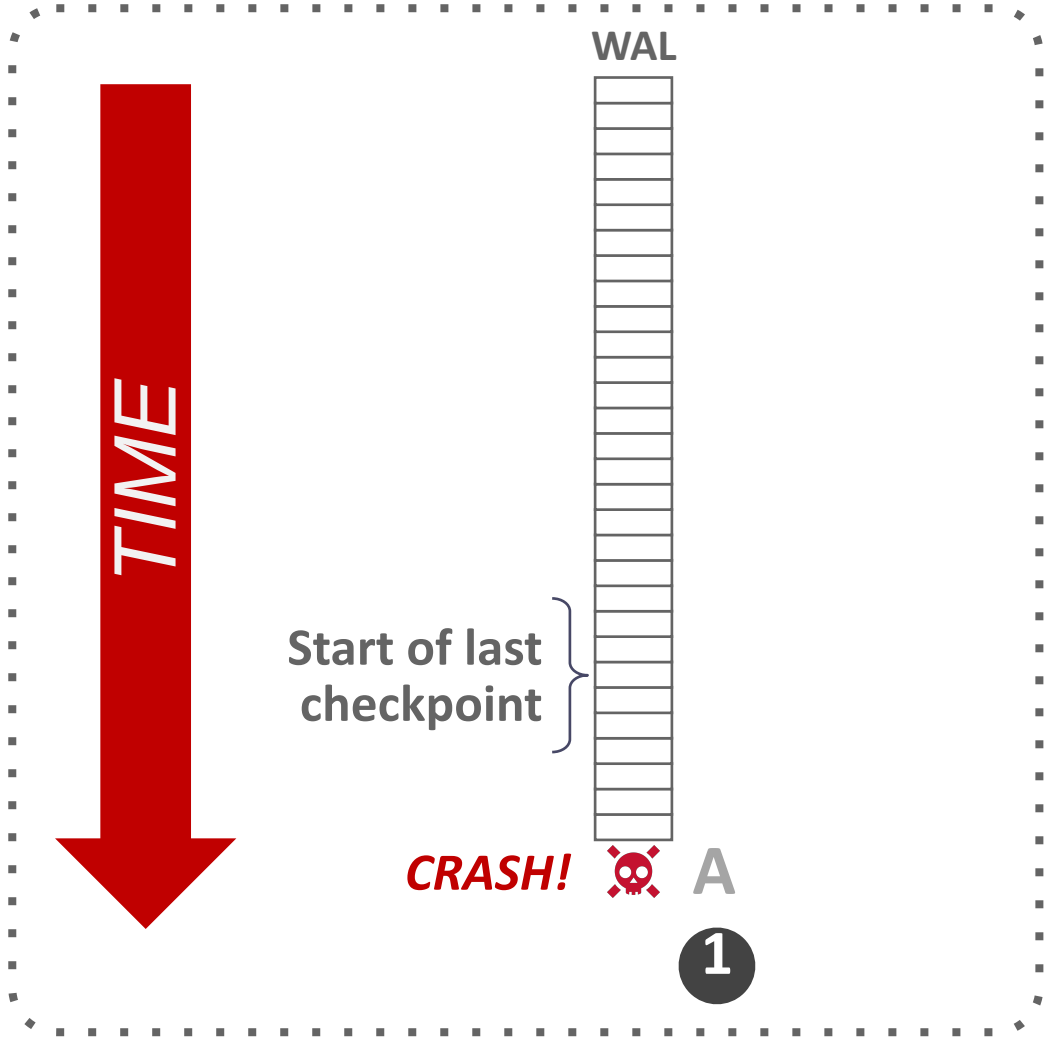
ARIES: recovery phase overview

Start from last **BEGIN-CHECKPOINT** found via **MasterRecord**

Analysis: Figure out which txns committed or failed since checkpoint

Redo: Repeat all actions

Undo: Reverse effects of failed txns



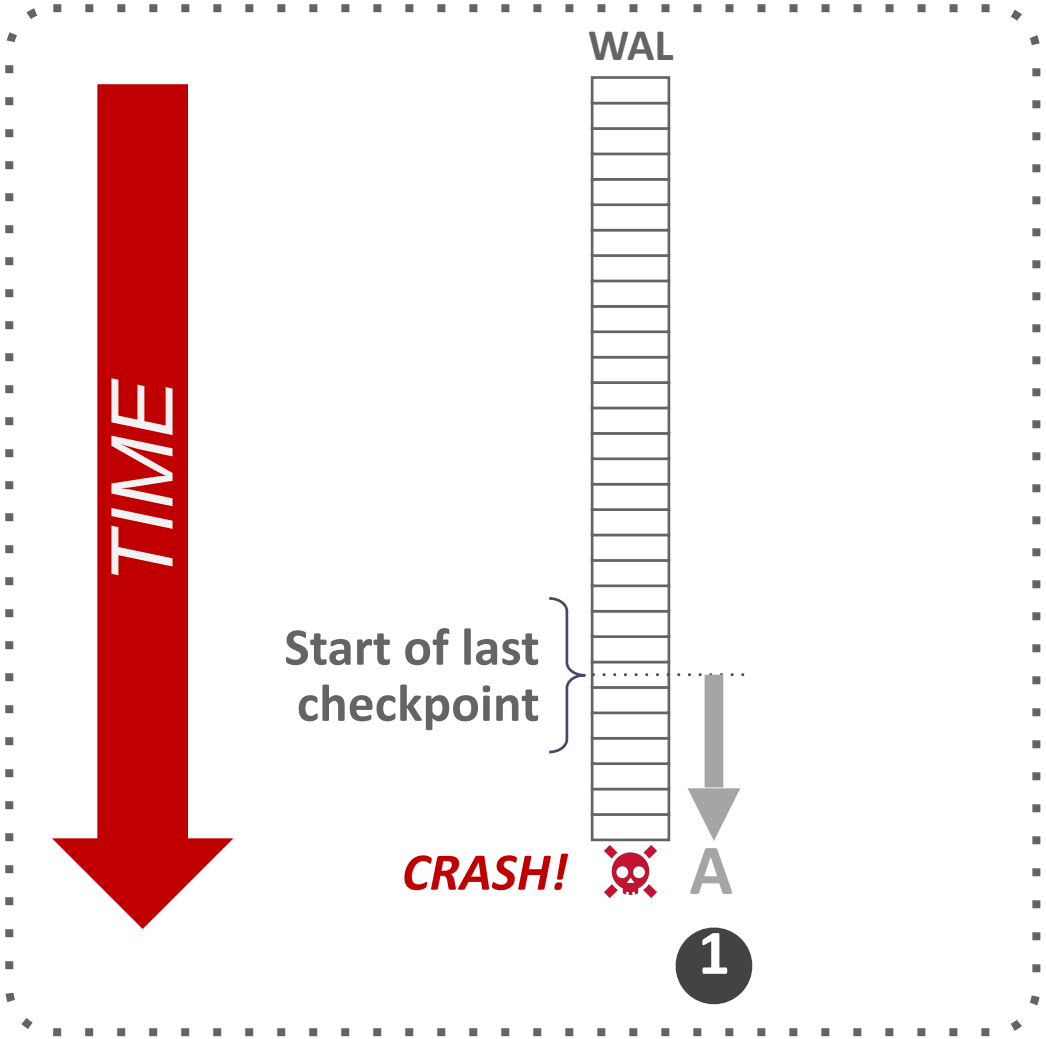
ARIES: recovery phase overview

Start from last **BEGIN-CHECKPOINT** found via **MasterRecord**

Analysis: Figure out which txns committed or failed since checkpoint

Redo: Repeat all actions

Undo: Reverse effects of failed txns



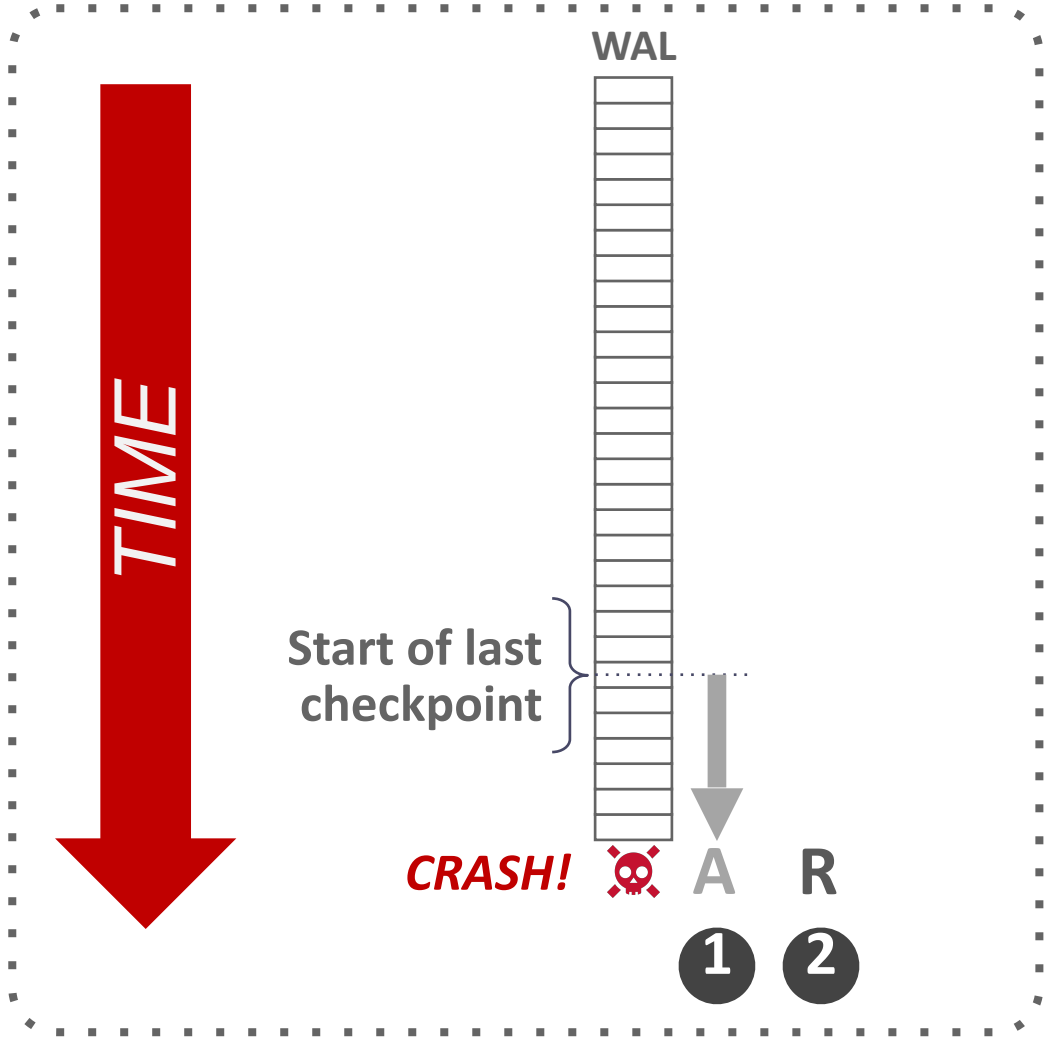
ARIES: recovery phase overview

Start from last **BEGIN-CHECKPOINT** found via **MasterRecord**

