CS 839: Design the Next-Generation Database

Lecture 11: NVM2

Xiangyao Yu

2/25/2020

Announcements

Upcoming deadlines:

- Form groups: **Feb. 27**
- Proposal due: **Mar. 10**

Fill this Google sheet for course project information

- https://docs.google.com/spreadsheets/d/1W7ObfjLqjDChm49GqrLg49x6r4B28-f-PBpQPHX01Mk/edit?usp=sharing

Project Proposal

Use VLDB 2020 format

- https://vldb2020.org/formatting-guidelines.html

The proposal is **1-page** containing the following

- Project name
- Author list
- Abstract (1-2 paragraphs about your idea)
- Introduction (Why is the problem interesting; what's your contribution)
- Methodology (how do you plan to approach the problem)
- Task-list (Who works on what tasks of the project)
- Timeline (List of milestones and when you plan to achieve them)

Submit proposal by March 3 to https://wisc-cs839-ngdb20.hotcrp.com

Discussion Highlights

How does memory-mode affect the design?

- Will be faster when data fits in DRAM
- Need to take care of logging
- Memory mode can ease programming
- Just use existing main-memory DB without change

Advantage of app-direct mode over memory mode

- Directly manage replacement policy
- Larger aggregated memory space
- Logging can potentially be simplified
- Allows hot/cold data separation

How would you design NVM-DB differently?

- Better recovery structures that use NVM
- Minimize writes to NVM
- Use memory-mode or the dual-mode
- Replace SSD with NVM (cost?)
- Build LSM-tree based storage system

Today's Paper

Write-Behind Logging

Joy Arulraj
Carnegie Mellon University
jarulraj@cs.cmu.edu

Matthew Perron
Carnegie Mellon University
mperron@cmu.edu

Andrew Pavlo
Carnegie Mellon University
pavlo@cs.cmu.edu

ABSTRACT

The design of the logging and recovery components of database management systems (DBMSs) has always been influenced by the difference in the performance characteristics of volatile (DRAM) and non-volatile storage devices (HDD/SSDs). The key assumption has been that non-volatile storage is much slower than DRAM and only supports block-oriented read/writes. But the arrival of new non-volatile memory (NVM) storage that is almost as fast as DRAM with fine-grained read/writes invalidates these previous design choices.

This paper explores the changes that are required in a DBMS to

random write latency. During transaction processing, if the DBMS were to overwrite the contents of the database before committing the transaction, then it must perform random writes to the database at multiple locations on disk. It works around this constraint by flushing the transaction's changes to a separate log on disk with only sequential writes on the critical path of the transaction. This method is referred to as *write-ahead logging* (WAL).

But emerging *non-volatile memory* (NVM) technologies are poised to upend these assumptions. NVM storage devices support low latency reads and writes similar to DRAM, but with persistent writes

VLDB 2016

Today's Agenda

Intel Optane fault tolerance features

Database logging

Write-behind logging

NVM Fault Tolerance

CLFLUSH

- Flushes a single cache line out of cache (invalidate). Multiple CLFLUSH instructions execute one by one without concurrency.

CLFLUSHOPT

- Similar to CLFLUSH but multiple CLFLUSHOPT instructions can execute in parallel.

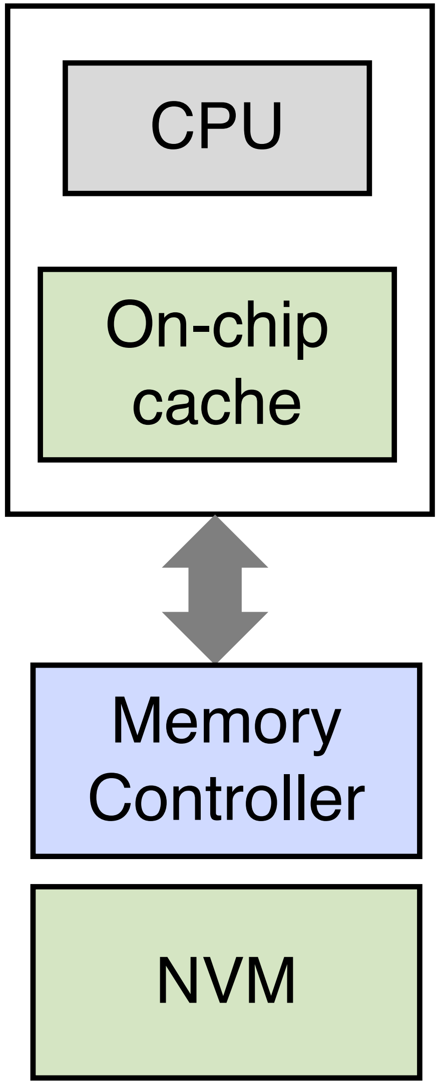
CLWB

- Cache line write back: Similar to CLFLUSHOPT but the cacheline can stay valid (in shared state) in the cache.

SFENCE

- Store fence. Ensure all previous stores are persistent once the instruction completes.

Asynchronous DRAM Refresh (ADR)



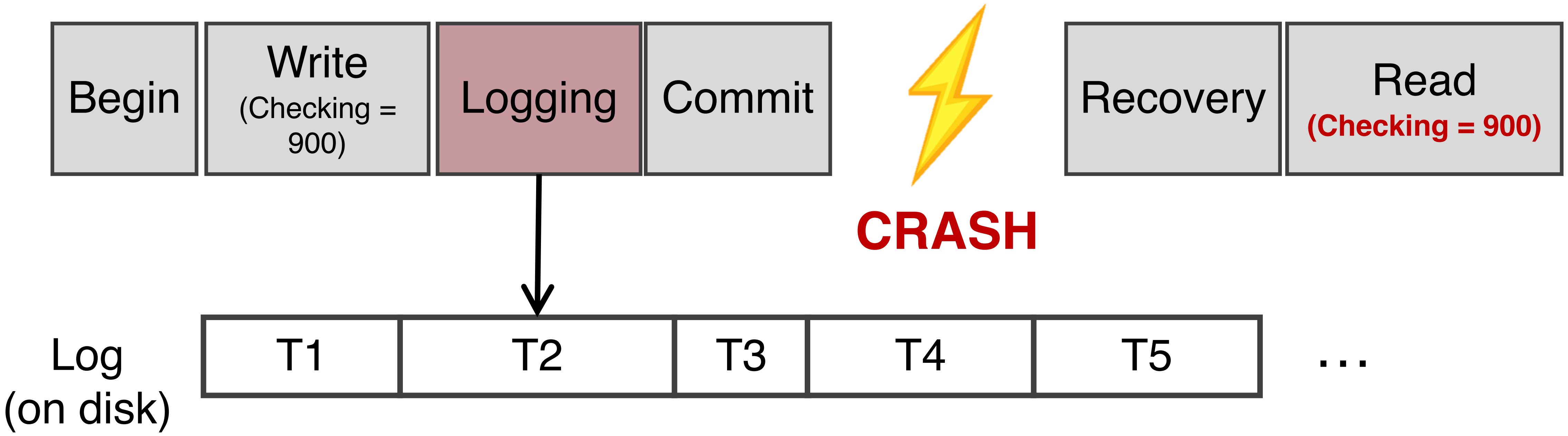
- Stores reaching the memory controller (MC) are guaranteed to be persistent
- Reducing latency of persistent store

