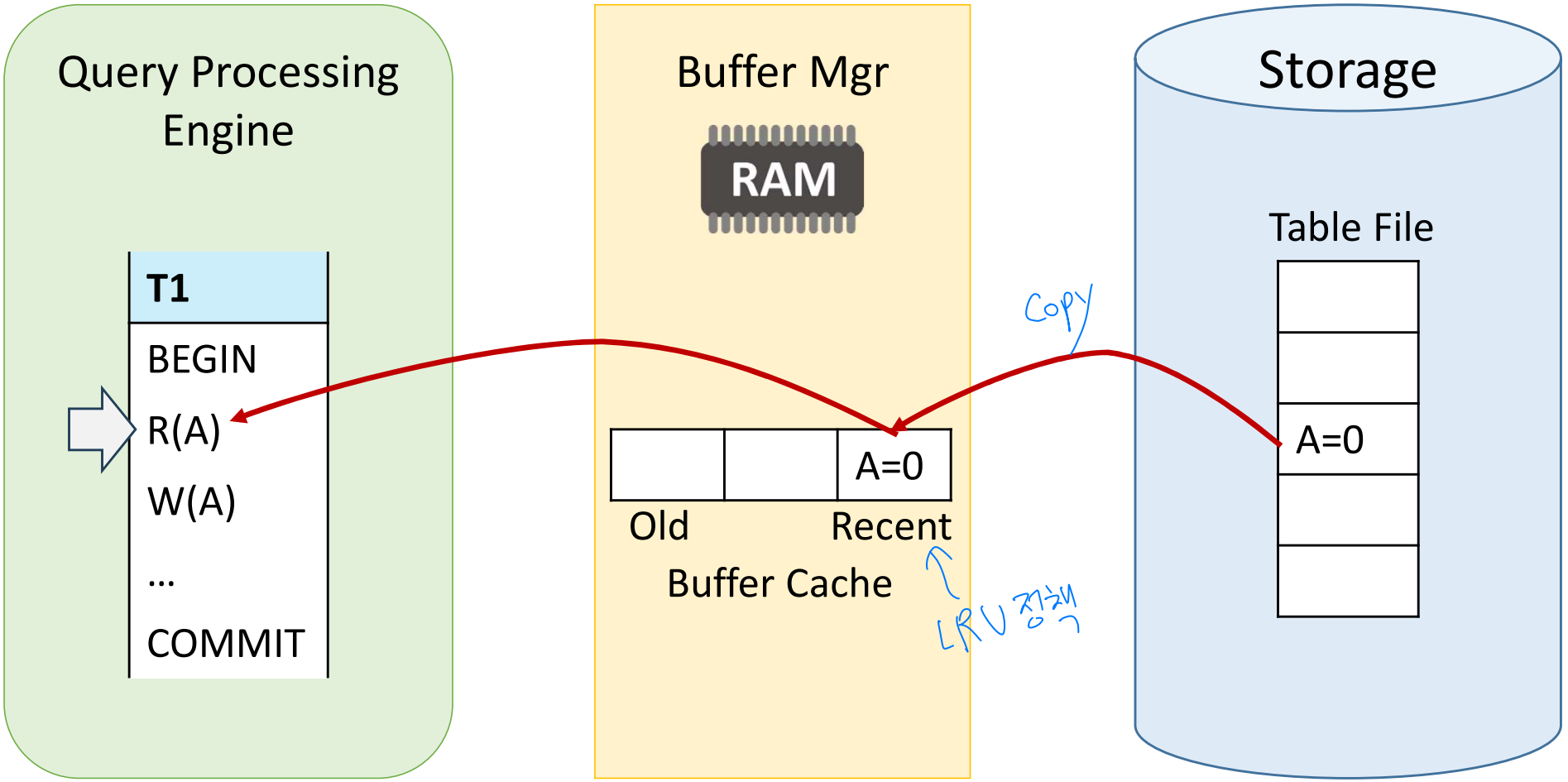
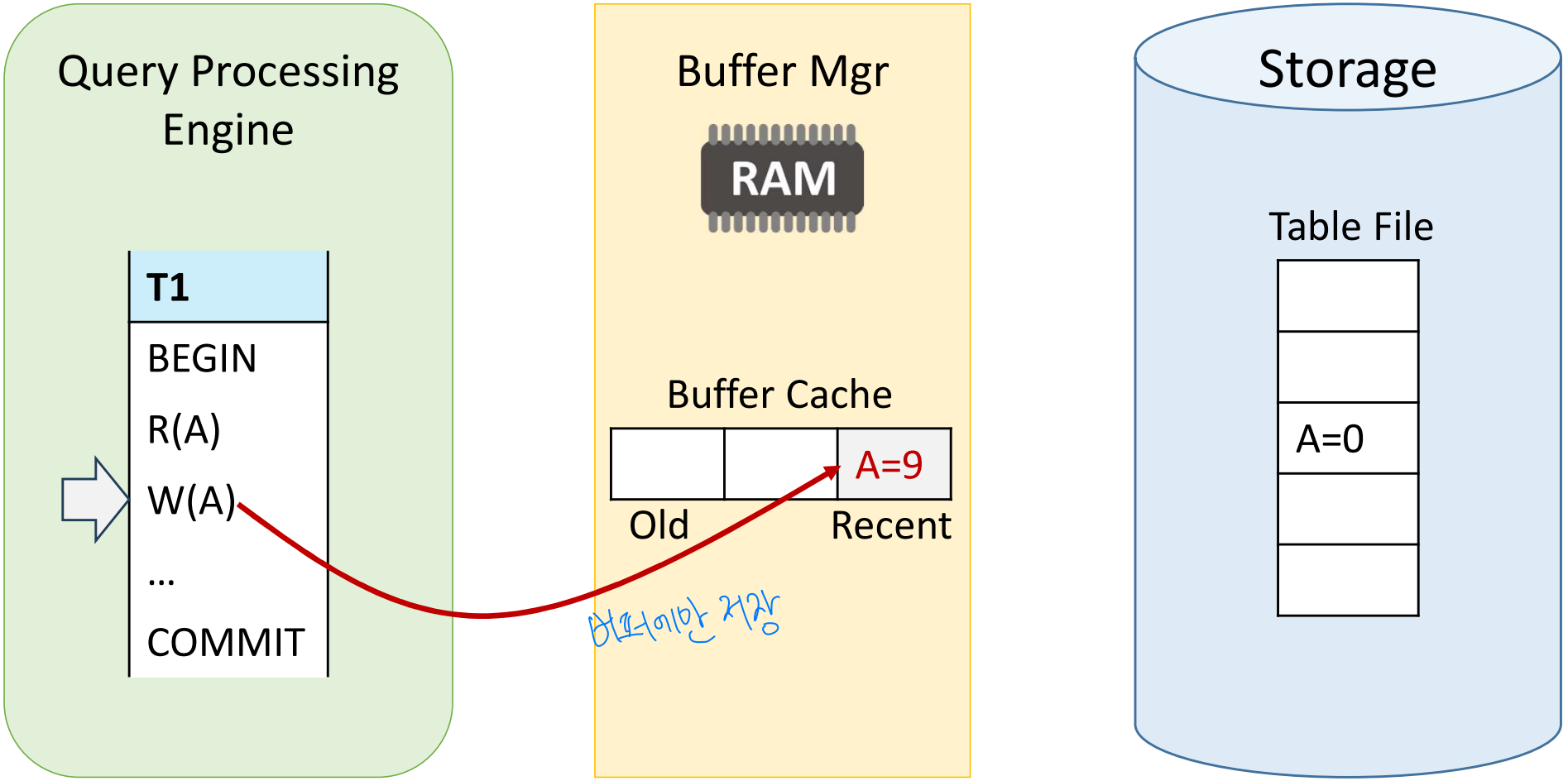
Database Systems
Lecture21 – Chapter 19: Recovery System

