Transactions: Recovery, Shadow-Paging, & **OATSdb** V2 Milestone

COM 3563: Database Implementation

Avraham Leff

Yeshiva University

avraham.leff@yu.edu

COM3563: Fall 2020

Today's Lecture: Overview

1. DBMS Recovery: Terminology & Concepts

2. Shadow Paging

3. **OATSdb** v2 Milestone: Providing Persistence to a Main-Memory Database

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

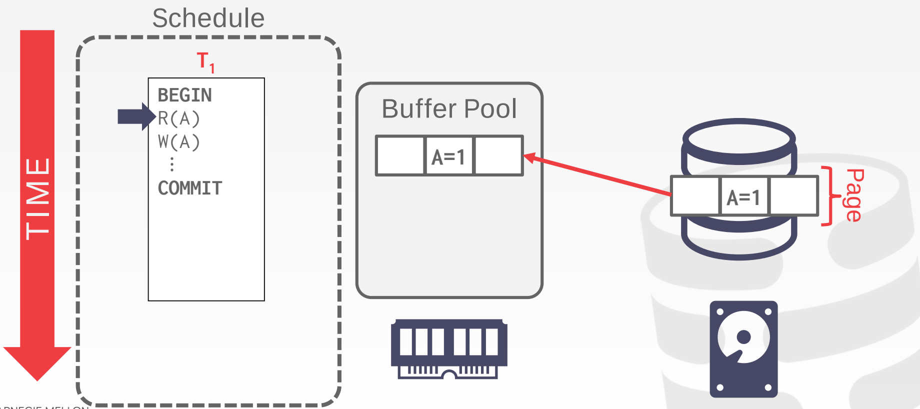
Context

- As you know, transactions are characterized by the famous set of ACID properties
- Today: begin a set of lectures that focus on the D of ACID
- Alternatively: how does a DBMS recover after a failure has occurred?
- We'll begin by outlining the intrinsic issues that must be addressed by any DBMS recovery scheme
- Then: how implementation strategies must deal with real-life™ memory hierarchy and storage characteristics

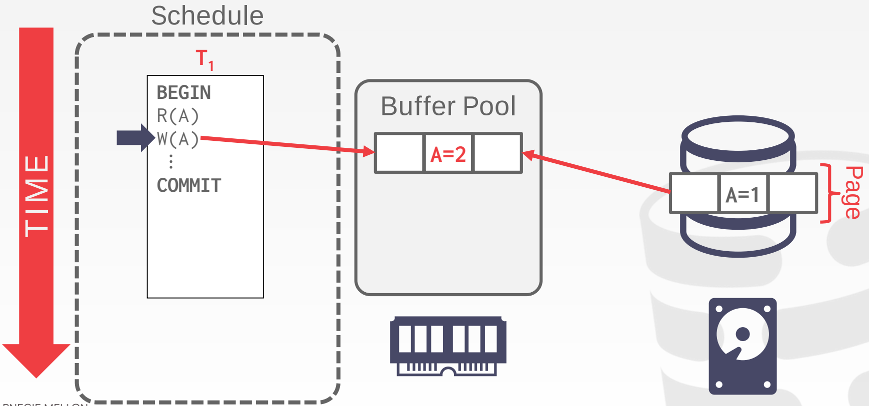
Recovery Algorithms: Intrinsic Issues

- Suppose T_i transfers \$50 from account A to account B
  - Two updates: subtract \$50 from A and add \$50 to B
  - Tx semantics require that updates to A and B be written to the database
- The system may crash (we'll define this more precisely later) after one of these modifications have been made but before both of them are made
- If we modify the database before the tx commit: failure implies that the database may be left in an inconsistent state
- If we modify the database only after the tx commit: failure implies that the database may be left in an inconsistent state (if crash "just after" tx commits)
- Recovery algorithms have two parts
  - Actions taken during "happy path" tx processing to ensure enough information exists to recover from failures
  - Actions taken after a failure to ensure that the database is recovered to a valid ACID state

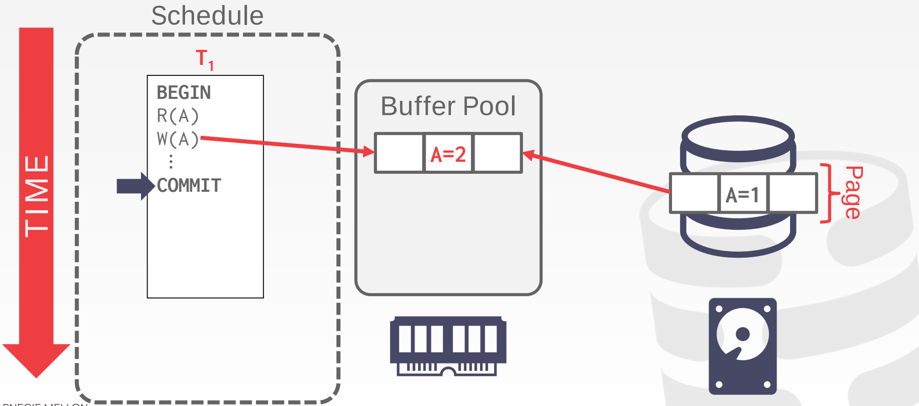
MOTIVATION



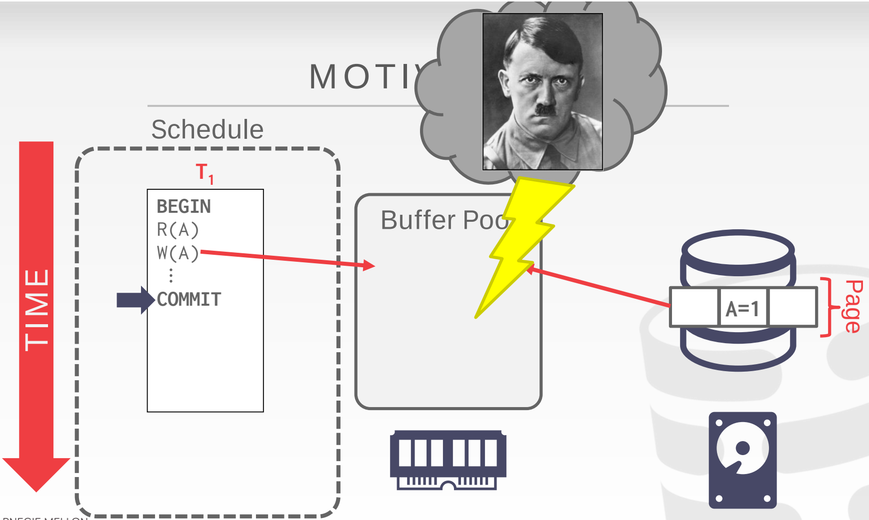
MOTIVATION



MOTIVATION



MOTI



Types of Storage

- The set of failure types depends partly on the set of storage types used by the DBMS
- Volatile storage: data do not persist after power failure
  - Examples: DRAM, SRAM (both are forms of random access memory (or RAM), typically used to provide "fast, main-memory")
- Non-volatile storage: data persist even after losing power
  - Example: solid-state drive (SSD)
  - Example: hard disk drive (HDD)
  - (See e.g., this article)