Database Logging

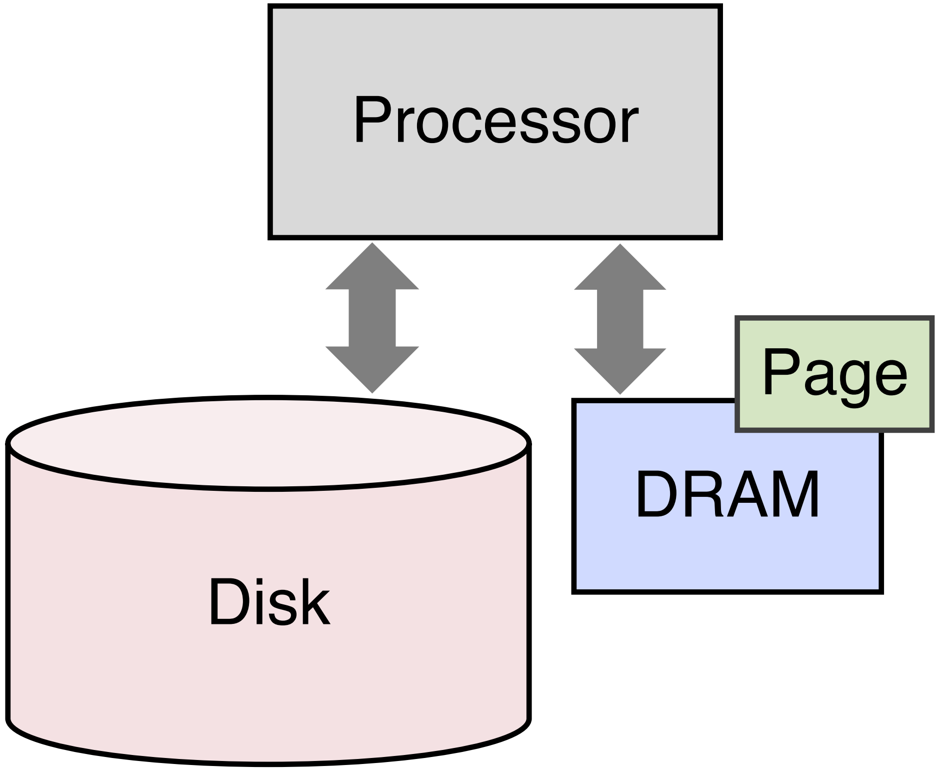
Recap: Write Ahead Logging (Lecture 2)

Log to persistent storage before commit

Initially
checking = 1000

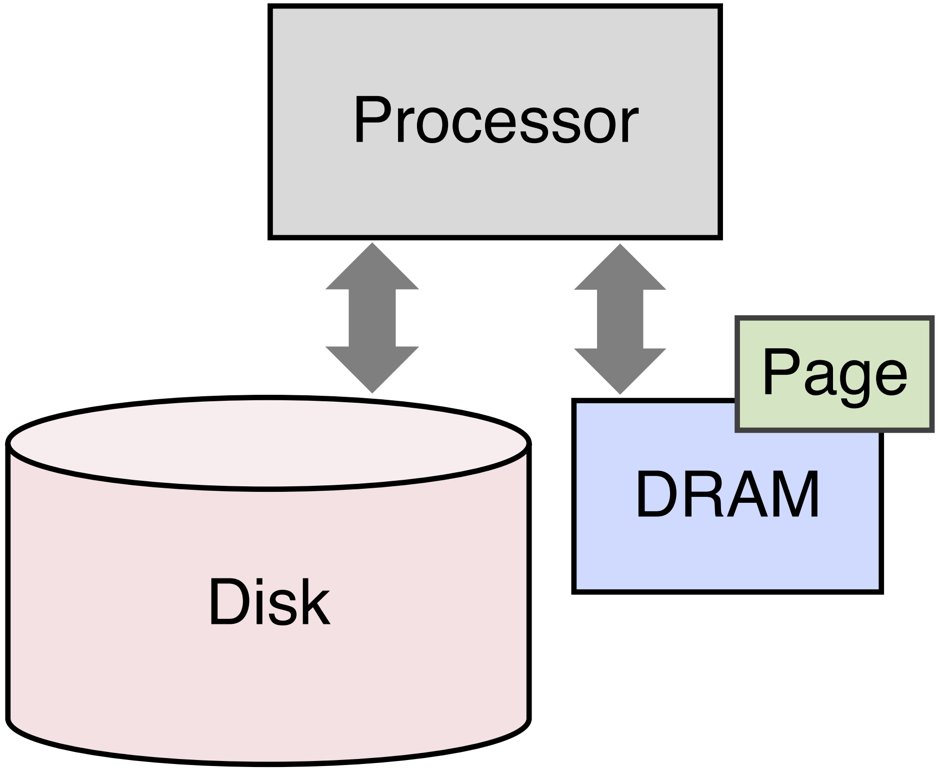


Logging in Disk-Based Databases



System must recover to a valid state no matter when crash occurs

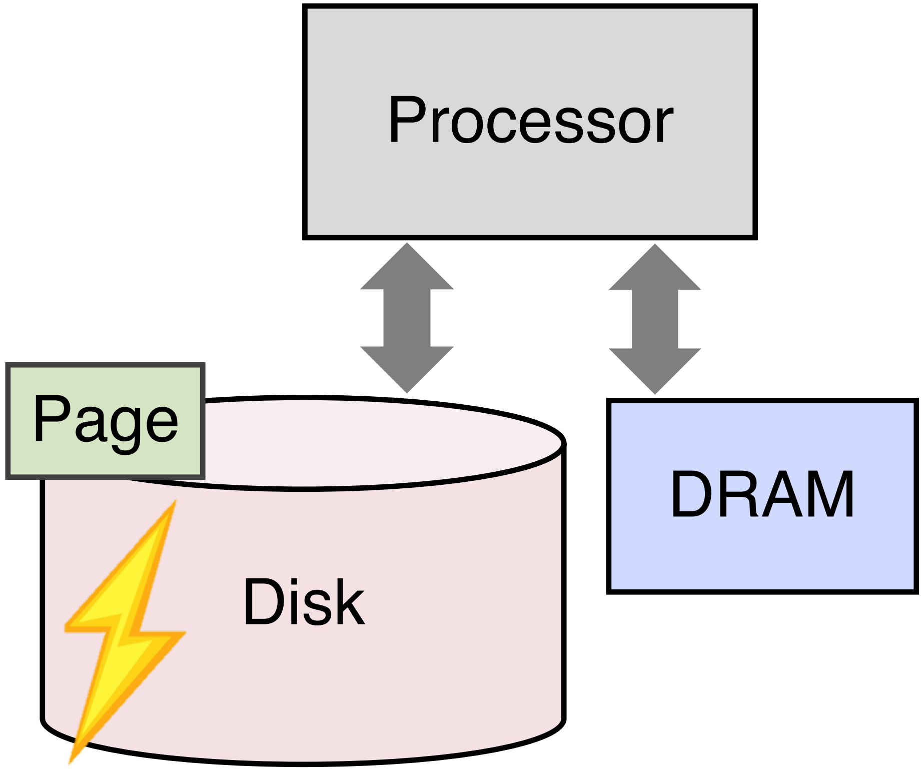
Logging in Disk-Based Databases



System must recover to a valid state no matter when crash occurs

How does a processor update a page?

