Transactions: Recovery & Log-Based Algorithms

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. DBMS Data Access
2. Log-Based Recovery
3. Checkpointing The Log
4. Log-Based Recovery "Drill Down"
5. Drilling Down Into "Steal + No-Force"

Some material from lectures created by Professor Pavlo (CMU) (used with permission)

Context

- Last lecture: began discussion of transactional recovery & persistence
  - Real-life durability provided by a (lower) level in the storage hierarchy
  - Challenge: how to coordinate state changes across multiple levels of storage
- We examined a shadow-page implementation strategy
- Today: log-based recovery strategies

Shadow Paging: Advantages & Disadvantages

Advantages: these are the "no-steal + force" advantages

- Undo is easy: remove the shadow pages, master copy and database root are fine in their current state
- Redo is even easier: this operation is never needed ☺

Disadvantages

- Copying the entire page table is expensive!
  - But note: a more clever implementation can refine to only copy paths that lead to pages that have been modified
- Commit overhead is high:
  - DBMS must "flush" every updated page, the page table itself, and the root
- Disk gets fragmented, must be defragmented to reorganize the DBMS's "live data" on disk

> Observation: conceptually elegant, but databases must have good performance

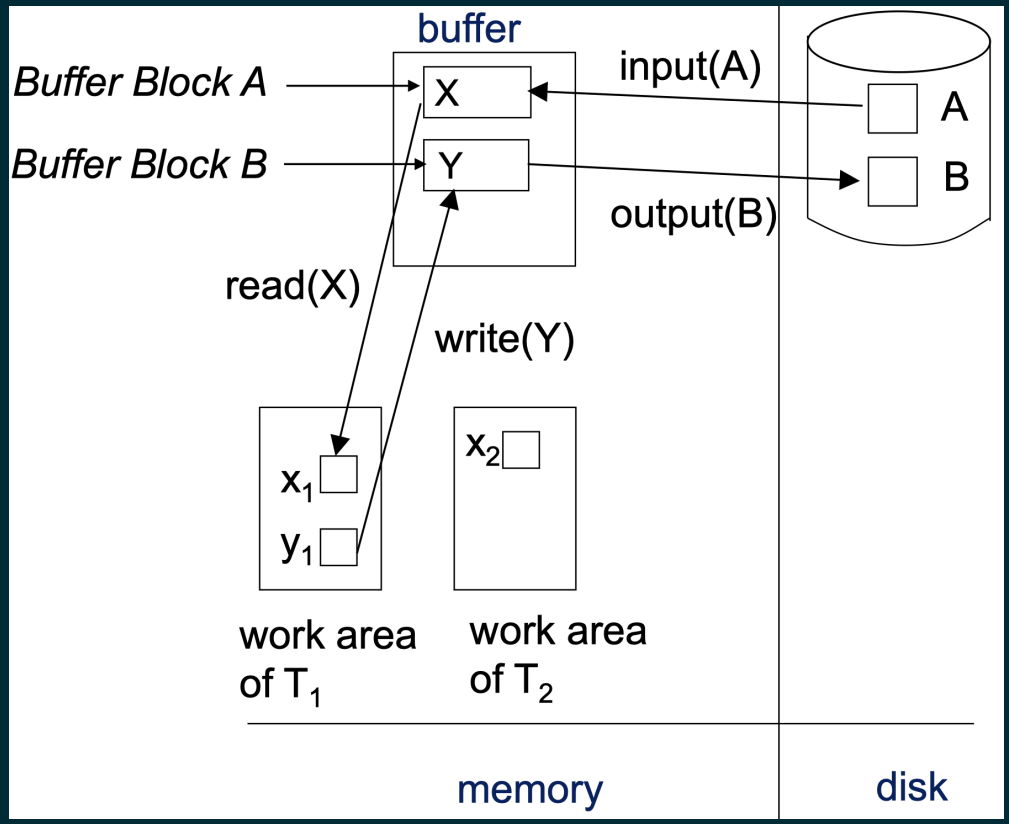
Segue To Log-Based Approaches

- If DBMS performs random writes, the OS must flush non-contiguous pages to disk
- For most storage media, this is a slow process
- Relatively speaking, sequential writes are much faster than random writes
- Key idea: devise a recovery scheme that allows for sequential writes

DBMS Data Access

- We've touched on these points before, but will do it more thoroughly now
- Transactions
  - Access data by "reading in" from disk into main memory
  - After modifying data, transactions must "write out" from main-memory to disk
- I/O is done in "page block" units
  - Assumption: size of a single datum fits into a single block
- Terminology:
  - Blocks residing on disk are called physical blocks
  - Corresponding blocks residing in main-memory are called buffer blocks
- Each Tx_i has a private "work-area" which maintains its local copy of the data that it reads and writes

Transactional Data Access



- Tx must transfer data between DBMS buffer blocks and its private work-area
- So: read and write operations are performed by the tx with respect to DBMS buffer blocks
  - The DBMS moves data to/from disk as necessary

Forcing Data To Disk

- Q: when does the DBMS write a buffer block to disk?
- Intuitively, you'd think that disk writes should be done immediately
  - After all: we want to keep disk state as up-to-date as possible
  - But: a buffer page can contain many data items, from multiple txs
  - Other txs may still be accessing those data items!
  - Performance will suffer if we require the DBMS to do a disk write every time that a tx writes to a disk buffer
- A: disk writes are performed at the DBMS's discretion
  - Example: when it runs out of buffer space
  - Example: transaction algorithms require that the disk state be updated now
- Q: what happens if the transaction commits a write (to the buffer block) and the system crashes before the DBMS disk writes?

Log-Based Recovery & Atomicity (I)