Analysis: Figure out which txns committed or failed since checkpoint

Redo: Repeat all actions

Undo: Reverse effects of failed txns



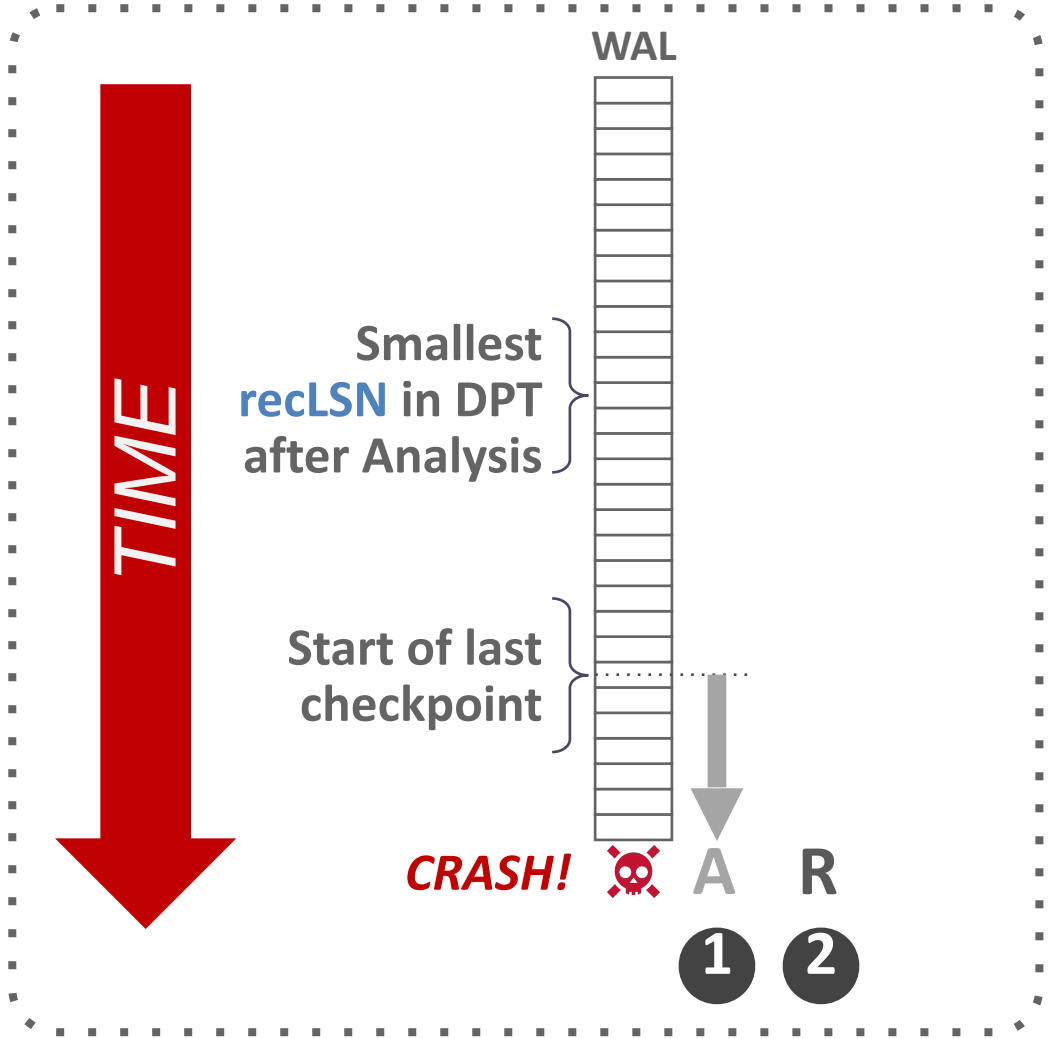
ARIES: recovery phase overview

Start from last **BEGIN-CHECKPOINT** found via **MasterRecord**

Analysis: Figure out which txns committed or failed since checkpoint

Redo: Repeat all actions

Undo: Reverse effects of failed txns



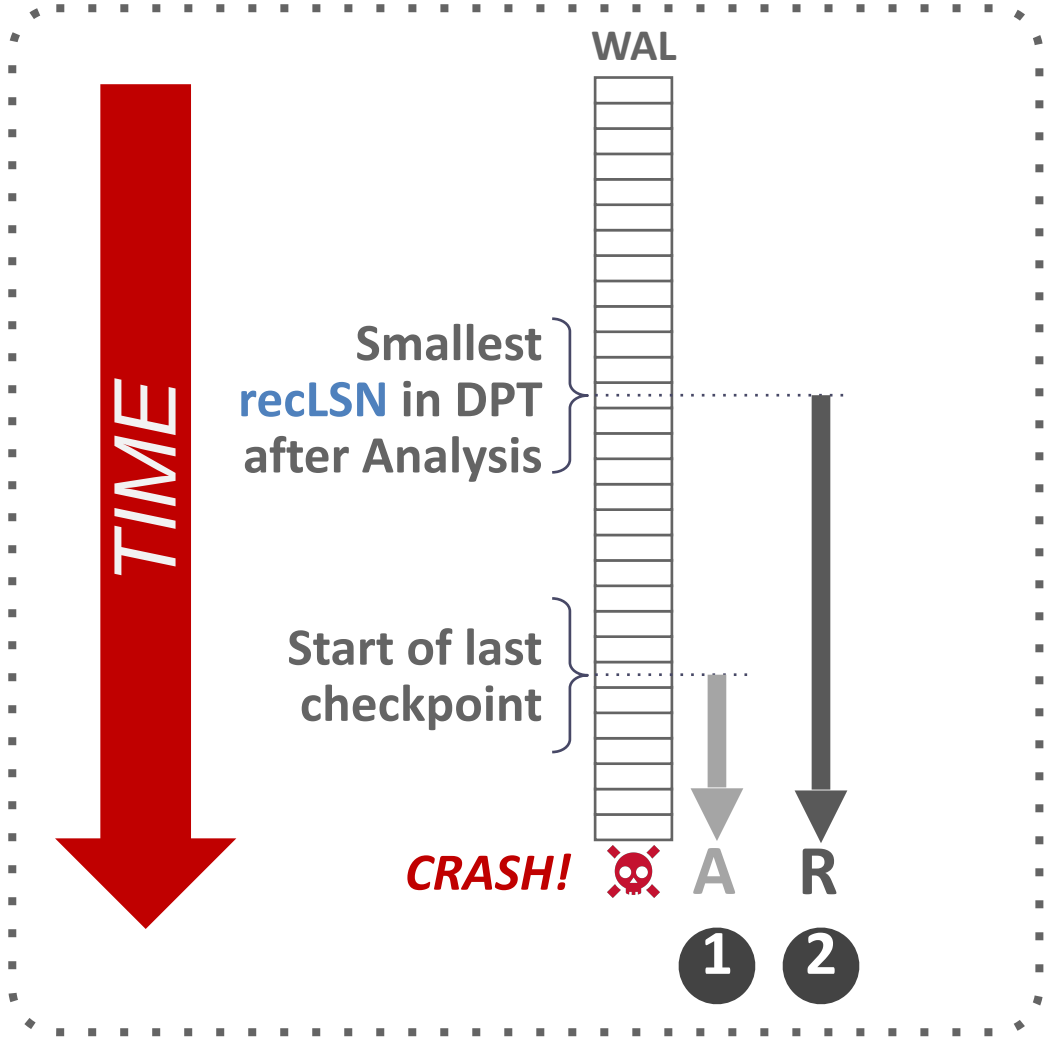
ARIES: recovery phase overview

Start from last **BEGIN-CHECKPOINT** found via **MasterRecord**

Analysis: Figure out which txns committed or failed since checkpoint

Redo: Repeat all actions

Undo: Reverse effects of failed txns



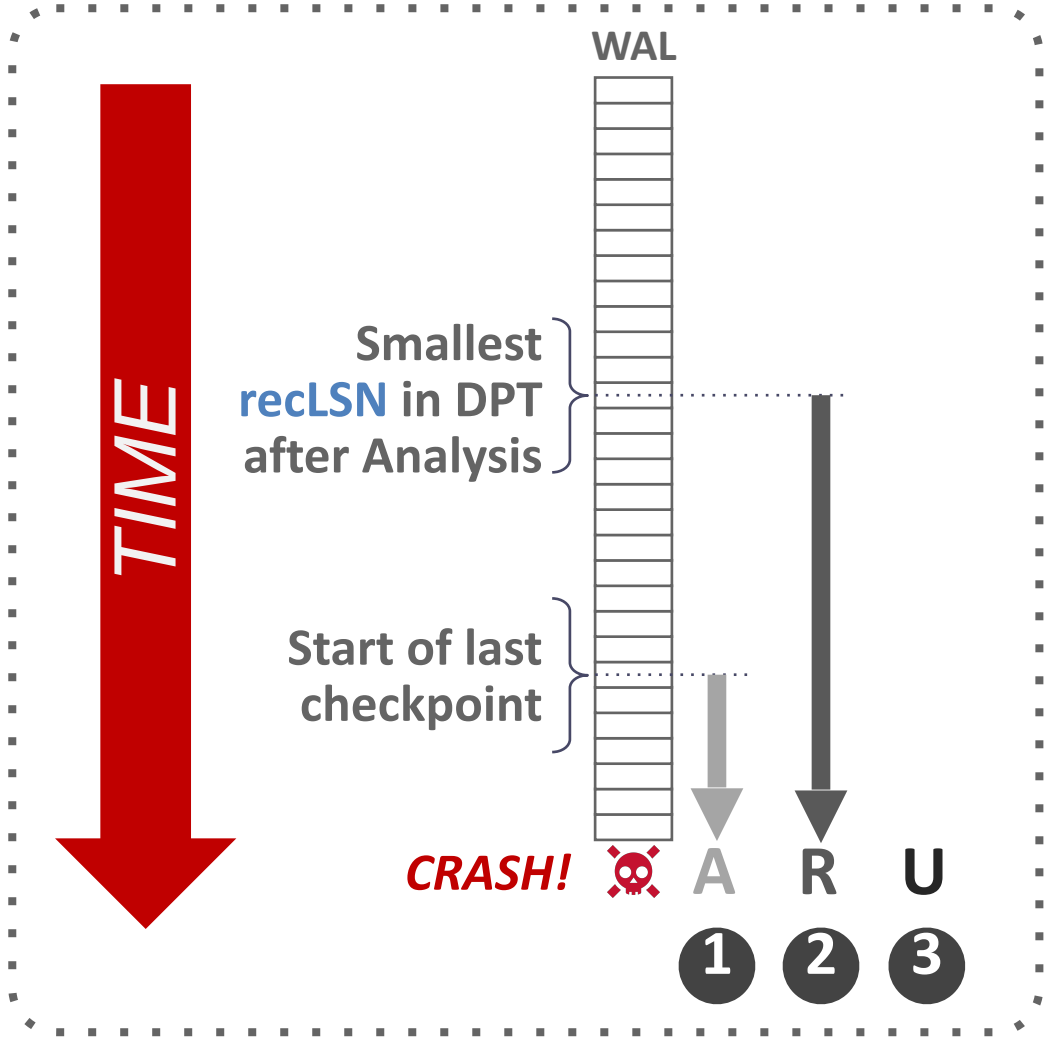