Logging in Disk-Based Databases



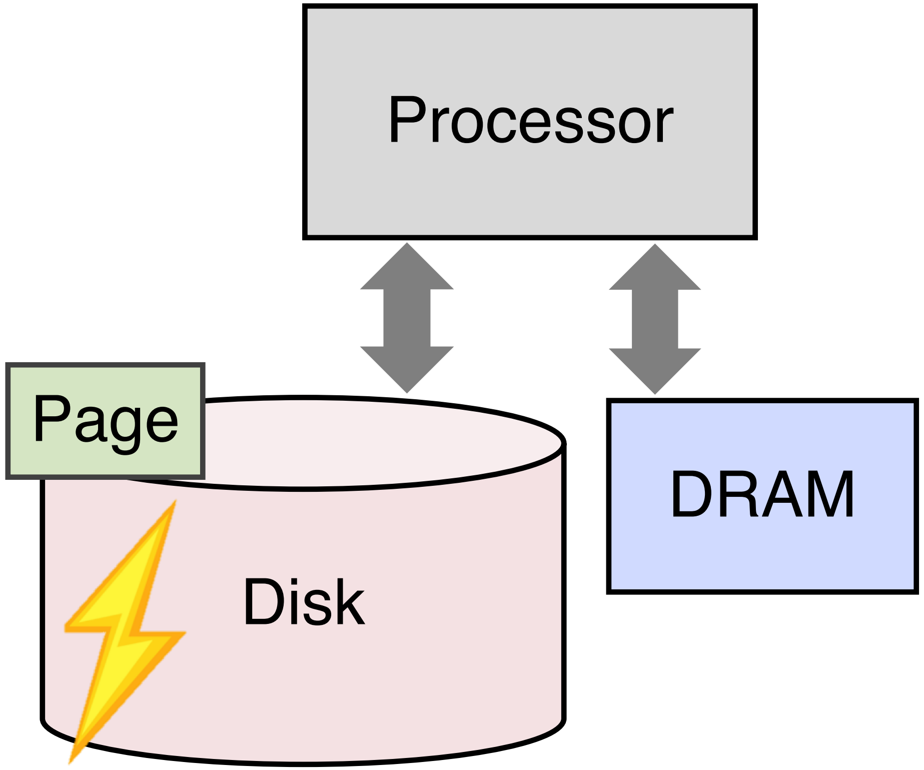
System must recover to a valid state no matter when crash occurs

How does a processor update a page?

What if the page is evicted to disk and the system crashes?

- The transaction may not have committed but the dirty page cannot be rolled back

Logging in Disk-Based Databases



System must recover to a valid state no matter when crash occurs

How does a processor update a page?

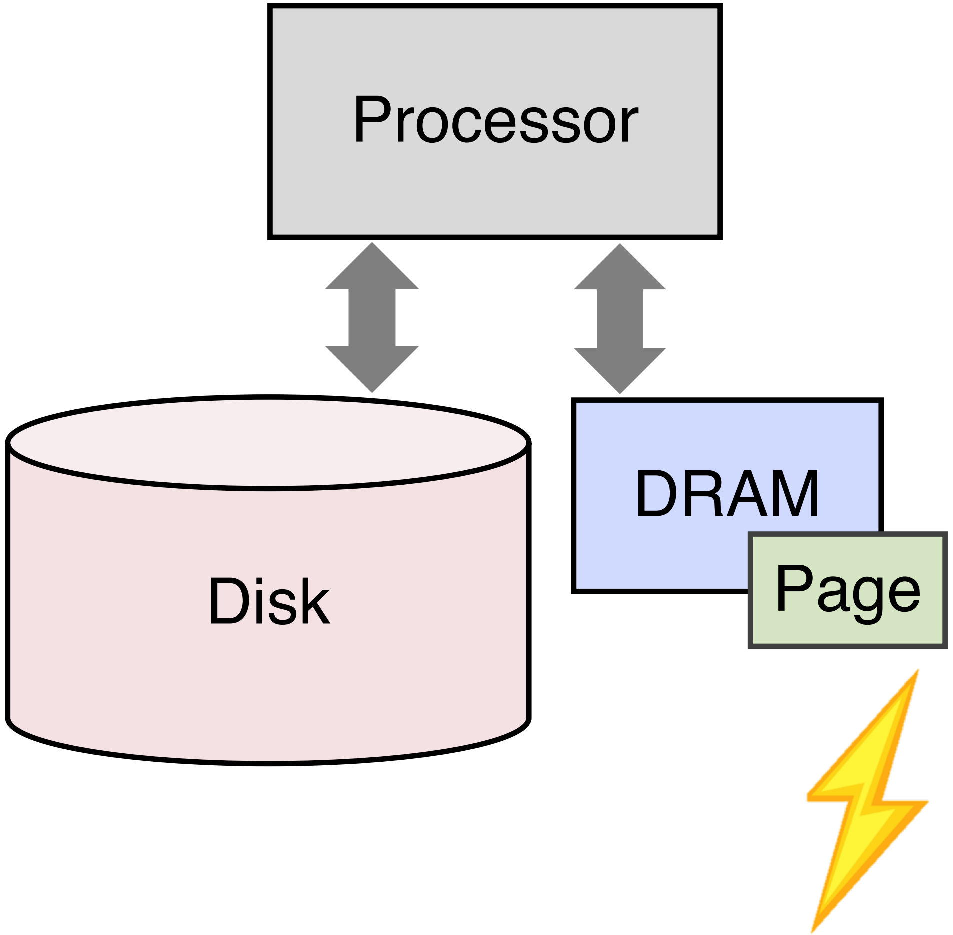
What if the page is evicted to disk and the system crashes?

- The transaction may not have committed but the dirty page cannot be rolled back

Design decision:

Steal vs. **no steal**

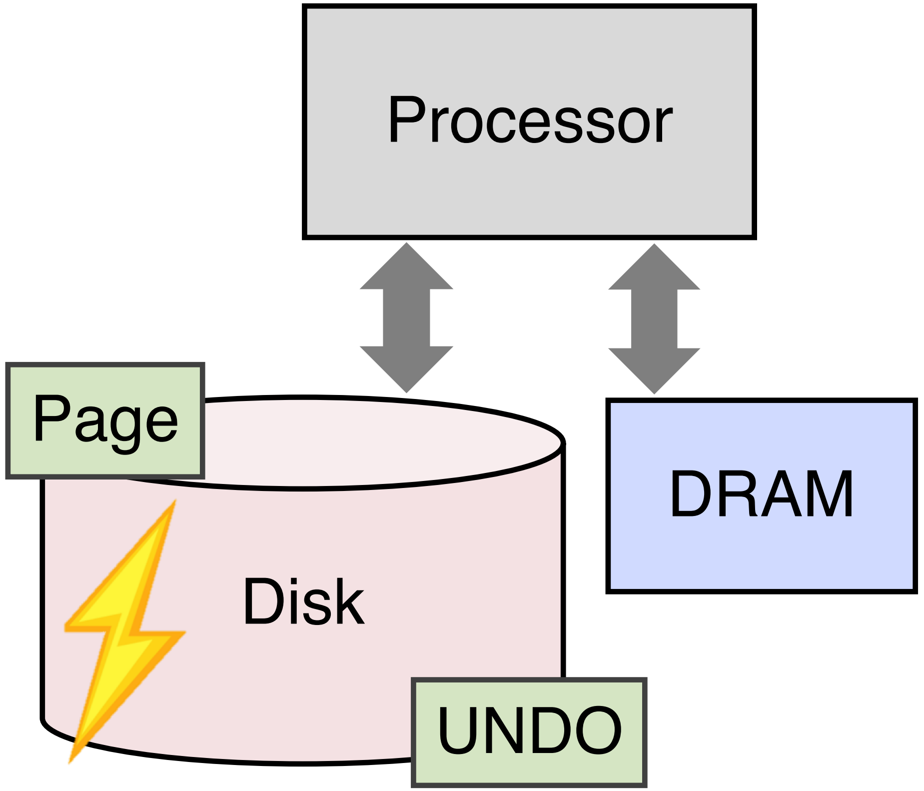
Steal vs. No Steal



No steal: dirty pages stay in DRAM

- Processor can directly update a page
- Main memory database

Steal vs. No Steal



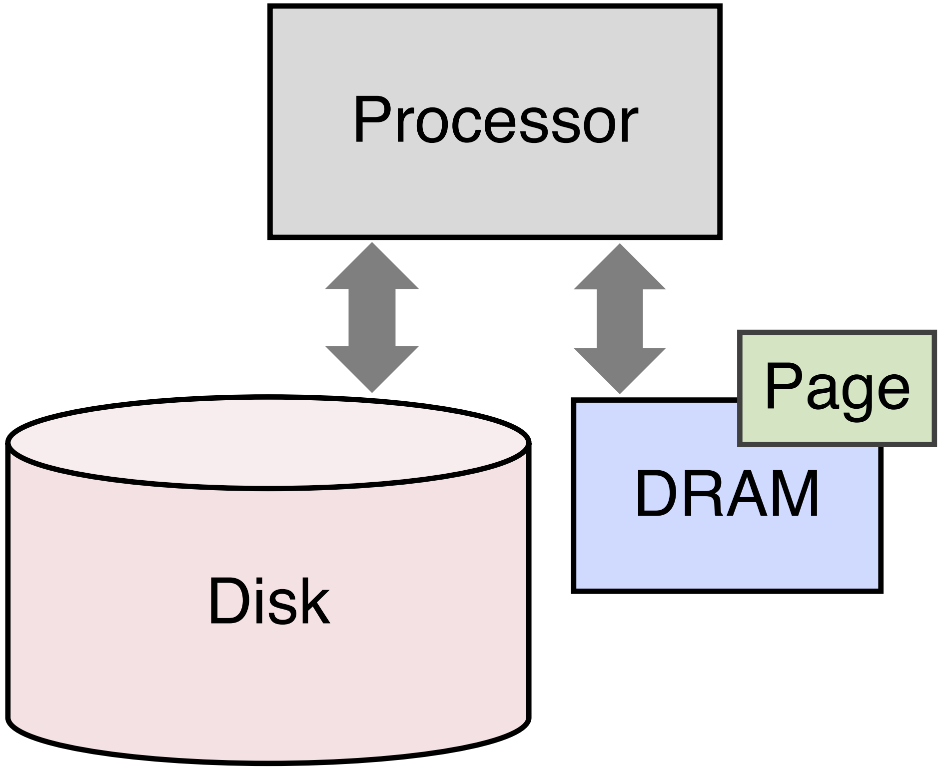
No steal: dirty pages stay in DRAM

- Processor can directly update a page
- Main memory database

Steal: dirty pages may overwrite pages on disk

- Must flush **UNDO log** (before-image) to disk before writing to the page

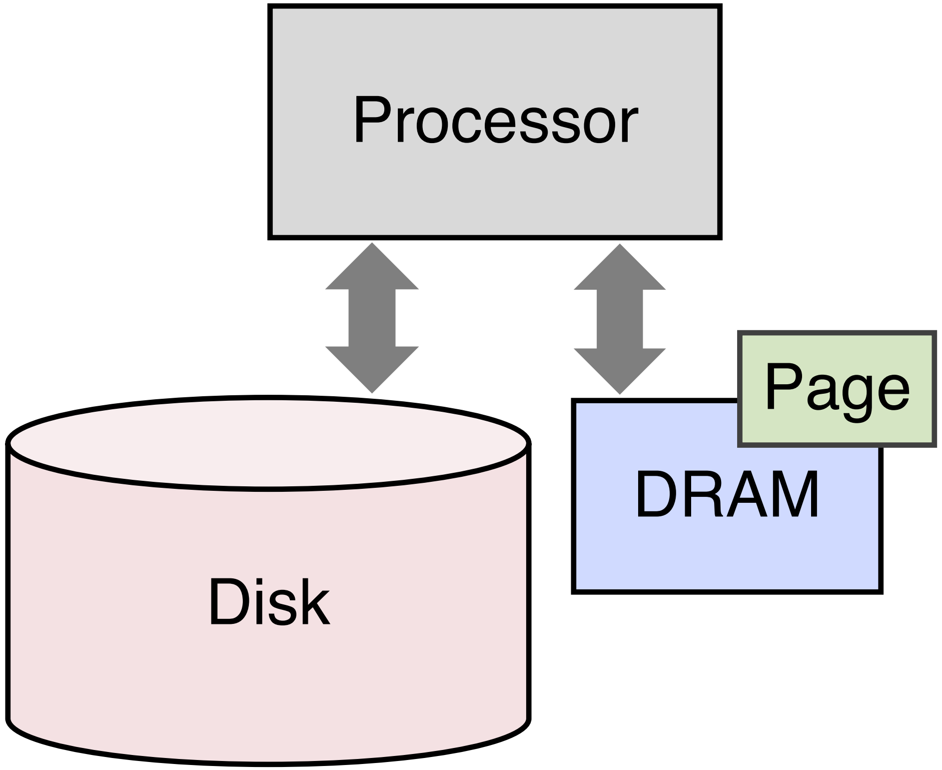
Logging in Disk-Based Databases



System must recover to a valid state no matter when crash occurs

How does a processor commit a transaction?

