Carnegie Mellon Univ.
Dept. of Computer Science
15-415/615 - DB Applications

C. Faloutsos – A. Pavlo

Lecture#23: Crash Recovery

(R&G ch. 18)

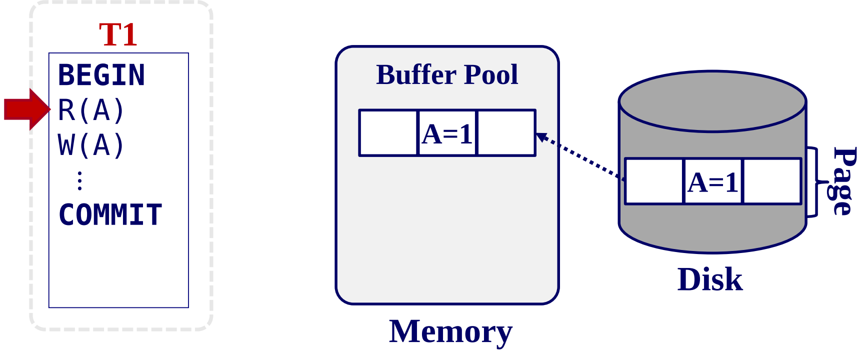
Last Class

- Basic Timestamp Ordering
- Optimistic Concurrency Control
- Multi-Version Concurrency Control

Today's Class

- Overview
- Write-Ahead Log
- Checkpoints
- Logging Schemes
- Recovery Protocol
- Shadow Paging

Motivation



T1

```
BEGIN
R(A)
W(A)
 ⋮
COMMIT
```

Buffer Pool

A=1

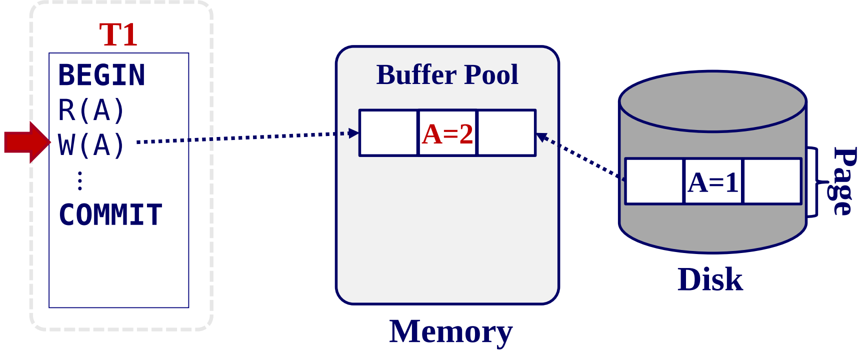
Memory

A=1

Page

Disk

Motivation



T1

BEGIN

R(A)

W(A)

⋮

COMMIT

Buffer Pool

A=2

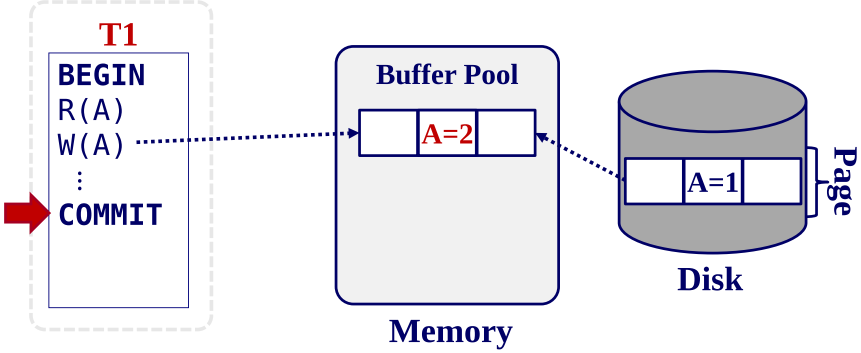
A=1

Page

Disk

Memory

Motivation



T1

BEGIN

R(A)

W(A)

⋮

COMMIT

Buffer Pool

A=2

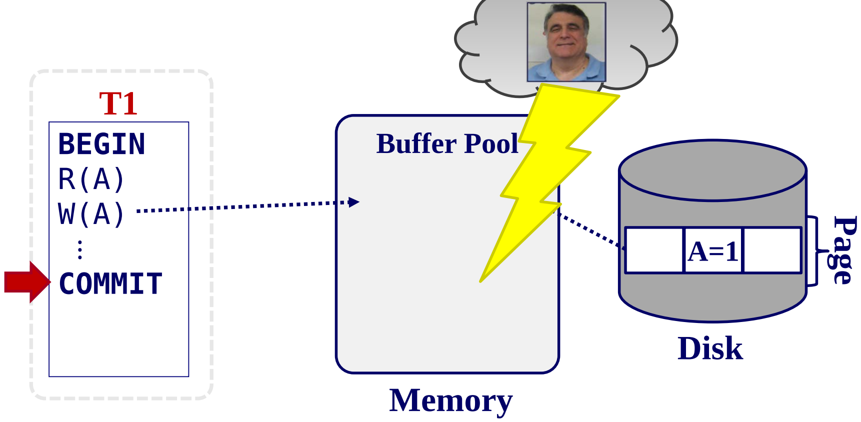
A=1

Page

Disk

Memory

Motivation



T1

BEGIN

R(A)

W(A)

⋮

COMMIT

Buffer Pool

A=1

Page

Disk

Memory

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.

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.

Crash Recovery

- DBMS is divided into different components based on the underlying storage device.
- Need to also classify the different types of failures that the DBMS needs to handle.

Storage Types

- **Volatile Storage:**
  - Data does not persist after power is cut.
  - Examples: DRAM, SRAM
- **Non-volatile Storage:**
  - Data persists after losing power.
  - Examples: HDD, SDD
- **Stable Storage:**
  - A non-existent form of non-volatile storage that survives all possible failures scenarios.

Use multiple storage devices to approximate.

Failure Classification

- Transaction Failures
- System Failures
- Storage Media Failures

Transaction Failures

- **Logical Errors:**
  - Transaction cannot complete due to some internal error condition (e.g., integrity constraint violation).
- **Internal State Errors:**
  - DBMS must terminate an active transaction due to an error condition (e.g., deadlock)

System Failures

- **Software Failure:**
  - Problem with the DBMS implementation (e.g., uncaught divide-by-zero exception).
- **Hardware Failure:**
  - The computer hosting the DBMS crashes (e.g., power plug gets pulled).
  - **Fail-stop Assumption:** Non-volatile storage contents are assumed to not be corrupted by system crash.