ARIES: recovery phase overview

Start from last **BEGIN-CHECKPOINT** found via **MasterRecord**

Analysis: Figure out which txns committed or failed since checkpoint

Redo: Repeat all actions

Undo: Reverse effects of failed txns



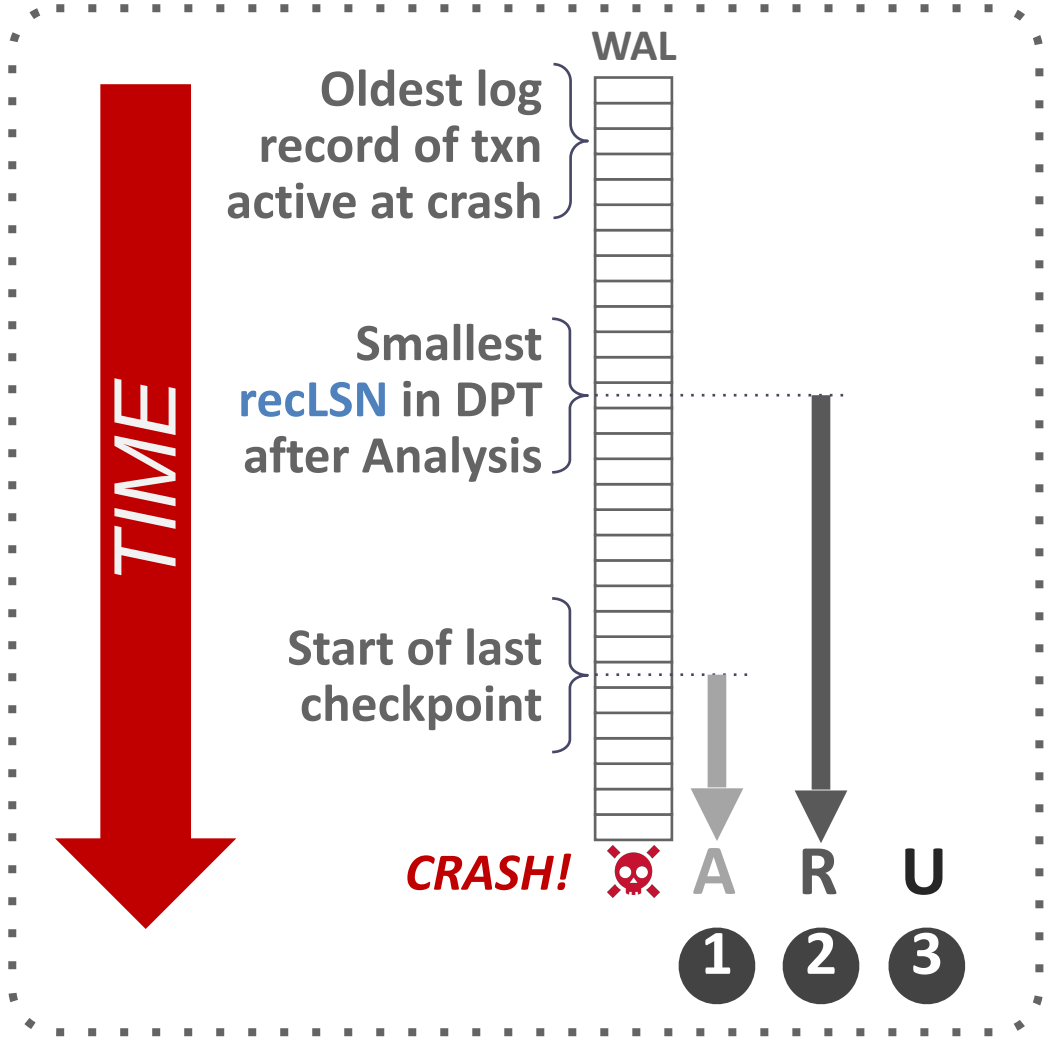
ARIES: recovery phase overview

Start from last **BEGIN-CHECKPOINT** found via **MasterRecord**

Analysis: Figure out which txns committed or failed since checkpoint

Redo: Repeat all actions

Undo: Reverse effects of failed txns



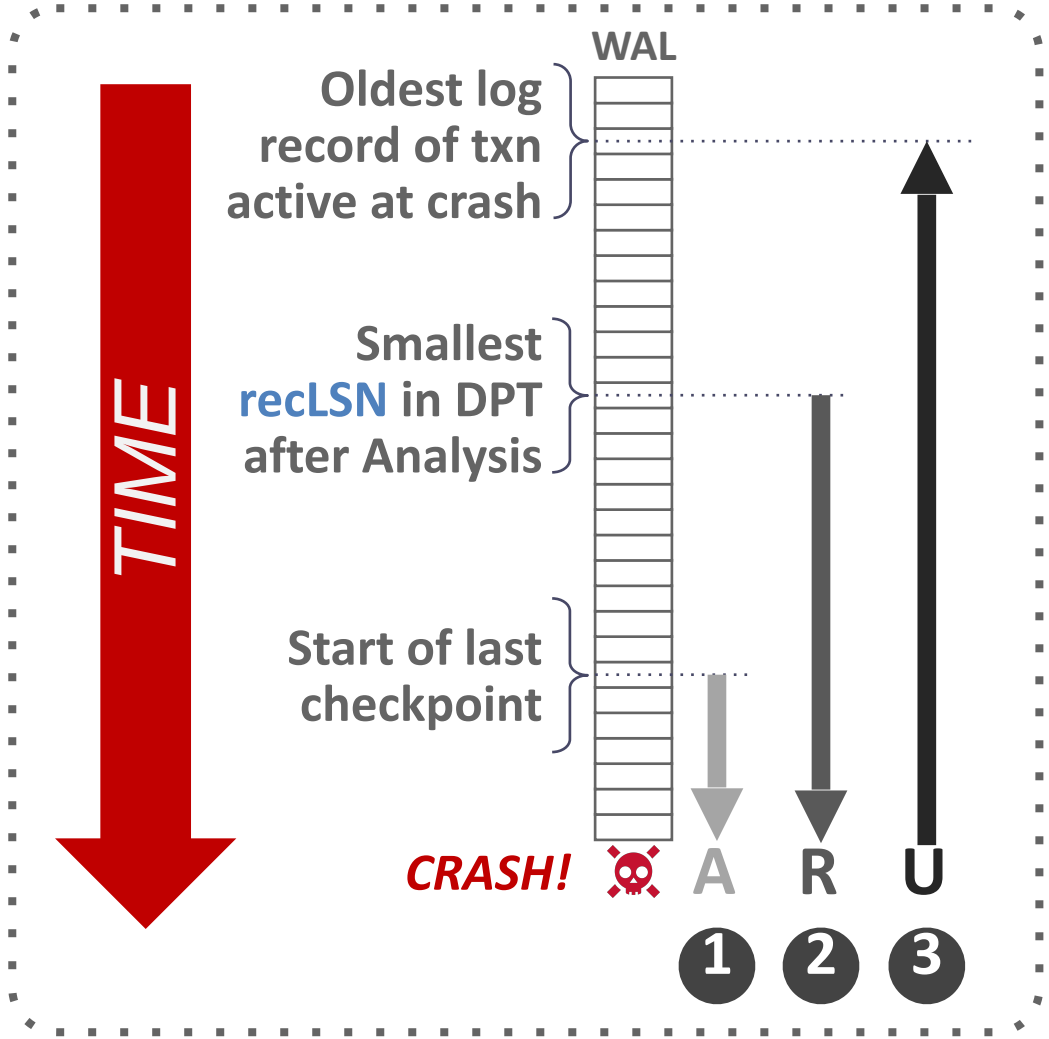
ARIES: recovery phase overview

Start from last **BEGIN-CHECKPOINT** found via **MasterRecord**

Analysis: Figure out which txns committed or failed since checkpoint