Logging in Disk-Based Databases



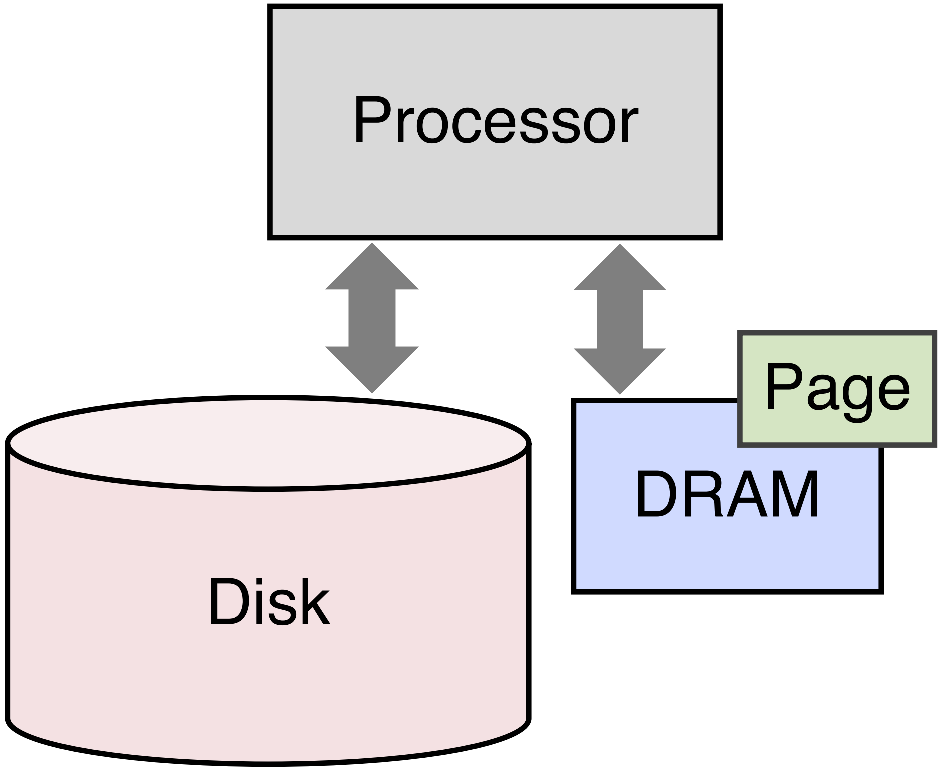
System must recover to a valid state no matter when crash occurs

How does a processor commit a transaction?

What if the system crashes before the page is evicted to disk?

- The transaction may have committed but the page is lost

Logging in Disk-Based Databases



System must recover to a valid state no matter when crash occurs

How does a processor commit a transaction?

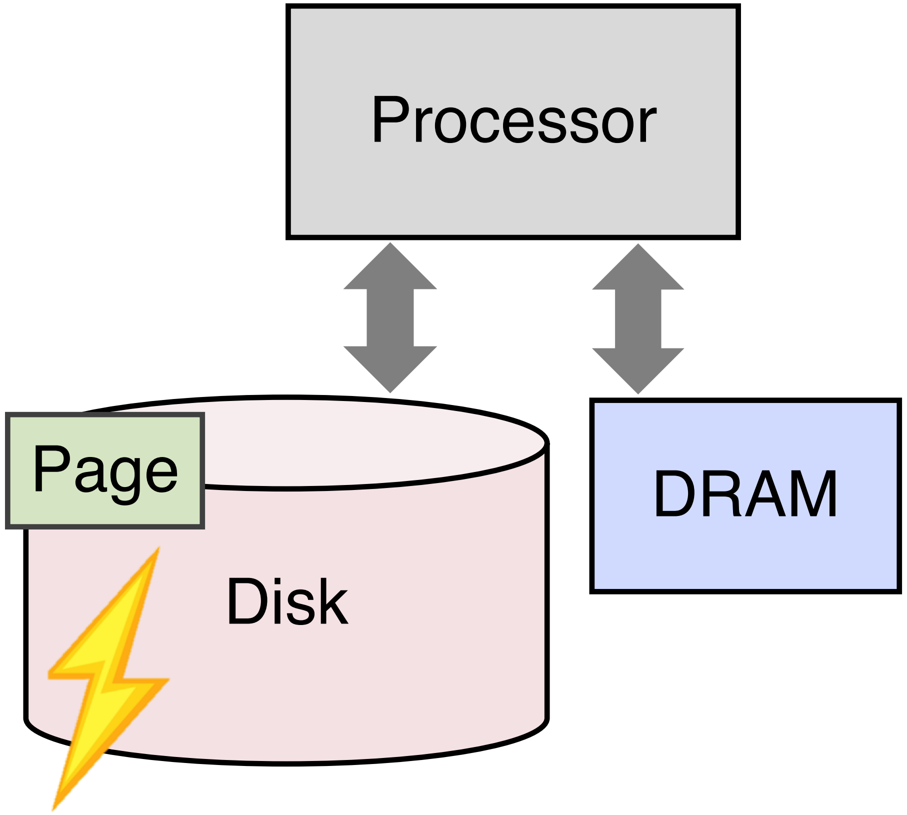
What if the system crashes before the page is evicted to disk?

- The transaction may have committed but the page is lost

Design decision:

Force vs. **no force**

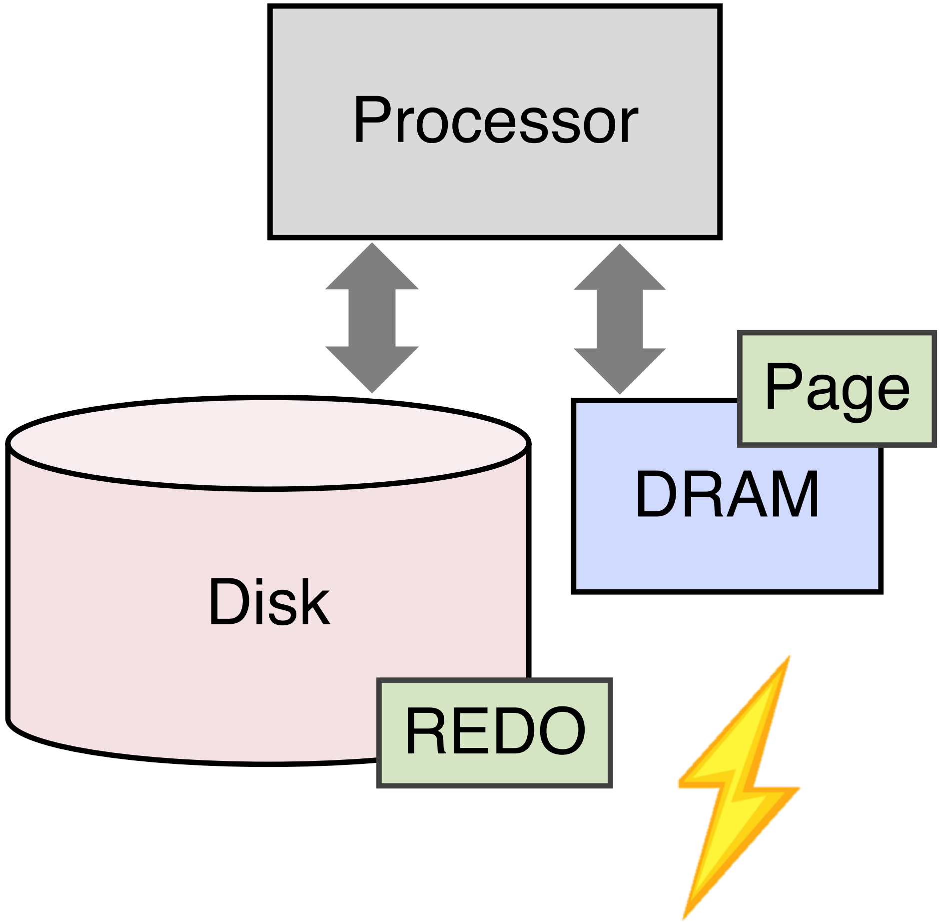
Force vs. No Force



Force: All modified pages written back to disk before commit

- Can commit transaction after all pages are forced to disk

Force vs. No Force



Force: All modified pages written back to disk before commit

- Can commit transaction after all pages are forced to disk

No Force: Modified pages may stay in main memory

- Flush **REDO log** (after-image) to disk before committing the transaction

Steal/No-Steal, Force/No-Force

	Steal	No Steal
Force	**UNDO** only	No REDO nor UNDO
No Force	**REDO and UNDO** logging (ARIES)	**REDO** only