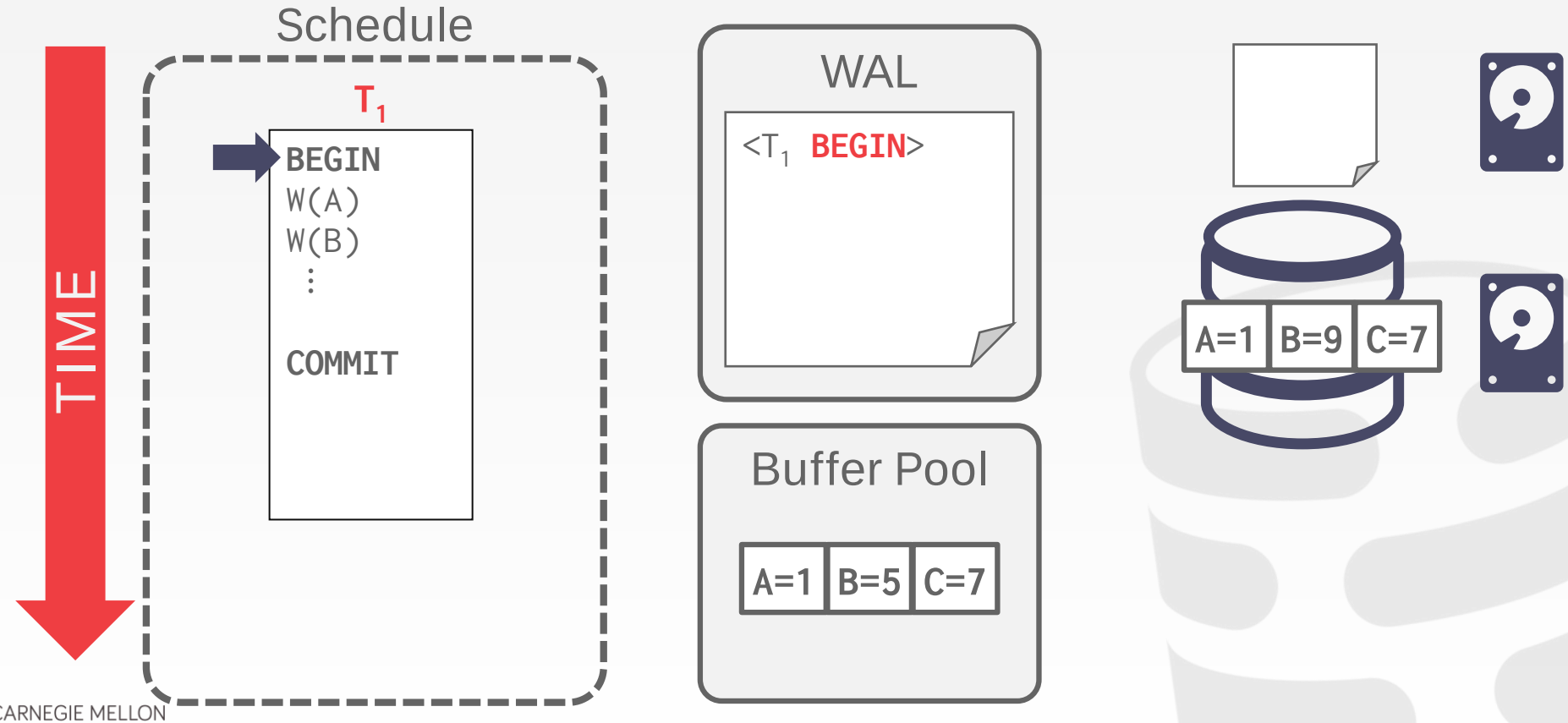
- To ensure atomicity despite failures, the DBMS writes records to a log (maintained on stable storage)
- The log records describe a transaction's operations: what data was modified and how that data was modified
- After a failure, the DBMS "replays the log" to:
  - Redo committed transactions
  - Undo uncommitted transactions

- This is the key insight of log-based recovery algorithms
- A log record describing a change must be written to stable storage before the "real database" change is made
- Hence: "Write-Ahead Log" (or WAL) property

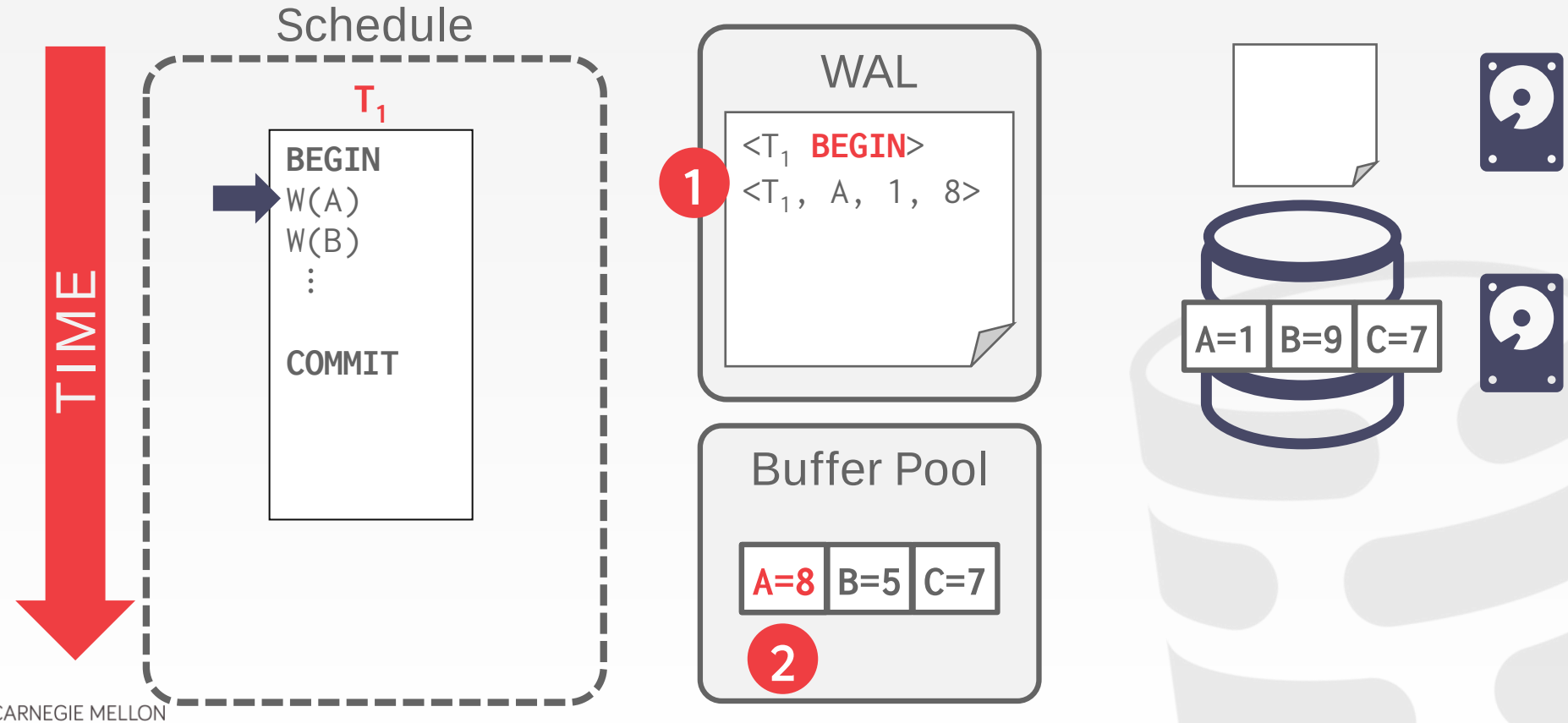
Log-Based Recovery & Atomicity (II)

- DBMS records each tx's execution by inserting a begin record into the log
- When a tx completes "normally", DBMS inserts a commit record into the log
- Between these "marker" records, the DBMS inserts "change" records that specify a tx's changes on a "per-datum" basis
  - Transaction id
  - Datum id
  - Before image (or BFIM)
  - After image (or AFIM)
- The BFIM is used to perform undo operations
- The AFIM is used to perform redo operations
- As you'd expect, the DBMS will insert a abort record to indicate that the specified tx was aborted