- You know what type of storage doesn't exist?
  - Stable storage: *"non-volatile storage that survives all possible failure scenarios"*
  - Simply doesn't exist (at least "right now") ☺

Failure Classification (I)

- Transaction failures: we've discussed these in previous lectures!
  - "Logical" errors: tx must be aborted due to some application-specific problem (e.g., the C in ACID)
  - "Internal State" errors: DBMS must abort a transaction because of concurrency problems (e.g., deadlock)
    - The specifics depend on the DBMS concurrency control algorithms
- DBMS "system failures": can be either hardware or software problems
  - Software: uncaught exceptions (think *ArithmeticException* ☹)
  - Hardware: computer crashes (think *"someone cut the power"*)
  - Key assumption: "fail-stop" behavior
    - Meaning: "Such a processor automatically halts in response to any internal failure and does so before the effects of that failure become visible"
    - Specifically: non-volatile storage contents are assumed to uncorrupted by system crash

Failure Classification (II)

- Storage media failures: meaning, *"non-repairable hardware failure"*
  - Example: a "disk head" crash or similar disk failure destroys all or part of non-volatile storage
- Key point: we can't prevent this from happening, but we can detect such failure events
  - Example: disk controllers use checksums to detect failures
- Recovery strategy: must restore the database from archived storage
  - Recovery strategy for "storage media" failure differs from strategy for "system crash" failure (previous slide)
  - DBMS use multiple integrity checks to prevent corruption of disk data
  - See discussion of "Redundant Array of Inexpensive Disks"

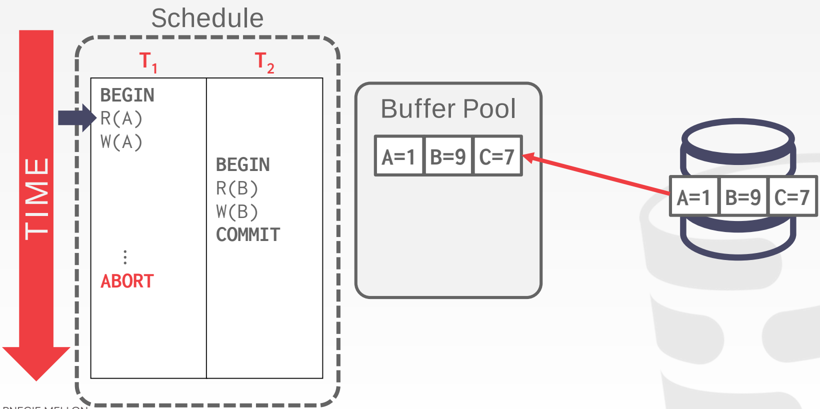
Multi-Tiered Memory: A Fact Of Life

- Given the relative cost of:
  - Expensive, but fast volatile storage
  - *Versus* cheap, but slow non-volatile storage ...
- Database storage is comprised of (at least) two tiers of memory
  - All data are "resident on disk" (cheap, non-volatile storage)
  - But: "in use" data are also resident in faster volatile storage ("main-memory")
- Database runtime
  1. First copy target record(s) into main-memory
  2. Perform read/write operations in main-memory
  3. Write "dirty records" back to disk
- Terminology: *A buffer pool is an area of main memory that has been allocated by the DBMS for the purpose of caching table and index data as it is read from disk*

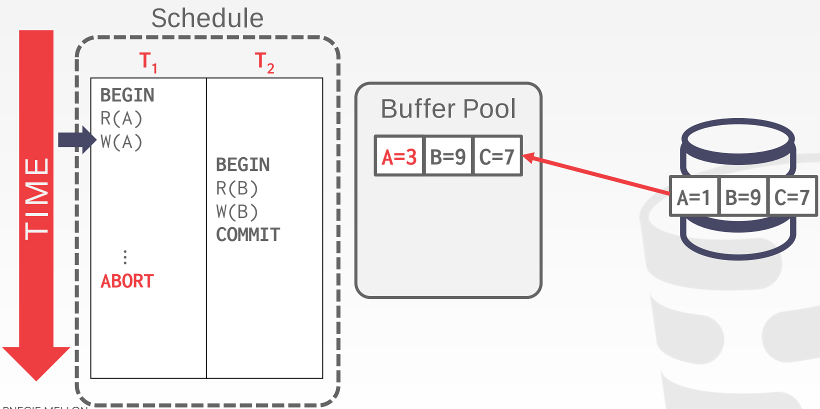
Multi-Tiered Memory: Implications For Recovery

- The "recovery algorithm" issues that we previously discussed apply even for a "main-memory-only" database
  - Example: **OATSdb** v1 milestone ☺
- But in (more realistic) databases that do provide persistence ...multi-tiered memory organization makes things even more complicated for a recovery algorithm ☹
- DBMS must ensure the following
  - Changes made during a transaction are durable ("written to disk") once the tx has been committed
  - No partial changes are visible if the tx is aborted
- Helps to think of two "recovery primitives" (applied to all memory tiers)
  - Undo: DBMS removes the effects of an incomplete or aborted transaction
  - Redo: DBMS reinstates the effects of a committed transaction (even if those results weren't previously "visible")