Beomseok Nam (남범석)
bnam@skku.edu

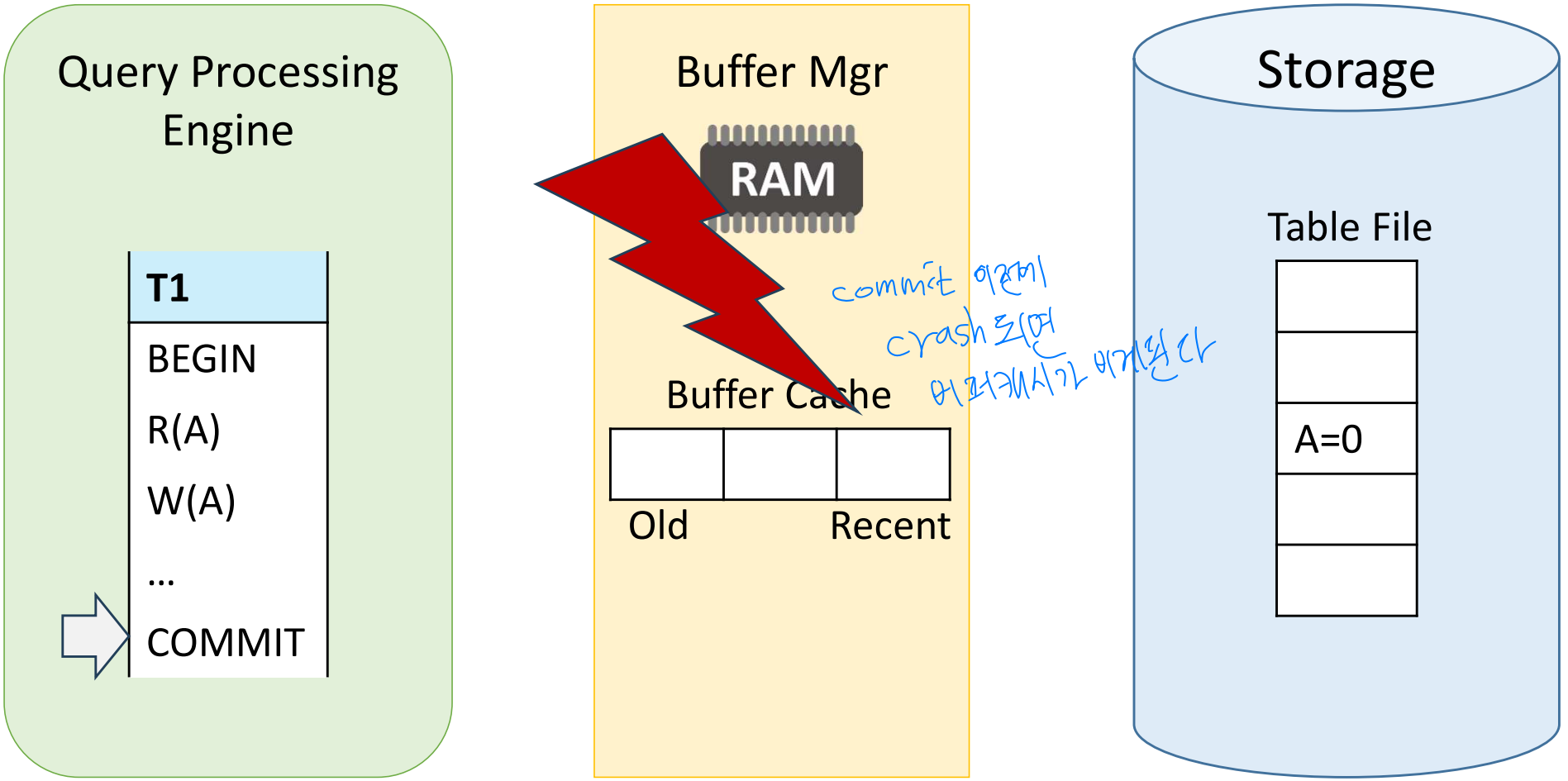
Buffer Manager - Review



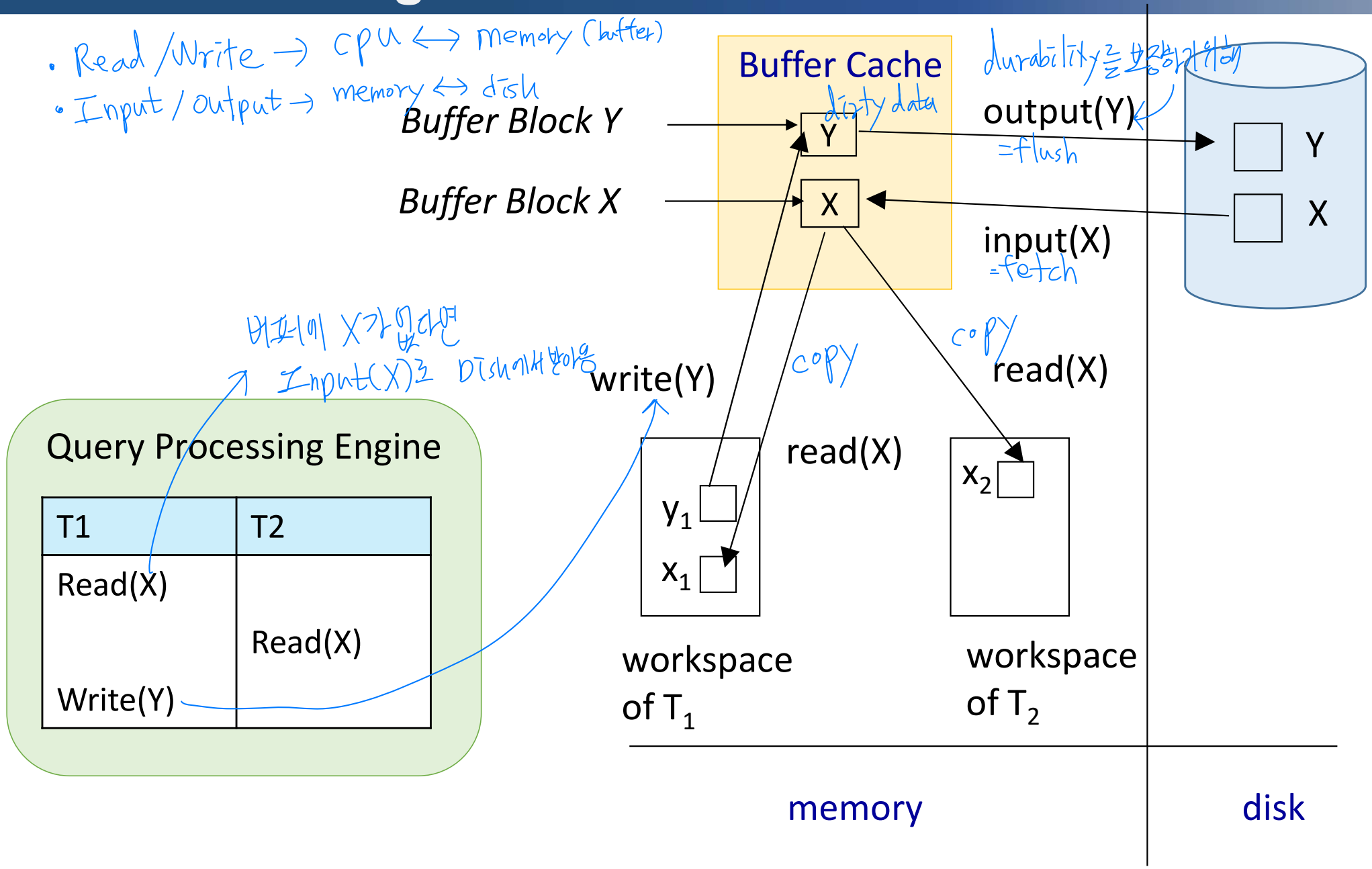
Buffer Manager - Review



Buffer Manager - Review

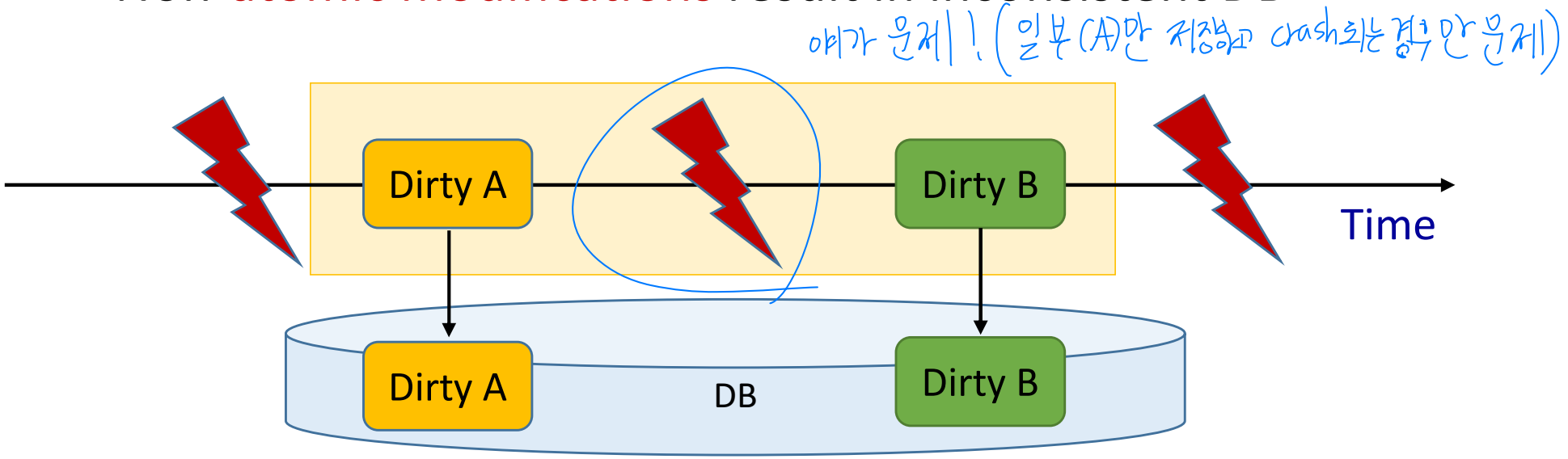


Buffer Manager for Concurrent Transactions



Recovery

- Suppose transaction T_i transfers \$50 from account *A* to *B*
  - subtract 50 from A
  - add 50 to B
- T_i requires both updates to A and B to be written to DB.
  - A failure may occur before both of modifications are made.
  - Not modifying the database may result in lost updates if failure occurs just after transaction commits
  - Non-atomic modifications result in inconsistent DB



UNDO vs. REDO

abort된 txn가 준 영향을 모두 삭제

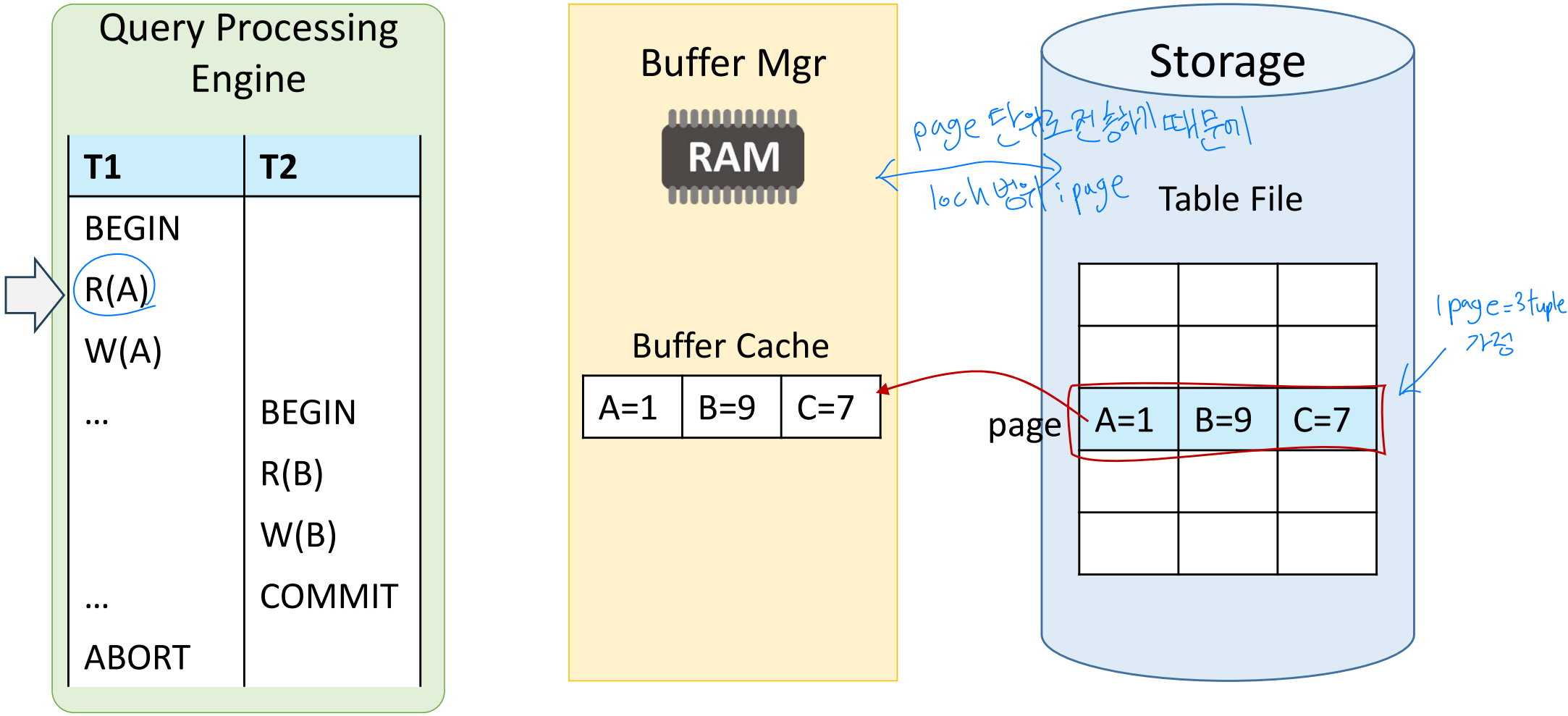
- **Undo**: The process of removing the effects of an incomplete or aborted txn.

commit전에 crash발생 → crash 이전의 내용을 다시 적용

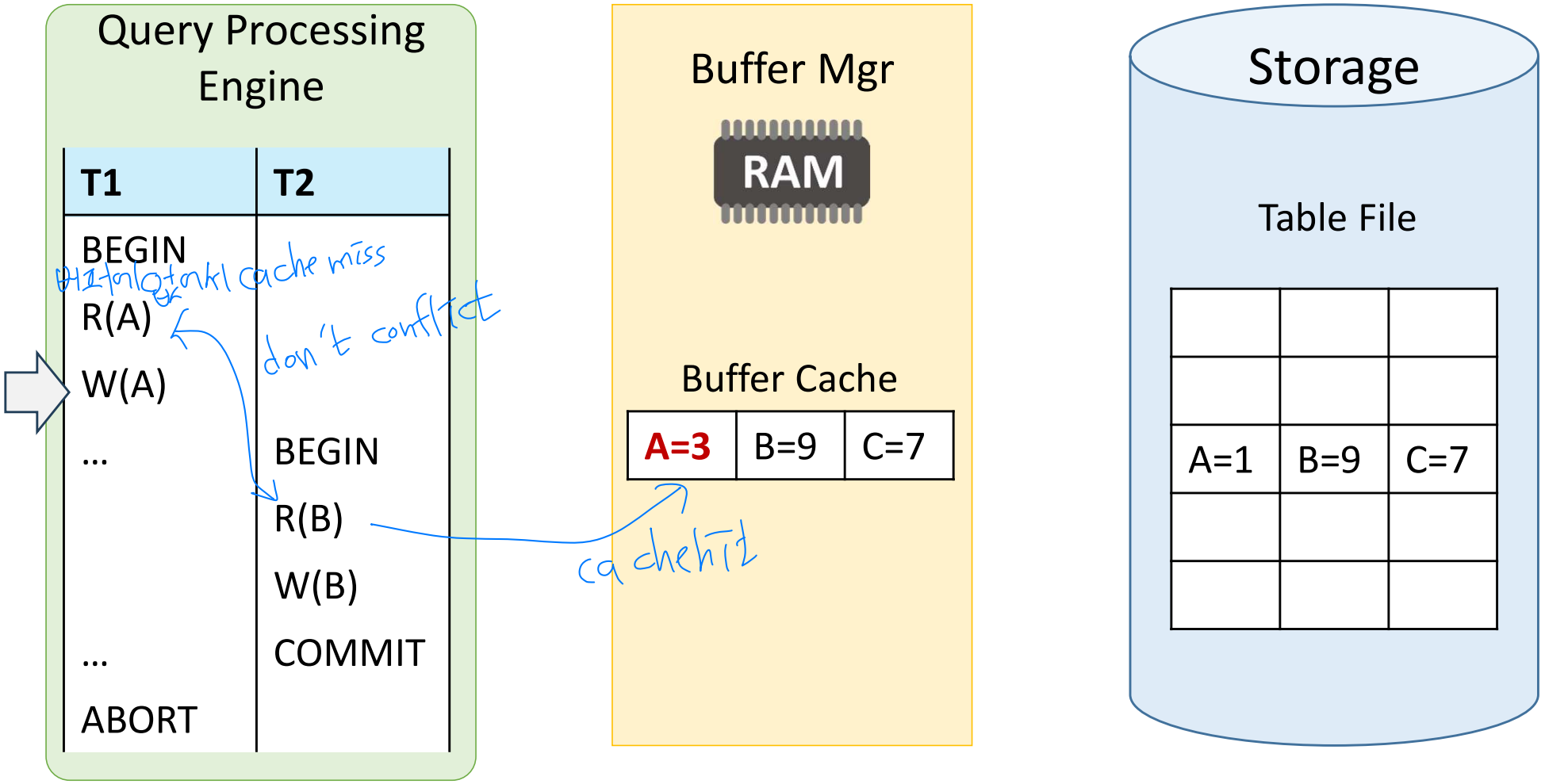
- **Redo**: The process of re-applying the effects of a committed txn for durability.