[1] Philip Bernstein, Vassos Hadzilacos, Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, 1987

Steal/No-Steal, Force/No-Force

	Steal	No Steal
Force	**UNDO** only	No REDO nor UNDO
No Force	**REDO and UNDO** logging (ARIES)	**REDO** only

Disk-based DB

[1] Philip Bernstein, Vassos Hadzilacos, Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, 1987

Steal/No-Steal, Force/No-Force

	Steal	No Steal
Force	**UNDO** only	No REDO nor UNDO
No Force	**REDO and UNDO** logging (ARIES)	**REDO** only
	Disk-based DB	**Main memory DB**

[1] Philip Bernstein, Vassos Hadzilacos, Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, 1987

Steal/No-Steal, Force/No-Force

	Steal	No Steal
Force	**UNDO** only	No REDO nor UNDO
No Force	**REDO and UNDO** logging (ARIES)	**REDO** only

NVM DB (Force row)

Disk-based DB (Steal / No Force)

Main memory DB (No Steal / No Force)

[1] Philip Bernstein, Vassos Hadzilacos, Nathan Goodman, *Concurrency Control and Recovery in Database Systems*, 1987

Steal/No-Steal, Force/No-Force

	Steal	No Steal
Force	**UNDO only**	No REDO/UNDO
No Force	REDO and UNDO	REDO only

UNDO only:

- Flush UNDO record before each update to database

Steal/No-Steal, Force/No-Force

	Steal	No Steal
Force	**UNDO only**	No REDO/UNDO
No Force	REDO and UNDO	REDO only

UNDO only:

- Flush UNDO record before each update to database
- After all updates are forced to persistent storage, flush COMMIT record
- Transaction commits after COMMIT record is persistent

Steal/No-Steal, Force/No-Force

	Steal	No Steal
Force	**UNDO only**	No REDO/UNDO
No Force	REDO and UNDO	REDO only

UNDO only:

- Flush UNDO record before each update to database
- After all updates are forced to persistent storage, flush COMMIT record
- Transaction commits after COMMIT record is persistent
- UNDO records of a transaction can be ignored after the transaction commits

Steal/No-Steal, Force/No-Force

	Steal	No Steal
Force	**UNDO only**	No REDO/UNDO
No Force	REDO and UNDO	REDO only

UNDO only:

- Flush UNDO record before each update to database
- After all updates are forced to persistent storage, flush COMMIT record
- Transaction commits after COMMIT record is persistent
- UNDO records of a transaction can be ignored after the transaction commits
- Recovery: UNDO uncommitted transactions

Steal/No-Steal, Force/No-Force

	Steal	No Steal
Force	UNDO only	**No REDO/UNDO**
No Force	REDO and UNDO	REDO only

All of a transaction's updates recorded in persistent storage in a single atomic operation

Steal/No-Steal, Force/No-Force

	Steal	No Steal
Force	UNDO only	**No REDO/UNDO**
No Force	REDO and UNDO	REDO only

All of a transaction's updates recorded in persistent storage in a single atomic operation

Shadow version algorithm (No UNDO, no REDO):

- Maintain two copies of directories (D^0 and D^1) that point to the location of records, use a master bit M to indicate the master copy
- Transaction writes all updates to unused location in persistent storage and update D^{1-M}
- Atomically M = 1-M
- Update D^{1-M}

Multi-Version Database

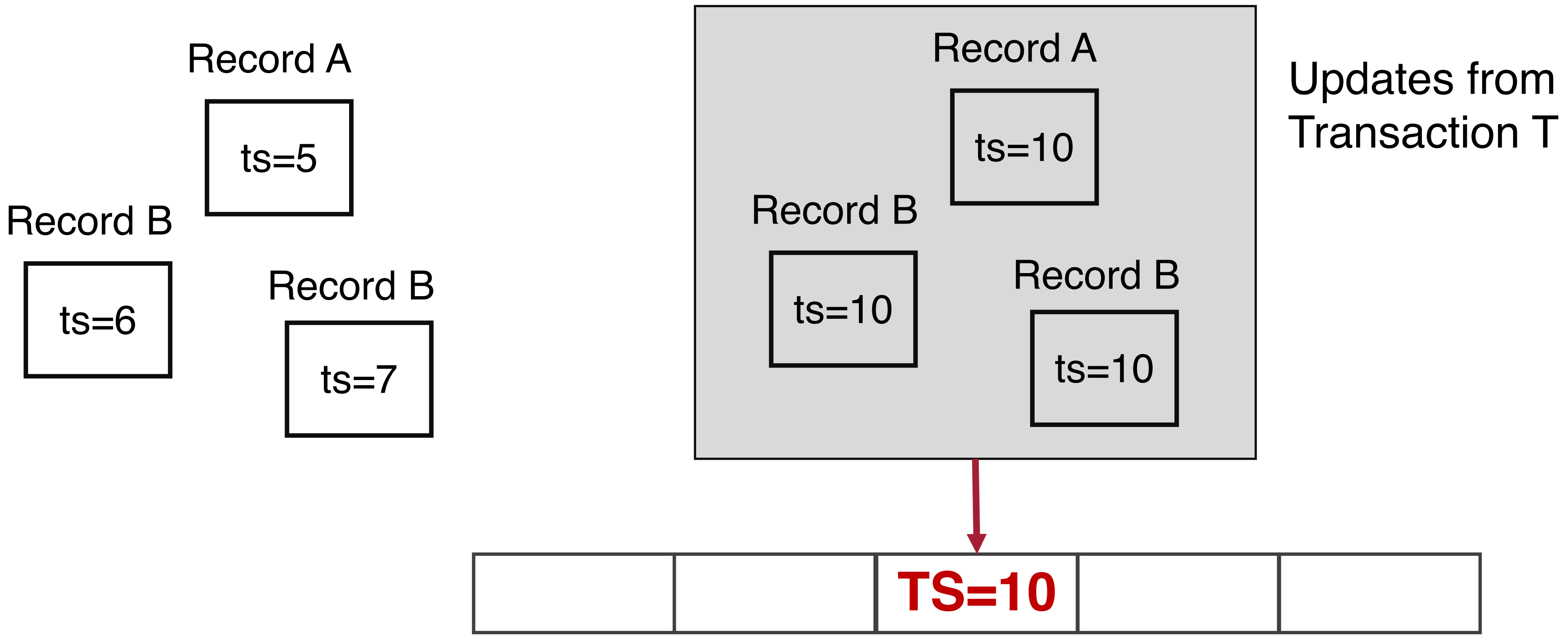
	Steal	No Steal
Force	UNDO only	**No REDO/UNDO**
No Force	REDO and UNDO	REDO only

All of a transaction's updates recorded in persistent storage in a single atomic operation

For an MVCC database, each update writes to a new version (with a transaction-specific version ID), which naturally achieves no steal