Storage Media Failure

- **Non-Repairable Hardware Failure:**
  - A head crash or similar disk failure destroys all or part of non-volatile storage.
  - Destruction is assumed to be detectable (e.g., disk controller use checksums to detect failures).

- No DBMS can recover from this. Database must be restored from archived version.

Problem Definition

- Primary storage location of records is on non-volatile storage, but this is much 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.

Problem Definition

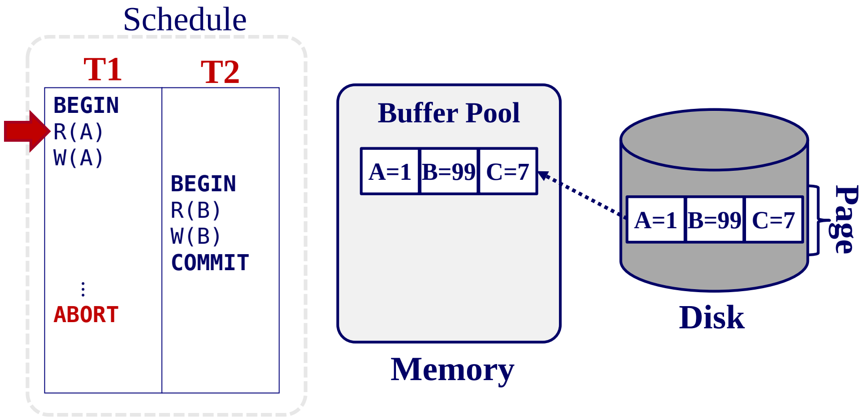
- Need to ensure:
  - The changes for any txn are durable once the DBMS has told somebody that it committed.
  - No changes are durable if the txn aborted.

Undo vs. Redo

- **Undo:** The process of removing the effects of an incomplete or aborted txn.
- **Redo:** The process of re-instating the effects of a committed txn for durability.

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

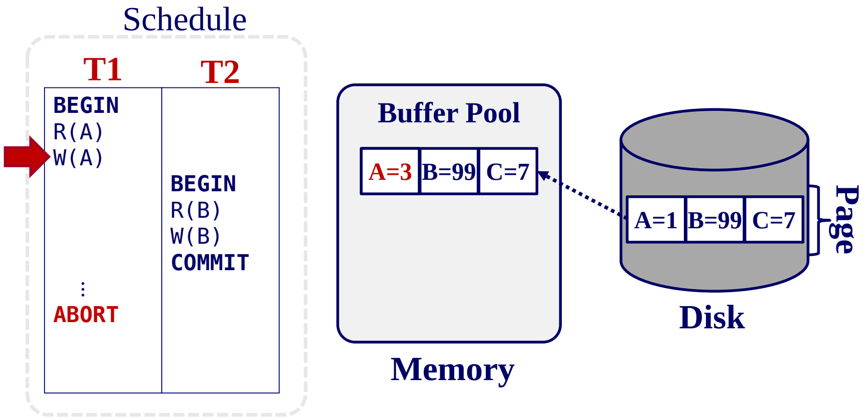
Buffer Pool Management



Schedule

T1	**T2**
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(B)
	W(B)
	COMMIT
⋮	
ABORT	

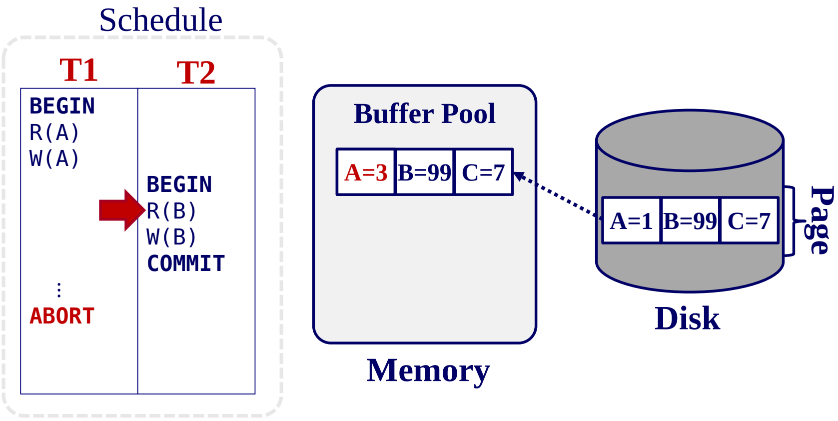
Buffer Pool Management



Schedule

T1	**T2**
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(B)
	W(B)
	COMMIT
⋮	
ABORT	