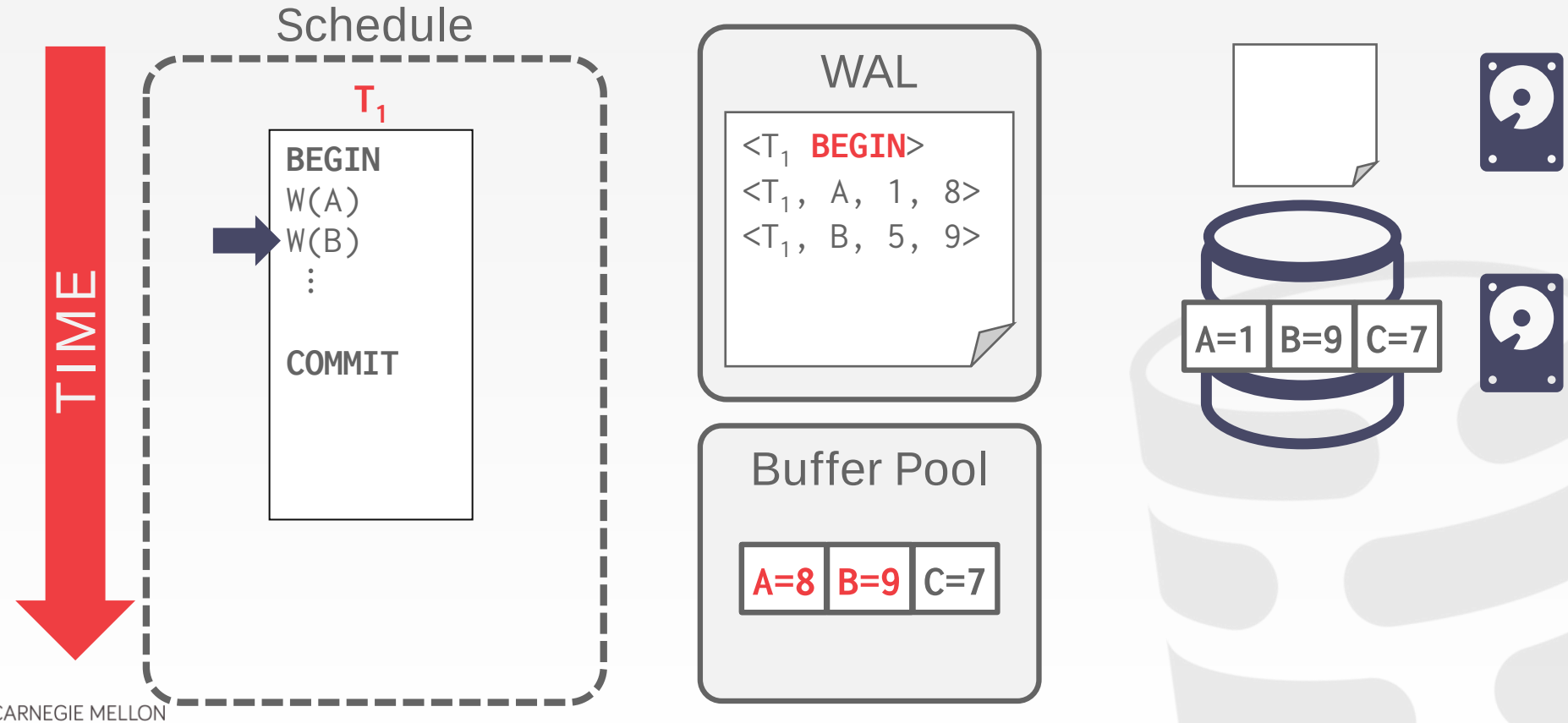
WAL – EXAMPLE


Schedule
TIME
T1
BEGIN
W(A)
W(B)
COMMIT
WAL
<T1 BEGIN>
Buffer Pool
A=1
B=5
C=7
A=1
B=9
C=7

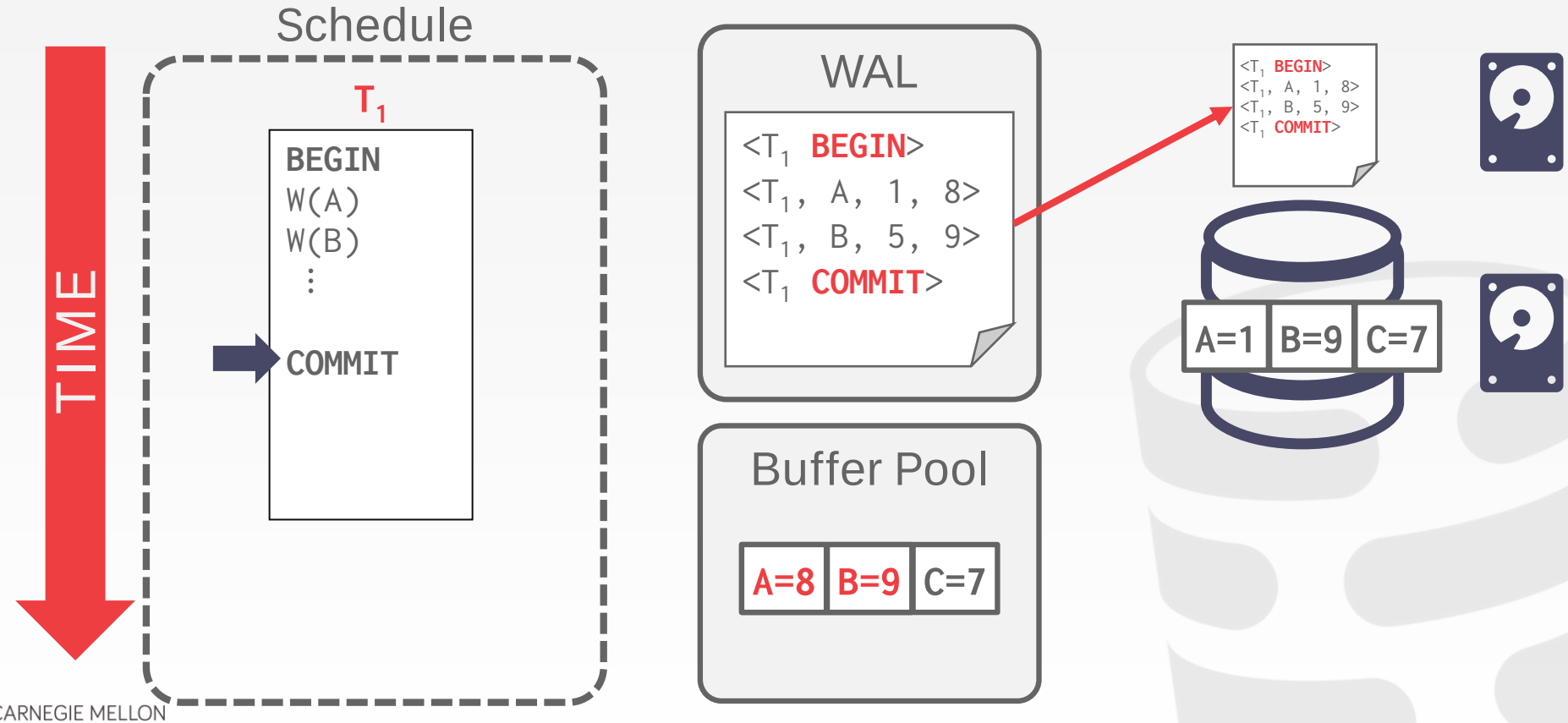
WAL – EXAMPLE


Schedule
TIME
T1
BEGIN
W(A)
W(B)
COMMIT
WAL
1
<T1 BEGIN>
<T1, A, 1, 8>
Buffer Pool
A=8
B=5
C=7
2
A=1
B=9
C=7