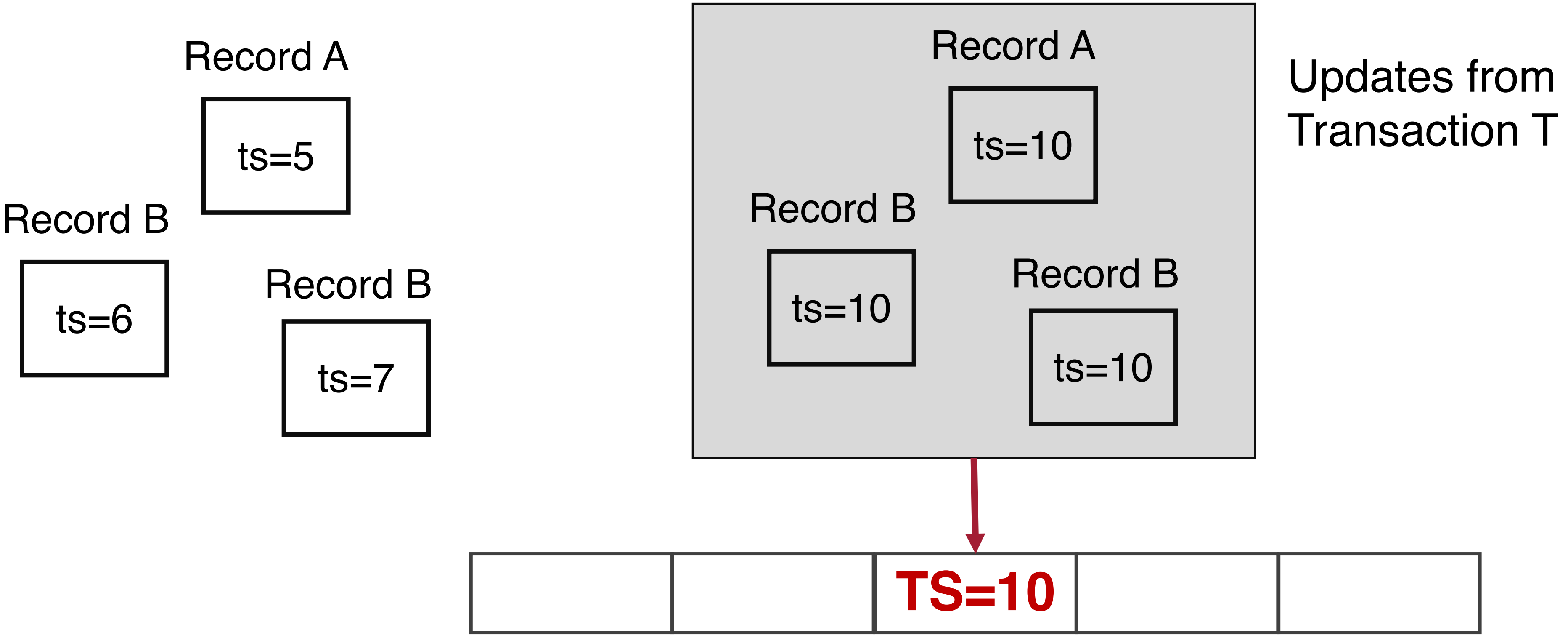
Now need a mechanism to make new versions of a transaction visible using a single **atomic** operation