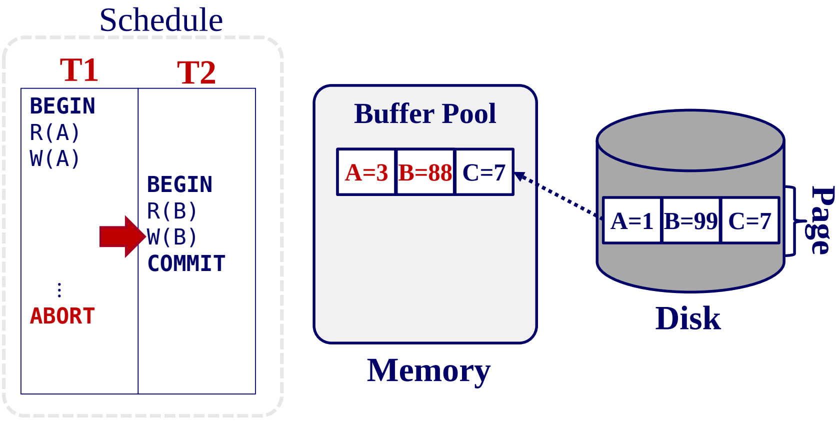
Buffer Pool Management



Schedule

T1	T2
BEGIN R(A) W(A)	
	BEGIN R(B) W(B) COMMIT
⋮ ABORT	

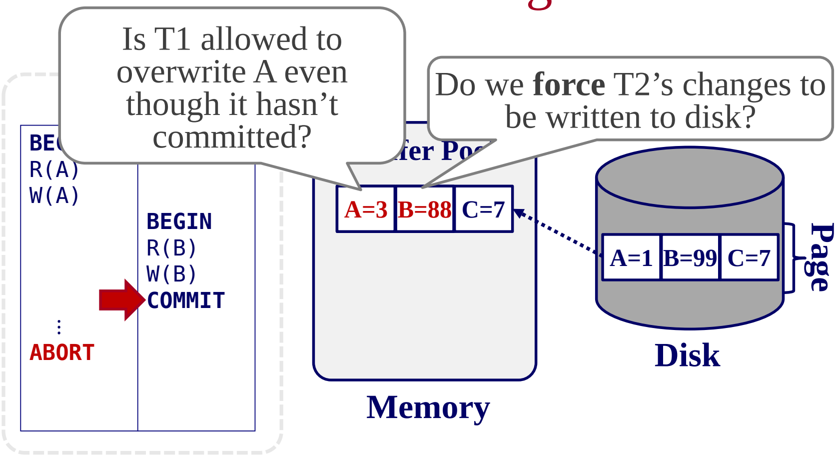
Buffer Pool Management



Schedule

T1	T2
BEGIN R(A) W(A)	
	BEGIN R(B) W(B) COMMIT
⋮ ABORT	

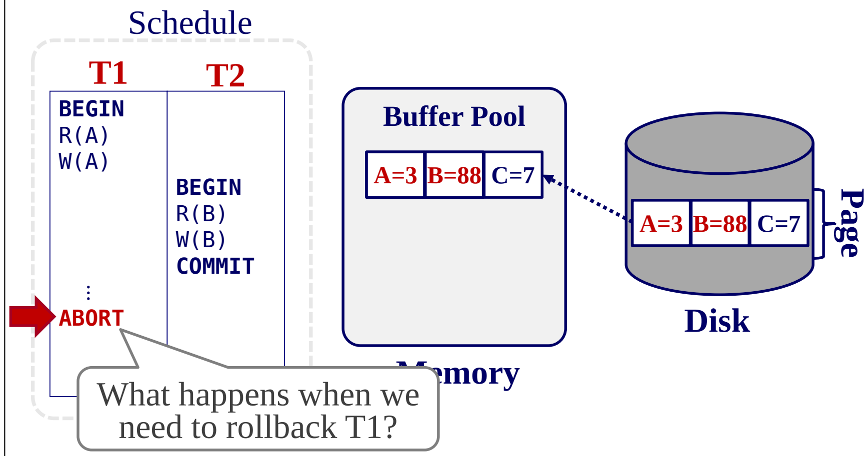
Buffer Pool Management



Is T1 allowed to overwrite A even though it hasn’t committed?

Do we **force** T2’s changes to be written to disk?

Buffer Pool Management



Schedule

T1	T2
BEGIN R(A) W(A)	
	BEGIN R(B) W(B) COMMIT
⋮ ABORT	

What happens when we need to rollback T1?

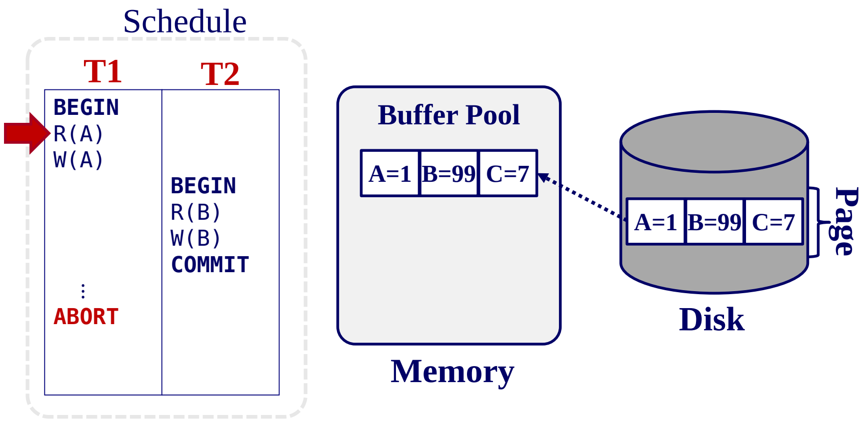
Buffer Pool – Steal Policy

- Whether the DBMS allows an uncommitted txn to overwrite the most recent committed value of an object in non-volatile storage.
  - **STEAL:** Is allowed.
  - **NO-STEAL:** Is not allowed.

Buffer Pool – Force Policy

- Whether the DBMS ensures that all updates made by a txn are reflected on non-volatile storage before the txn is allowed to commit:
  - **FORCE:** Is enforced.
  - **NO-FORCE:** Is not enforced.
- Force writes makes it easier to recover but results in poor runtime performance.

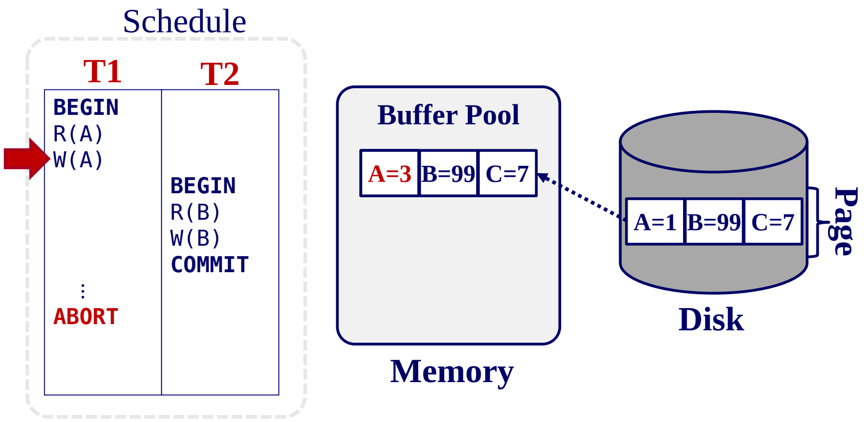
NO-STEAL + FORCE



Schedule

T1	T2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(B)
	W(B)
	COMMIT
⋮	
ABORT	

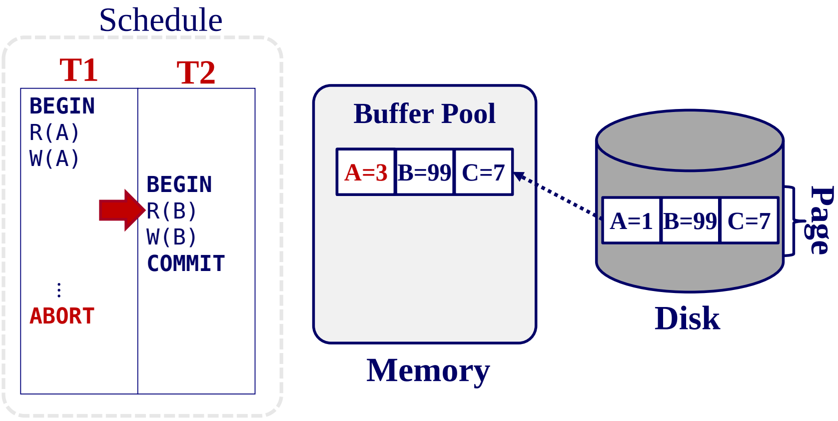
NO-STEAL + FORCE



Schedule

T1	T2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(B)
	W(B)
	COMMIT
⋮	
ABORT	