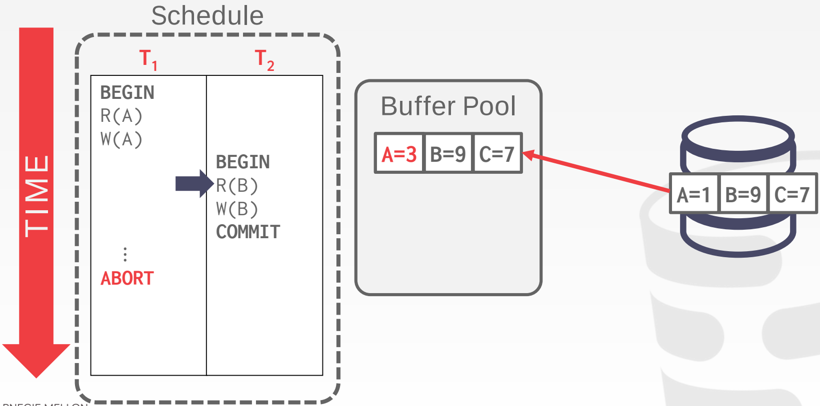
BUFFER POOL


Schedule
TIME
T_1
T_2
BEGIN
R(A)
W(A)
⋮
ABORT
BEGIN
R(B)
W(B)
COMMIT
Buffer Pool
A=1
B=9
C=7
A=1
B=9
C=7

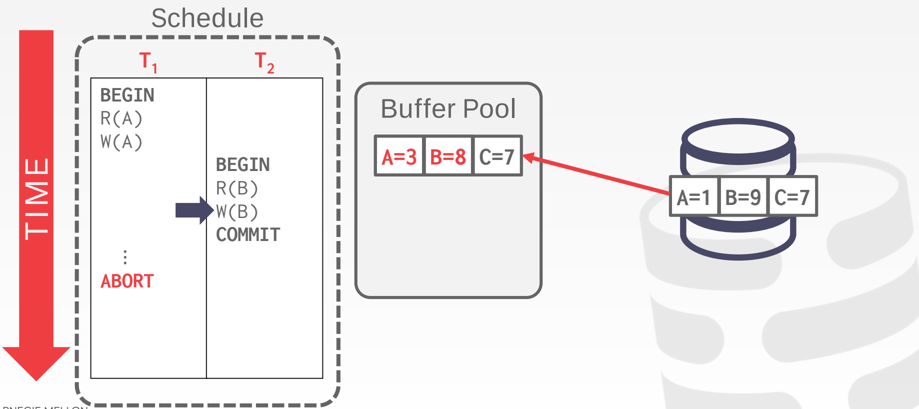
BUFFER POOL


Schedule
TIME
T1
T2
BEGIN
R(A)
W(A)
⋮
ABORT
BEGIN
R(B)
W(B)
COMMIT
Buffer Pool
A=3
B=9
C=7
A=1
B=9
C=7

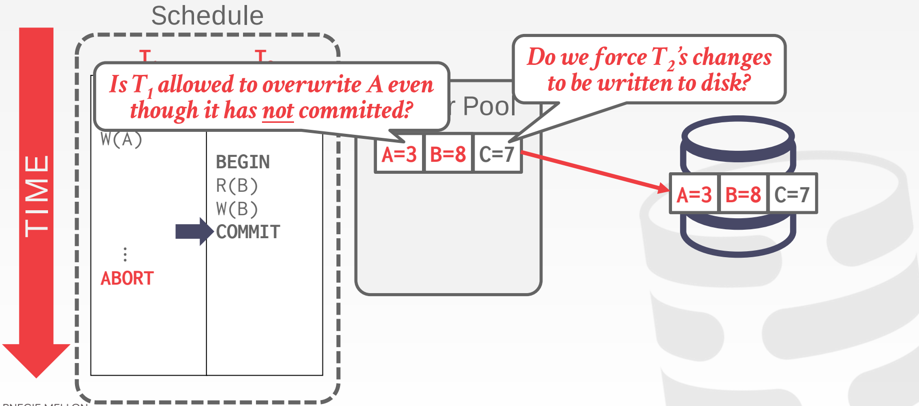
BUFFER POOL


Schedule
TIME
T_1
T_2
BEGIN
R(A)
W(A)
⋮
ABORT
BEGIN
R(B)
W(B)
COMMIT
Buffer Pool
A=3
B=9
C=7
A=1
B=9
C=7

BUFFER POOL

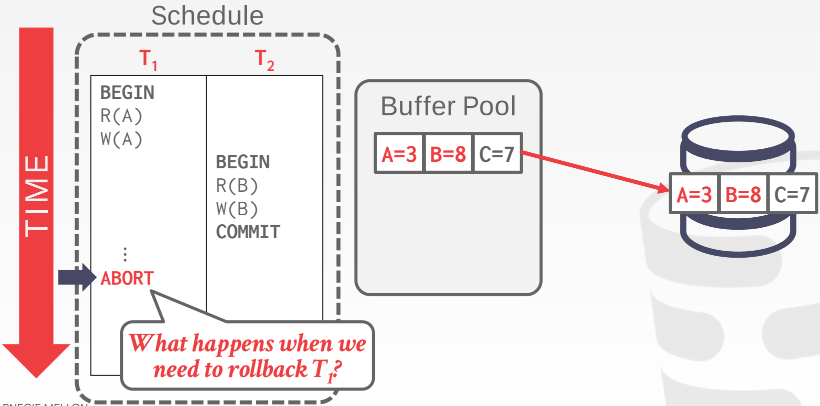

Schedule
TIME
T_1
T_2
BEGIN
R(A)
W(A)
ABORT
BEGIN
R(B)
W(B)
COMMIT
Buffer Pool
A=3
B=8
C=7
A=1
B=9
C=7

BUFFER POOL



Schedule
T
T
W(A)
BEGIN
R(B)
W(B)
COMMIT
⋮
ABORT
TIME
Is T_1 allowed to overwrite A even though it has not committed?
r Pool
Do we force T_2's changes to be written to disk?
A=3
B=8
C=7
A=3
B=8
C=7

BUFFER POOL


Schedule
T1
T2
BEGIN
R(A)
W(A)
BEGIN
R(B)
W(B)
COMMIT
ABORT
TIME
Buffer Pool
A=3
B=8
C=7
A=3
B=8
C=7
What happens when we need to rollback T1?