Redo: Repeat all actions

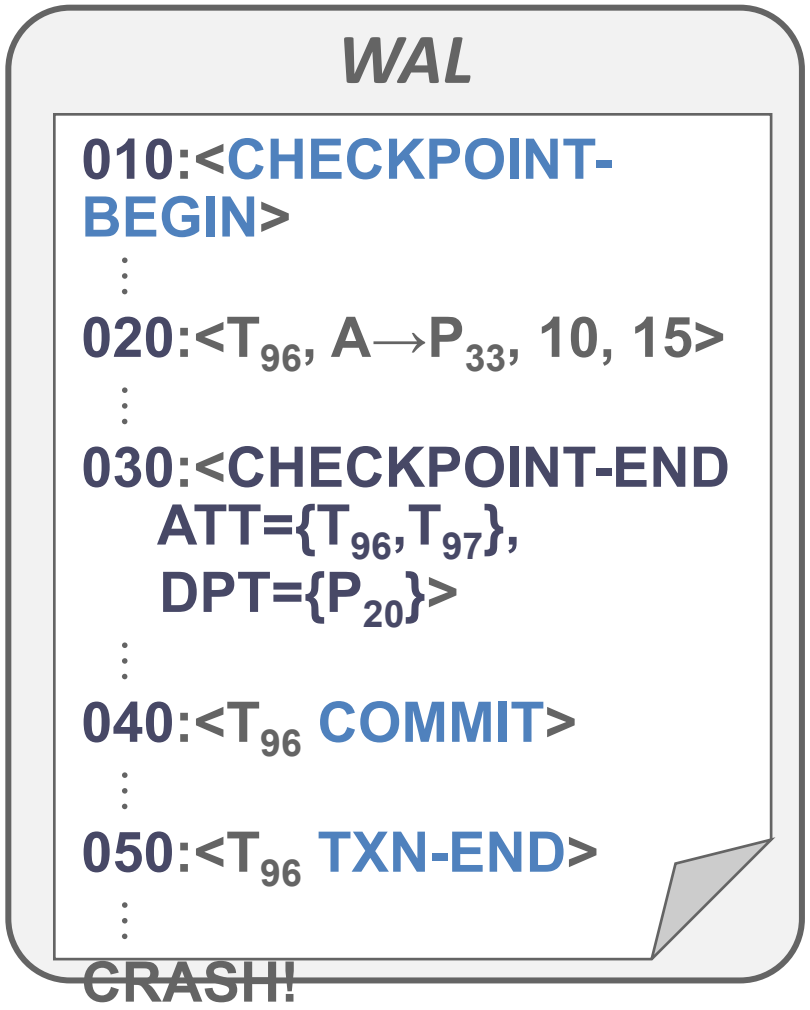
Undo: Reverse effects of failed txns



Analysis phase

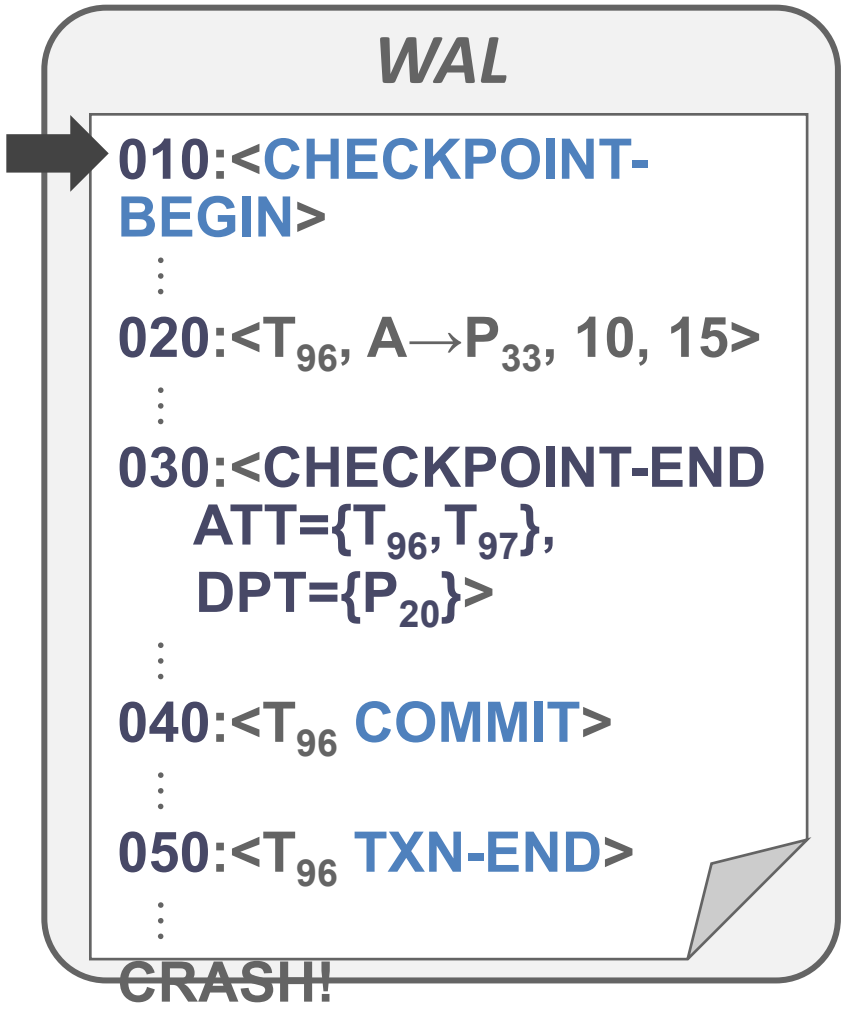
- Scan the log forward from last successful checkpoint
  - If the DBMS finds a **TXN-END** record, remove its corresponding txn from **ATT**
- All other records:
  - If txn not in **ATT**, add it with status **undo**
  - On commit, change txn status to **COMMIT**
- For update log records
  - If page **P** not in **DPT**, add **P** to **DPT**, set its **recLSN=LSN**
- At the end of the phase:
  - **ATT** identifies which txns were active at the time of crash
  - **DPT** identifies which dirty pages might have made it to disk