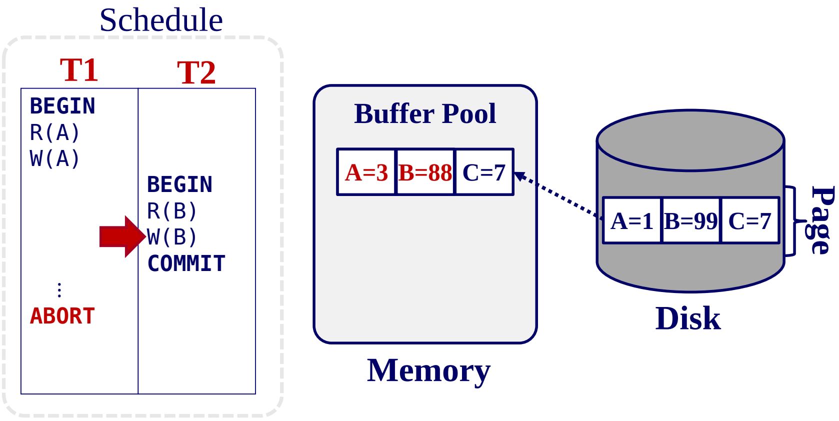
NO-STEAL + FORCE



Schedule

T1	T2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(B)
	W(B)
	COMMIT
⋮	
ABORT	

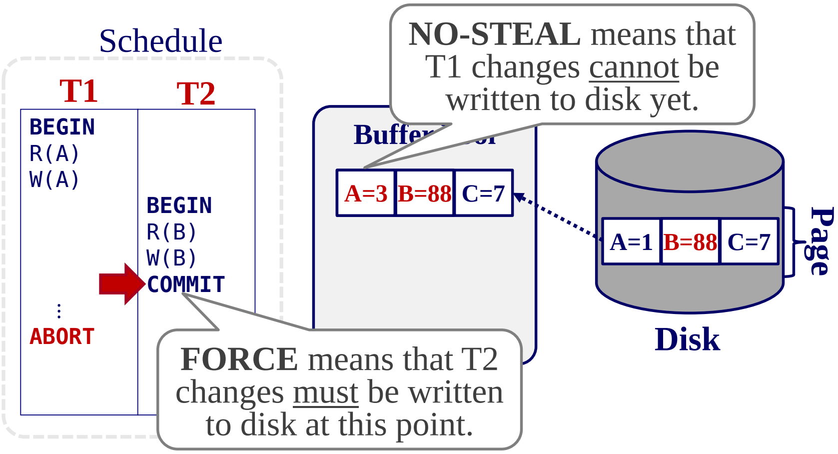
NO-STEAL + FORCE



Schedule

T1	T2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(B)
	W(B)
	COMMIT
⋮	
ABORT	

NO-STEAL + FORCE



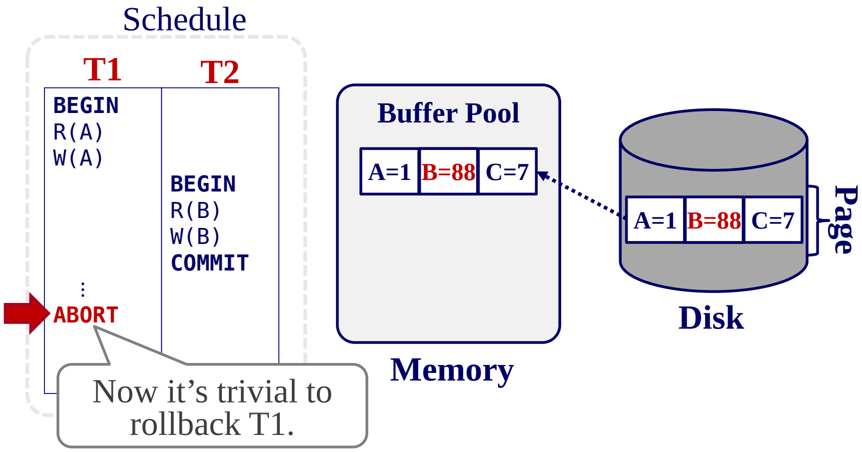
Schedule

T1	T2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(B)
	W(B)
	COMMIT
⋮	
ABORT	

NO-STEAL means that T1 changes cannot be written to disk yet.

FORCE means that T2 changes must be written to disk at this point.

NO-STEAL + FORCE



Schedule

T1	T2
BEGIN	
R(A)	
W(A)	
	BEGIN
	R(B)
	W(B)
	COMMIT
⋮	
ABORT	

Now it’s trivial to rollback T1.

NO-STEAL + FORCE

- This approach is the easiest to implement:
  - Never have to undo changes of an aborted txn because the changes were not written to disk.
  - Never have to redo changes of a committed txn because all the changes are guaranteed to be written to disk at commit time.
- But this will be slow…
- What if txn modifies the entire database?

Today's Class

- Overview
- Write-Ahead Log
- Checkpoints
- Logging Schemes
- Recovery Protocol
- Shadow Paging

Write-Ahead Log

- Record the changes made to the database in a log ***before*** the change is made.
  - Assume that the log is on stable storage.
  - Log contains sufficient information to perform the necessary undo and redo actions to restore the database after a crash.
- Buffer Pool: **STEAL** + **NO-FORCE**

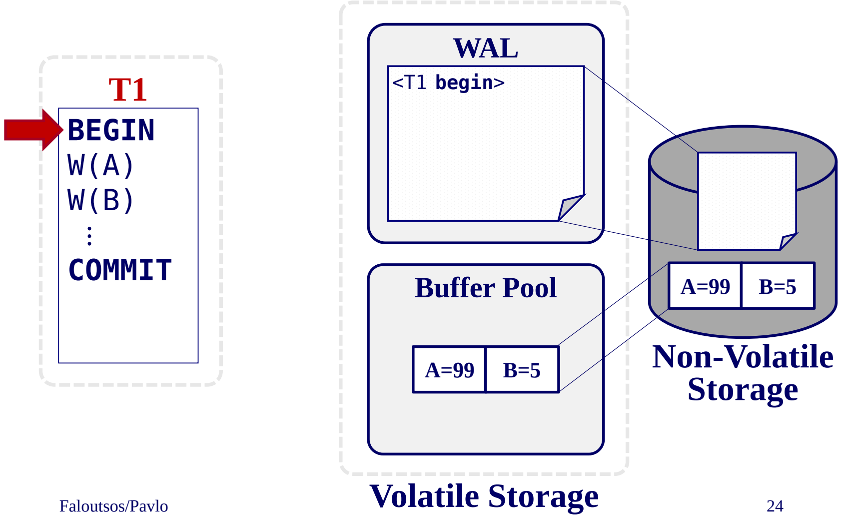
Write-Ahead Log Protocol

- All log records pertaining to an updated page are written to non-volatile storage before the page itself is allowed to be over-written in non-volatile storage.
- A txn is not considered committed until all its log records have been written to stable storage.