- How the DBMS supports this Undo/Redo depends on how it implements the Buffer Manager

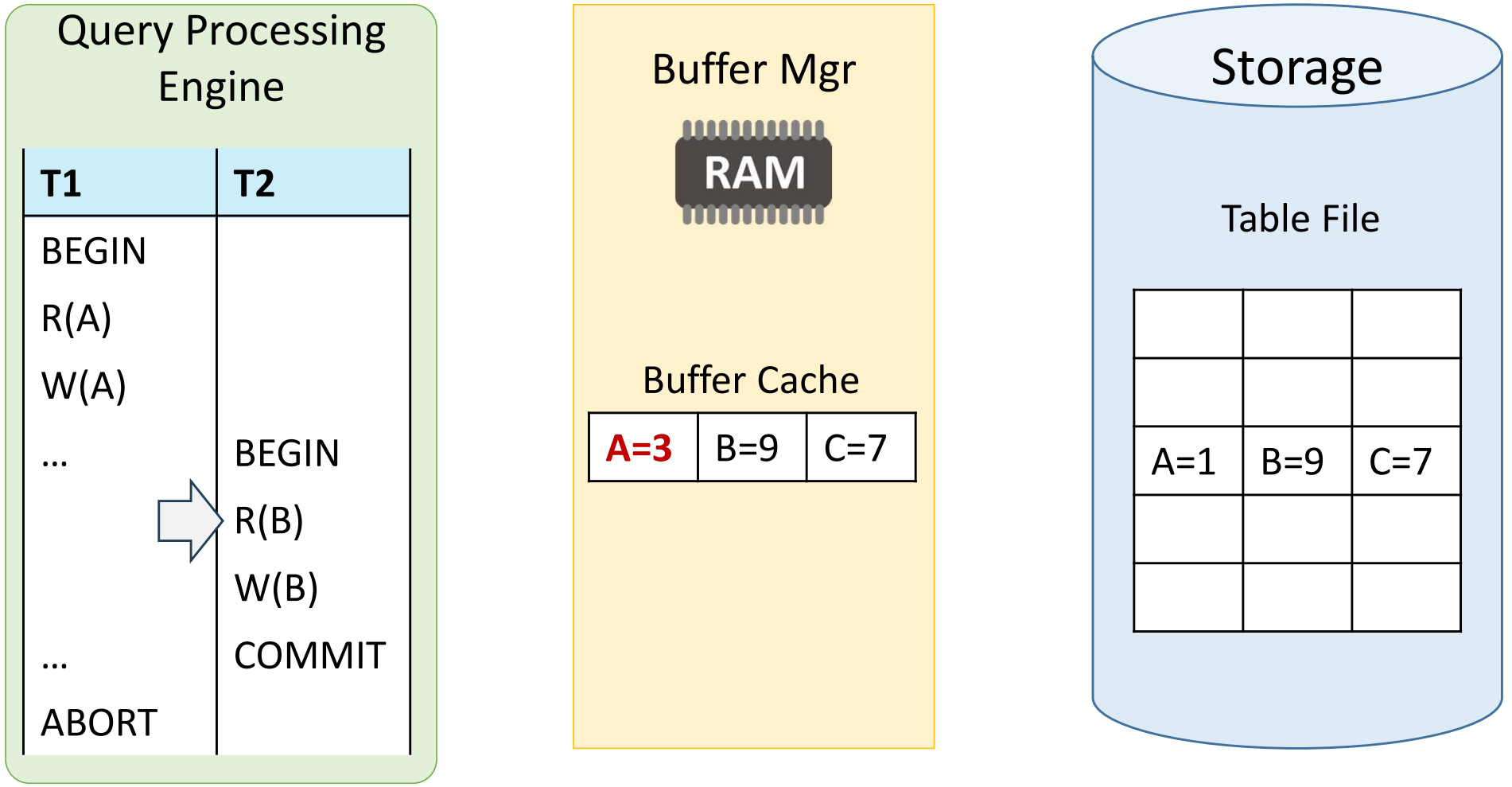
Buffer Manager



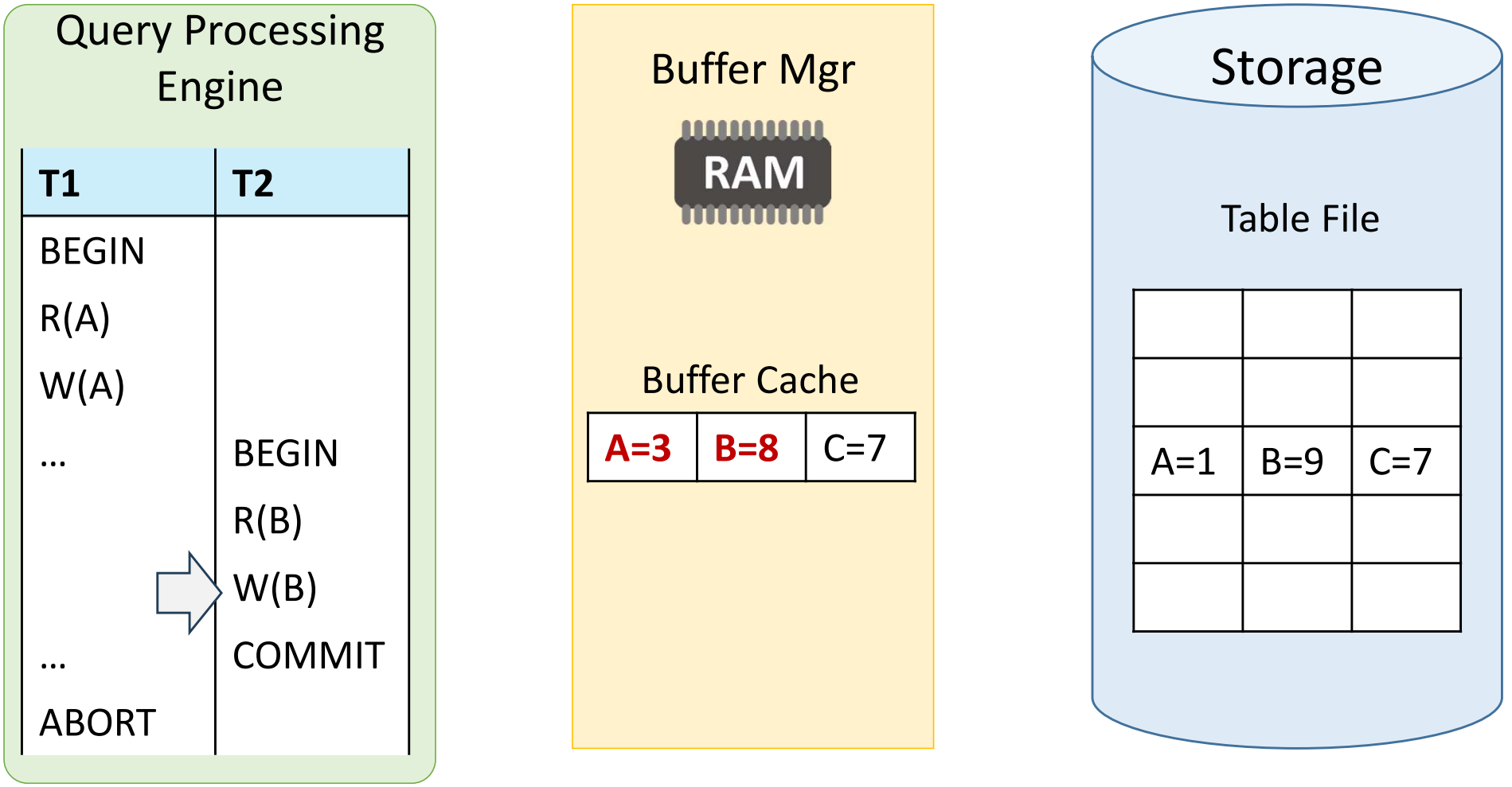
Buffer Manager



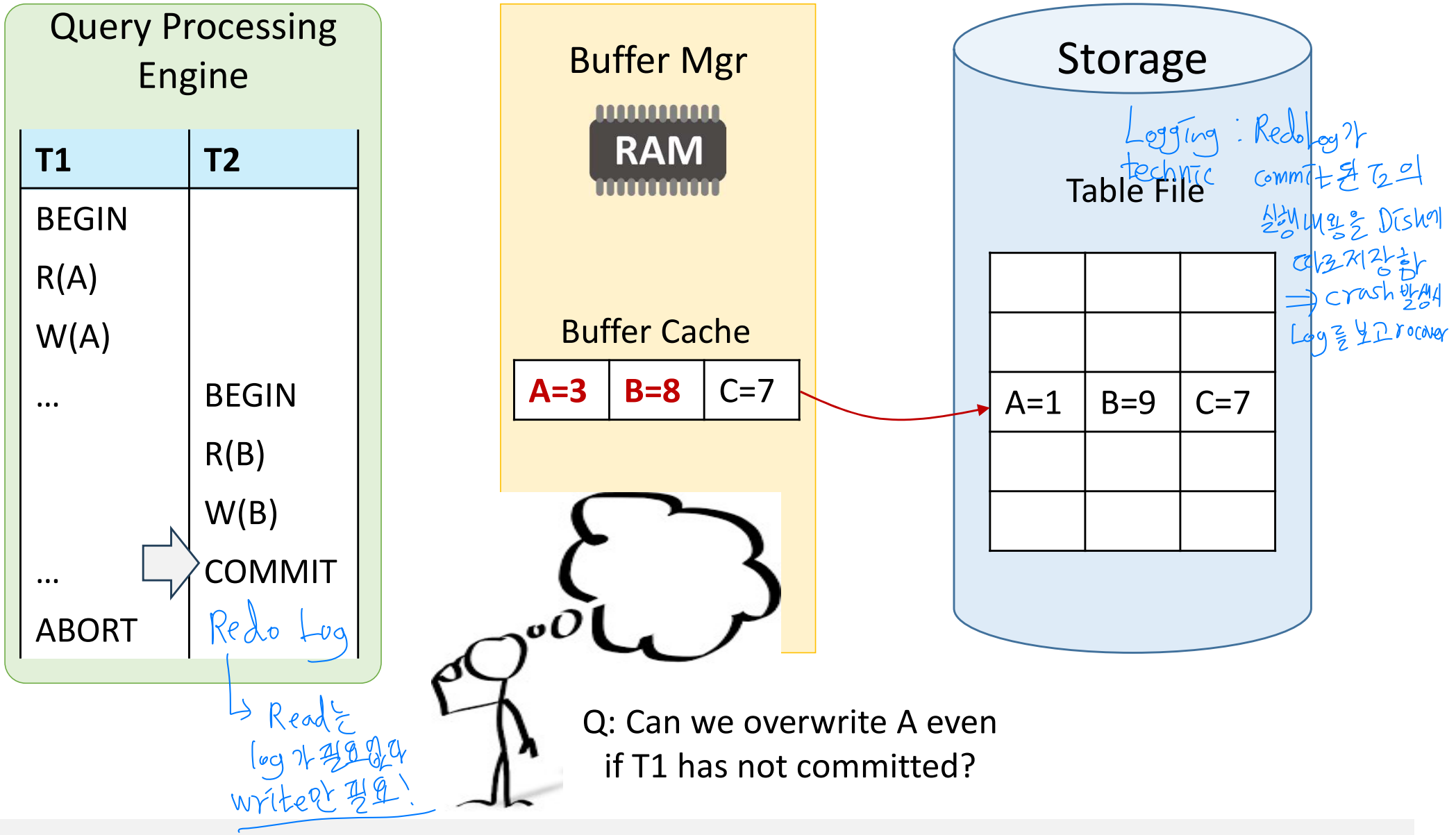
Buffer Manager



Buffer Manager



Buffer Manager



Steal Policy

- Whether the DBMS allows an uncommitted transaction to overwrite the most recent committed value of an object in non-volatile storage.

