

# COMP 421: Files & Databases

## Lecture 20: It's Log!

# Announcements

**Reminder:** Project 3 due 12/3, final exam 12/12

**Final:** Cumulative, focus on material after midterm

**Scheduling Question:** What to do on 12/3?

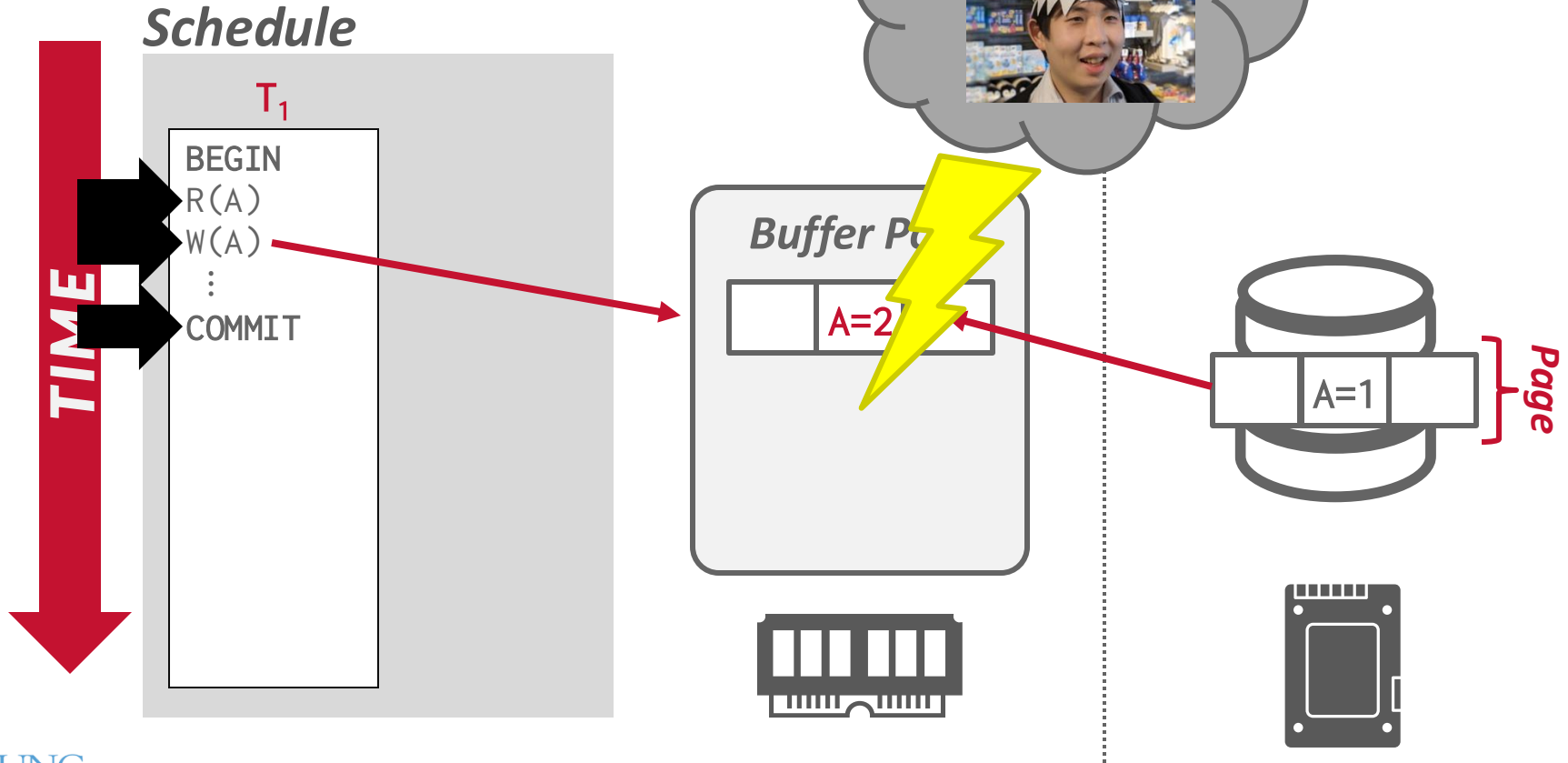
# Last Class

We discussed multi-version concurrency control (MVCC) and how it effects the design of the entire DBMS architecture.

A DBMS's concurrency control protocol gives it **Atomicity + Consistency + Isolation**.

We now need ensure **Atomicity + Durability**...

# MOTIVATI



# Crash Recovery

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

Recovery algorithms have two parts:

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

*Today*

# Today's Agenda

Buffer Pool Policies

Shadow Paging

Write-Ahead Log

Logging Schemes

Checkpoints

# Observation

The database'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 writes 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 told somebody that it committed.
- No partial changes are durable if the txn aborted.