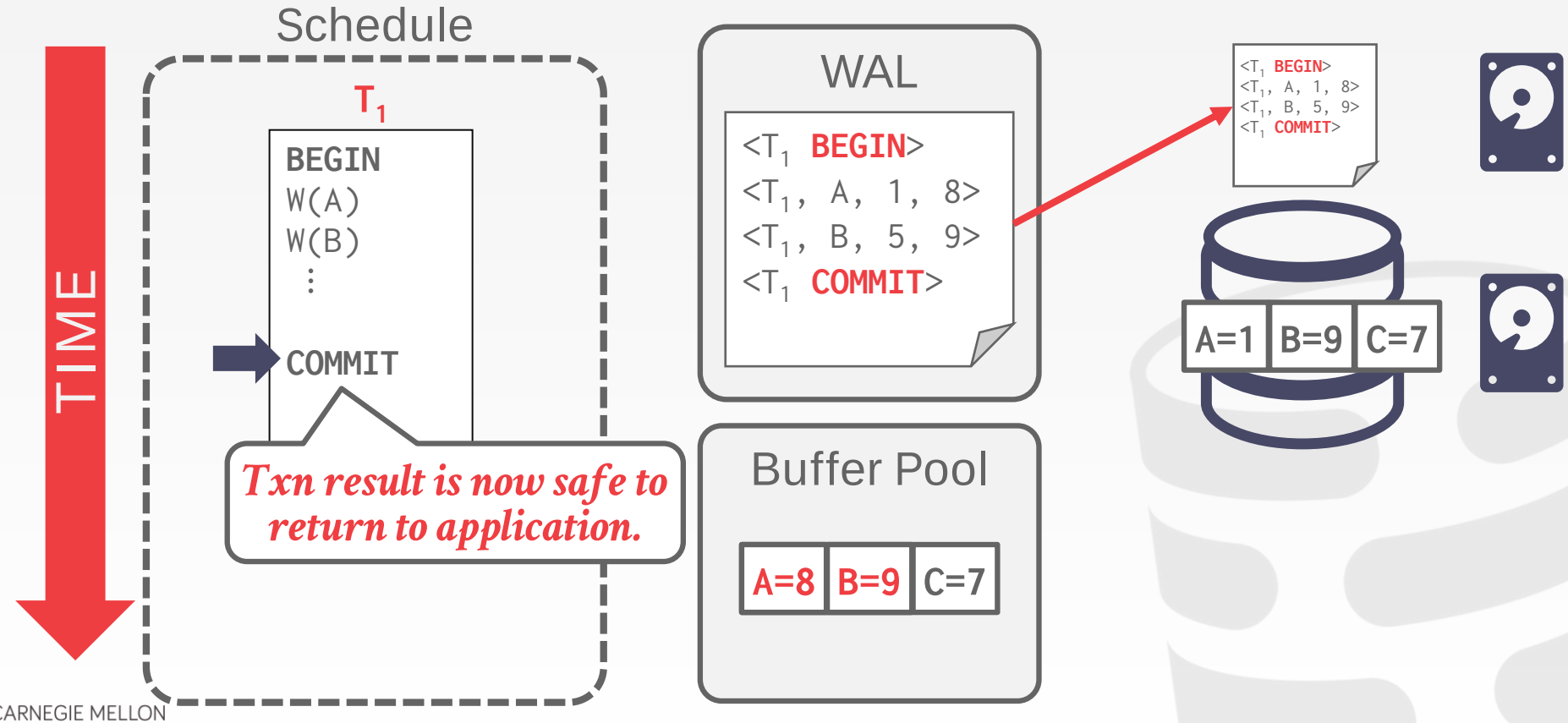
WAL – EXAMPLE


Schedule
TIME
T1
BEGIN
W(A)
W(B)
COMMIT
WAL
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
Buffer Pool
A=8
B=9
C=7
A=1
B=9
C=7

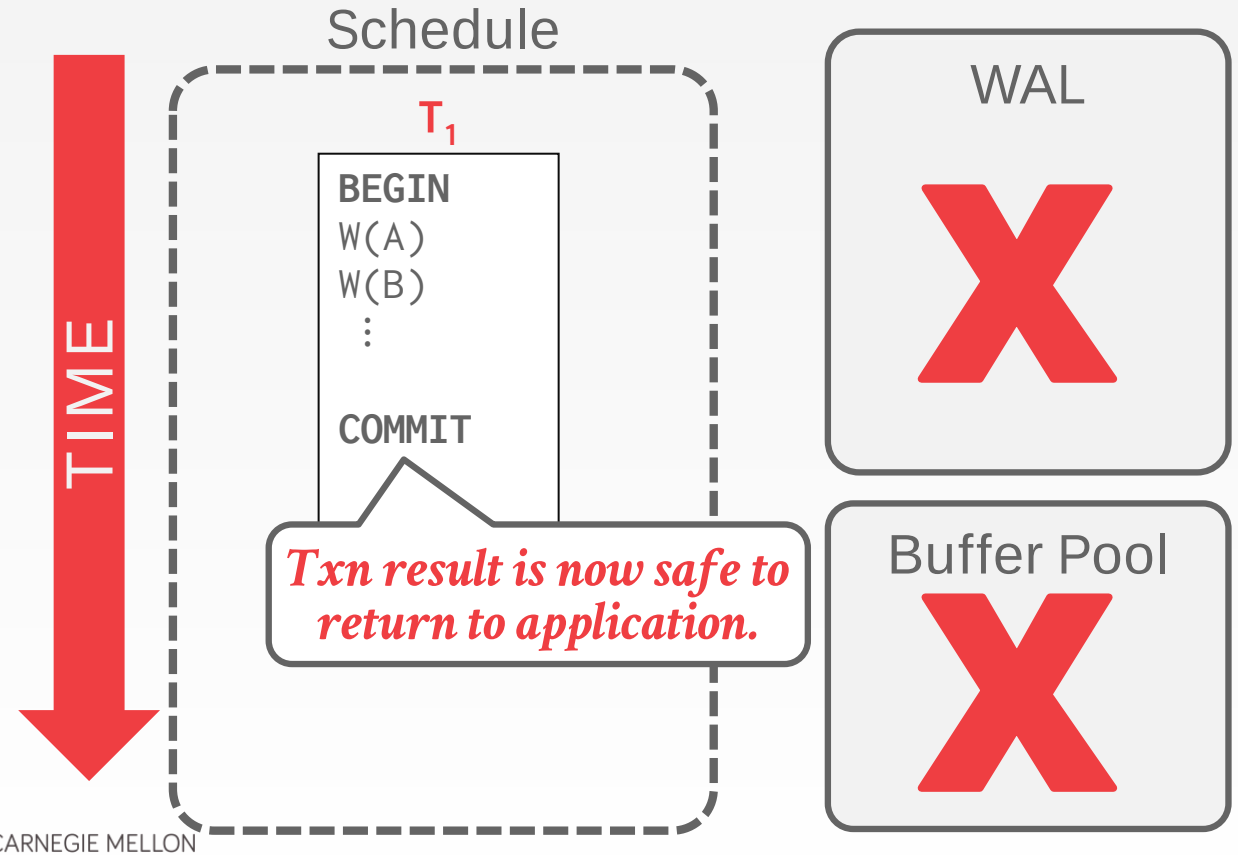
WAL – EXAMPLE

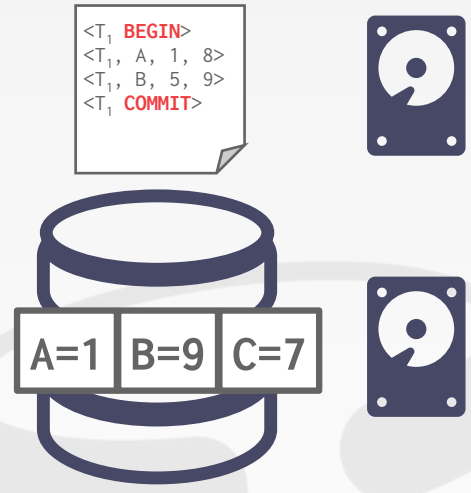

Schedule
TIME
T1
BEGIN
W(A)
W(B)
COMMIT
WAL
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
<T1 COMMIT>
Buffer Pool
A=8
B=9
C=7
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
<T1 COMMIT>
A=1
B=9
C=7