Analysis phase



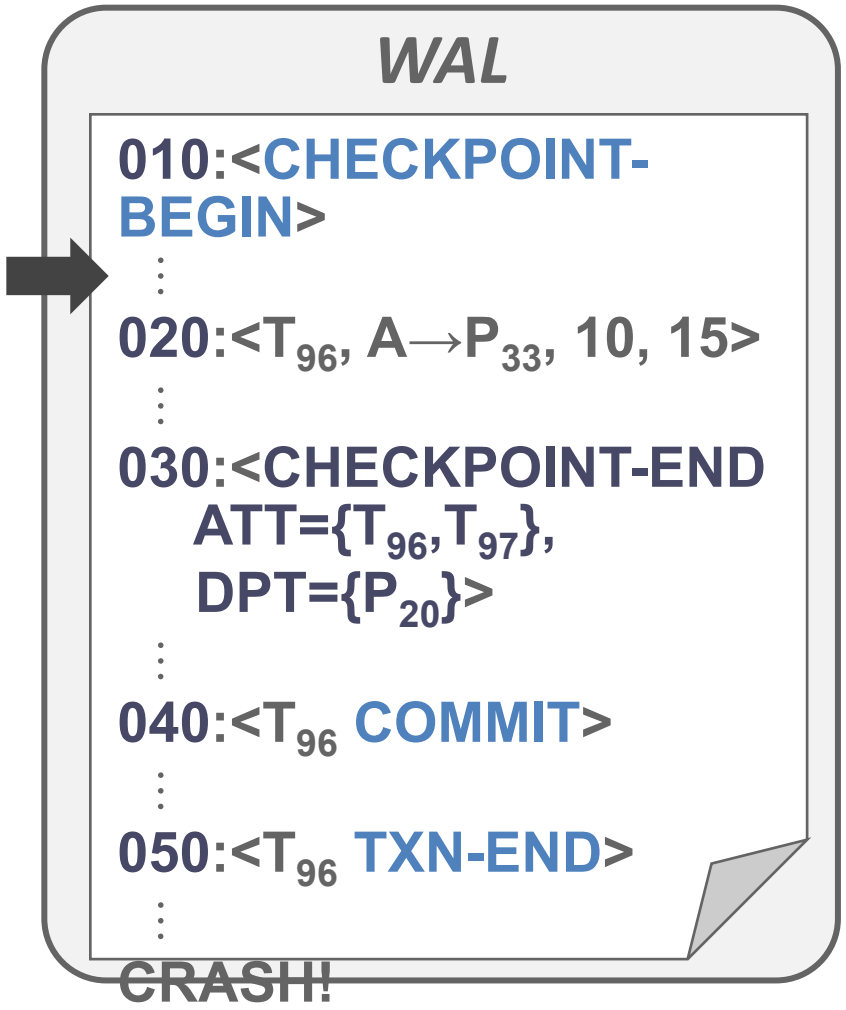
LSN	ATT	DPT
010		
020		
030		
040		
050		

Analysis phase



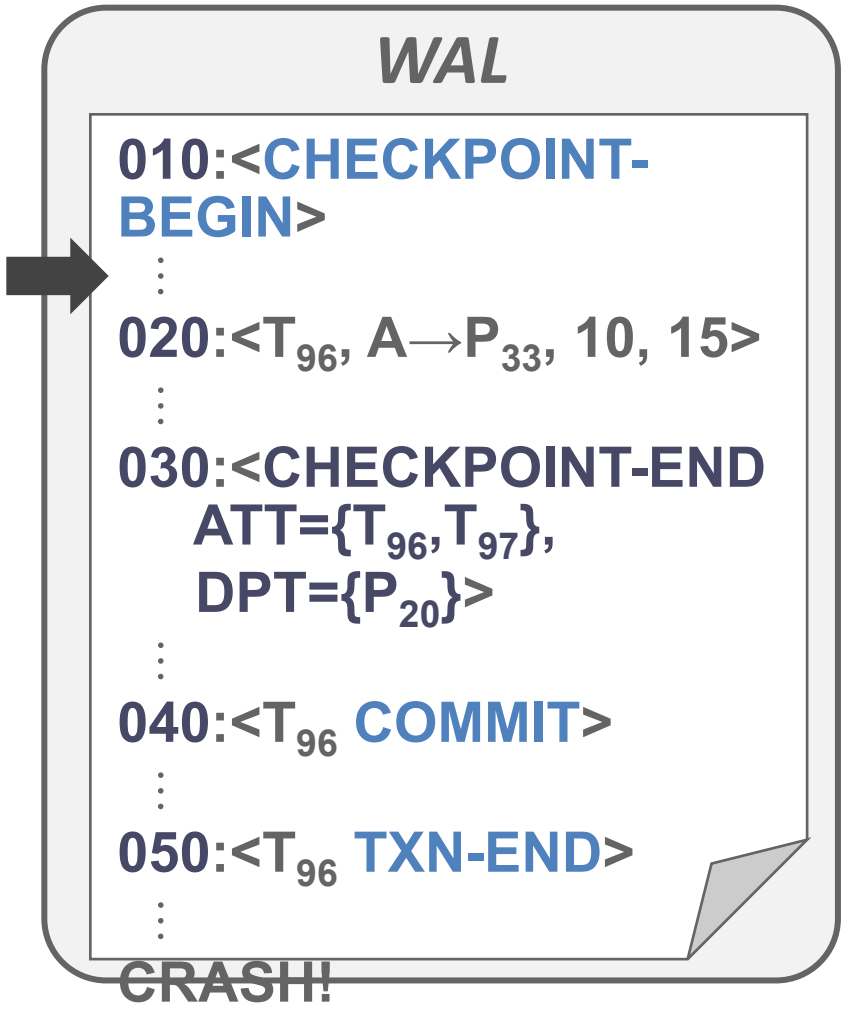
LSN	ATT	DPT
010		
020		
030		
040		
050		

Analysis phase