Write-Ahead Log Protocol

- Write a **`<BEGIN>`** record to the log for each txn to mark its starting point.
- Log record format:
  - **`<txnId, objectId, beforeValue, afterValue>`**
- When a txn finishes, the DBMS will:
  - Write a **`<COMMIT>`** record on the log
  - Make sure that all log records are flushed before it returns an acknowledgement to application.

Write-Ahead Log – Example



T1

```
BEGIN
W(A)
W(B)
 ⋮
COMMIT
```

WAL

```
<T1 begin>
```

Buffer Pool

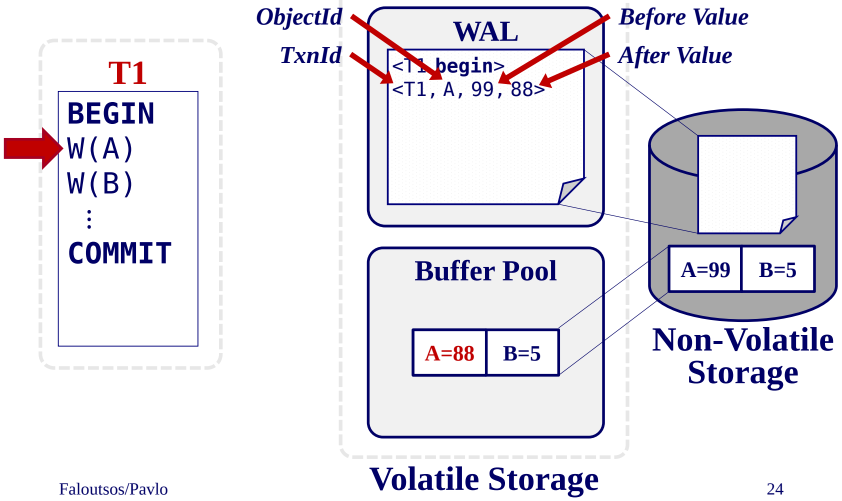
A=99 B=5

A=99 B=5

Non-Volatile Storage

Volatile Storage

Write-Ahead Log – Example



ObjectId

TxnId

Before Value

After Value

T1

```
BEGIN
W(A)
W(B)
 ⋮
COMMIT
```

WAL

```
<T1 begin>
<T1, A, 99, 88>
```

Buffer Pool

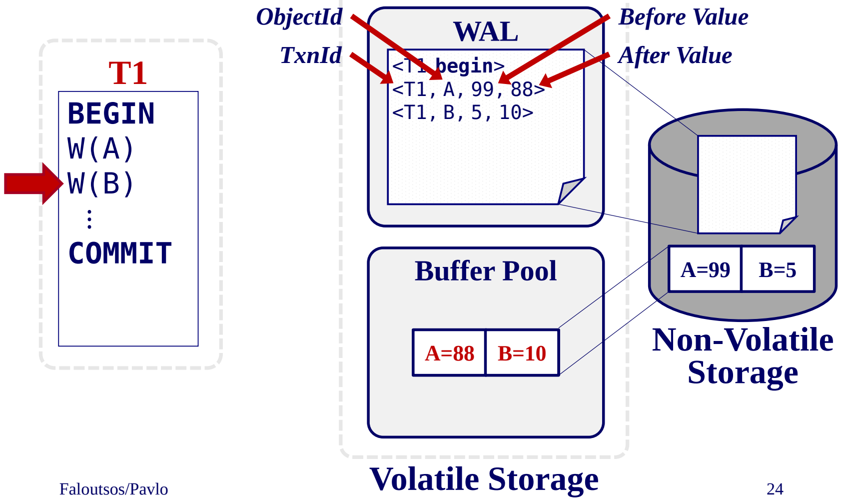
A=88 B=5

A=99 B=5

Non-Volatile Storage

Volatile Storage

Write-Ahead Log – Example



ObjectId

TxnId

Before Value

After Value

T1

```
BEGIN
W(A)
W(B)
 ⋮
COMMIT
```

WAL

```
<T1 begin>
<T1, A, 99, 88>
<T1, B, 5, 10>
```

Buffer Pool

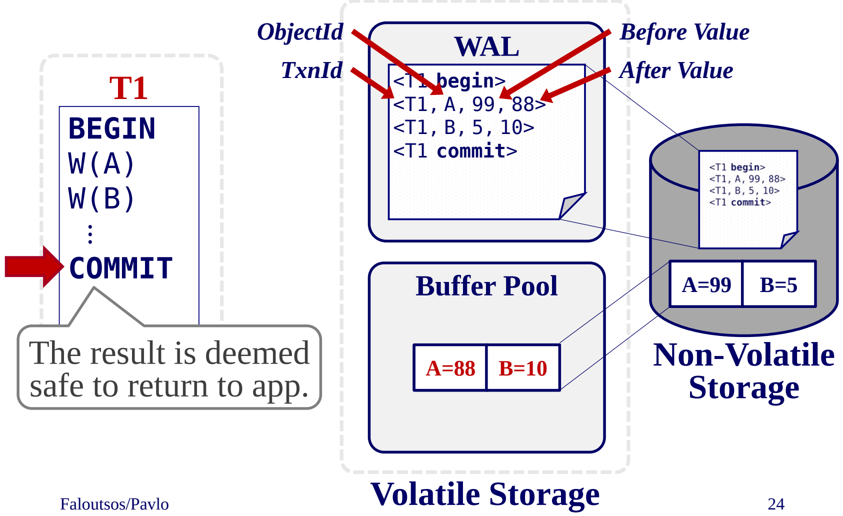
A=88 B=10

A=99 B=5

Non-Volatile Storage

Volatile Storage

Write-Ahead Log – Example



T1

BEGIN
W(A)
W(B)
⋮
COMMIT

The result is deemed safe to return to app.