# Undo vs. Redo

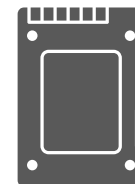
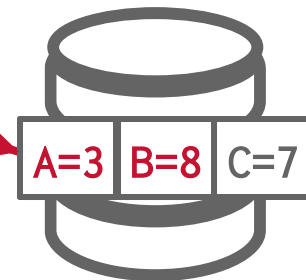
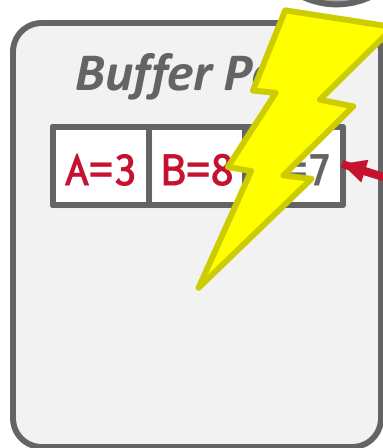
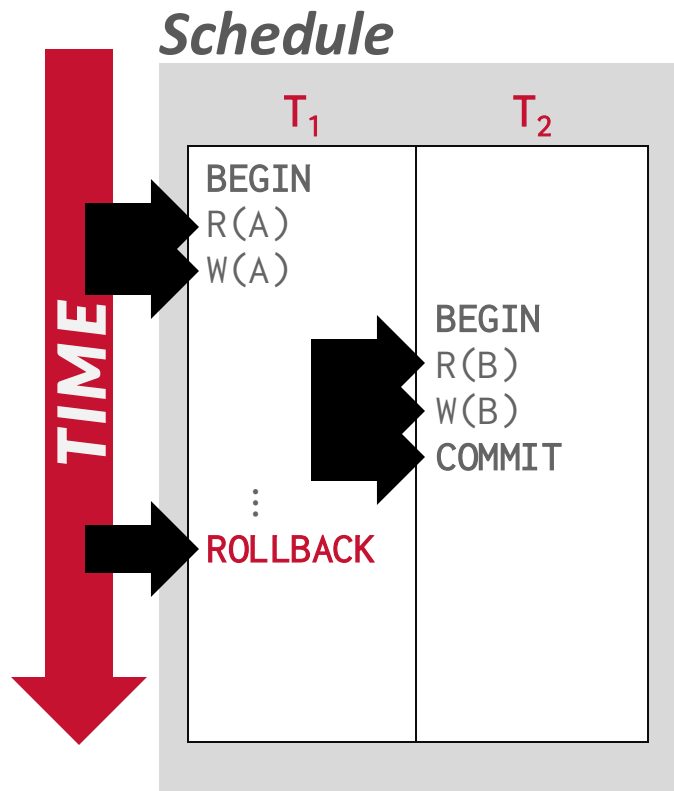
**Undo:** The process of removing the effects of an incomplete or aborted txn.

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

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



# Buffer Po



# Steal Policy

Whether the DBMS can evict a dirty object in the buffer pool modified by an uncommitted txn and overwrite the most recent committed version of that object in non-volatile storage.