LSN	ATT	DPT
010		
020		
030		
040		
050		

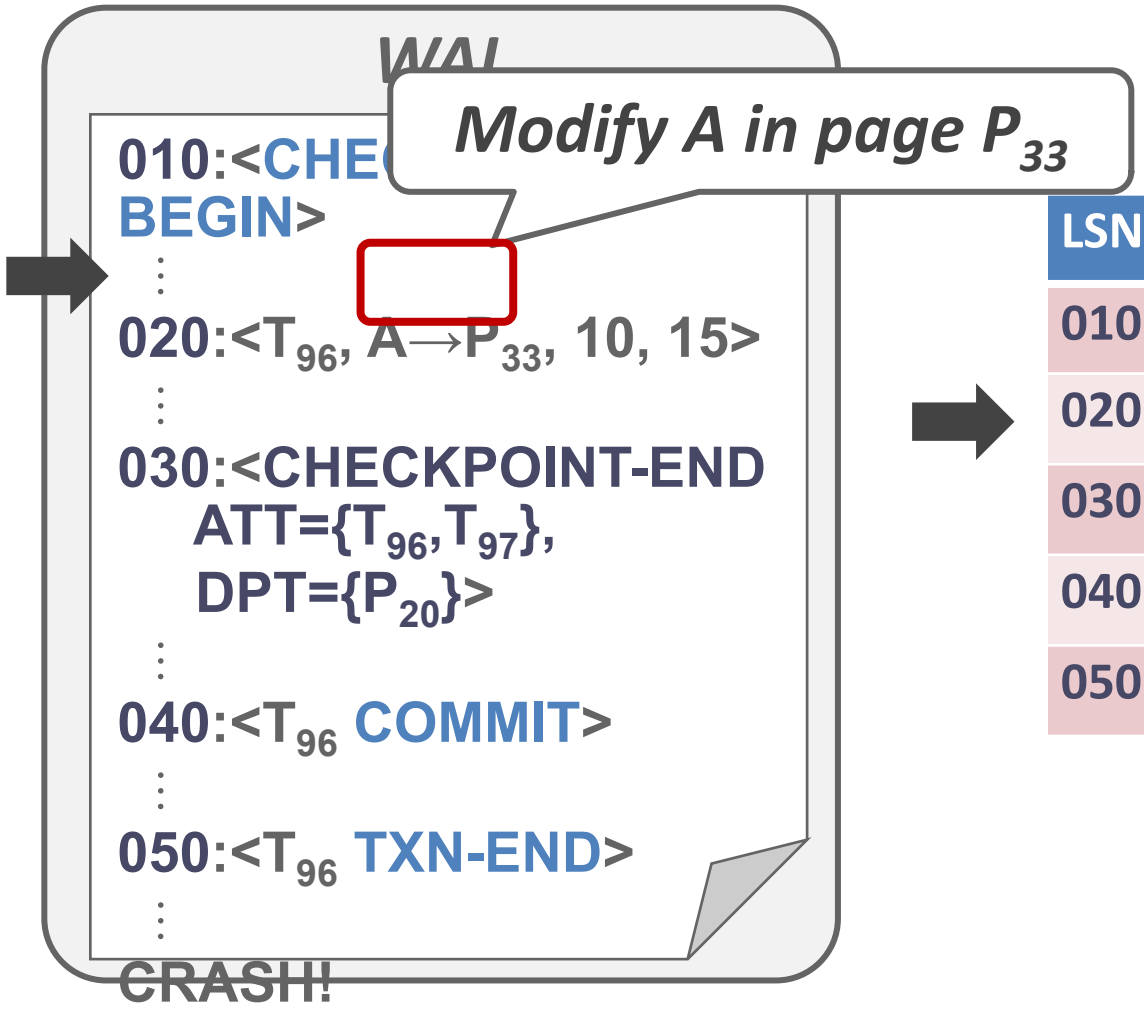
Analysis phase



LSN	ATT	DPT
010		
020	(T_{96},U)	
030		
040		
050		

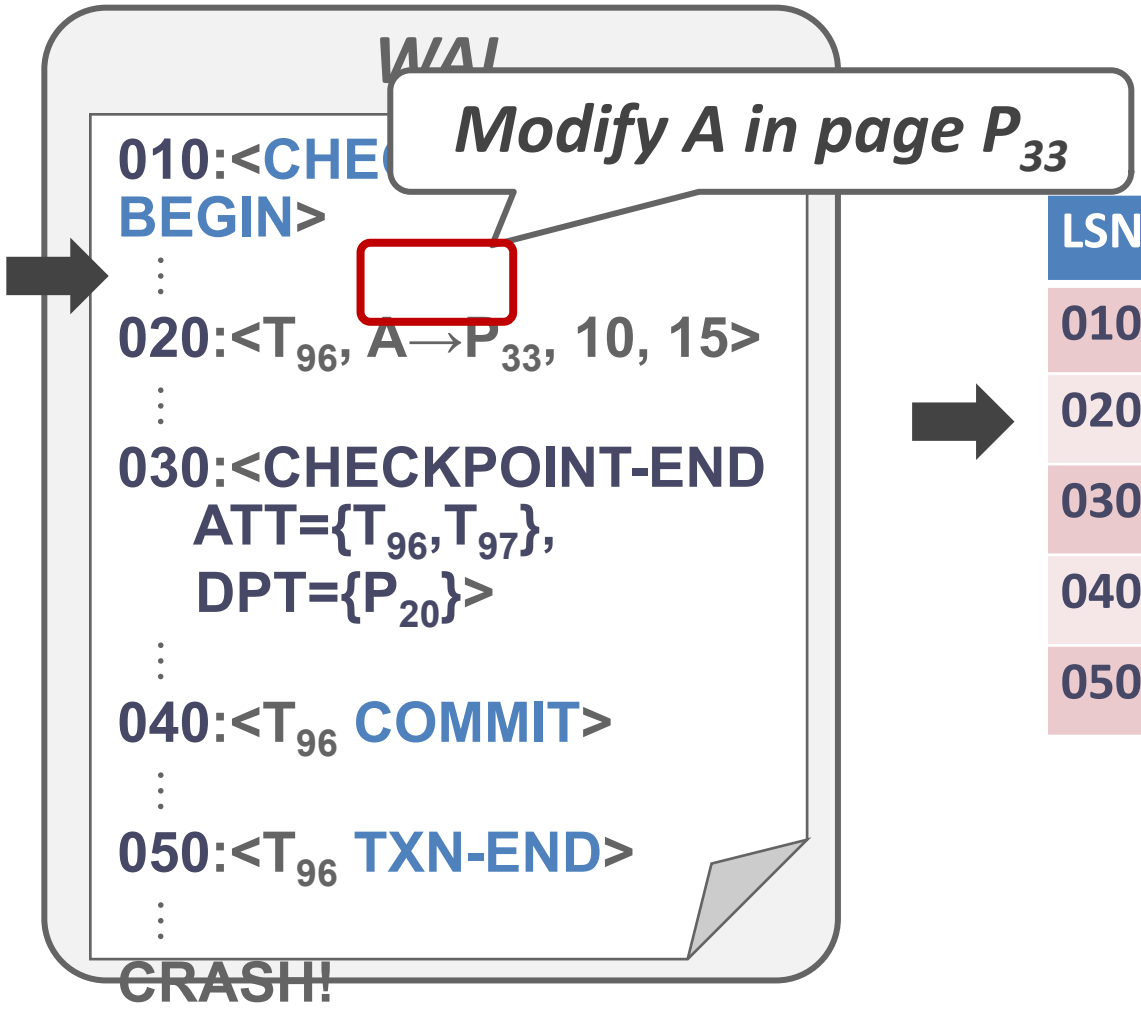
(TxnId, Status)