Logging technic의 관점

대부분 DBMS에서 "Steal+No-Force" 구조사용

- **STEAL vs. NO-STEAL**
  - **STEAL**: Uncommitted transactions **can** overwrite committed data on disk. ⇒ commit되지 않은 내용을 디스크에 덮어쓰기 / 만약 abort 등의 상황으로 consistent상태로 돌아가고싶다 → log보면서 recover (Inconsistent db가됨)
  - **NO-STEAL**: Uncommitted transactions **cannot** overwrite committed data. ⇒ commit되지 않은 내용 덮어쓰지않기 / buffer에서 copy본을 만들어서 commit된 데이터만 Flush

ex) A=3 | B=8 | C=7

↓ copy

A=1 | B=8 | C=7 → Flush

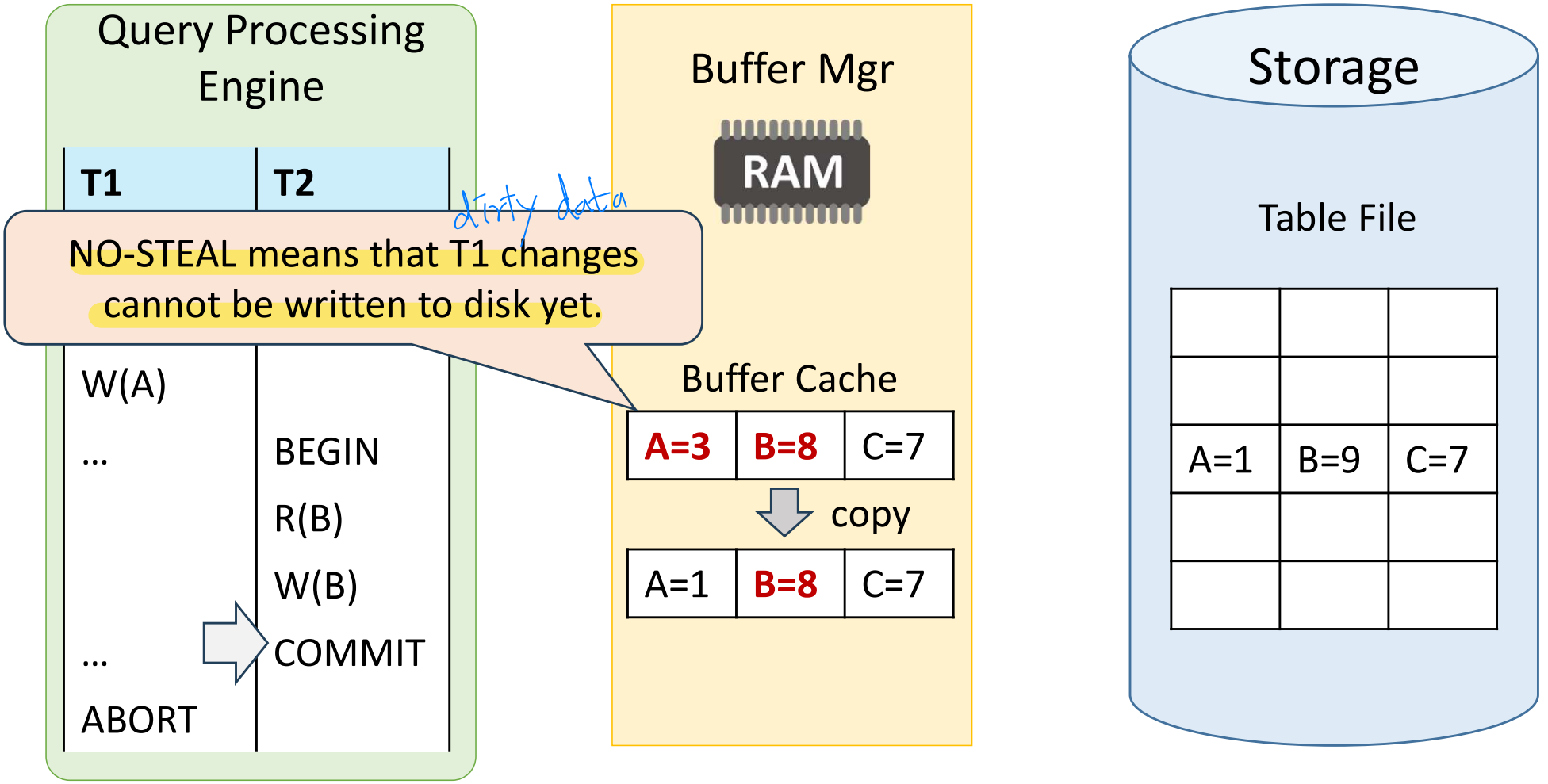
<Buffer>

Force Policy

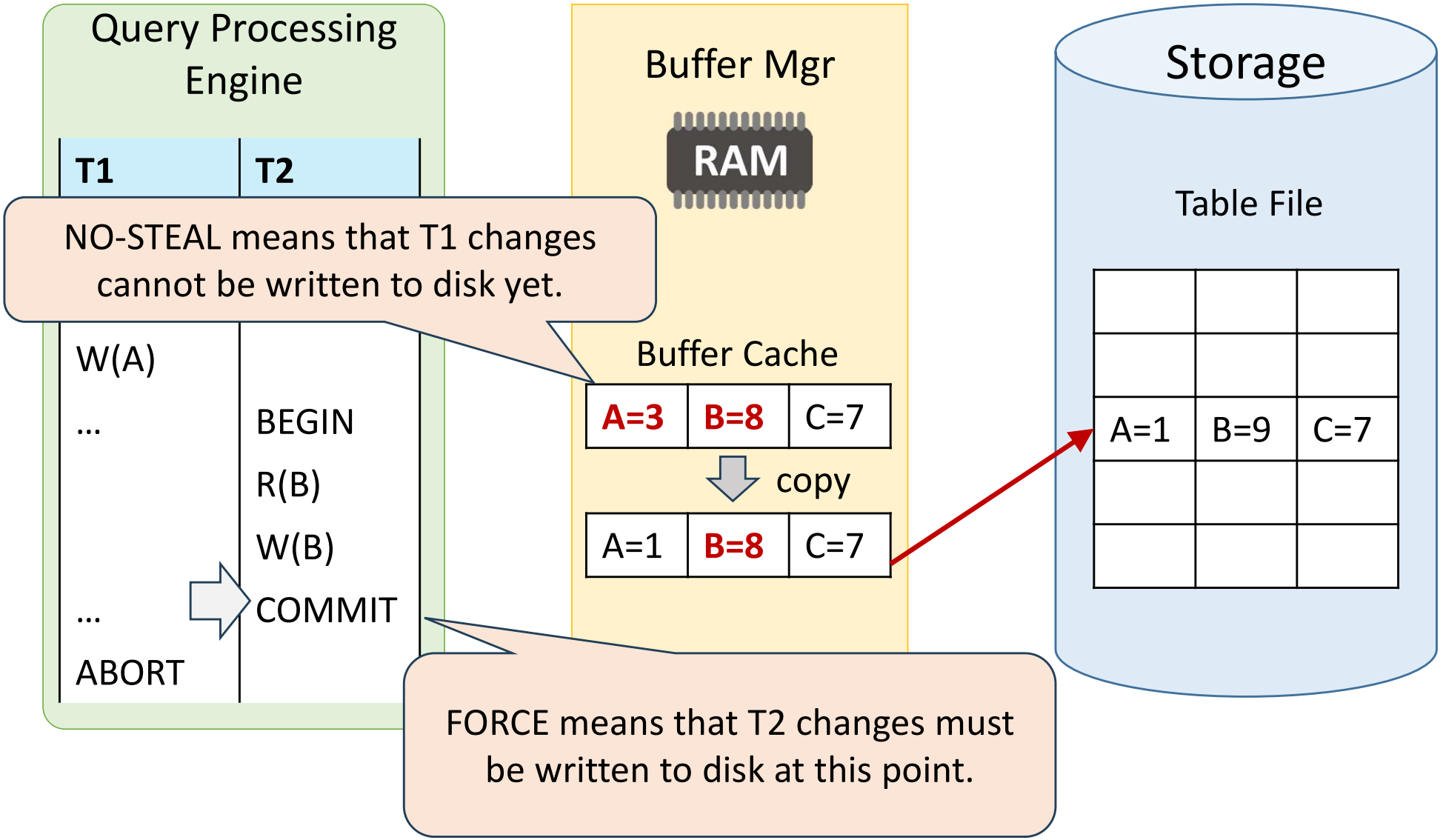
- Whether the DBMS requires that all updates made by a transaction are reflected on storage before the transaction can commit.

- **FORCE vs. NO-FORCE**
  - **FORCE**: All updates must be written to disk **before commit**
  - **NO-FORCE**: Updates **don't need to be written to disk** at commit time

NO-STEAL + FORCE



NO-STEAL + FORCE



NO-STEAL + FORCE

- This approach is the easiest to implement:

  → **Never have to undo** changes of an aborted transaction because the changes were not written to disk.