**STEAL:** Eviction + overwriting is allowed.

**NO-STEAL:** Eviction + overwriting is not allowed.

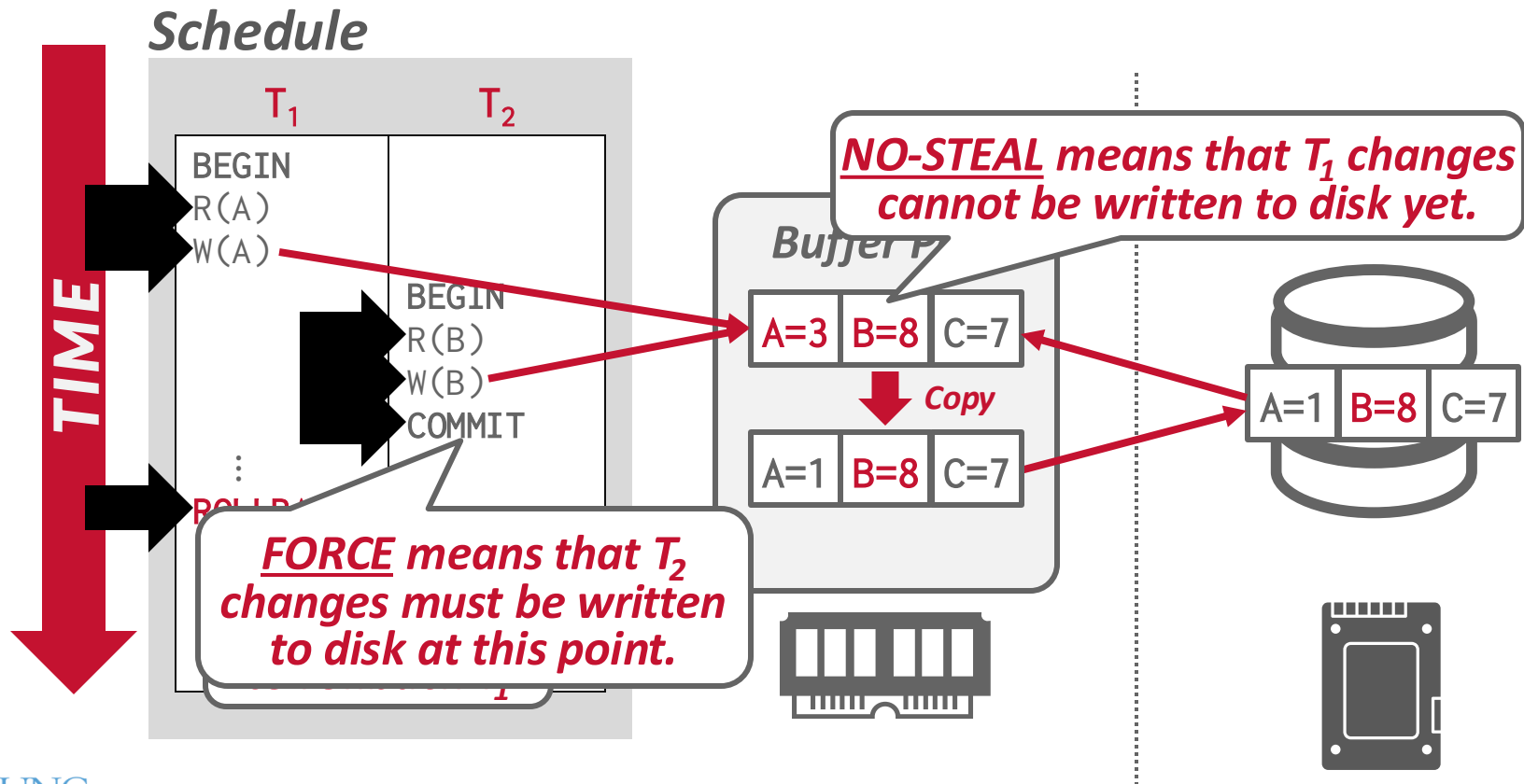
# Force Policy

Whether the DBMS requires that all updates made by a txn are written back to non-volatile storage before the txn can commit.

**FORCE:** Write-back is required.

**NO-FORCE:** Write-back is not required.

# No-Steal + Force



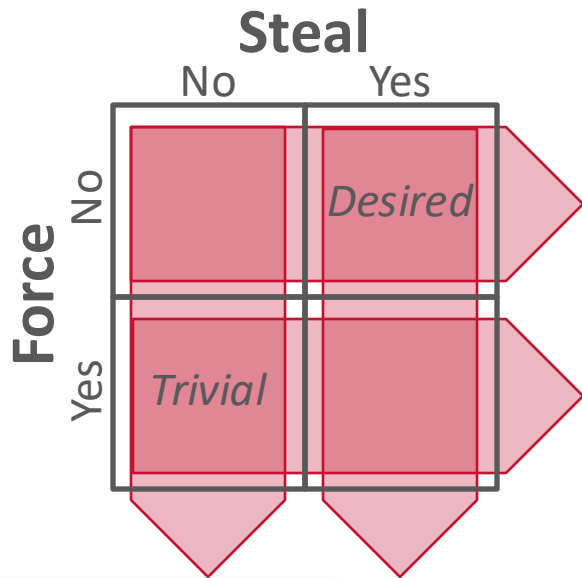
# No-Steal + Force

This approach is the easiest to implement:

- Never have to undo changes of an aborted txn because the changes were not written to disk.
- Never have to redo changes of a committed txn because all the changes are guaranteed to be written to disk at commit time (assuming atomic hardware writes).

Previous example cannot support **write sets** that exceed the amount of physical memory available.

# Buffer Pool + WAL



## No-Force

Concern: Crash before a page is flushed to disk. Durability?