Analysis phase



LSN	ATT	DPT
010		
020	(T_{96},U)	
030		
040		
050		

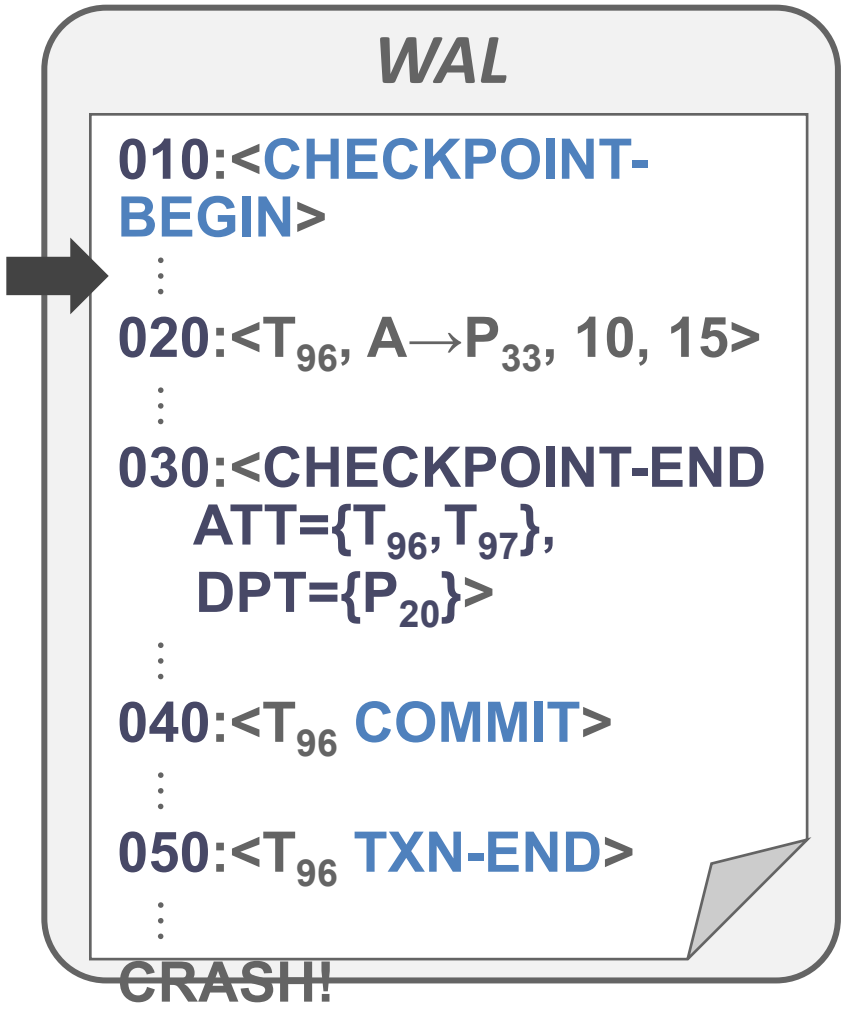
Analysis phase



LSN	ATT	DPT
010		
020	(T_{96},U)	(P_{33},020)
030		
040		
050		

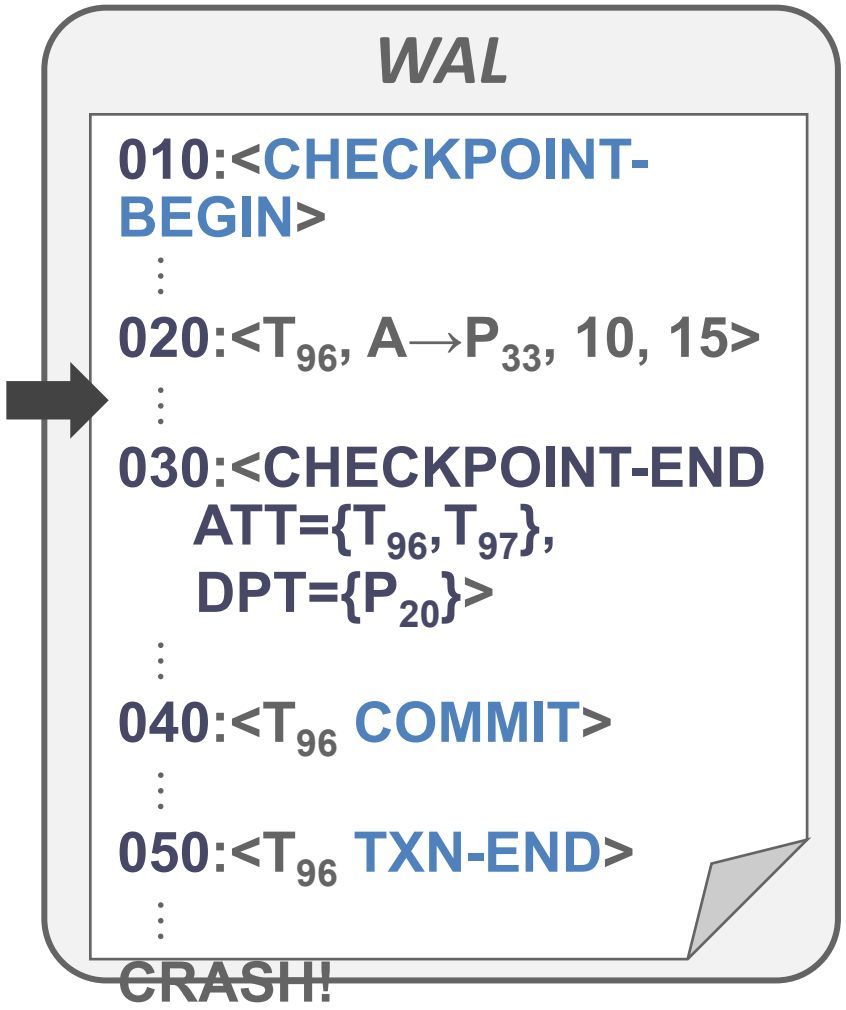
(PageId, RecLSN)