  → **Never have to redo** changes of a committed transaction because all the changes are guaranteed to be written to disk at commit time (assuming atomic hardware writes).

commit된 data만 Flush하기에 DB가 consistent함(가장 최근 데이터가 아닐수도..)
⇒ redo, undo (logging) 필요없다

Shadow Paging

- **Shadow paging** is useful if transactions execute serially
- Idea: Use *two* page tables

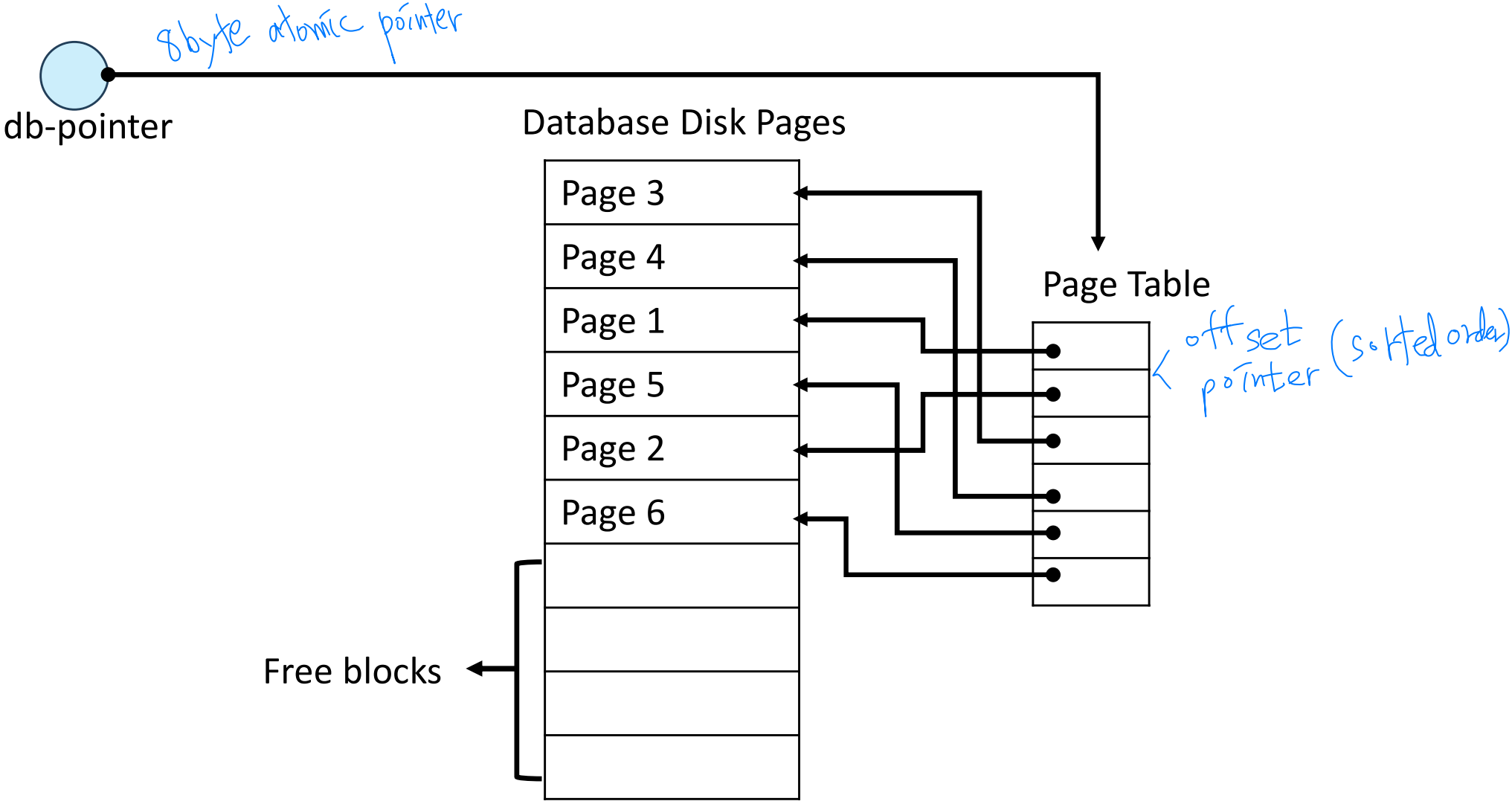
1. the **current page table**
   - used for data accesses during execution of the transaction.
2. the **shadow page table**
   - Shadow page table is never modified during execution

- Whenever any page is about to be written for the first time
  - Copy this page on an unused page
  - Update the current page table to point to the copy
  - Perform the update on the copy
  - To commit, it swaps the current and shadow page tables

  - Buffer Pool Policy: STEAL + NO-FORCE

Page Table

- Mapping between logical pages and physical pages

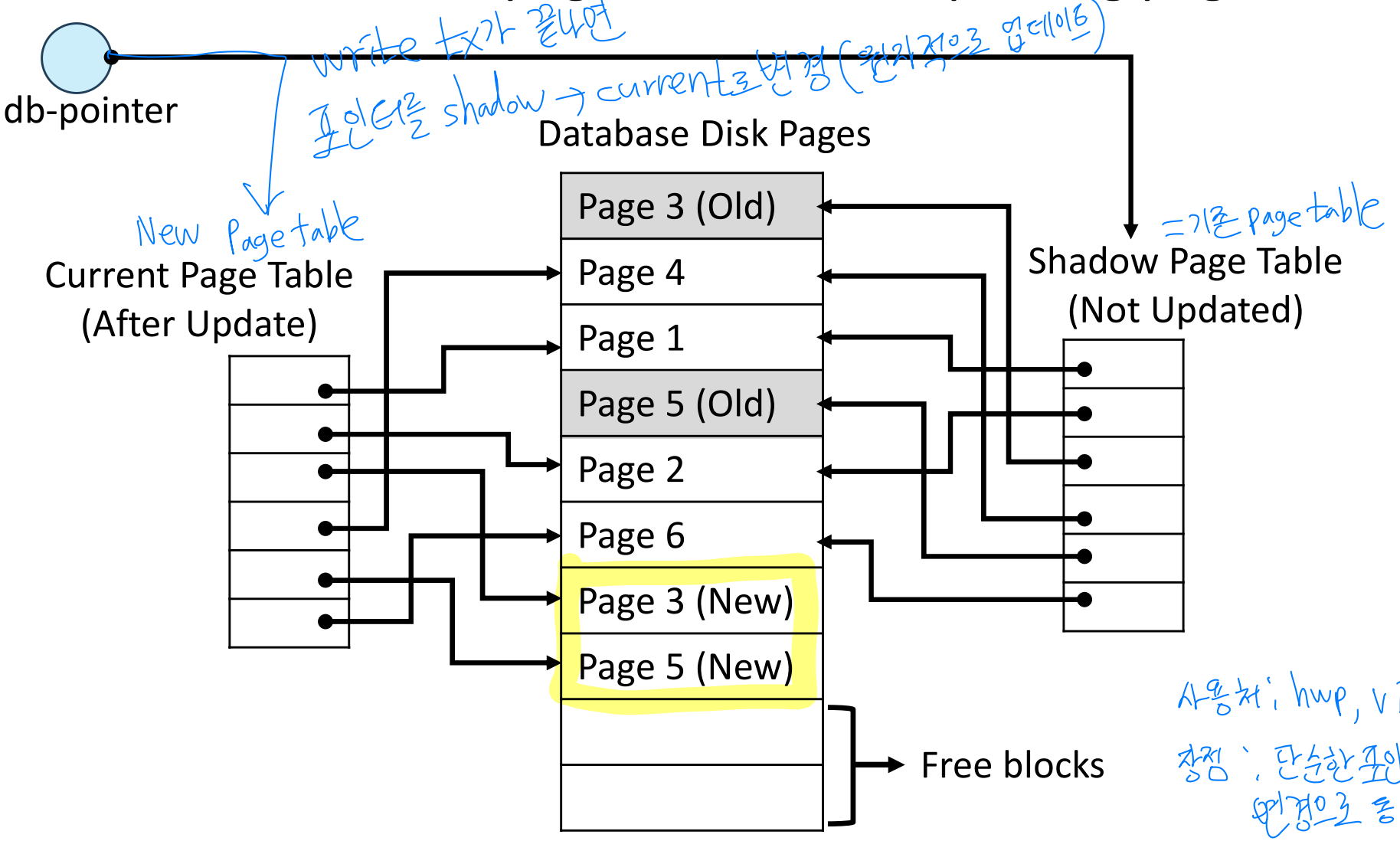


Shadow Paging

- To commit a transaction :
  1. Flush all modified pages in main memory to disk
  2. Output current page table to disk
  3. Make the current page table the new shadow page table
     - keep a pointer to the shadow page table at a fixed (known) location on disk.
- When *the pointer to shadow page table has been written*, transaction is committed.

Shadow Paging

- Shadow and current page tables after updating page 3, 5



Pros and Cons of Shadow Paging

- Advantages
  - no overhead of writing log records (log 사용X)
  - **Recovery is not needed** (단순한 포인터 변경으로 recover 가능)
    - new transactions can start right away, using the shadow page table.

- Disadvantages
  - **Copying the entire page table is very expensive**
  - Commit overhead is still high
    - Must flush every updated page, and page table
  - Old pages need **garbage collection** → old page들을 shadowpage가 필요없어지면 재활용할 수 있는 공간으로 만들어줌
    - Free pages not referenced by current/shadow page tables
  - **Concurrent transactions not supported** ⇒ 다른 tx 접근이 많은상황에서 쓰기 힘들다
    (↑ 2개의 write tx가 충돌하는 상황에서는 1개만 성공함)

WAL (Write-Ahead Log)

- Keep a **separate log file** that records all changes made by transactions.
- The **log must be on stable storage**
- Log contains enough info to support **undo** and **redo**
- **Before** flushing a data object to disk,
  → The corresponding **log records must be written to disk**

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

WAL (Write-Ahead Log)

- All **log records** for a transaction are first **staged in volatile storage** (e.g., buffer cache)
- **Before** a page is written to non-volatile storage,
  → All **associated log records** must be flushed to disk
- A transaction is **not committed** until **all its log records** are written to **stable storage**
- Updates don't need to be flushed to DB when commits (No-Force)

dirty page flush되기전에 log가 disk에 기록되어야한다

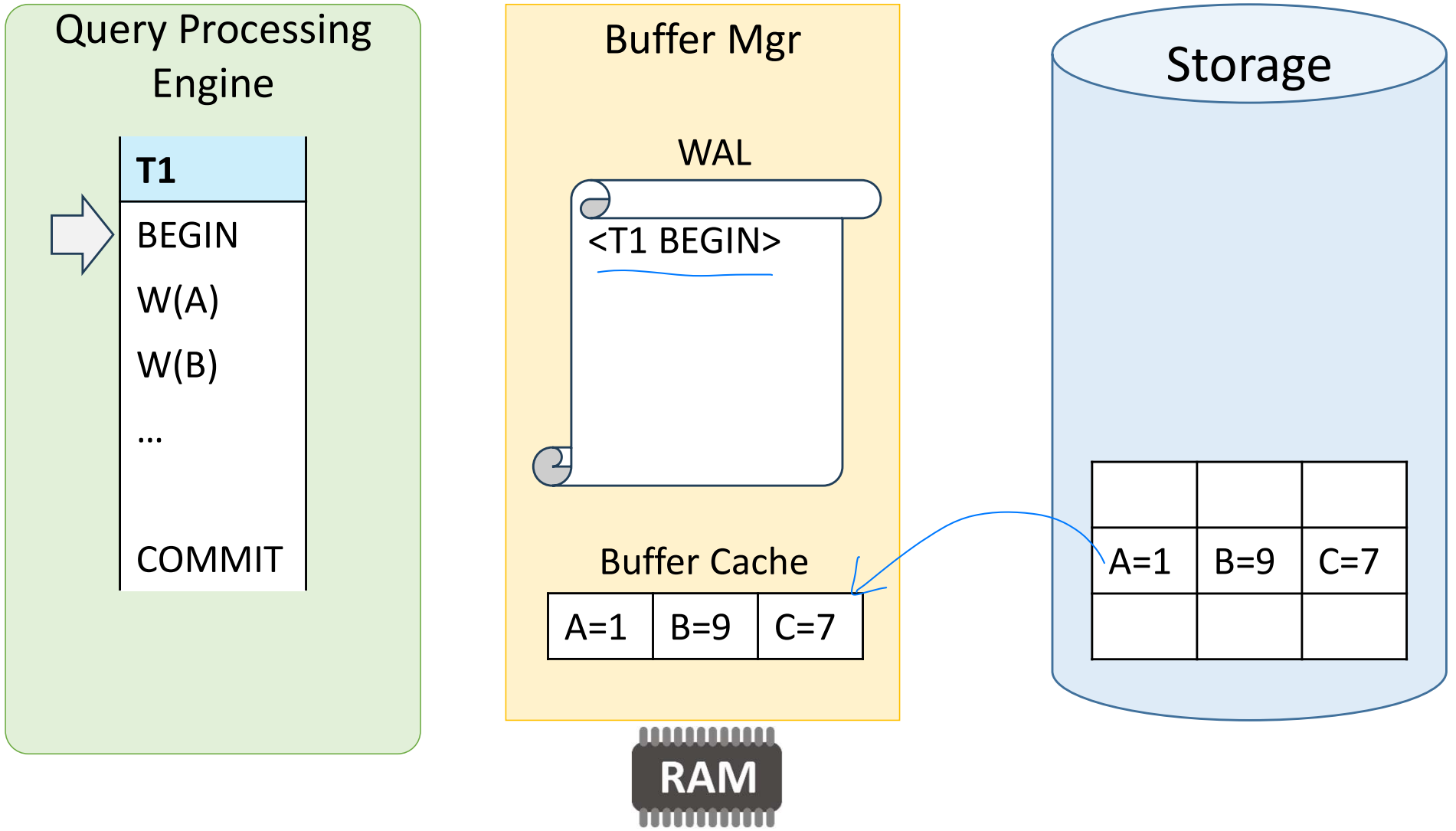
WAL (Write-Ahead Log) – Log Entries

- At the **start** of each transaction T_i:
  → Write a <T_i BEGIN> record to the log