Managing The Buffer Pool: Two (Independent) Policy Decisions

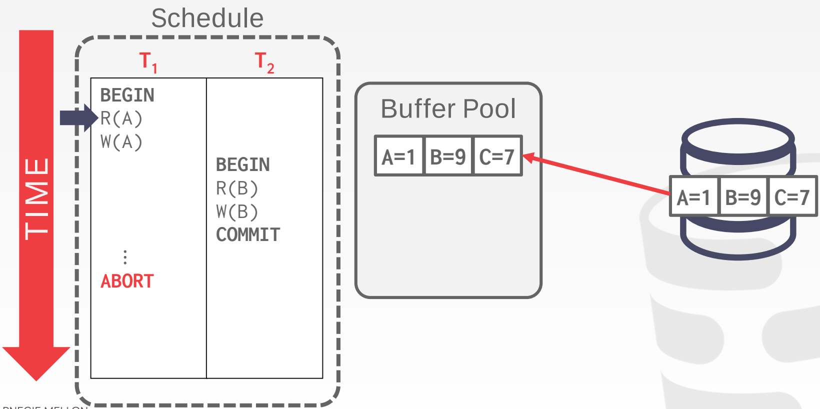
- Steal policy: whether the DBMS allows an uncommitted tx to overwrite the most recent committed value written to disk
  - Boolean-valued: "steal" means the DBMS will allow the write even though it's being performed by an uncommitted transaction
- Force policy: whether the DBMS requires that all tx updates be written to disk before the tx commits
  - Boolean-valued: "force" means the DBMS will enforce this policy

Shadow Paging

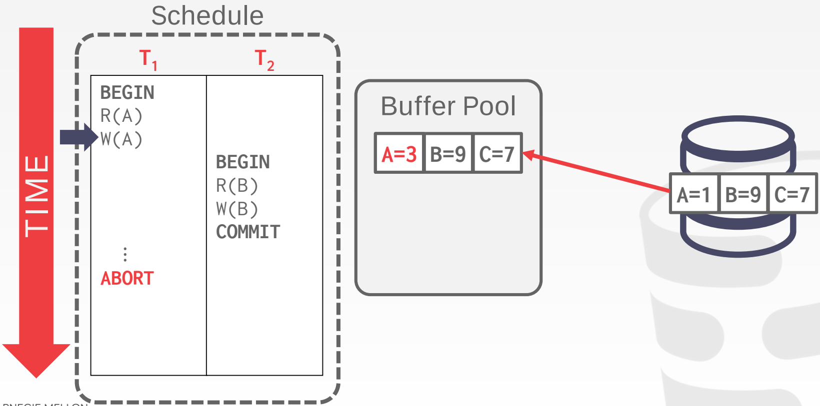
Introduction

- The shadow-paging recovery technique is an example of a "no-steal + force" selection from the recovery menu that we just discussed
- It's not commonly used in real systems because it has serious disadvantages
- But: we'll still spend time discussing shadow-paging because
  - It nicely illustrates the implications of selecting one or the other recovery policy options
  - It can be implemented in a straightforward fashion, may be useful for your **OATSdb** project ☺
- We'll first walk-through a scenario to illustrate what the "no-steal + force" terms mean
- Followed by a discussion of the shadow paging implementation