WAL – EXAMPLE

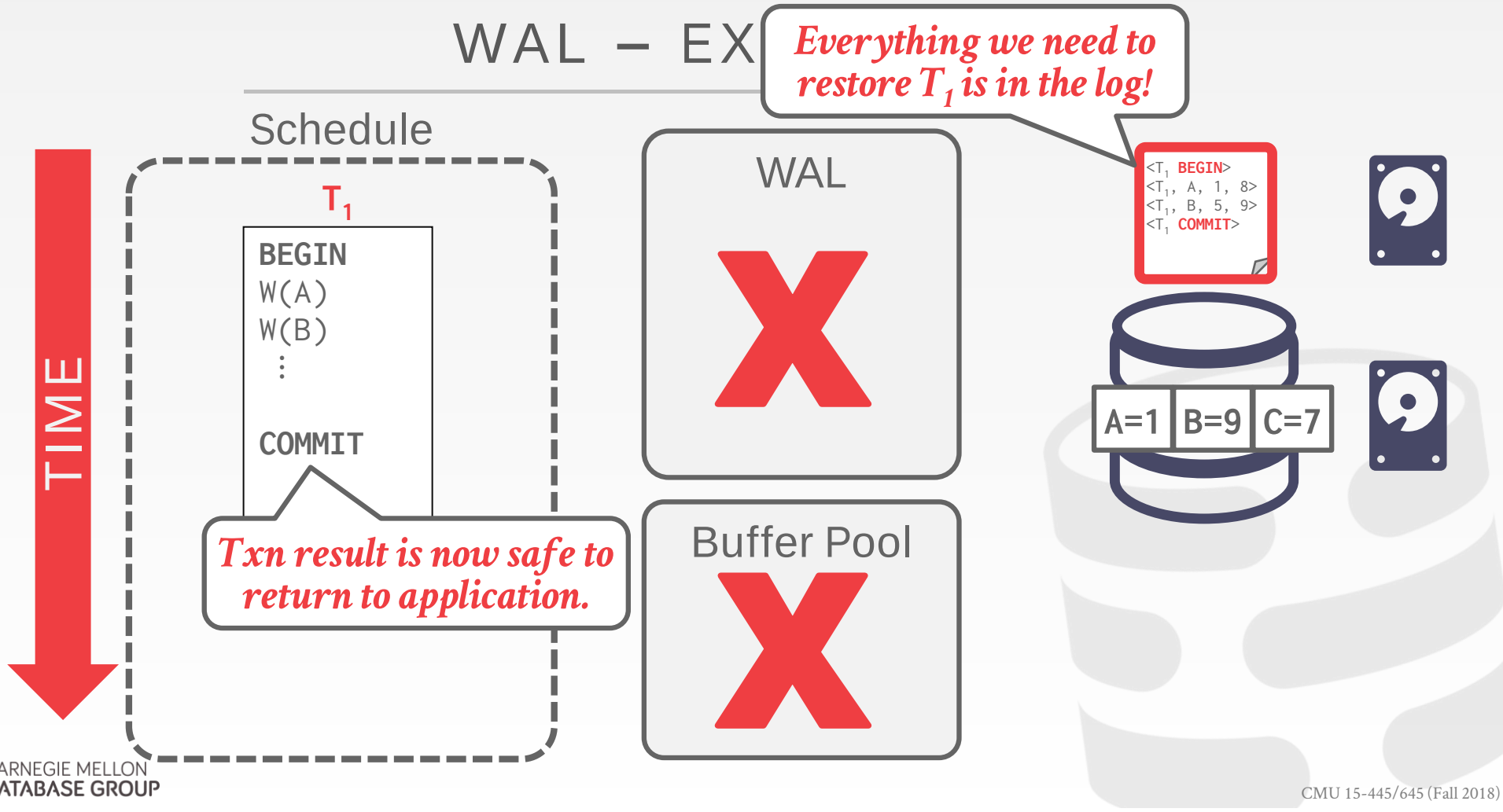

Schedule
TIME
T1
BEGIN
W(A)
W(B)
COMMIT
Txn result is now safe to return to application.
WAL
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
<T1 COMMIT>
Buffer Pool
A=8
B=9
C=7
<T1 BEGIN>
<T1, A, 1, 8>
<T1, B, 5, 9>
<T1 COMMIT>
A=1
B=9
C=7

WAL – EXAMPLE


Schedule
TIME
T1
BEGIN
W(A)
W(B)
COMMIT
Txn result is now safe to return to application.
WAL
Buffer Pool


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

WAL – EX


Everything we need to restore T_1 is in the log!
Schedule
T_1
BEGIN
W(A)
W(B)
⋮
COMMIT
Txn result is now safe to return to application.
TIME
WAL
Buffer Pool
<T_1 BEGIN>
<T_1, A, 1, 8>
<T_1, B, 5, 9>
<T_1 COMMIT>
A=1
B=9
C=7

Modifying The Database (I)

- Key point: a transaction has modified the database only if it performs a write to a buffer block or disk pages
- Modifications to the private "work-area" don't count!
- But when should the tx modify the database?
- You'd think that database modification should occur only after the tx has committed
  - Advantage: nothing has to be "undone" if the tx fails to commit ☺
  - We refer to this strategy as deferred database modification
- But: this strategy implies that main-memory usage is duplicated
  - Data are maintained in buffer blocks
  - Data are also maintained in txs's private work area (until the tx commits)

Modifying The Database (II)

- Databases tend to address the problems of the deferred database modification strategy by using a different strategy: immediate database modification
  - Meaning: tx changes are allowed to modify the database even before the tx has completed
- Implications with respect to a log-based recovery strategy:
  - DBMS must write a log record before it modifies the database
  - Also: the log record must be "forced to disk" immediately
- Because the log records are safely recorded, OK for the DBMS to
  - Write the corresponding buffer blocks to disk whenever it wants
    - Before or even after the tx commits
  - Also OK for the DBMS to write blocks to disk in a different order from the order in which txs committed the writes

Immediate DB Modification: Example

Log	Write	Output
<T_0 **start**>		
<T_0, A, 1000, 950>		
<T_0, B, 2000, 2050>		
	A = 950	
	B = 2050	
<T_0 **commit**>		
<T_1 **start**>		
<T_1, C, 700, 600>		
	C = 600	
		B_B , B_C
<T_1 **commit**>		
		B_A

B_C output before T_1 commits

B_A output after T_0 commits

Note: B_X denotes block containing X.