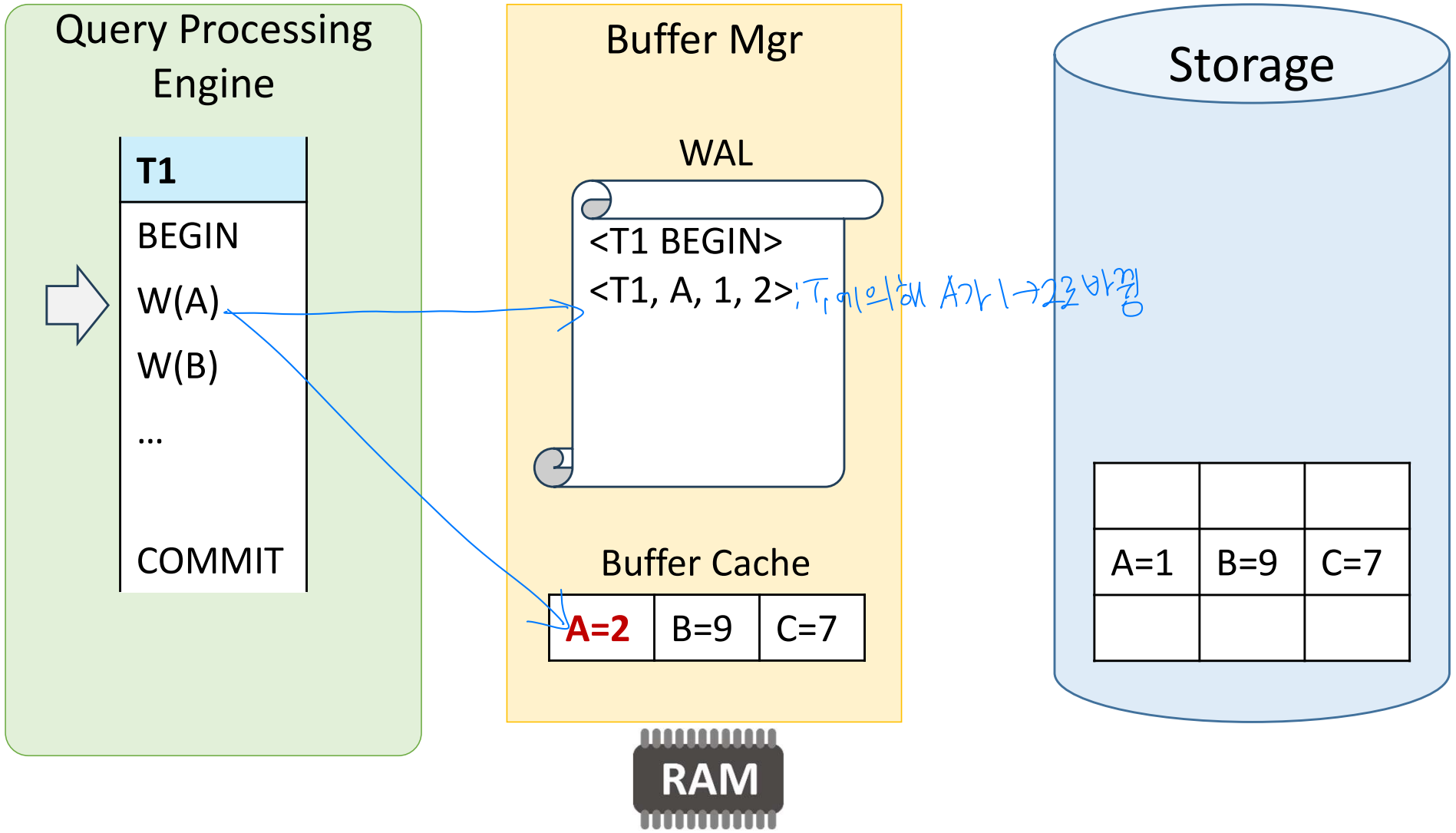
- When the transaction **completes**:
  → **Make sure all log records** are flushed to stable storage
  → Write a <T_i COMMIT> record to the log
  → **Only then** return a commit acknowledgment to the application

- When the transaction **writes**:
  → **Write a log entry** <T_i, X, V_1, V_2> that contains information about the change to a single object:
  - Transaction Id (T_i)
  - Object Id (X)
  - Before Value for UNDO (V_1)
  - After Value for REDO (V_2)

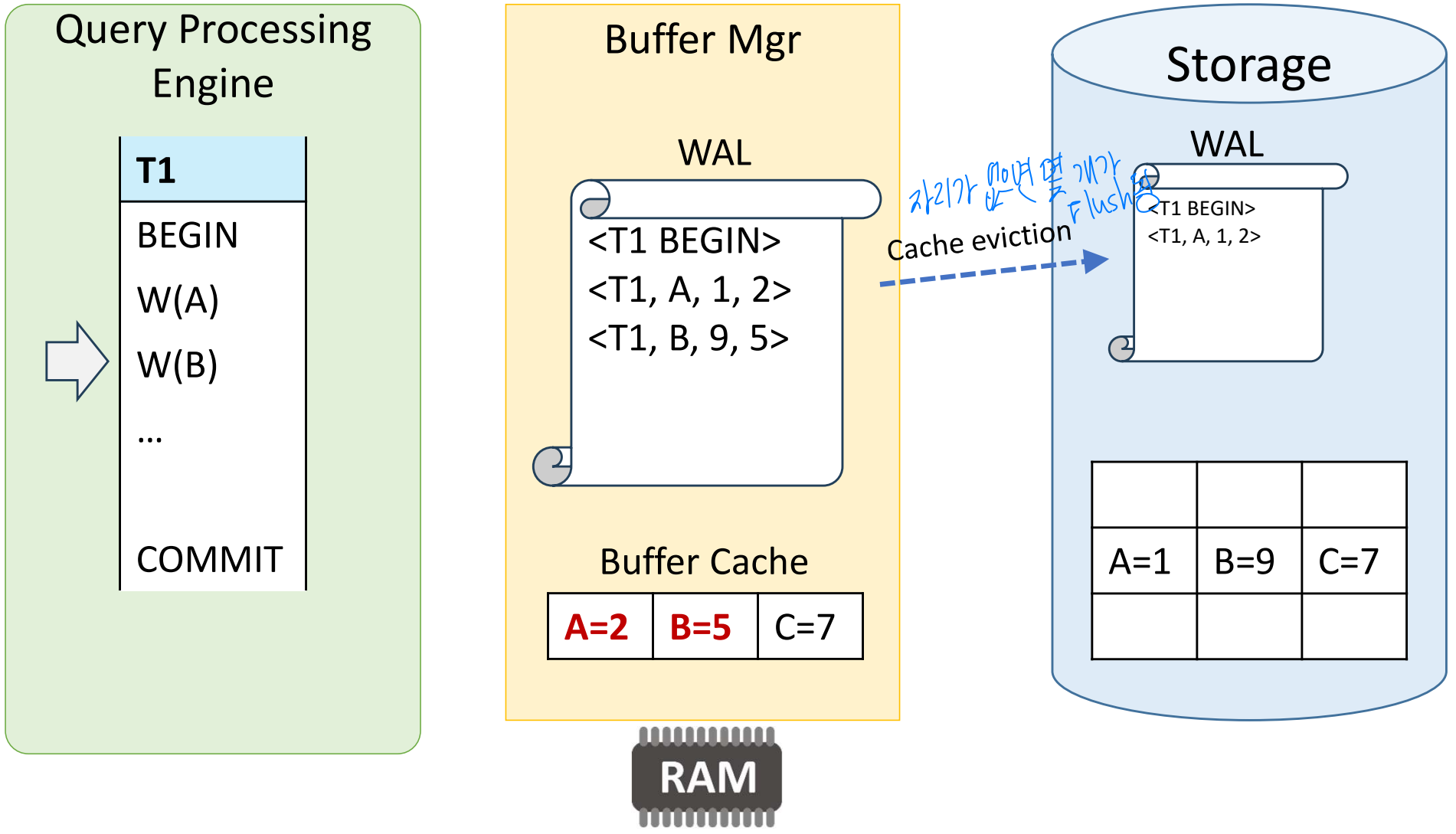
WAL – Example


Query Processing Engine
T1
BEGIN
W(A)
W(B)
…
COMMIT
Buffer Mgr
WAL
<T1 BEGIN>
Buffer Cache
A=1
B=9
C=7
RAM
Storage
A=1
B=9
C=7

WAL – Example

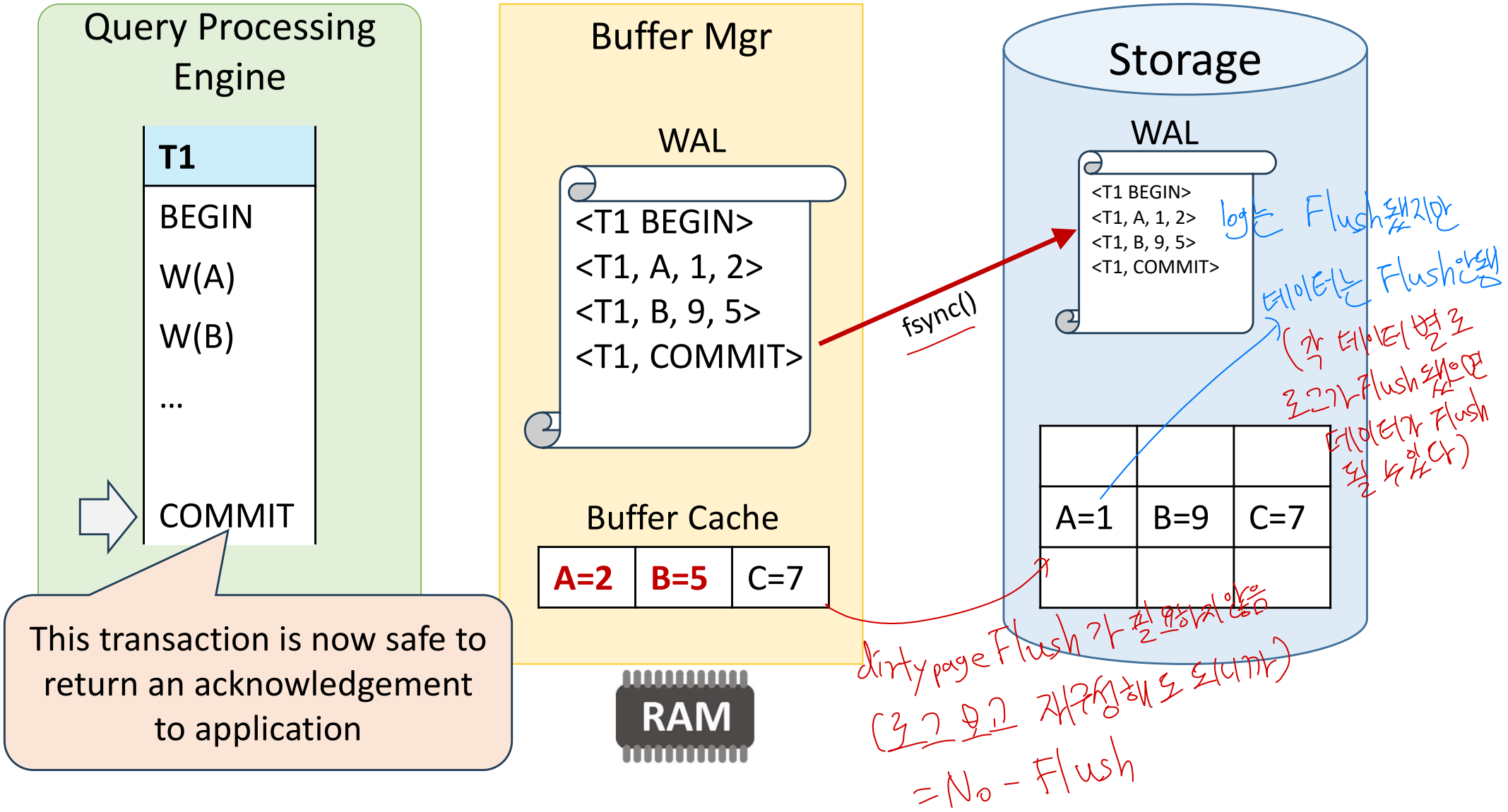

Query Processing Engine
T1
BEGIN
W(A)
W(B)
…
COMMIT
Buffer Mgr
WAL
<T1 BEGIN>
<T1, A, 1, 2>
Buffer Cache
A=2
B=9
C=7
RAM
Storage
A=1
B=9
C=7