## 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?

# Shadow Paging

Instead of copying the entire database, the DBMS copies pages on write to create two versions:

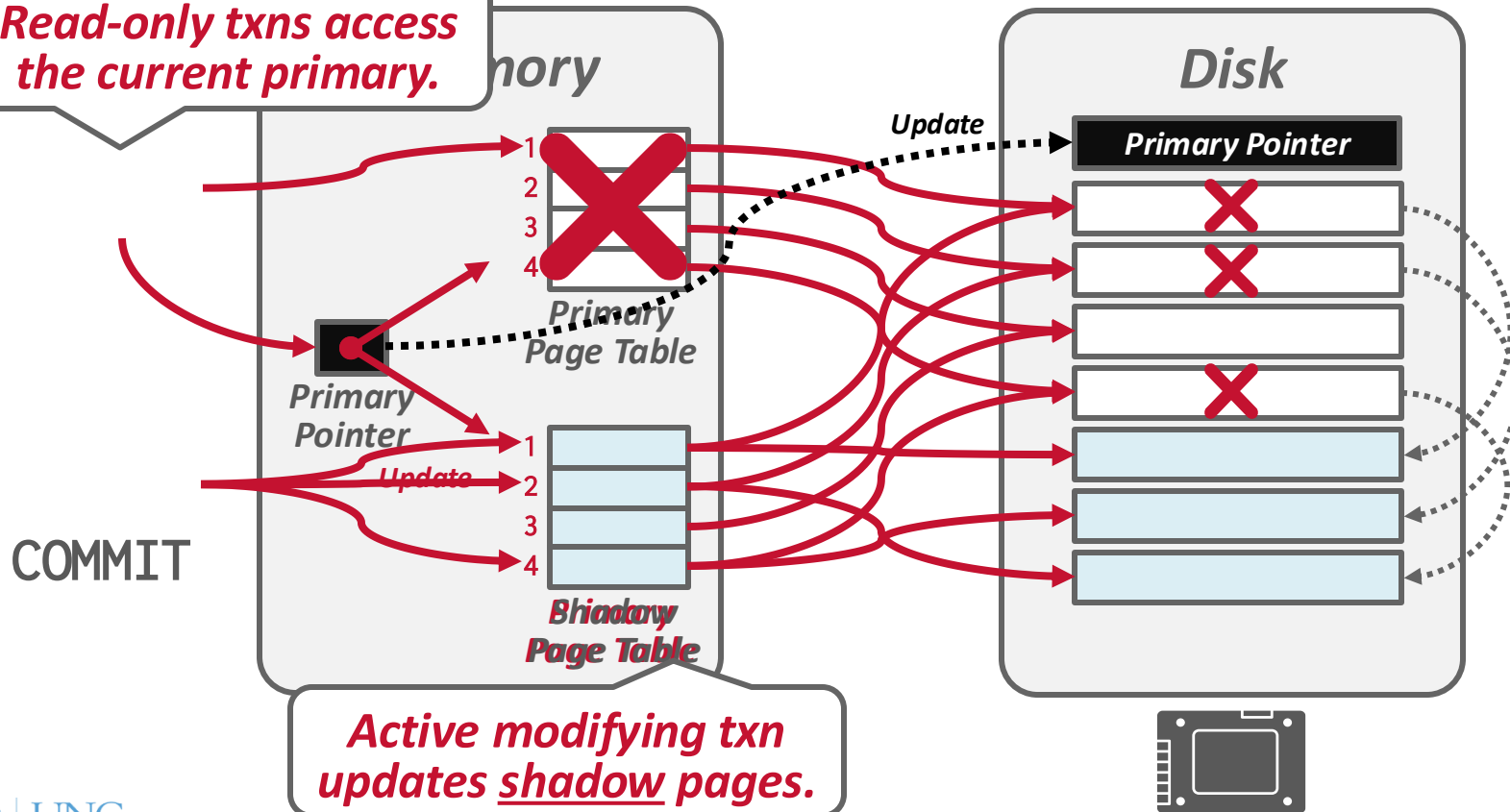
- **Primary**: Contains only changes from committed txns.
- **Shadow**: Temporary database with changes made from uncommitted txns.

To install updates when a txn commits, overwrite the root so it points to the shadow, thereby swapping the master and shadow.

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

# Shadow Paging – Example

*Read-only txns access the current primary.*





# Shadow Paging – Undo/Redo

Supporting rollbacks and recovery is easy with shadow paging.

**Undo:** Remove the shadow pages. Leave the primary and the DB root pointer alone.

**Redo:** Not needed at all.

# Shadow Paging – Disadvantages

Copying the entire page table is expensive:

- Use a page table structured like a B+tree (LMDB).
- No need to copy entire tree, only need to copy paths in the tree that lead to updated leaf nodes.