ObjectId *TxnId* **WAL** *Before Value* *After Value*

```
<T1 begin>
<T1, A, 99, 88>
<T1, B, 5, 10>
<T1 commit>
```

Buffer Pool

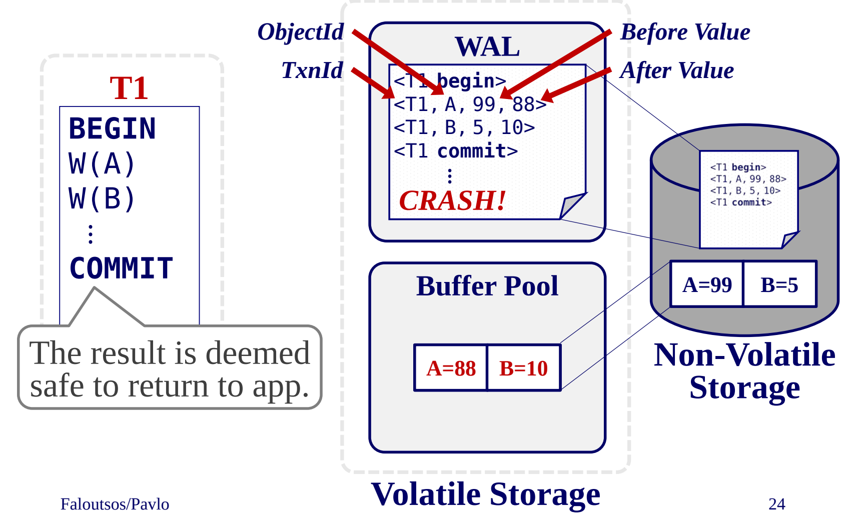
A=88 B=10

Volatile Storage

A=99 B=5

Non-Volatile Storage

Write-Ahead Log – Example



T1

BEGIN
W(A)
W(B)
⋮
COMMIT

The result is deemed safe to return to app.

ObjectId *TxnId* **WAL** *Before Value* *After Value*

```
<T1 begin>
<T1, A, 99, 88>
<T1, B, 5, 10>
<T1 commit>
⋮
CRASH!
```

Buffer Pool

A=88 B=10

Volatile Storage

A=99 B=5

Non-Volatile Storage

WAL – Implementation Details

- When should we write log entries to disk?
  - When the transaction commits.
  - Can use **group commit** to batch multiple log flushes together to amortize overhead.
  - Every time the txn executes an update?
  - Once when the txn commits?

WAL – Deferred Updates

- Observation: If we prevent the DBMS from writing dirty records to disk until the txn commits, then we don't need to store their original values.

WAL

```
<T1 begin>
<T1, A, 99, 88>
<T1, B, 5, 10>
<T1 commit>
```

WAL – Deferred Updates

- Observation: If we prevent the DBMS from writing dirty records to disk until the txn commits, then we don't need to store their original values.

WAL

<T1 **begin**>
<T1, A, ~~99~~, 88>
<T1, B, ~~5~~, 10>
<T1 **commit**>

WAL – Deferred Updates

- Observation: If we prevent the DBMS from writing dirty records to disk until the txn commits, then we don't need to store their

Replay the log and redo each update.

<T1 **begin**>
<T1, A, 88>
<T1, B, 10>
<T1 **commit**>
CRASH!

Simply ignore all of T1's updates.

<T1 **begin**>
<T1, A, 88>
<T1, B, 10>
CRASH!

WAL – Deferred Updates

- This won't work if the change set of a txn is larger than the amount of memory available.
- The DBMS cannot undo changes for an aborted txn if it doesn't have the original values in the log.
- We need to use the **STEAL** policy.

WAL – Buffer Pool Policies

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

Runtime Performance

Undo + Redo

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

No Undo + No Redo

Recovery
erformance

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

Today's Class

- Overview
- Write-Ahead Log
- Checkpoints
- Logging Schemes
- Recovery Protocol
- Shadow Paging

Checkpoints

- The WAL will grow forever.
- After a crash, the DBMS has to replay the entire log which will take a long time.
- The DBMS periodically takes a **checkpoint** where it flushes all buffers out to disk.

Checkpoints

- Output onto stable storage all log records currently residing in main memory.
- Output to the disk all modified blocks.
- Write a `<CHECKPOINT>` entry to the log and flush to stable storage.

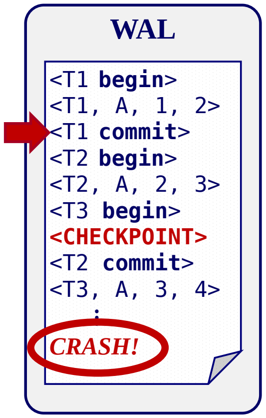
Checkpoints

WAL

```
<T1 begin>
<T1, A, 1, 2>
<T1 commit>
<T2 begin>
<T2, A, 2, 3>
<T3 begin>
<CHECKPOINT>
<T2 commit>
<T3, A, 3, 4>
        :
```

CRASH!

Checkpoints

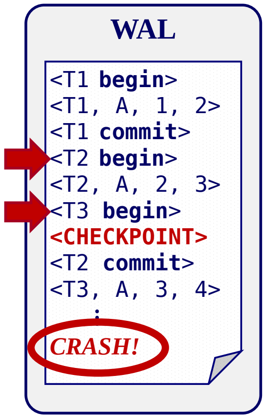


WAL

```
<T1 begin>
<T1, A, 1, 2>
<T1 commit>
<T2 begin>
<T2, A, 2, 3>
<T3 begin>
<CHECKPOINT>
<T2 commit>
<T3, A, 3, 4>
 :
CRASH!
```

- Any txn that committed before the checkpoint is ignored (T1).

Checkpoints



WAL

```
<T1 begin>
<T1, A, 1, 2>
<T1 commit>
<T2 begin>
<T2, A, 2, 3>
<T3 begin>
<CHECKPOINT>
<T2 commit>
<T3, A, 3, 4>
 :
CRASH!
```

- Any txn that committed before the checkpoint is ignored (T1).
- T2 + T3 did not commit before the last checkpoint.
  - Need to <u>redo</u> T2 because it committed after checkpoint.
  - Need to <u>undo</u> T3 because it did not commit before the crash.

Checkpoints – Challenges

- We have to stall all txns when take a checkpoint to ensure a consistent snapshot.
- Scanning the log to find uncommitted txns can take a long time.
- Not obvious how often the DBMS should take a checkpoint…

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.