WAL – Example

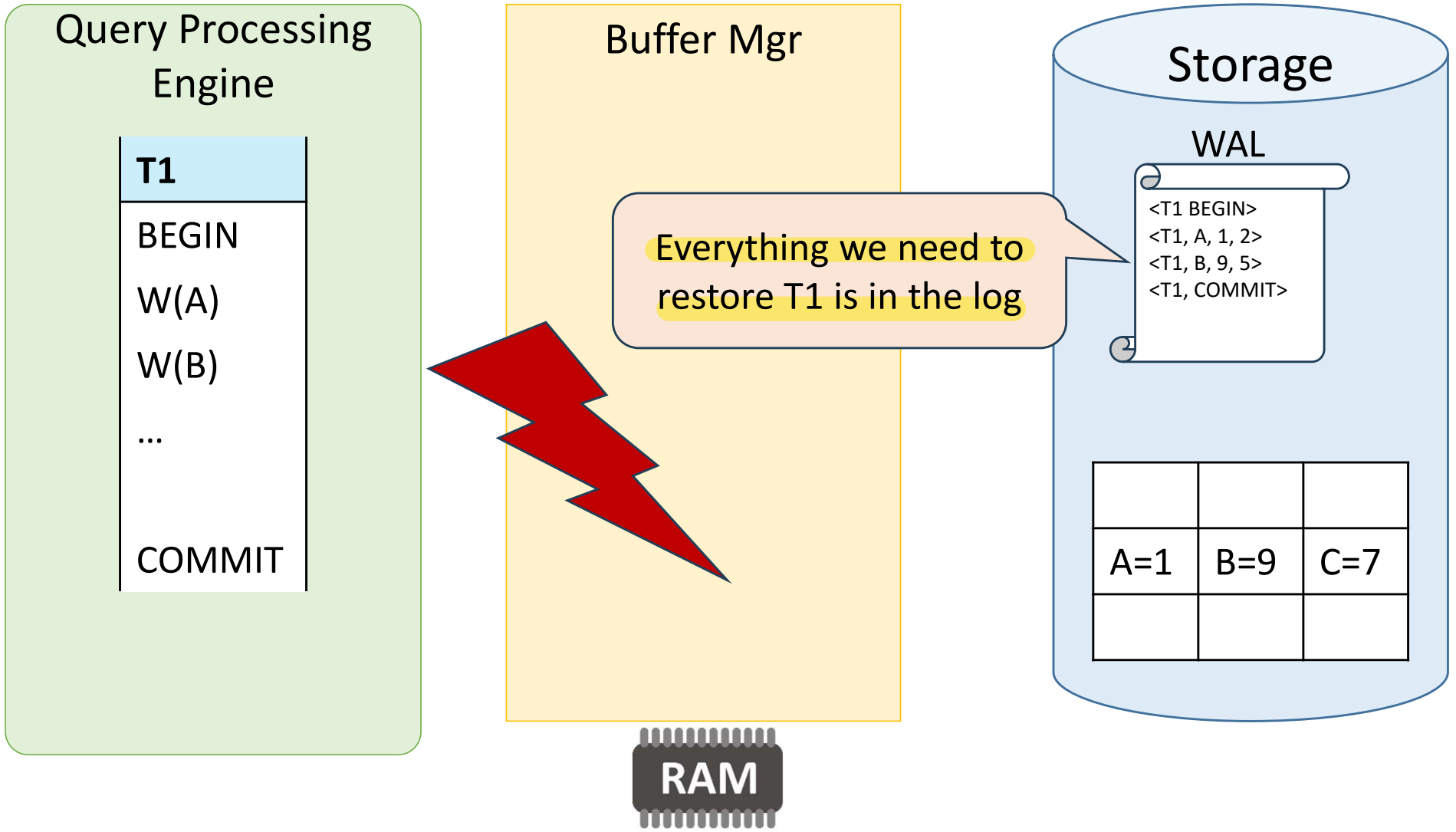

Query Processing Engine
T1
BEGIN
W(A)
W(B)
...
COMMIT
Buffer Mgr
WAL
<T1 BEGIN>
<T1, A, 1, 2>
<T1, B, 9, 5>
Buffer Cache
A=2
B=5
C=7
RAM
Cache eviction
Storage
WAL
<T1 BEGIN>
<T1, A, 1, 2>
A=1
B=9
C=7

WAL – Example

Write Ahead log → write 전에 logging하라



WAL – Example


Query Processing Engine
T1
BEGIN
W(A)
W(B)
…
COMMIT
Buffer Mgr
RAM
Everything we need to restore T1 is in the log
Storage
WAL
<T1 BEGIN>
<T1, A, 1, 2>
<T1, B, 9, 5>
<T1, COMMIT>
A=1
B=9
C=7

No-Force + Steal vs Steal + No-Force

- Almost every DBMS uses STEAL+NO FORCE (WAL)

No-Flush가 빠르다!
(log가 크기가 작아서
Flush하는게 더 빠르다)

Runtime Performance

	NO-STEAL	STEAL
NO-FORCE	-	Fast 빠르게
FORCE	Slow	-

Recovery Performance

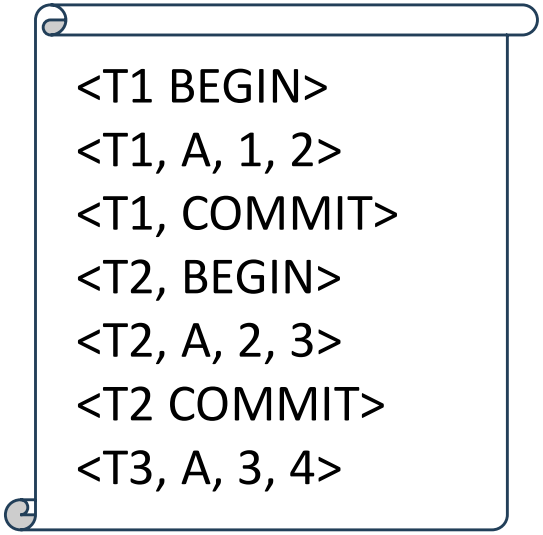
	NO-STEAL	STEAL
NO-FORCE	-	Slow 로그를 다 읽어야해서
FORCE	Fast	-

UNDO
REDO

No UNDO
No REDO

Concurrency Transactions and WAL

- **Single buffer** and **single log** are shared by all transactions
- A buffer page may be updated by **multiple transactions**
- Log records from different transactions can be **interleaved**



- Key Assumption: *if a transaction T_i modifies an item, no other transaction can modify the same item until T_i commits/aborts*
  - i.e., No Dirty Read, as enforced by Strict 2PL

Crash Recovery: Redo and Undo

- **REDO if...**
  - Log **contains** <T_i START>
  - And also **contains** <T_i COMMIT> or <T_i ABORT>
    → T_i **must be redone**

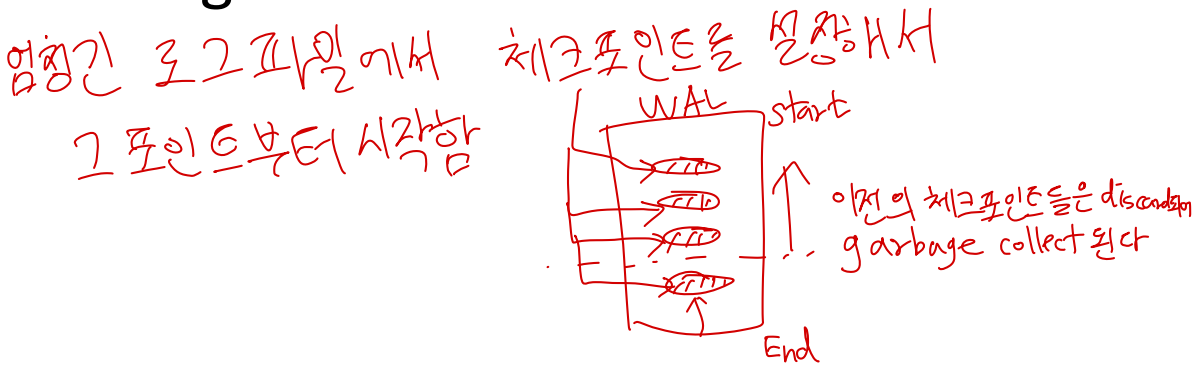
- **UNDO if...**
  - Log **contains** <T_i START>
  - But **does not contain** <T_i COMMIT> or <T_i ABORT>
    → T_i **must be undone**

Checkpoints

- The WAL file keeps growing indefinitely
  - After a crash, the DBMS must replay the **entire** log
  - Recovery becomes very slow

- **Checkpoint** to the Rescue
  - DBMS **periodically flushes all dirty pages** to disk
  - This **marks a safe point** in the log
  - After a crash, recovery can **start from the last checkpoint**, not from the beginning of the log

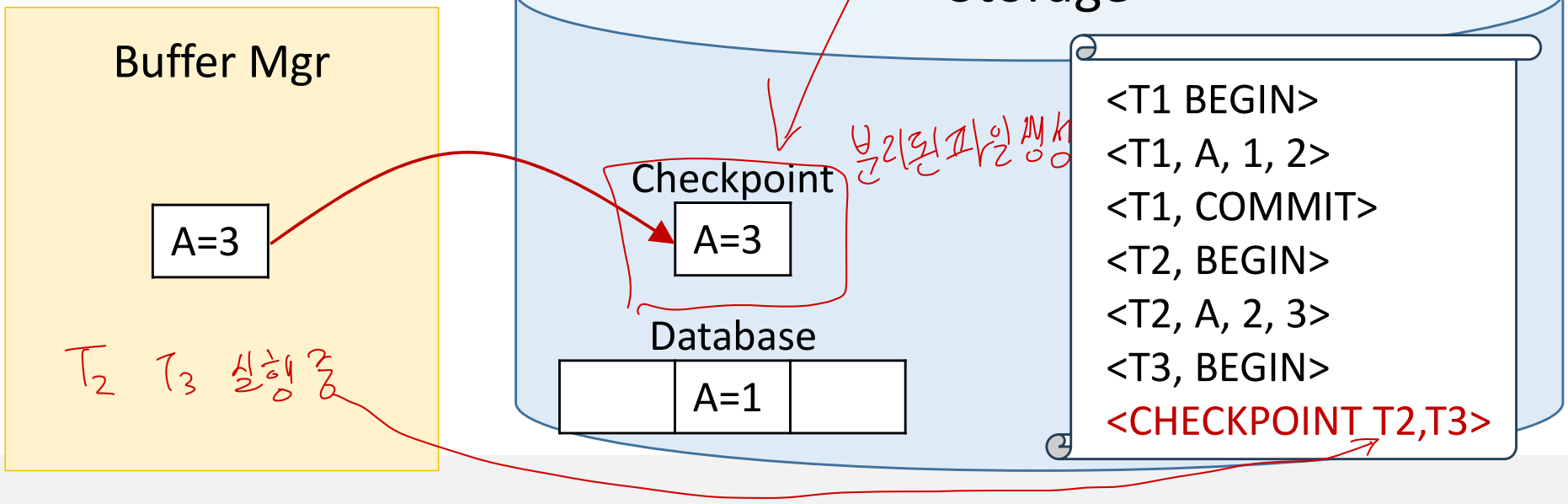
→ **Faster recovery**



Checkpoints (Cont.)