Commit overhead is high:

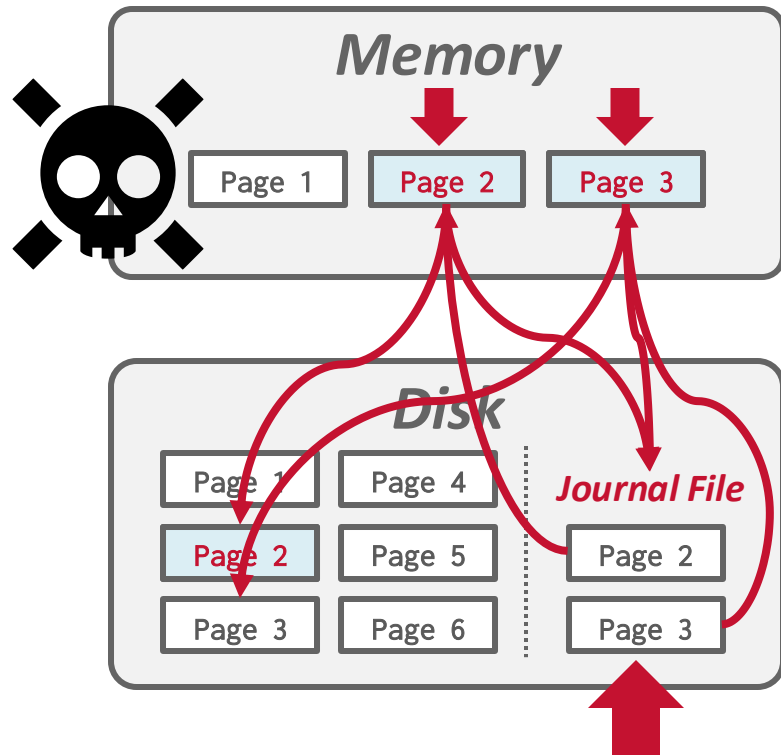
- Flush every updated page, page table, and root.
- Data gets fragmented (bad for sequential scans).
- Need garbage collection.
- Only supports one writer txn at a time or txns in a batch.

# SQLite (Pre-2010)

When a txn modifies a page, the DBMS copies the original page to a separate journal file before overwriting primary version.

→ Called rollback mode.

After restarting, if a journal file exists, then the DBMS restores it to undo changes from uncommitted txns.

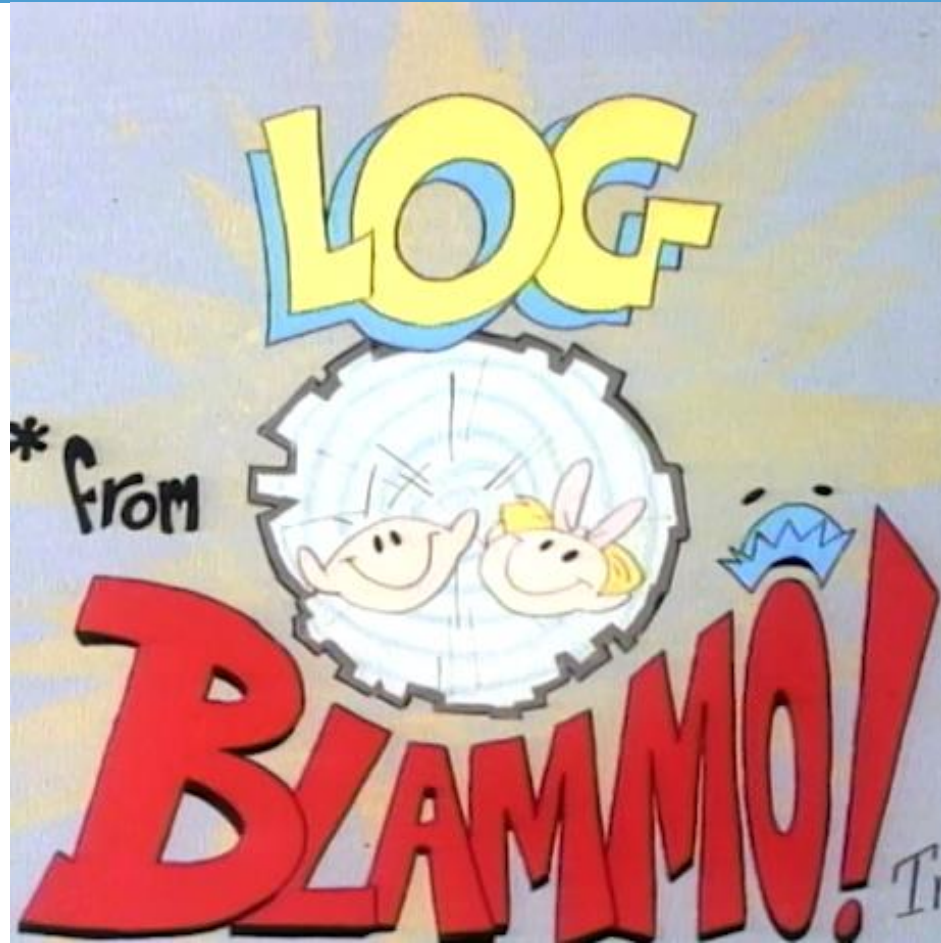


# Observation

Shadowing page requires the DBMS to perform writes to random non-contiguous pages on disk.

We need a way for the DBMS convert random writes into sequential writes...