Atomic Visibility



Atomically flush commit timestamp to log --> transaction commit

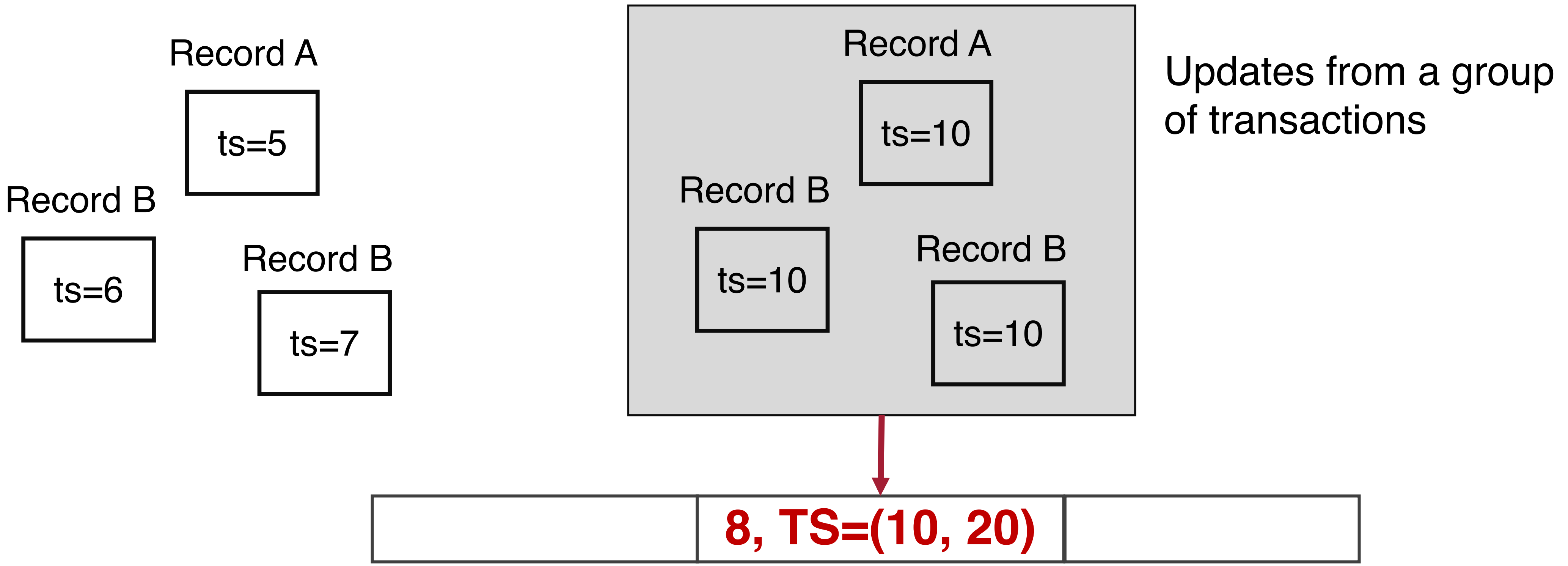
Atomic Visibility



Atomically flush commit timestamp to log --> transaction commit

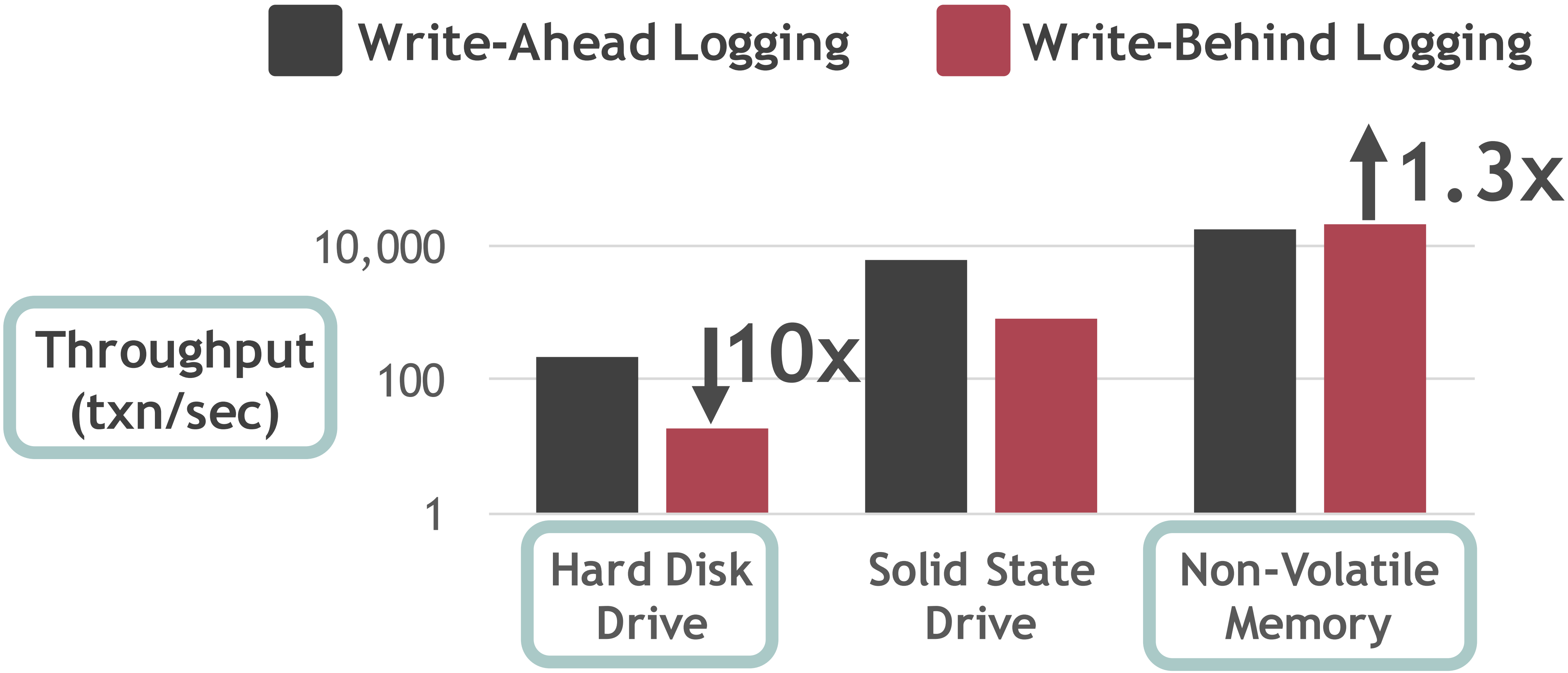
Downside: during recovery, difficult to decide what records have committed

Write-Behind Logging



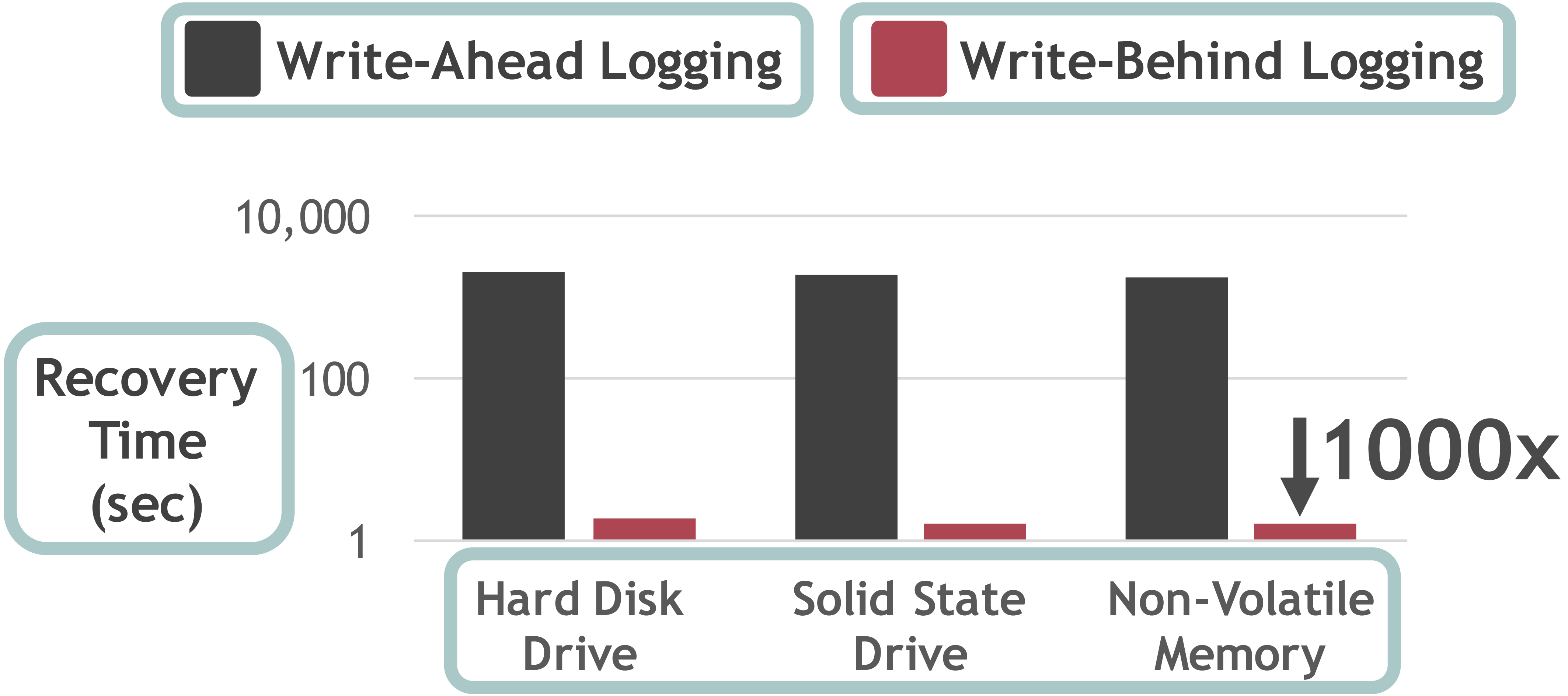
- Group commit TS=(c_p, c_d)
- All transactions before c_p have committed except listed outliers
- No transactions after c_d could have started
- During recovery, ignore versions between c_p and c_d and outliers

PERFORMANCE



Slide from https://www.cc.gatech.edu/~jarulraj/talks/2016.wbl.pdf

APPLICATION AVAILABILITY



Slide from https://www.cc.gatech.edu/~jarulraj/talks/2016.wbl.pdf

Summary

NVM: new device in the storage hierarchy

- Byte-addressable
- Non-volatile

Taking advantage of both byte-addressability and non-volatility to improve performance of fault tolerance

- Force + steal ---> UNDO only
- Force + MVCC ---> No UNDO, No REDO

NVM – Q/A

Physical vs. logical logging?

256 GB of DRAM (AWS *u-24tb1.metal* has 24 TB main memory)

Torn writes: Only part of a multi-sector update are written successfully to disk

Value-based vs. operational logging?

WBL in the three-tier BM architecture

Group Discussion

Distributed databases today require high availability (i.e., data replication) and recovery from a different machine; what does this mean for NVM-based fault tolerance?

What are the advantages of REDO only, UNDO only, and write-behind logging with respective to each other?

How does WBL work in the three-tier architecture from last lecture?

Before Next Lecture

Submit discussion summary to https://wisc-cs839-ngdb20.hotcrp.com

- **Deadline: Wednesday 11:59pm**

Submit review for

- Joins in a Heterogeneous Memory Hierarchy: Exploiting High-Bandwidth Memory
- [optional] Fundamental Latency Trade-offs in Architecting DRAM Caches