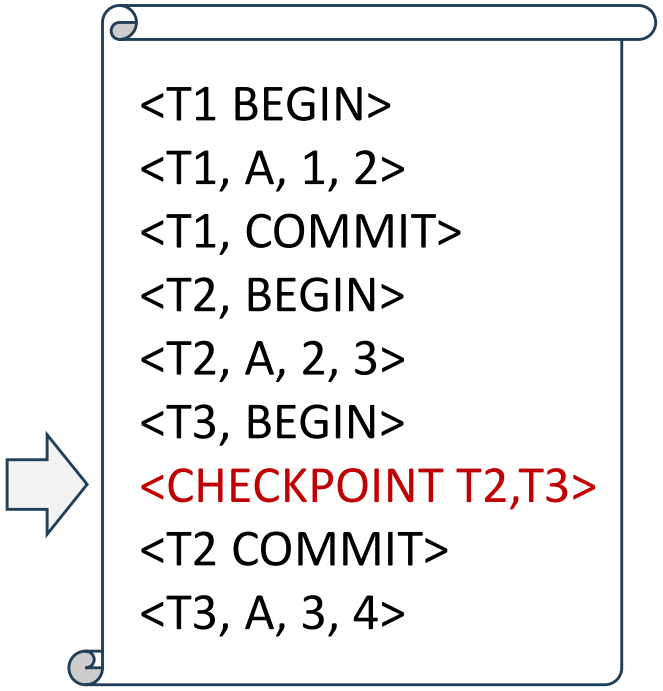
- Perform **Periodic Checkpointing**

1. **Pause all active queries**
2. **Flush all in-memory log records** to stable storage
3. **Flush all dirty buffer pages** to disk
4. **Write <checkpoint L>** to the log
   → L: List of all **active transactions** at the time of checkpoint
5. **Resume** queries



Checkpoints (Cont.)

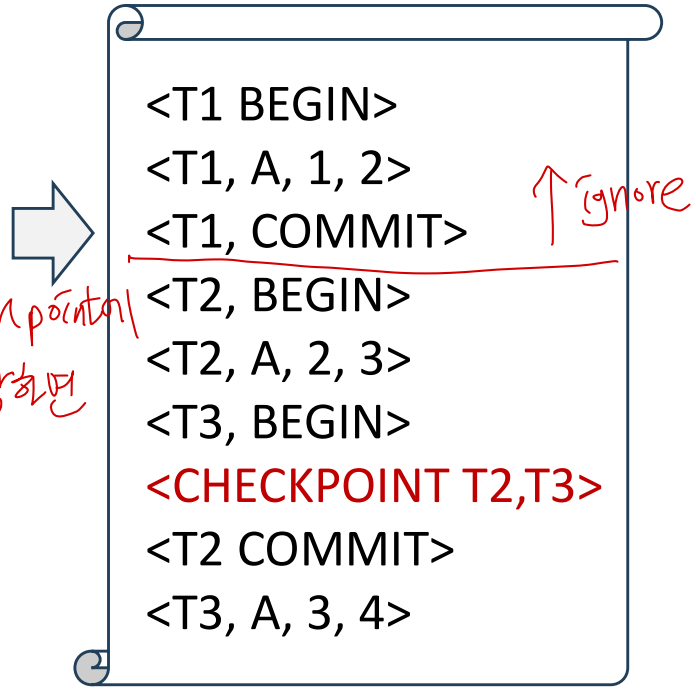
- Use **Checkpoints for Recovery**
- Use the <CHECKPOINT> record as the starting point for analyzing the WAL.



Checkpoints (Cont.)

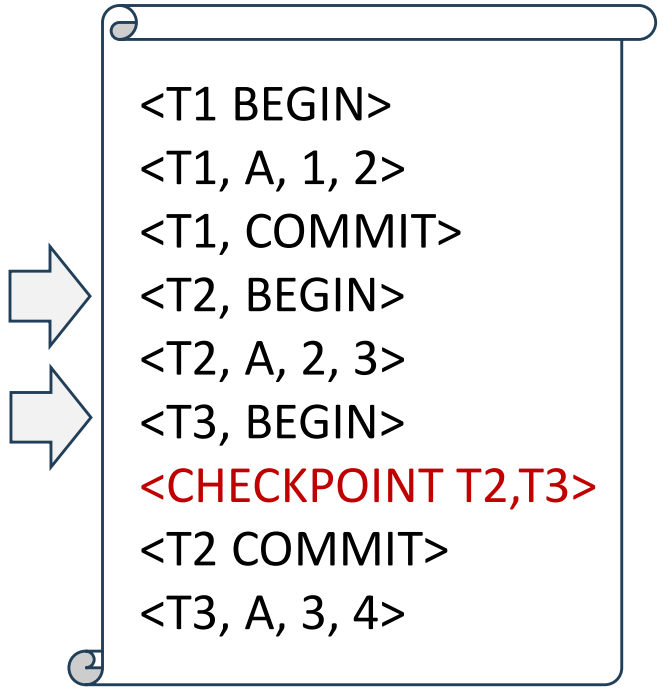
- Use **Checkpoints for Recovery**
- Use the <CHECKPOINT> record as the starting point for analyzing the WAL.
- Any transaction that committed before the checkpoint is ignored (T1).

↳ T_1이 A를 수정했으므로 버퍼에 존재
⇒ checkpoint를 설정할 때
모든 dirty page를 flush 하므로 checkpoint에
A가 저장됨/그리고 checkpoint부터 시작하면
저장된 A와 C.P → 버퍼 캐시로 옮겨짐



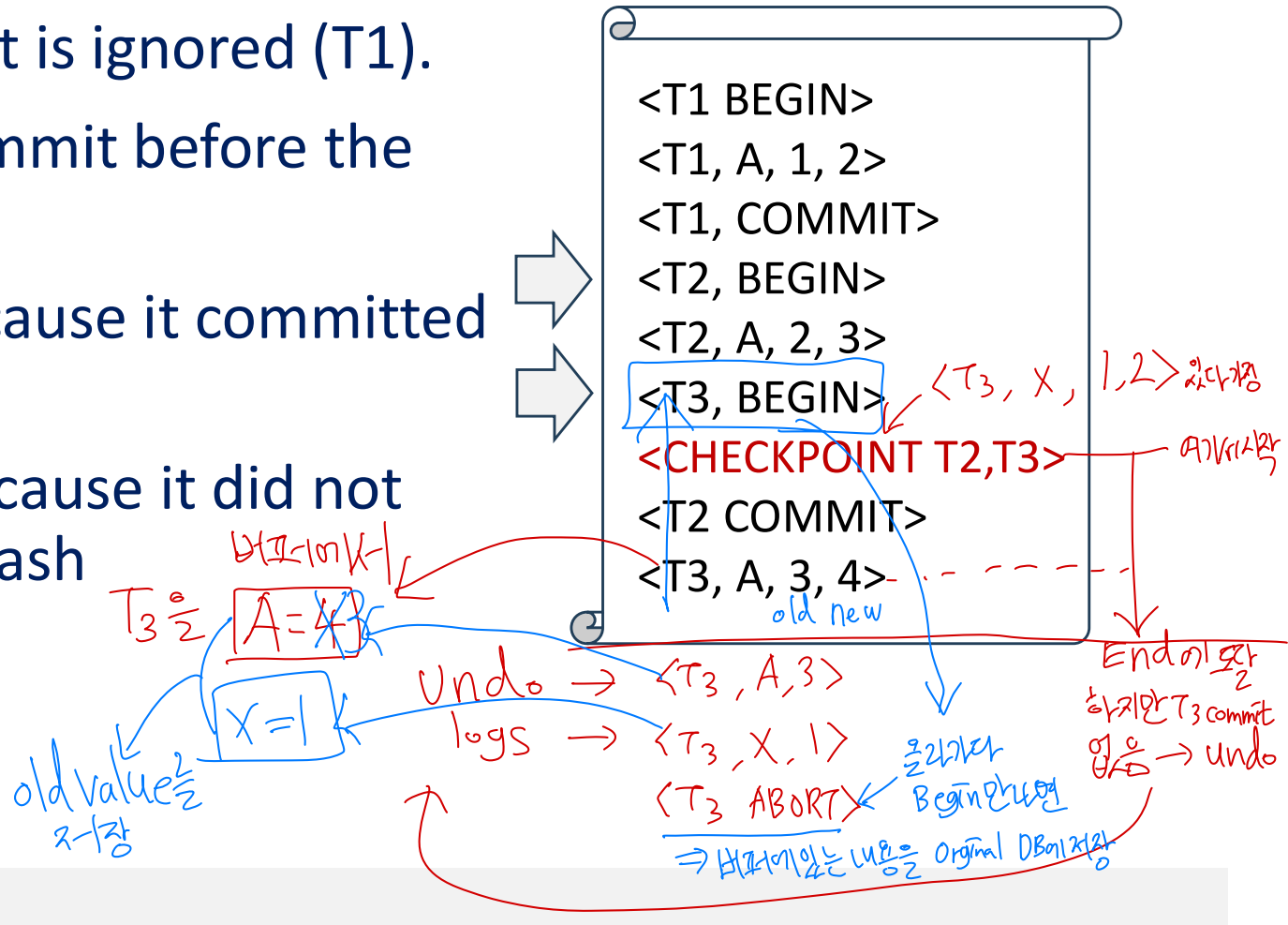
Checkpoints (Cont.)

- Use **Checkpoints for Recovery**
- Use the <CHECKPOINT> record as the starting point for analyzing the WAL.
- Any transaction that committed before the checkpoint is ignored (T1).
- T2 and T3 did not commit before the last checkpoint



Checkpoints (Cont.)

- Use **Checkpoints for Recovery**
- Use the <CHECKPOINT> record as the starting point for analyzing the WAL.
- Any transaction that committed before the checkpoint is ignored (T1).
- T2 and T3 did not commit before the last checkpoint
- Need to REDO T2 because it committed after checkpoint.
- Need to UNDO T3 because it did not commit before the crash



Recovery Algorithm with Checkpointing

- **Logging** (during normal operation):
  - $<T_i$ **start**$>$ at transaction start
  - $<T_i, X_j, V_{old}, V_{new}>$ for each update, and
  - $<T_i$ **commit**$>$ at transaction end

- **Transaction rollback (during normal operation)**
  - T_i : the transaction to be rolled back
  - Scan log backwards, and for each $<T_i, X_j, V_{old}, V_{new}>$
    - Perform the undo, i.e., $V_{old} \rightarrow X_j$
    - Write a log record $<T_i, X_j, V_{old}>$
      - such log records are called **compensation log records**
  - Once the record $<T_i$ **start**$>$ is found stop the scan and write the log record $<T_i$ **abort**$>$

Recovery Algorithm with Checkpointing (Cont.)

- **Recovery from failure**: Two phases
  - **Redo phase**: replay updates of all transactions, whether they committed, aborted, or are incomplete 체크포인트에서 내려오는 단계(↓)
  - **Undo phase:** undo all incomplete transactions 다시 올라가는 단계(↑)

- **Redo phase**:
  1. Find last <**checkpoint** *L*> record, and set undo-list to *L*.
  2. Scan forward from above <**checkpoint** *L*> record
     1. For each log record $<T_i, X_j, V_1, V_2>$ or $<T_i, X_j, V_2>$, redo it by writing V_2 to X_j
     2. For each log record $<T_i$ **start**>, add T_i to undo-list
     3. For each log record $<T_i$ **commit**> *or* $<T_i$ **abort**>, remove T_i from undo-list

Recovery Algorithm (Cont.)

- **Undo phase:**

  Scan log backwards from end

  1. For each log record $<T_i, X_j, V_1, V_2>$ where T_i is in undo-list:
     1. perform undo by writing V_1 to X_j.
     2. write a log record $<T_i, X_j, V_1>$
  2. For each log record $<T_i$ **start**$>$ where T_i is in undo-list,
     1. Write a log record $<T_i$ **abort**$>$
     2. Remove T_i from undo-list
  3. Stop when undo-list is empty
     - i.e., $<T_i$ **start**$>$ has been found for every transaction in undo-list

- After undo phase completes, normal transaction processing can commence

Example of Recovery