Today's Class

- Overview
- Write-Ahead Log
- Checkpoints
- Logging Schemes
- Recovery Protocol
- Shadow Paging

Logging Schemes

- **Physical Logging:** Record the changes made to a specific location in the database.
  - Example: Position of a record in a page.
- **Logical Logging:** Record the high-level operations executed by txns.
  - Example: The **`UPDATE`**, **`DELETE`**, and **`INSERT`** queries invoked by a txn.

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.
  - Hard to determine which parts of the database may have been modified by a query before crash.
  - Also takes longer to recover because you must re-execute every txn all over again.

Physiological Logging

- Hybrid approach where log records target a single page but do not specify data organization of the page.
- This is the most popular approach.

Logging Schemes

```
INSERT INTO X VALUES(1,2,3);
```

Physical

```
<T1,
 Table=X,
 Page=99,
 Offset=4,
 Record=(1,2,3)>

<T1,
 Index=X_PKEY,
 Page=45,
 Offset=9,
 Key=(1,Record1)>
```

Logical

```
<T1,
 “INSERT INTO X
  VALUES(1,2,3)”>
```

Physiological

```
<T1,
 Table=X,
 Page=99,
 Record=(1,2,3)>

<T1,
 Index=X_PKEY,
 IndexPage=45,
 Key=(1,Record1)>
```

Today’s Class

- Overview
- Write-Ahead Log
- Checkpoints
- Logging Schemes
- Recovery Protocol
- Shadow Paging

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's Class – ARIES

- **A**lgorithms for **R**ecovery and **I**solation **E**xploiting **S**emantics
  - Write-ahead Logging
  - Repeating History during Redo
  - Logging Changes during Undo

ARIES

- Developed at IBM during the early 1990s.
- Considered the "gold standard" in database crash recovery.
    - Implemented in DB2.
    - Everybody else more or less implements a variant of it.

C. Mohan
IBM Fellow

ARIES – Main Ideas

- **Write-Ahead Logging:**
    - Any change is recorded in log on stable storage before the database change is written to disk.
- **Repeating History During Redo:**
    - On restart, retrace actions and restore database 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.

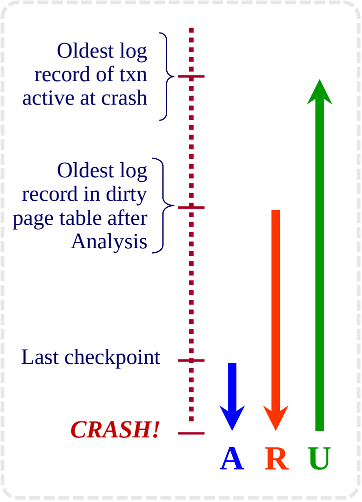
ARIES – Main Ideas

- Write Ahead Logging
    - Fast, during normal operation
    - Least interference with OS (i.e., **STEAL**, **NO FORCE**)
- Fast (fuzzy) checkpoints
- On Recovery:
    - Redo everything.
    - Undo uncommitted txns.

ARIES – Recovery Phases

- **Analysis:** Read the WAL to identify dirty pages in the buffer pool and active txns at the time of the crash.
- **Redo:** Repeat all actions starting from an appropriate point in the log.
- **Undo:** Reverse the actions of txns that did not commit before the crash.

ARIES - Overview



- Start from last checkpoint found via ***Master Record***.
- Three phases.
  - – **Analysis** - Figure out which txns committed or failed since checkpoint.
  - – **Redo** all actions (repeat history)
  - – **Undo** effects of failed txns.

Additional Crash Issues

- What happens if system crashes during the Analysis Phase? During the Redo Phase?
- How do you limit the amount of work in the Redo Phase?
  - – Flush asynchronously in the background.
- How do you limit the amount of work in the Undo Phase?
  - – Avoid long-running txns.

Today’s Class

- Overview
- Write-Ahead Log
- Checkpoints
- Logging Schemes
- Recovery Protocol
- Shadow Paging

Shadow Paging

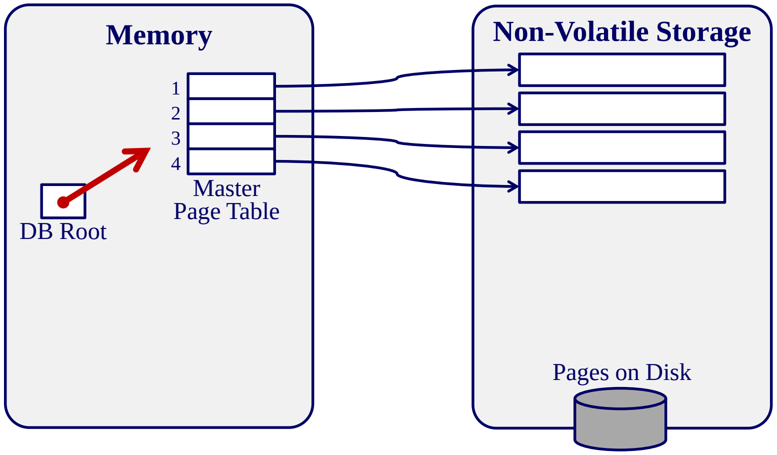
- Maintain two separate copies of the database (master, shadow)
- Updates are only made in the shadow copy.
- When a txn commits, atomically switch the shadow to become the new master.
- Buffer Pool: **NO-STEAL** + **FORCE**

Shadow Paging

- Database is a tree whose root is a single disk block.
- There are two copies of the tree, the master and shadow
  - The root points to the master copy.
  - Updates are applied to the shadow copy.