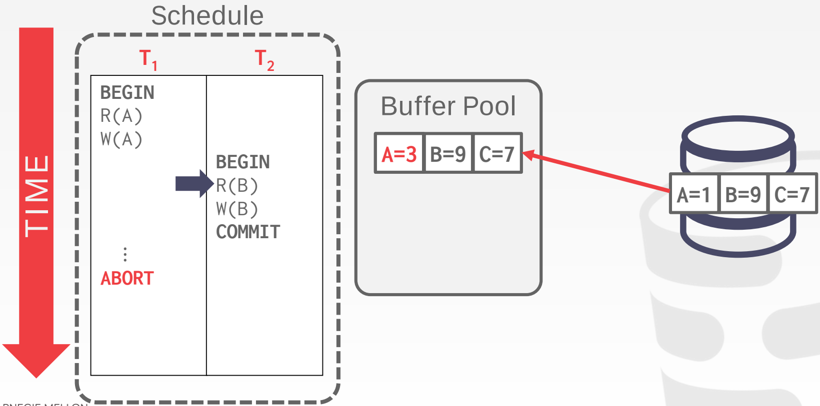
NO-STEAL + FORCE



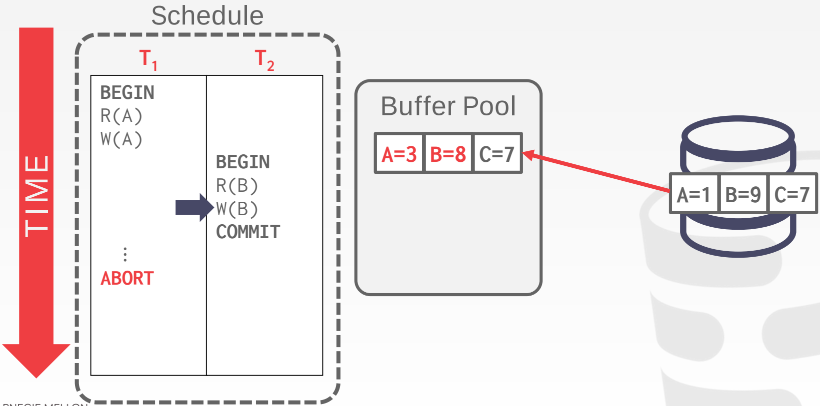
NO-STEAL + FORCE



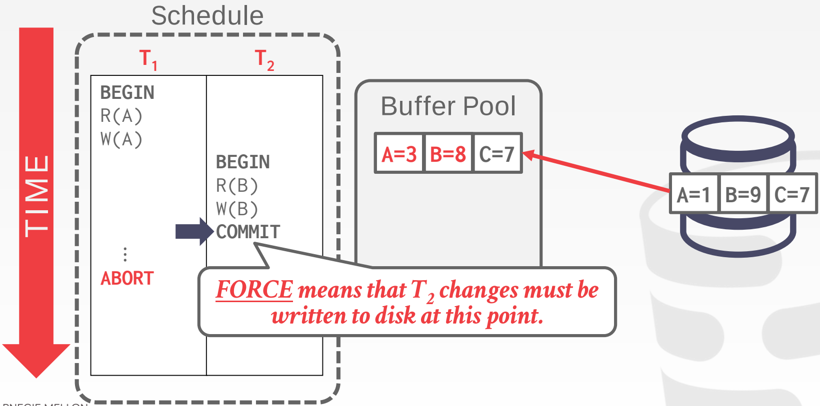
NO-STEAL + FORCE



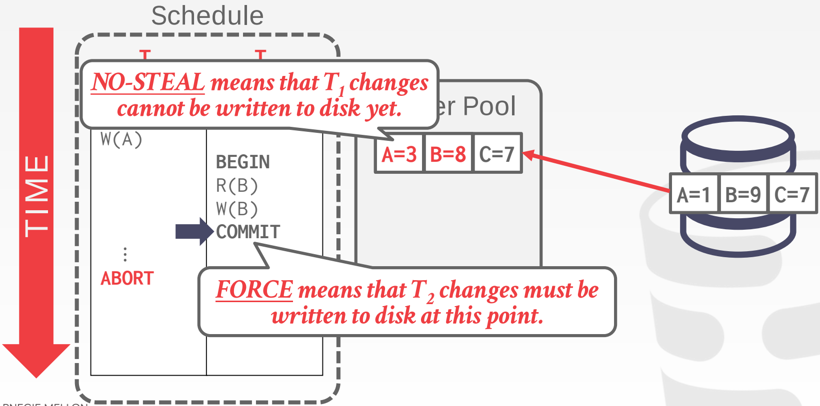
NO-STEAL + FORCE



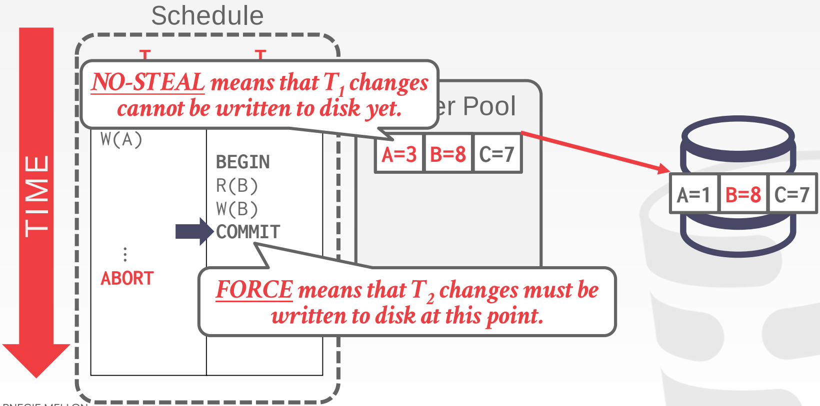
NO-STEAL + FORCE



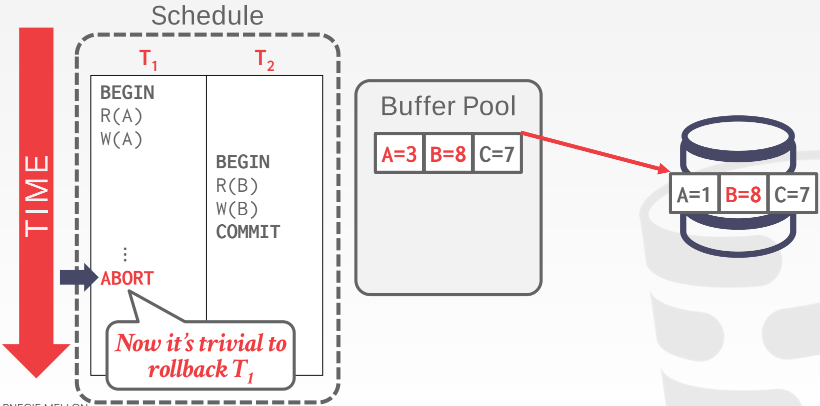
NO-STEAL + FORCE



NO-STEAL + FORCE



NO-STEAL + FORCE



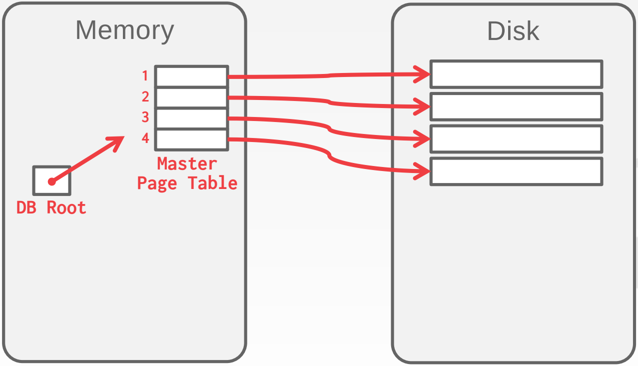
Advantages of “No-Steal + Force”

- Easy to implement ☺
- Tx abort? No problem, changes made by the tx were not written to disk, so nothing has to be “undone”
- Tx commit? No work has to be “redone” because all works is guaranteed to be written to disk at commit time
- Shadow paging: an implementation technique for “no-steal + force”
- DBMS maintains two copies of the database: master copy and shadow copy
- Tx updates are only applied to the shadow
- At commit time, DBMS atomically transforms shadow copy into the new master

Shadow Paging: Key Idea

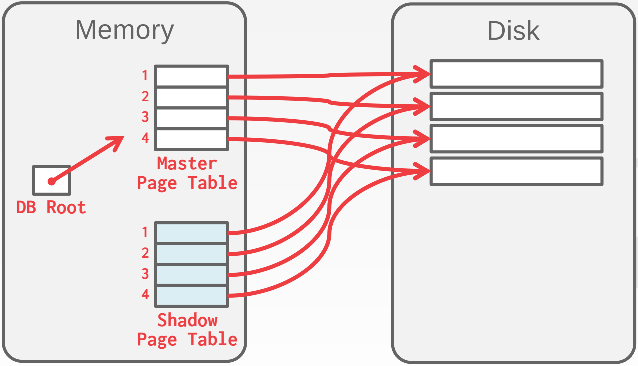
- (We're ignoring (for now) the disadvantages of the shadow paging approach, focusing on the *"at commit time, atomically transform shadow copy ⇒ master copy"* idea)
- It's not enough to have two copies of the database: some state has to record which copy is the current master copy!
- We'll refer to this piece of state as the "database root": it points to the current master copy
- At commit time, "swizzle the pointer" to point to the (current) shadow copy
  - The master & shadow have now switched roles in an atomic operation
- Before "commit", none of the transaction's updates were part of the "master copy"
  - They were written to disk, but not (conceptually) the "database disk"
- After "commit": the transaction's updates are now part of the "database disk"