Concurrency Control and Recovery

- One advantage of log-based recovery is that it dovetails neatly with implementing concurrency
- All transactions share a single disk buffer and a single log
- Can intersperse records for different txs in the same log
- A buffer block may contain data that has been updated by multiple txs
- Q: how do we provide isolation if txs share the same disk buffer?
  - Scenario: Tx_1 updates A, then Tx_2 updates A and commits, then Tx_1 aborts
- A: DBMS ensures that once Tx_i has modified data, that data is no longer visible to other txs until Tx_i has committed or aborted
  - Use an "exclusive locks" scheme, hold the locks until tx ends
  - Also know as (strict) 2PL ☺

Log-Based Recovery

Log-Based Recovery & Tx Commit

- We can now identify a single "atomic" event at which we can say that a transaction has "committed" ...
- *"The DBMS commits a tx when its commit log record is written to stable storage"*
- Recall that all previous log records for that tx must have already been written to disk before the "commit" record is written
- If the system crashes before the commit record is written ...
  - No problem: the tx will be rolled back (undone) when the DBMS brings the system back up
- If the system crashes after the commit record is written ...
  - No problem: the tx will be redone and committed when the DBMS brings the system back up

Log-Based Recovery: Undo & Redo Operations

undo(Tx_i):

- Restores the value of all data items updated by Tx_i to their old values
  - The DBMS works backwards from the last log record for Tx_i
- Each time a datum X is restored to its BFIM V, DBMS also inserts a "special" log record $< Tx_i, X, V >$
- When Tx_i work has been completely "undone", DBMS inserts a "special" $< Tx_i, abort >$ into the log (we'll explain why this is useful later)

redo(Tx_i):

- Sets the value of all data updated by Tx_i to the new values
  - The DBMS works forwards from the first log record for Tx_i
- No need for the DBMS to insert "special" records during this operation
  - Even if subsequent crash, the current set of records suffice to perform an idempotent "redo"

Recovering From Failure (I)

- Previous slide defines the undo and redo operations in the context of a log-based implementation
- Now: what does the DBMS do when recovering after failure?
- Transaction Tx_i needs to be undone if the log:
  - Does contain the record $< Tx_i \ start >$
  - But contains neither a $< Tx_i \ commit >$ record nor a $< Tx_i \ abort >$ record
- Transaction Tx_i needs to be redone if the log:
  - Contains both a $< Tx_i \ start >$ record and either a $< Tx_i \ commit >$ record or a $< Tx_i \ abort >$ record
- That last "or clause" needs more explanation (next slide)

Recovering From Failure (II)

- Q: why does the DBMS need to perform a redo if the log contains an "abort" record for that tx?
- A: first observe that this algorithm doesn't break anything ☺
  - Recall that the "abort" record is only inserted into the log by the undo operation
  - The undo operation inserts special $< Tx_i, X, V >$ records that set X to its old value
  - Executing these operations during an redo will preserve the correct semantics of the "aborted" transaction
  - Yes, it's redundant, but this algorithm makes recovery code considerably "cleaner" (more later)

Immediate DB Modification Recovery Example

Below we show the log as it appears at three instances of time.

(a)	(b)	(c)
<T_0 start>	<T_0 start>	<T_0 start>
<T_0, A, 1000, 950>	<T_0, A, 1000, 950>	<T_0, A, 1000, 950>
<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>	<T_0, B, 2000, 2050>
	<T_0 commit>	<T_0 commit>
	<T_1 start>	<T_1 start>
	<T_1, C, 700, 600>	<T_1, C, 700, 600>
		<T_1 commit>

Recovery actions in each case above are:

(a) undo (T_0): B is restored to 2000 and A to 1000, and log records <T_0, B, 2000>, <T_0, A, 1000>, <T_0, **abort**> are written out

(b) redo (T_0) and undo (T_1): *A* and *B* are set to 950 and 2050 and C is restored to 700. Log records <T_1, C, 700>, <T_1, **abort**> are written out.

(c) redo (T_0) and redo (T_1): A and B are set to 950 and 2050 respectively. Then *C* is set to 600

Checkpointing The Log

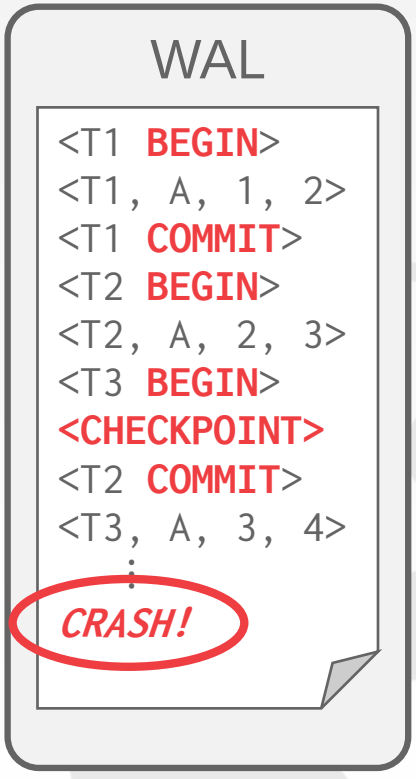
Checkpoints: Motivation

- The log-based "recovery algorithm" sketch that we've just discussed does work ☺
- But: "as is", will perform terribly ☹
- Note: the DBMS log will grow unboundedly as the system processes more and more txs
  - After a crash, the DBMS must replay the entire log
  - This will take a really long time ☹
- Key observation: once a tx has actually written its buffers to disk, the corresponding log records are obsolete
- Hence periodic checkpointing to streamline future recovery activity:
  1. DBMS outputs (and flushes) any main-memory log records to stable storage
  2. DBMS outputs (and flushes) any modified buffer blocks to stable storage
  3. Write a log record $< \textit{checkpoint } L >$ to stable storage
     - L is a list of "all txs active at the time of checkpoint"
- Note: potential performance hit since must stop all updates during the checkpointing process

Checkpointing: Recovery Algorithm

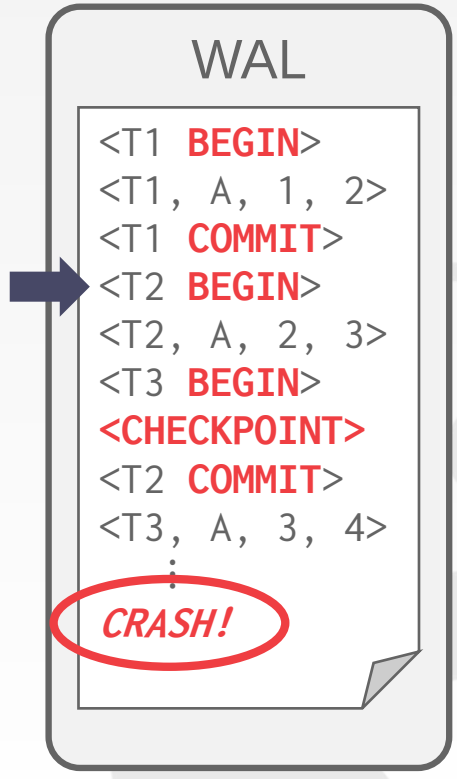
- With checkpointing, system modifies recovery algorithm
  - Determine the most recent tx that started before latest checkpoint
  - Determine the set of txs that started after this tx
- DBMS scan backwards from end of log to find the most recent $< \textit{checkpoint } L >$ record
- Only txs that are in L or started after the checkpoint need to be redone or undone
  - We know that txs that committed or aborted before the checkpoint already have all their updates propagated to stable storage
- Note: undo operations require that we scan backwards until a record $< Tx_i \textit{ start} >$ is found for every $Tx_i \in L$
- Safe to discard all log records prior to the earliest $< Tx_i \textit{ start} >$ record