# It's Log!



# Write-Ahead Log (WAL)

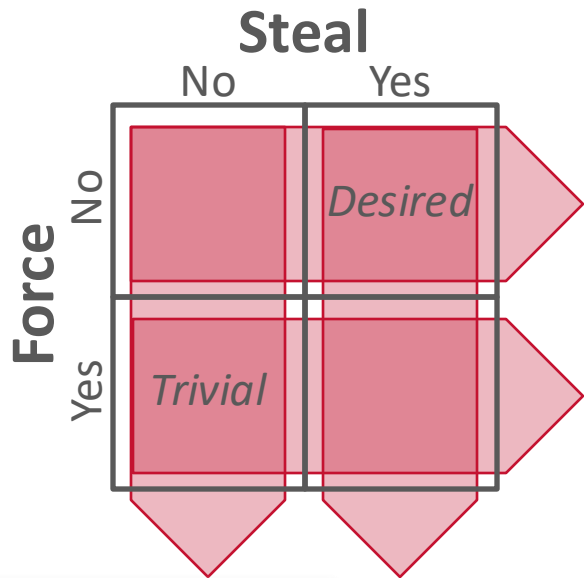
Maintain a log file separate from data files that contains the changes that txns make to database.

- Assume that the log is on stable storage.
- Log contains enough information to perform the necessary undo and redo actions to restore the database.

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



## 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 stages all a txn's log records in volatile storage (usually backed by buffer pool).

All log records pertaining to an updated page are written to non-volatile storage before the page itself is over-written in non-volatile storage.

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



# WAL Protocol

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

Append a record every time a txn changes an object:

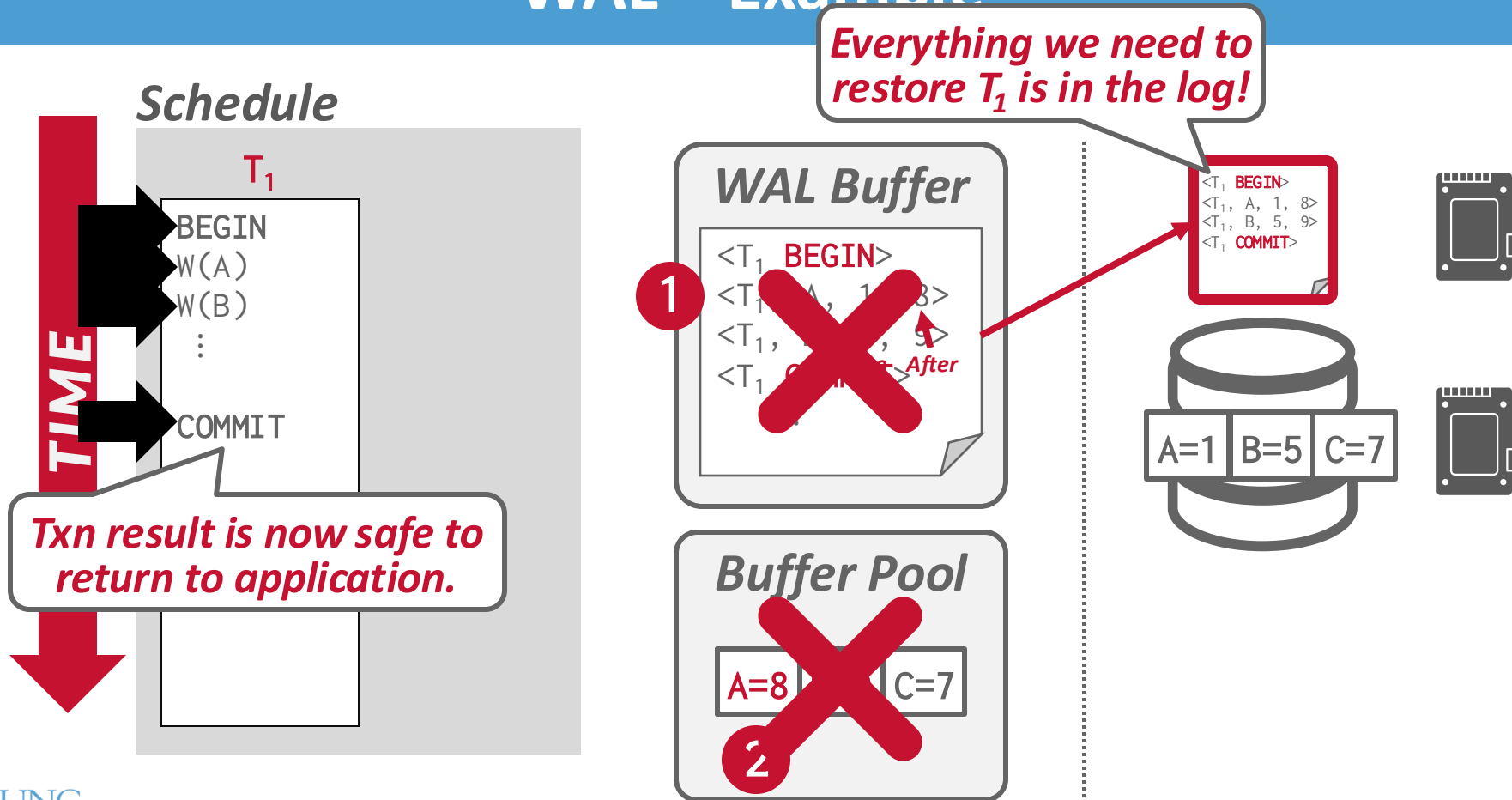
- Transaction Id
- Object Id
- Before Value (**UNDO**)
- After Value (**REDO**)