Portions courtesy of the great Phil Bernstein

Shadow Paging – Example

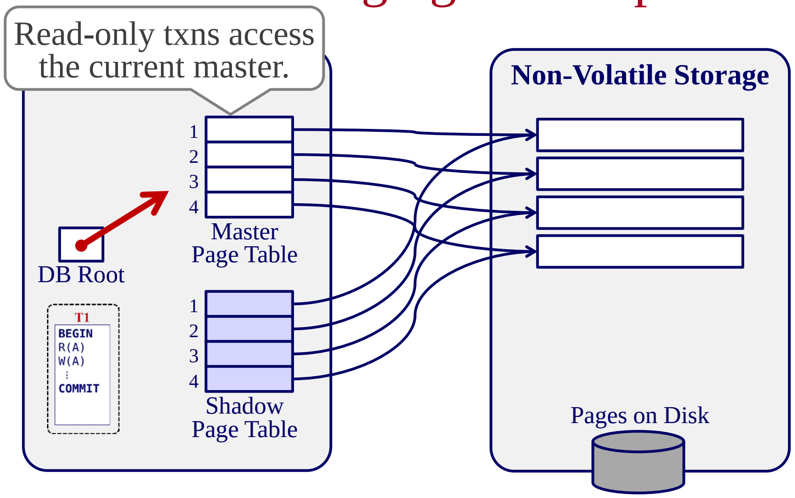


Shadow Paging

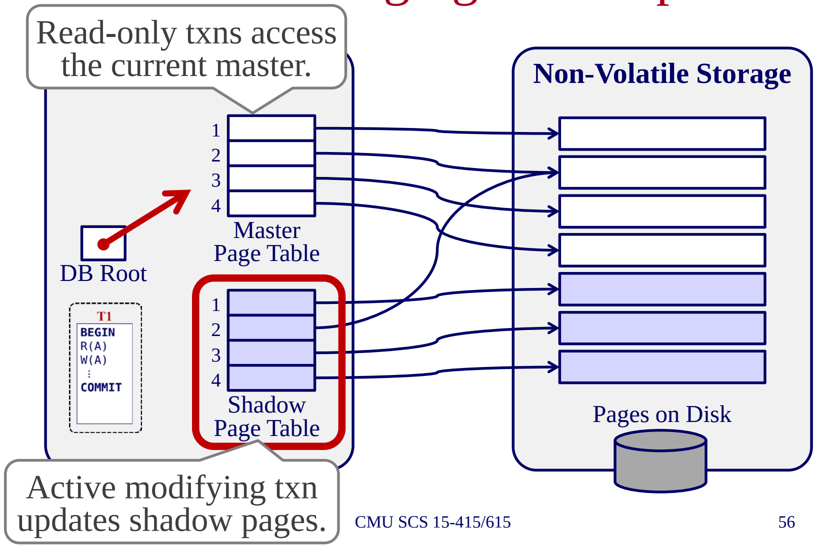
- To install the updates, overwrite the root so it points to the shadow, thereby swapping the master and shadow:
  - Before overwriting the root, none of the transaction's updates are part of the disk-resident database
  - After overwriting the root, all of the transaction's updates are part of the disk-resident database.

Portions courtesy of the great Phil Bernstein

Shadow Paging – Example


Read-only txns access the current master.

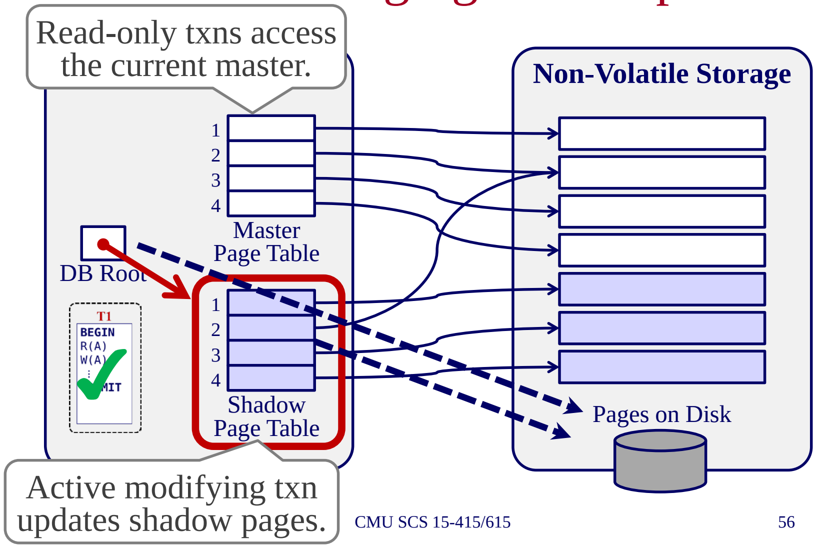
Shadow Paging – Example



Read-only txns access the current master.

Active modifying txn updates shadow pages.

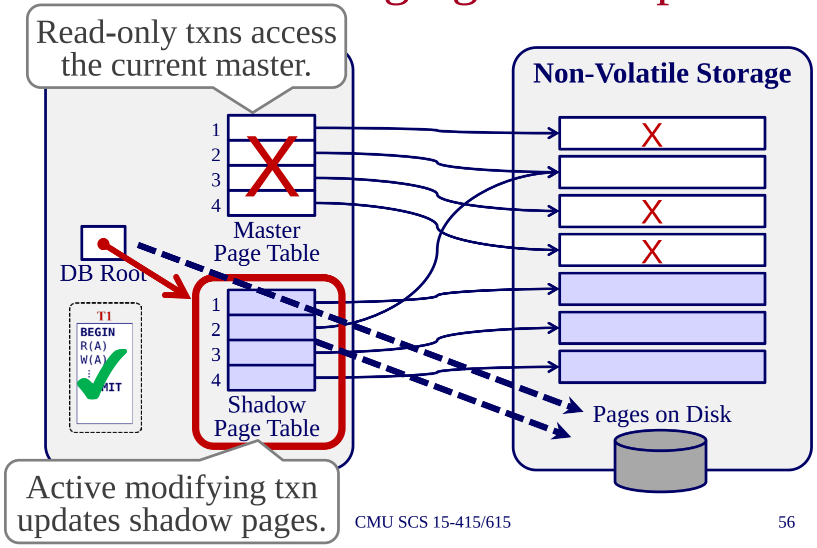
Shadow Paging – Example



Read-only txns access the current master.

Active modifying txn updates shadow pages.

Shadow Paging – Example



Read-only txns access the current master.

Active modifying txn updates shadow pages.

Shadow Paging – Undo/Redo

- Supporting rollbacks and recovery is easy.
- **Undo:**
  - Simply remove the shadow pages. Leave the master and the DB root pointer alone.
- **Redo:**
  - Not needed at all.

Shadow Paging – Advantages

- No overhead of writing log records.
- Recovery is trivial.

Shadow Paging – Disadvantages

- Copying the entire page table is expensive:
  - Use a page table structured like a B+tree
  - 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, & root.
  - Data gets fragmented.
  - Need garbage collection.

Summary

- Write-Ahead Log to handle loss of volatile storage.
- Use incremental updates (i.e., **STEAL**, **NO-FORCE**) with checkpoints.
- On recovery: undo uncommitted txns + redo committed txns.

Conclusion

- Recovery is really hard.
- Be thankful that you don’t have to write it yourself.