CHECKPOINTS



CHECKPOINTS

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



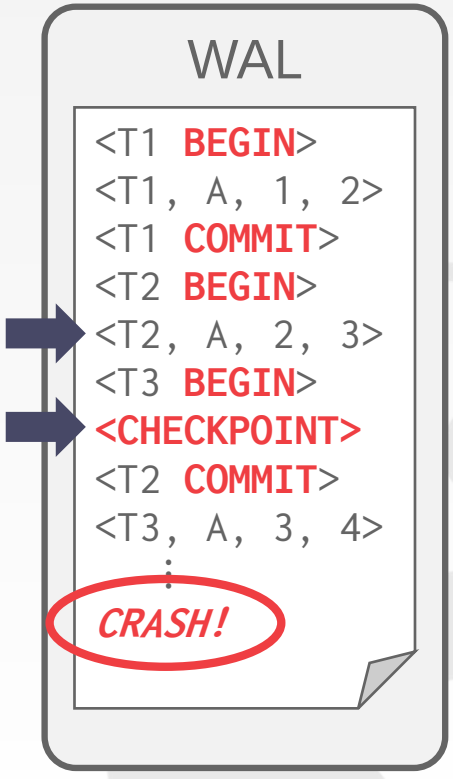
CHECKPOINTS

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

T_2 + T_3 did not commit before the last checkpoint.

→ Need to redo T_2 because it committed after checkpoint.

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



Checkpointing: Issues

- The advantages of checkpointing are obvious ☺
- But note the following issues ...
- System must suspend all tx processing while it takes a checkpoint
  - Otherwise: you get an inconsistent checkpoint ☹
- What is optimal checkpoint frequency?
- Too frequent? ⇒ performance degrades
  - Too much overhead from flushing buffers
- Infrequent? ⇒ performance degrades
  - Checkpoint will take a long time, and subsequent recovery process will be too long
- Also: scanning the log to find uncommitted txns can take a long time

Log-Based Recovery “Drill Down”

Recovery Algorithm: Normal Operations

- Logging
  - $< Tx_i \ start >$ when tx begins
  - Each update: $< Tx_i, X_j, V_1, V_2 >$
  - $< Tx_i \ commit >$ when tx ends
- Tx_i rollback
  - Scan log backwards from the end, and for each log record of Tx_i of the form $< Tx_i, X_j, V_1, V_2 >$ ...
    - Perform an undo by setting value of X_i to V_1
    - Write a compensation record of the form $< Tx_i, X_j, V_1 >$
  - Stop the scan when you see the record $< Tx_i \ start >$
    - Write a $< Tx_i \ abort >$ log record

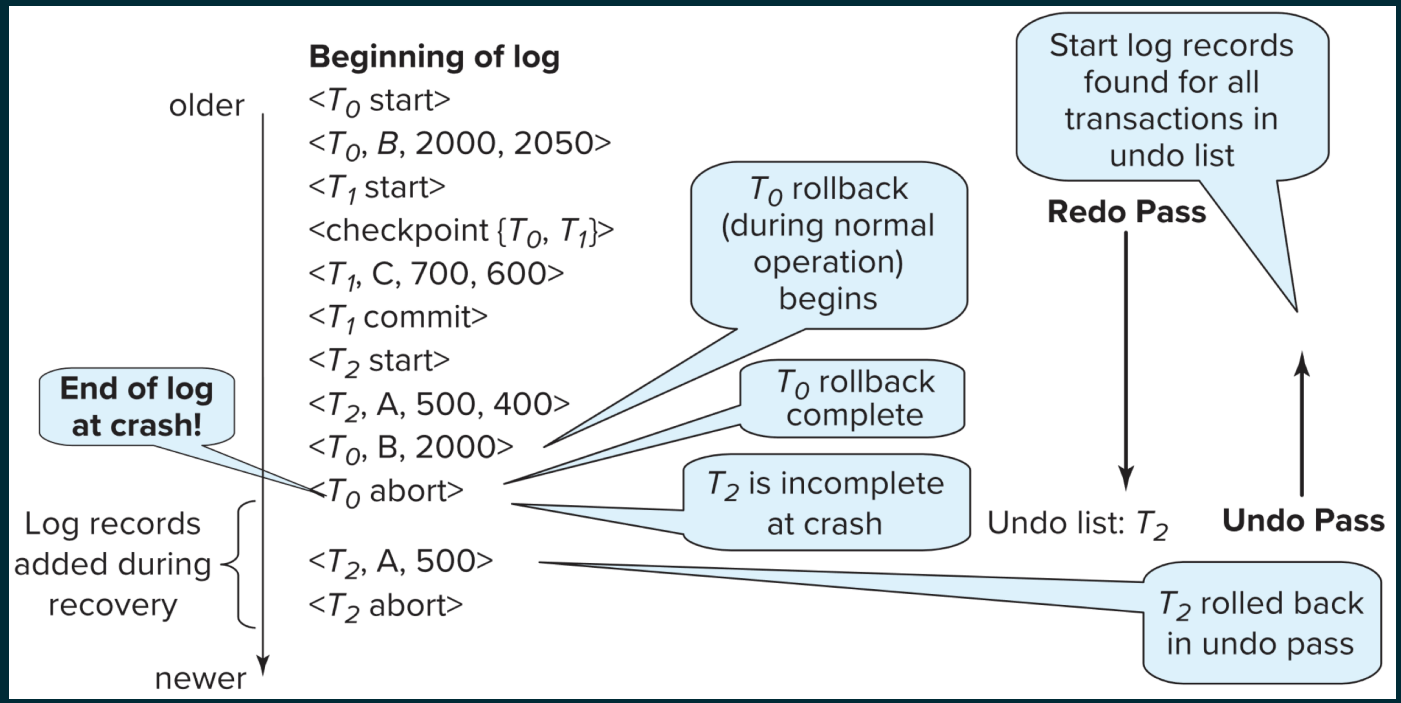
Recovery Algorithm: Recovery Process (I)

- Log-based recovery consists of two phases
  - Redo phase: replay updates of all transactions, whether they committed, aborted, or are incomplete
  - Undo phase: undo all incomplete transactions
- Redo phase
  1. Find last $<$ *checkpoint L* $>$ record, and set ‘‘`undo-list`’’ to L
  2. Scan forward from the $<$ *checkpoint L* $>$ record
     - 2.1 Whenever either a record $< Tx_i, X_j, V_1, V_2 >$ or $< Tx_i, X_j, V_2 >$ record is found, “redo it” by writing setting X_j to V_2
     - 2.2 Whenever a $< Tx_i$ *start* $>$ record is found, add Tx_i to `undo-list`
     - 2.3 Whenever either a $< Tx_i$ *commit* $>$ record or $< Tx_i$ *abort* $>$ record is found, remove Tx_i from `undo-list`
- At the end of the redo phase, the `undo-list` contains all incomplete txs
  - Meaning: txs that neither committed nor completed “normal” rollback before system crash

Recovery Algorithm: Recovery Process (II)