SHADOW PAGING – EXAMPLE



Txn T_1

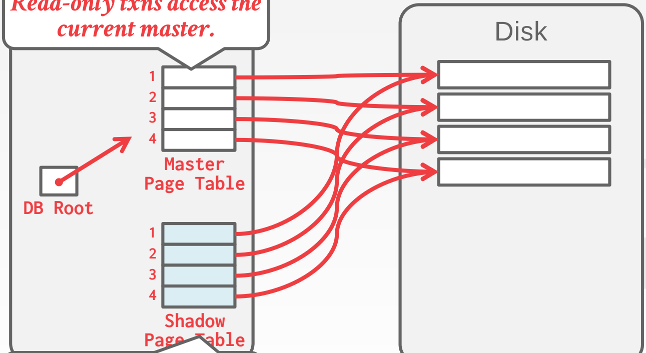
SHADOW PAGING – EXAMPLE



Txn T_1

SHADOW PAGING – EXAMPLE

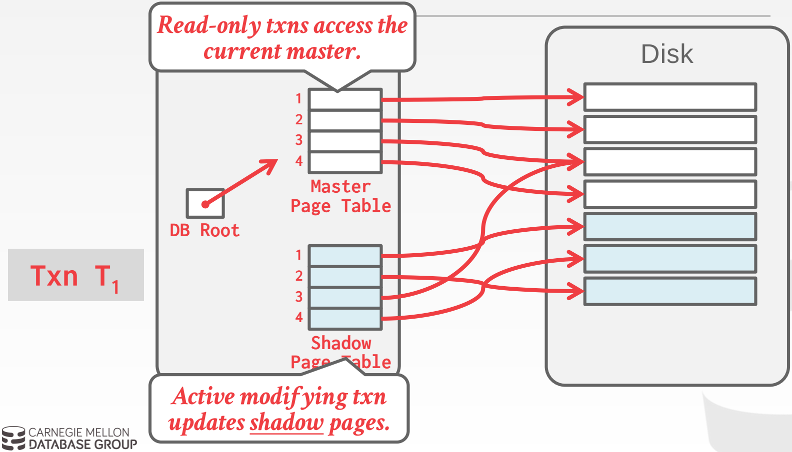
Read-only txns access the current master.



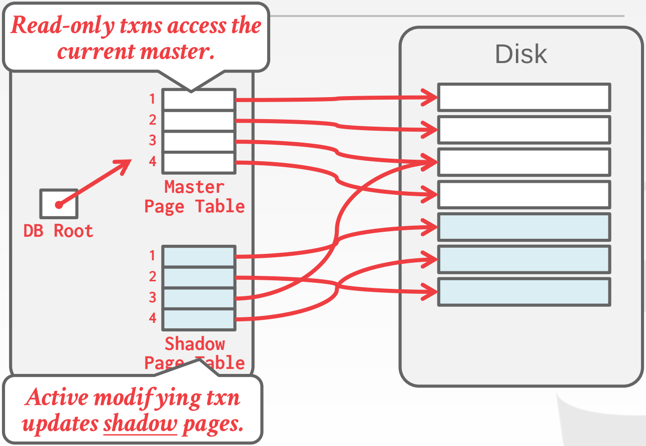
Txn T_1

Active modifying txn updates shadow pages.

SHADOW PAGING – EXAMPLE



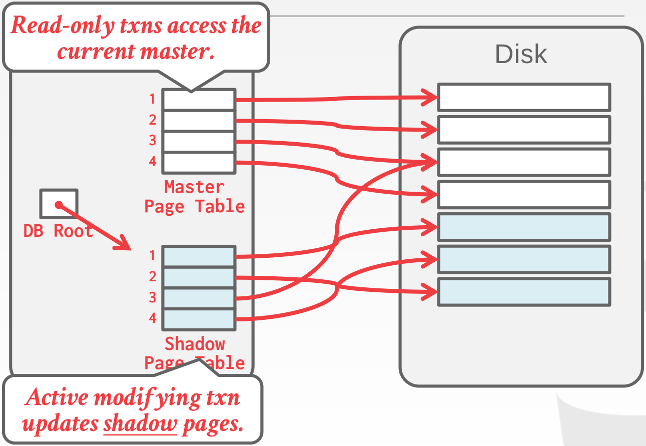
SHADOW PAGING – EXAMPLE



Txn T_1

COMMIT

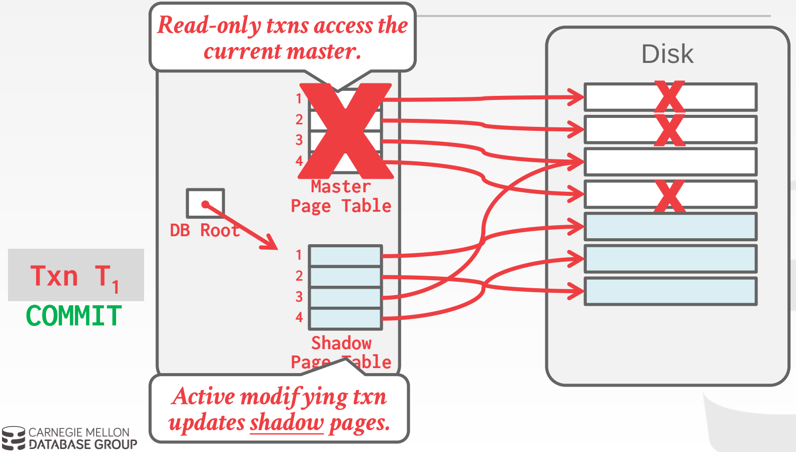
SHADOW PAGING – EXAMPLE



Txn T_1

COMMIT

SHADOW PAGING – EXAMPLE



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

OATSdb v2 Milestone: Providing Persistence to a Main-Memory Database

OATSdb: Where Are We Now?

- v1 milestone gave us a functional main-memory database
  - Including transactions!
- We used a lock-based technique to implement concurrency control
- We combined the lock-based technique with timeouts
  - Txs that block for more than the “timeout” period are rolled back by the DBMS
  - DBMS “wakes up” the blocked tx when relevant lock is released by the previous tx

OATSdb: Where Are Going?