 ***Not necessary if  
using append-only  
MVCC***

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



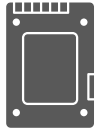
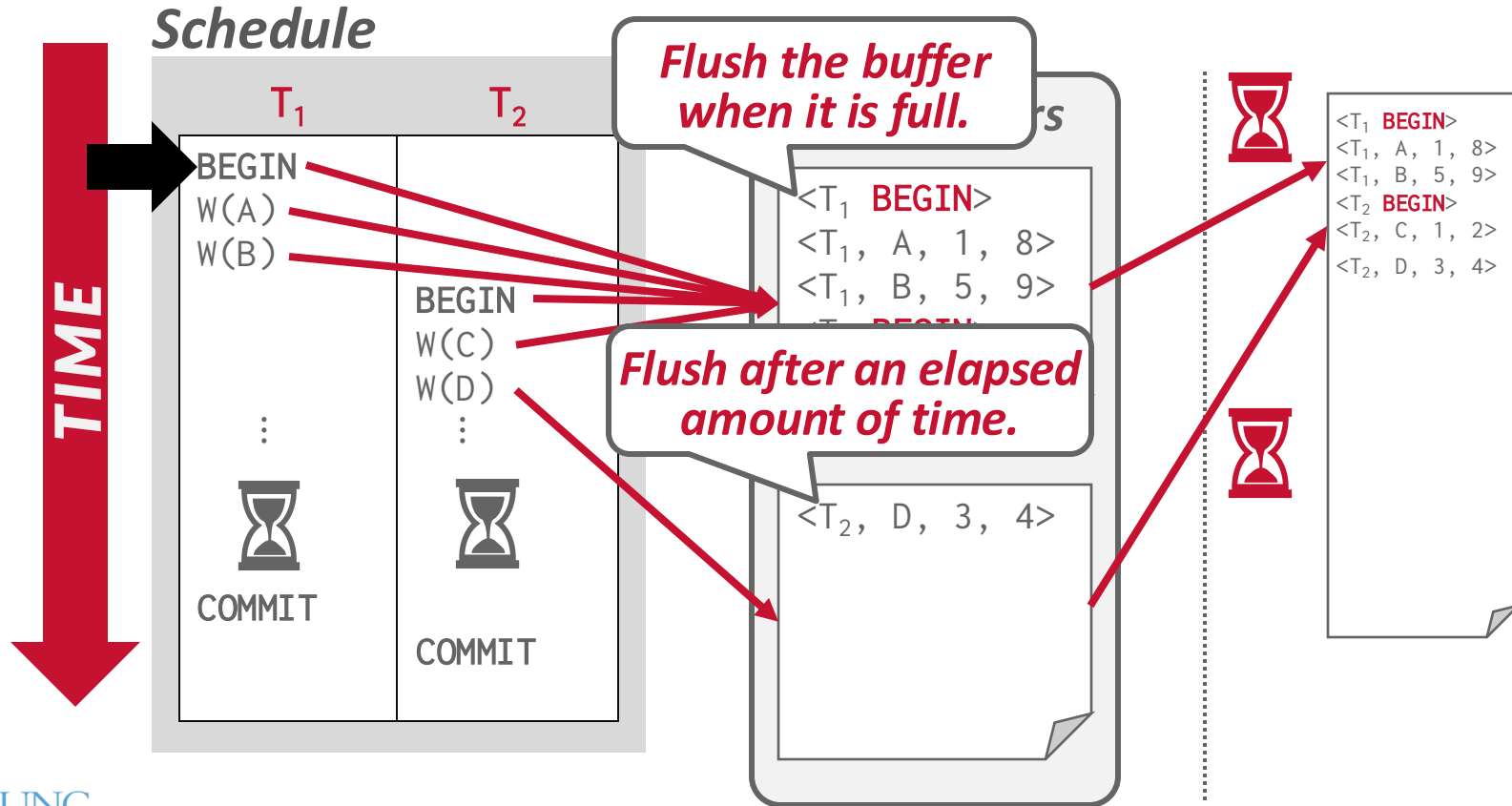
# WAL – Implementation

Flushing the log buffer to disk every time a txn commits will become 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).