Analysis phase



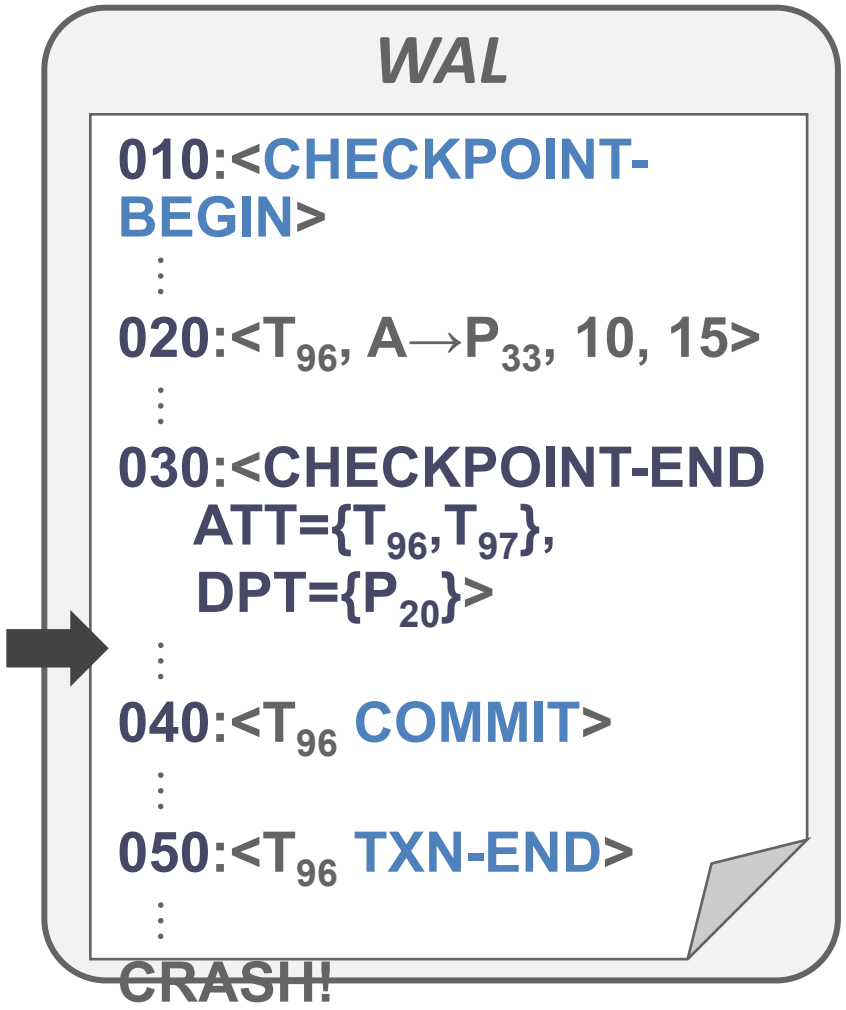
LSN	ATT	DPT
010		
020	(T_{96},U)	(P_{33},020)
030		
040		
050		

Analysis phase



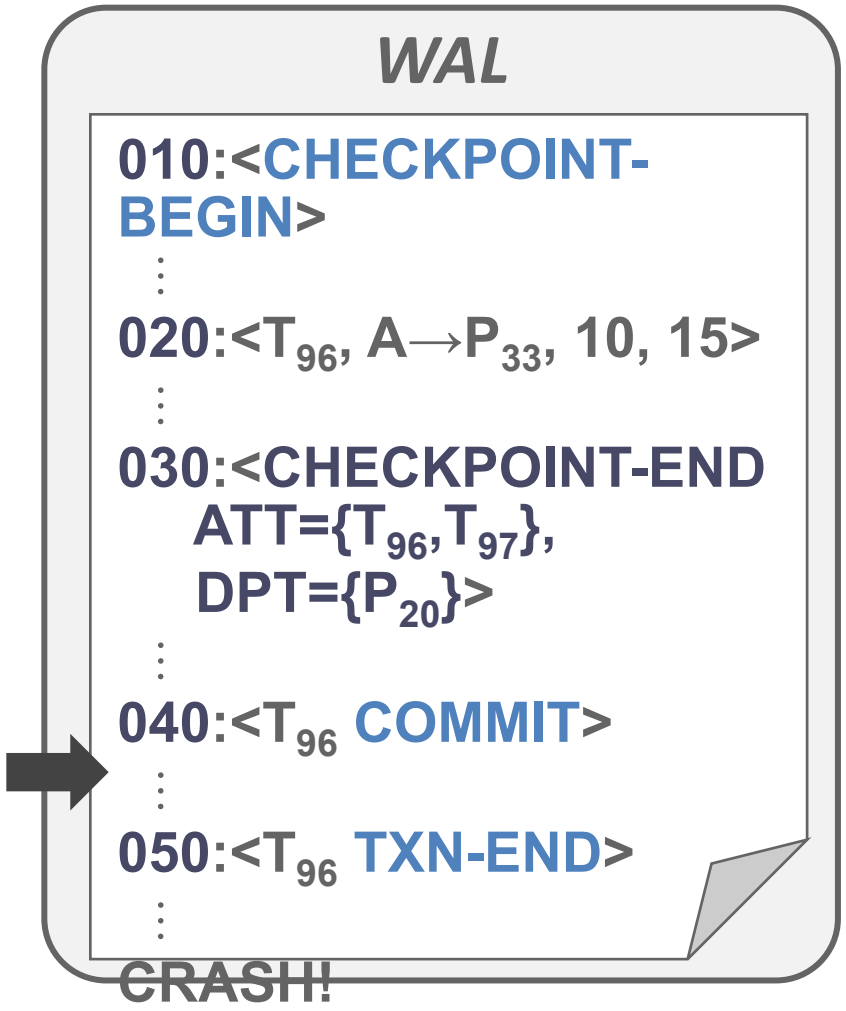
LSN	ATT	DPT
010		
020	(T_{96},U)	(P_{33},**020**)
030	(T_{96},U), (T_{97},U)	(P_{33},**020**), (P_{20},**008**)
040		
050		

Analysis phase



LSN	ATT	DPT
010		
020	(T_{96},U)	(P_{33},020)
030	(T_{96},U), (T_{97},U)	(P_{33},020), (P_{20},008)
040	(T_{96},C), (T_{97},U)	(P_{33},020), (P_{20},008)
050		

Analysis phase



LSN	ATT	DPT
010		
020	(T_{96},U)	(P_{33},020)
030	(T_{96},U), (T_{97},U)	(P_{33},020), (P_{20},008)
040	(T_{96},C), (T_{97},U)	(P_{33},020), (P_{20},008)
050	(T_{97},U)	(P_{33},020), (P_{20},008)

Redo phase

- Goal: Repeat history to reconstruct the DB state at the moment of crash

→ Reapply all updates (even aborted txns) and redo CLRs

- Scan forward from the log record containing smallest **recLSN** in **DPT**
- For each update log record or **CLR** with a given **LSN**, redo the actions unless:
  - The affected page is not in **DPT,** or
  - The affected page is in **DPT**, but that log record's **LSN** is less than the page's **recLSN**
  - Log record's **LSN <= pageLSN**

    DBMS must fetch page from the disk to read page value

Redo phase

- To redo an action
  - Reapply logged update
  - Set **pageLSN** to log record's **LSN**
  - No additional logging, no forced flushes

- At the end of redo phase, write **TXN-END** log records for all txns with status **C** and remove them from **ATT**

Undo phase

- Undo all txns that were active at the time of crash and therefore never commit
  - These are all the txns with **U** status in the **ATT** after the Analysis phase

- Process them in reverse LSN order using the **lastLSN** to speed up traversal
  - At each step, pick the largest **lastLSN** across all transactions in the ATT
  - Traverse **lastLSN** in the same order, but in reverse, for how the updates happen originally
- Write a **CLR** for every modification

Summary: ARIES

- Main ideas of ARIES:
  - WAL with **Steal** + **No-force**
  - Fuzzy checkpoints (snapshot of dirty page IDs)
  - Redo everything since the earliest dirty page
  - Undo txns that never commit
  - Write CLRs when undoing, to survive failures during restarts

- Log sequence numbers:
  - LSNs identify log records; linked into backward chains per txn via prevLSN
  - pageLSN allows comparison of data page and log records