- v2 will add persistence to **OATSdb**
- Currently: all data are lost when **OATSdb** (Java) process exits
- We want data to persist even after the process exits
  - When we restart **OATSdb**, from data perspective, we want to be in exactly the same state as when we exited

Approach: Use the File System

- Q: what storage media can we use to provide persistence?
- Note: no <u>fundamental</u> reason why main-memory is volatile and files are persistent …
- (When I joined IBM, group had just wound down a project that built a transactional system using *fault-tolerant, non-volatile main-memory*)
- Computer systems legacy: main-memory is tied to a non-persistent process model
- That said: files are persistent, main-memory is not ☹
  - For v2, we'll have to integrate disk files into **OATSdb**

v2 In a Nutshell

- Use Java Serialization to store object as serialized bytes in a file
  - v1 only uses serialization to create a byte stream
  - v2 writes that byte stream to a file!
- Each Map associated with its own file
- Transaction commit → serialize to file
- **OATSdb** process startup: deserialize files (if any) into *main-memory*

Not That Simple

- If all we had to do was provide persistence, this project milestone would be straightforward
- However: v2 must provide <u>transactional</u> persistence
  - State must be persisted on transaction boundaries
  - If **OATSdb** "goes down" in the middle of a transaction, system must come back up with the last committed transaction state
  - If **OATSdb** "goes down" after a transaction, system must come back up with the committed state
- Note: We're not going to worry about scenarios in which your laptop disk crashes ☺
  - Only worrying about process failures which cause the **OATSdb** process to exit

- Serializing a Java Map "per se" is insufficient
  - Remember: in **OATSdb**, a Map is accessed by name
    - So we must persist the name
  - But must also persist the Map key and value information
- In other words: must persist both Map data and Map metadata
- Example: must store information about *"which file* stores information about which Map?"

Disclaimer

- There are many possibilities for implementing the v2 requirements
- Examples
  - Encode metadata in the file name itself
  - Create a class that encapsulates both a Map instance and its metadata
  - Create a separate metadata class
- COM 3563 is all about empowering the individual!
- Plan: give you as much flexibility as possible
  - Lecture will spell out the minimal requirements

- Your v2 *DBMSImpl* implementation must implement the *ConfigurablePersistentDBMS* interface
- Motivation: would be nice if persistence function would be completely transparent to existing v1 **OATSdb** interfaces
  - I couldn't see a practical way to do this ☹
  - Example: tests need a way to do a "level set" on the database
- Note: in real-life™, such APIs would only be available to administrators
  - One approach: add credentials to the API plus a user registry
  - But: we won't worry about security issues for **OATSdb** ☺

ConfigurablePersistentDBMS APIs

```java
public interface ConfigurablePersistentDBMS
  extends ConfigurableDBMS
{
  /** Returns the disk usage in MB of this DBMS
   * instance.
   *
   * @return disk usage in MB for this DBMS
   */
  double getDiskUsageInMB();

  /** Delete all files and directories associated
   * with this DBMS instance from both disk and from
   * main-memory.  Effectively resets the database.
   *
   * IMPORTANT: the effects of this API on existing
   * transactions is undefined.  This method should
   * be invoked only when the system is quiescent.
   */
  void clear();
}
```

Concurrency

- Providing concurrency for v2 should be possible using the approach you used for v1
  - As usual, am assuming that you used the approach discussed in v1 lectures
- Key point: the main-memory components of **OATSdb** are unchanged in v2
  - No need for you to go to disk to fetch data except when starting the system
  - You still use main-memory locking and “shadow database” techniques to ensure that one transactional client doesn’t see another client’s activity

Atomicity

- The v2 complication involves atomicity ("all or nothing")
  - v1 only had to worry about main-memory data structures
  - As long as the system doesn't go down, you get the "all" behavior
  - If system goes down, you bring up a fresh, uninitialized database and you get the "nothing" behavior
- With v2, we're storing system data in a set of files
  - Note: implementation details don't matter for the point I'm making now
- If system goes down, what state are those files going to be in?
  - If not committing a transaction, may resemble v1 behavior
  - But: if DBMS fails while committing a transaction, will likely be a mess
    - Some files will be in the previous tx state
    - Some files will be in the committing tx state
- When the system comes back up, the database will now be corrupted ☹

OATSdb v2 Persistence & Atomicity: One Approach (I)

- Assume that database has three Maps: M_1, M_2, M_3 which are stored as separate files on disk
- DBMS maintains metadata in MD which is stored as a separate file on disk
  - Again: metadata details don't matter for this algorithm
- tx_1 successfully committed the Map state to disk
- tx_2 involves M_1 and M_2 and changes their state
- Key idea: maintain two sets of files
  1. First set: persisted state as of the last committed transaction
  2. Second set: current transaction state (that we wish to commit)
  3. "commit" implementation: system seamlessly (i.e., atomically) switches from one set to another

OATSdb v2 Persistence & Atomicity: One Approach (II)

Implementation details …

- A single M_j file
  - (in general: any system file, e.g., a metadata file)
  - is really a pair of files
    - One associated with “previous transaction state”
    - Other associated with “current transaction state”
- Not always a pair: e.g., if current transaction just created a new Map M_4
  - But conceptually a pair, with previous transaction containing the “null file”
  - Since the “previous transaction state” will not include a M_4 file
- Let’s call the first set of files $Files_1$ and the second set of files $Files_2$
  - Important: in our example, tx_2 can optimize such that M_3 file is actually the M_3 file in $Files_1$
    - Because tx_2 doesn’t have to create a new file to represents its M_3 state

OATSdb v2 Persistence & Atomicity: One Approach (III)

- Recall: the atomicity problem stems from the fact that Java doesn't provide us with an atomic, multi-file, write operation
  - Compare to `java.util.concurrent.atomic.AtomicLong`
- We'll assume that a single write operation to a single file behaves atomically
  - That is: with the "all or nothing" behavior that we want
- This isn't quite true, but can't be done correctly without special hardware support or (possibly) using the `java.nio.file.Files.move` API
  - Google this
  - So, we'll make this assumption to make our life easier ☺
- Trick: maintain a single file that stores a single "bit of information"
  - Is the system in tx_1 state or in tx_2 state?

OATSdb v2 Persistence & Atomicity: One Approach (IV)

Algorithm steps:

1. Write all $Files_2$ files to disk
2. Write all MD_2 files to disk
3. Write the "single bit file" to disk, changing its contents to point to the tx_2 state
   - Before `tx.commit`: file "said" tx_1
   - After `tx.commit`: file "says" tx_2

- What happens if the system crashes in the middle of the transaction?
- When the system comes back up will look at the "single bit file"
  - Either it says "tx_1" or it says "tx_2"
  - Based on this single bit of information, your system will deserialize the appropriate set of files from disk into main-memory
- We're done ☺

OATSdb v2 Persistence & Atomicity: One Approach (V)

- What happens when you want to commit a subsequent tx_3?
  - Do you have to keep incrementing (and persisting) a transaction "counter"?
- No: you're not persisting information about which transaction committed successfully!
- You're persisting information about which of two sets of system state are correct: $version_1$ or $version_2$
  - So you'll toggle back and forth between these two versions

Naming This Algorithm

- Can you think of a good name for this v2 atomicity algorithm?
- It generalizes the main-memory "shadow database" technique that we used to provide concurrency
- Here we're using a file-based "shadow database" to provide atomicity
- So: let's call it a shadow database algorithm ...