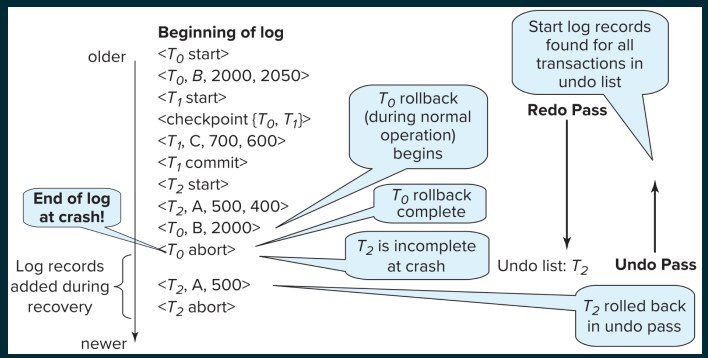
- Undo phase: scan log backwards from end
  1. Whenever a $< Tx_i, X_j, V_1, V_2 >$ record is found for a $Tx_i \in$ `undo-list`, perform same actions as for "normal tx rollback"
     1.1 Perform undo by setting X_j to V_1
     1.2 Write a log record $< Tx_i, X_j, V_1 >$
  2. Whenever a log record $< Tx_i \ start >$ is found for a $Tx_i \in$ `undo-list` ...
     2.1 Write a log record $< Tx_i \ abort >$
     2.2 Remove Tx_i from `undo-list`
  3. Stop when `undo-list` is empty
     - Meaning: $< Tx_i \ start >$ has been found for every transaction in undo-list
- After undo phase completes, resume "normal" tx processing

Log-Based Recovery: An Example (I)



- At the time that the system crashed
  - T_0 had already aborted
  - T_1 had already committed
  - List specified in checkpoint record contains T_0, T_1 (as "active txs")
    - `undo-list` initialized to T_0, T_1

Log-Based Recovery: An Example (II)



Redo phase

1. Scan forward, beginning from the checkpoint record
2. Remove T_1 from the `undo-list` when scan finds T_1 commit record
3. Add T_2 to the `undo-list` when scan finds T_2 start record
4. Remove T_0 from the `undo-list` when scan finds T_0 abort record

Undo phase: now, only T_2 is in the `undo-list`

1. Scan backwards from the end, finds a record in which T_2 updates A
2. Restore the old value of A
3. Insert a special “redo-only” record in the log
4. Scan continues, find T_2 start record, inserts an “abort” record for T_2

Drilling Down Into “Steal + No-Force”

Review: Atomicity, Durability, Buffer Management (I)

- The recovery manager is responsible for both tx atomicity and tx durability
- Atomicity: by undoing the actions of transactions that did not commit
- Durability: by redoing (all) the actions of committed transactions
- The recovery manager must interact with the buffer manager with respect to the following two policy decisions (we introduced this last lecture)
- Can the changes made to a buffer block pool by a tx be written to disk before the tx commits?
  - Yes, if a steal policy is being used
  - No, if a no-steal policy is being used
- Key point: the write-ahead logging algorithm that we've discussing works even with a "steal policy"
- And: "steal policy" (however counter-intuitive) supports txs that perform lots of updates
  - "No steal" policy causes the buffer to get filled with blocks that can't be written to disk, and txs can't proceed

Review: Atomicity, Durability, Buffer Management (II)

- When tx does commit, must we ensure that all its changes are immediately forced to disk?
  - Yes, if a force policy is used
  - No, if a no-force approach is used
- Key point: the write-ahead logging algorithm that we've discussing works even with a "no-force policy"
- And: "no-force" policy (however counter-intuitive) enables txs to commit faster than under the "force" policy
- Also: allows multiple updates to accumulate in the buffer before the system must write the buffer to disk

Policy Tradeoffs (I)

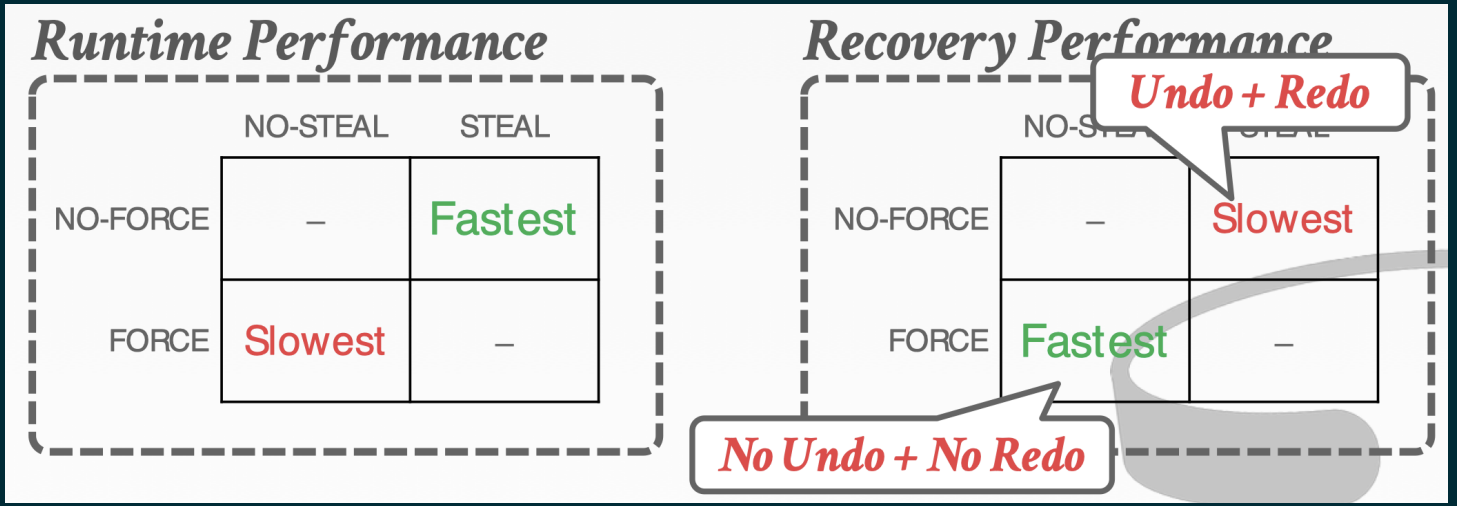
- "No-steal" policy advantages
  - DBMS implements undo as a "no-op" ☺
- "No-steal" policy disadvantages
  - All data modified by active txs must fit into the buffer pool (previous slide)
- "Force" policy advantages
  - DBMS implements redo as a "no-op" ☺
- "Force" policy disadvantages
  - Excessive I/O cost: a "hot page" will be written to disk many times

Policy Tradeoffs (II)

	No-Steal	Steal
Force	Trivial, but undesired	High I/O cost, but modified pages need not fit in the buffer pool
No-Force	Low I/O cost, but modified pages need to fit in the buffer pool	Low I/O cost, and modified pages need not fit in the buffer pool

- Recall: shadow-pages strategy is: NO-STEAL + FORCE
- In contrast, the log-based strategy is: "STEAL + NO-FORCE"

Policy Tradeoffs (III)



Based on previous discussion, the tradeoffs boil down to optimizing for "normal" runtime processing *versus* recovery processing

Today's Lecture: Wrapping it Up

DBMS Data Access

Log-Based Recovery

Checkpointing The Log

Log-Based Recovery "Drill Down"

Drilling Down Into "Steal + No-Force"

Readings

- Textbook discusses recovery in Chapter 19: focus is (almost exclusively) on log-based implementation
- You are responsible for the material through Chapter 19.6