# WAL – Group Commit



# Buffer Pool Policies

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

## Runtime Performance

	NO-STEAL	STEAL
NO-FORCE	—	<b>Fastest</b>
FORCE	<b>Slowest</b>	—

## Recovery Performance

	NO-STEAL	STEAL
NO-FORCE	—	<b>Slowest</b>
FORCE	<b>Fastest</b>	—

**No Undo + No Redo**

**Undo + Redo**

# Logging Schemes

## Physical Logging

- Record the byte-level changes made to a specific page.
- Example: **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 approach with byte-level changes for a single tuple identified by page id + slot number.
- Does not specify organization of the page.

# Logging Schemes

```
UPDATE foo SET val = XYZ WHERE id = 1;
```

## Physical

```
<T1,
  Table=X,
  Page=99,
  Offset=1024,
  Before=ABC,
  After=XYZ>
<T1,
  Index=X PKEY,
  Page=45,
  Offset=9,
  Key=(1,Record1)>
```

## Logical

```
<T1,
  Query="UPDATE foo
        SET val=XYZ
        WHERE id=1">
```

## Physiological

```
<T1,
  Table=X,
  Page=99,
  Slot=1,
  Before=ABC,
  After=XYZ>
<T1,
  Index=X PKEY,
  IndexPage=45,
  Key=(1,Record1)>
```

# Physical vs. Logical Logging

Logical logging requires less data written in each log record than physical logging.

Difficult to implement recovery with logical logging if you have concurrent txns running at lower isolation levels.

- Hard 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.



# Observation

The DBMS's WAL will grow forever.

After a crash, the DBMS must replay the entire log, which will take a long time.

The DBMS periodically takes a **checkpoint** where it flushes all buffers out to disk.

- This provides a hint on how far back it needs to replay the WAL after a crash.
- Notes all current, uncommitted transactions
- Truncate the WAL up to a certain safe point in time.

# Checkpoints

## Blocking / Consistent Checkpoint Protocol:

- Pause all queries.
- Flush all WAL records in memory to disk.
- Flush all modified pages in the buffer pool to disk.
- Write a **<CHECKPOINT>** entry to WAL and flush to disk.
- Resume queries.

# Checkpoints

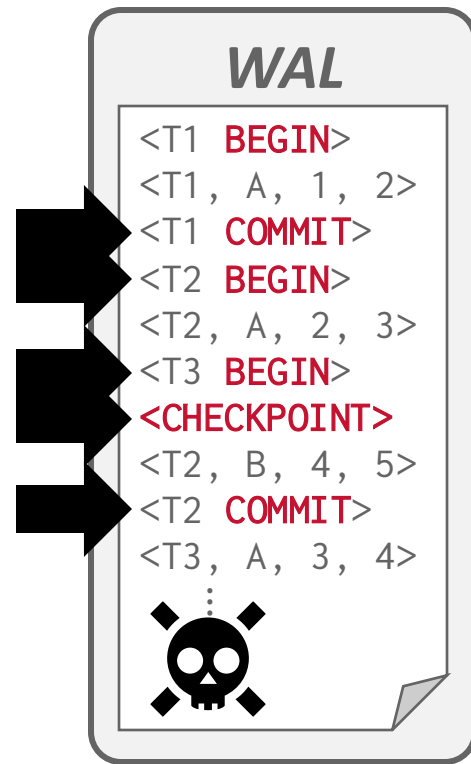
Use the **<CHECKPOINT>** record as the starting point for analyzing the WAL.

Any txn that committed before the checkpoint is ignored (**T<sub>1</sub>**).

**T<sub>2</sub>** + **T<sub>3</sub>** did not commit before the last checkpoint.

→ Need to redo **T<sub>2</sub>** because it committed after checkpoint.

→ Need to undo **T<sub>3</sub>** because it did not commit before the crash.



# Checkpoints – Challenges

In this example, the DBMS must stall txns when it takes a checkpoint to ensure a consistent snapshot.

→ We will see how to get around this problem next class.

Scanning the log to find uncommitted txns can take a long time.

→ Unavoidable but we will add hints to the **<CHECKPOINT>** record to speed things up next class.

How often the DBMS should take checkpoints depends on many different factors...

# Checkpoints – Frequency

Checkpointing too often causes the runtime performance to degrade.

→ System spends too much time flushing buffers.

But waiting a long time is just as bad:

→ The checkpoint will be large and slow.

→ Makes recovery time much longer.

Tunable option that depends on application recovery time requirements.

# Conclusion

Write-Ahead Logging 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.

# Next Class

Better Checkpoint Protocols.

Recovery with ARIES.