More accurately: this is an **OATSdb** rendering of the shadow paging recovery algorithm discussed in today's lecture ☺

Algorithm For System Startup

- This may be obvious, just making sure …
- The algorithm used during system startup is the mirror image of the commit algorithm

1. Read the "single bit file" from disk, determine whether it says tx_1 or tx_2. Without loss of generality, if that file says tx_1 …
2. Read all MD_1 files so as to load the appropriate metadata files for the most recent committed transaction
3. Read all $Files_1$ files, loading their state into main-memory Maps
4. Start servicing client requests ☺

Main-Memory & File Processing Code

- As mentioned, the file processing code gets invoked at two system events
  1. **OATSdb** startup
  2. Transaction commit
- The main-memory code continues to function as it did for v1
  - Handling Map API requests
  - Transaction rollback and related APIs
  - etc
- That said: the file processing code must be made aware of Map creation events
  - So it can persist that Map on transaction commit
  - Otherwise: no Maps will ever be persisted since the original Map set is the empty set ☺

Getting Comfortable With Java File APIs

- Keep in mind as you search the Web
  - Java started off with "old-style" File APIs (`java.io` package)
  - Then introduced newer "NIO" APIs (`java.nio` package, not directly affecting File APIs)
  - Then introduced even newer "NIO.2" File APIs (`java.nio.file` package)
- Some starter URLs
- Java I/O, NIO, and NIO.2
- Java NIO.2
- File manipulation
- I recommend using "NIO.2", but key point is to write utility methods that (1) you're comfortable with and (2) work ☺
- Note about serialization
  - v1 implementations only serialized Map values
  - v2 implementations must now serialize entire Map
    - So: key class must also be Java serializable

Testing: Some Tips

- Verify that your v1 test suite continues to pass for v2
- Add some tests for the *ConfigurablePersistentDBMS* APIs
- Give some thought to this question: can "unit tests" be used to validate your v2 implementation?
- Note: persistence only manifests after **OATSdb** process terminates and then restarts
  - Hard to construct JUnit test that sequence such events ☹
- As usual: I won't be surprised if you find a test framework or other solution to this problem
  - I took a different approach

Persistence Only Manifests On Restart

Basic Persistence Test

Use e.g., a shell-script to automate this sequence of steps

1. Bring up **OATSdb**
2. Create multiple Maps, multiple `Map.put` operations in a single tx
3. Commit tx
4. Shut down system
5. Bring it up again: are committed data still there?

- Perhaps have multiple "phases" (separated by *System.exit*)
- Each phase verifies previous phase's state is still there
  - Then change state of Map item or add new Map item
  - "lather, rinse, and repeat"

- Not that easy to test ☹
- We have to crash the system while it's in the middle of a commit operation
- Ideas?
- One approach
  1. Bring up **OATSdb**
  2. Create Map, insert Map entries, commit the tx
  3. Issue `System.exit()` concurrently with the commit operation
     - Threads can be scheduled to execute a Task with a given delay
  4. Bring system back up: are data corrupted?

Testing Transactional Persistence: Some Tips

- Your test should transform the database from one “well-known” state to another
  - And back again as necessary
  - Make it easy for test code to detect whether database in one or the other state
    - And to signal test failure if state is neither of these “well-known” states
- Note: I found that transactions that manipulated only a single Map and only small amounts of data, created such a small “footprint” that hard to have the `System.exit` intersect the `tx.commit`
  - Solution: increase the size of the transactional footprint

Testing Transactional Persistence: More Tips

- Even with a large transactional footprint, precise scheduling of the “DB crasher” thread is tricky
  - Do you know how fast your “commit” code is running?
- One approach: parameterize the delay in your test code
  - Then invoke the test with a range of delay parameters

Can You Spot v2 Limitations?

- Lecture already mentioned significant performance issues associated with shadow paging technique
- Here are some **OATSdb**-specific performance issues

- Inefficient to write an entire Map per commit
  - Disk operations take time
  - Serialization take time
- If only some Map state changed, we still have to serialize the entire Map
- Also: `commit` is a processing bottleneck
  - Can't commit concurrently
  - So bad performance for tx_1 commit slows down subsequent tx_i commits as well

Suggestions For Solving the Performance Problem?

- Serialization is too "heavy-weight"
- Instead: replace current (Hash)Map implementation with a B^+-Tree implementation of Map interface
  - The B^+-Tree index serves as the Map key index
- Result: now can write to disk at per-*MapEntry* granularity
  - B^+-Tree key is the Map key
  - B^+-Tree index is the address of the current *MapEntry*
- This is just food for thought, keep it in mind when we start digging in to database internals

Q: Is the following a v2 design problem?

- Every Map that is ever persisted to disk is brought into main-memory on startup
  - Tweak: could optimize to “load on demand”
- Assume: not enough room in main-memory to hold all disk Maps
- v2 design does not seem to be able to accommodate this (common) scenario!

Main-Memory Footprint: II

Can augment the API

- `setOffloadThreshold()` API might specify when items in main-memory must be offloaded to disk
- Doesn't have to be an API specifying ratio of main-memory to disk
  - Could simplify by using a global counter across all Maps rather than a per-Map threshold
  - Not track size of MapEntries: use the number of MapEntries as a proxy for main-memory footprint
  - Ignore (or not) memory used by "shadow database"

A: I claim that this is not a fundamental design issue because we can rely on the operating system use of virtual memory to swap main-memory Maps to and from disk as necessary

Today's Lecture: Wrapping it Up

DBMS Recovery: Terminology & Concepts

Shadow Paging

OATSdb v2 Milestone: Providing Persistence to a Main-Memory Database

Readings

- The textbook discusses transactional recovery and logging in Chapter 19

We'll discuss logging next lecture

- The textbook very briefly discusses "shadow paging" in a "Note" in Chapter 19.3.1