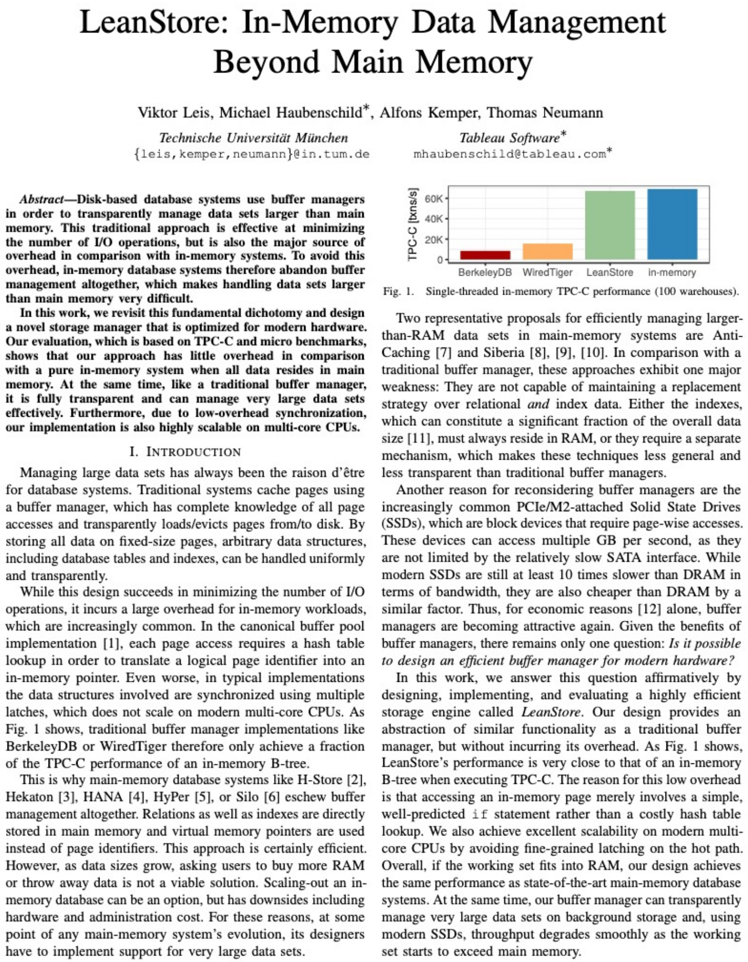
WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

CS 764: Topics in Database Management Systems

Lecture 5: Modern Buffer Management

Xiangyao Yu
9/21/2021

Today’s Paper: LeanStore



LeanStore: In-Memory Data Management Beyond Main Memory

Viktor Leis, Michael Haubenschild*, Alfons Kemper, Thomas Neumann

Technische Universität München
{leis,kemper,neumann}@in.tum.de

*Tableau Software**
mhaubenschild@tableau.com*

Abstract—**Disk-based database systems use buffer managers in order to transparently manage data sets larger than main memory. This traditional approach is effective at minimizing the number of I/O operations, but is also the major source of overhead in comparison with in-memory systems. To avoid this overhead, in-memory database systems therefore abandon buffer management altogether, which makes handling data sets larger than main memory very difficult.**

In this work, we revisit this fundamental dichotomy and design a novel storage manager that is optimized for modern hardware. Our evaluation, which is based on TPC-C and micro benchmarks, shows that our approach has little overhead in comparison with a pure in-memory system when all data resides in main memory. At the same time, like a traditional buffer manager, it is fully transparent and can manage very large data sets effectively. Furthermore, due to low-overhead synchronization, our implementation is also highly scalable on multi-core CPUs.

I. Introduction

Managing large data sets has always been the raison d’être for database systems. Traditional systems cache pages using a buffer manager, which has complete knowledge of all page accesses and transparently loads/evicts pages from/to disk. By storing all data on fixed-size pages, arbitrary data structures, including database tables and indexes, can be handled uniformly and transparently.

While this design succeeds in minimizing the number of I/O operations, it incurs a large overhead for in-memory workloads, which are increasingly common. In the canonical buffer pool implementation [1], each page access requires a hash table lookup in order to translate a logical page identifier into an in-memory pointer. Even worse, in typical implementations the data structures involved are synchronized using multiple latches, which does not scale on modern multi-core CPUs. As Fig. 1 shows, traditional buffer manager implementations like BerkeleyDB or WiredTiger therefore only achieve a fraction of the TPC-C performance of an in-memory B-tree.

This is why main-memory database systems like H-Store [2], Hekaton [3], HANA [4], HyPer [5], or Silo [6] eschew buffer management altogether. Relations as well as indexes are directly stored in main memory and virtual memory pointers are used instead of page identifiers. This approach is certainly efficient. However, as data sizes grow, asking users to buy more RAM or throw away data is not a viable solution. Scaling-out an in-memory database can be an option, but has downsides including hardware and administration cost. For these reasons, at some point of any main-memory system’s evolution, its designers have to implement support for very large data sets.

Fig. 1. Single-threaded in-memory TPC-C performance (100 warehouses).

Two representative proposals for efficiently managing larger-than-RAM data sets in main-memory systems are Anti-Caching [7] and Siberia [8], [9], [10]. In comparison with a traditional buffer manager, these approaches exhibit one major weakness: They are not capable of maintaining a replacement strategy over relational *and* index data. Either the indexes, which can constitute a significant fraction of the overall data size [11], must always reside in RAM, or they require a separate mechanism, which makes these techniques less general and less transparent than traditional buffer managers.

Another reason for reconsidering buffer managers are the increasingly common PCIe/M2-attached Solid State Drives (SSDs), which are block devices that require page-wise accesses. These devices can access multiple GB per second, as they are not limited by the relatively slow SATA interface. While modern SSDs are still at least 10 times slower than DRAM in terms of bandwidth, they are also cheaper than DRAM by a similar factor. Thus, for economic reasons [12] alone, buffer managers are becoming attractive again. Given the benefits of buffer managers, there remains only one question: *Is it possible to design an efficient buffer manager for modern hardware?*

In this work, we answer this question affirmatively by designing, implementing, and evaluating a highly efficient storage engine called *LeanStore*. Our design provides an abstraction of similar functionality as a traditional buffer manager, but without incurring its overhead. As Fig. 1 shows, LeanStore’s performance is very close to that of an in-memory B-tree when executing TPC-C. The reason for this low overhead is that accessing an in-memory page merely involves a simple, well-predicted `if` statement rather than a costly hash table lookup. We also achieve excellent scalability on modern multi-core CPUs by avoiding fine-grained latching on the hot path. Overall, if the working set fits into RAM, our design achieves the same performance as state-of-the-art main-memory database systems. At the same time, our buffer manager can transparently manage very large data sets on background storage and, using modern SSDs, throughput degrades smoothly as the working set starts to exceed main memory.

ICDE 2018

Agenda

Main-memory DB

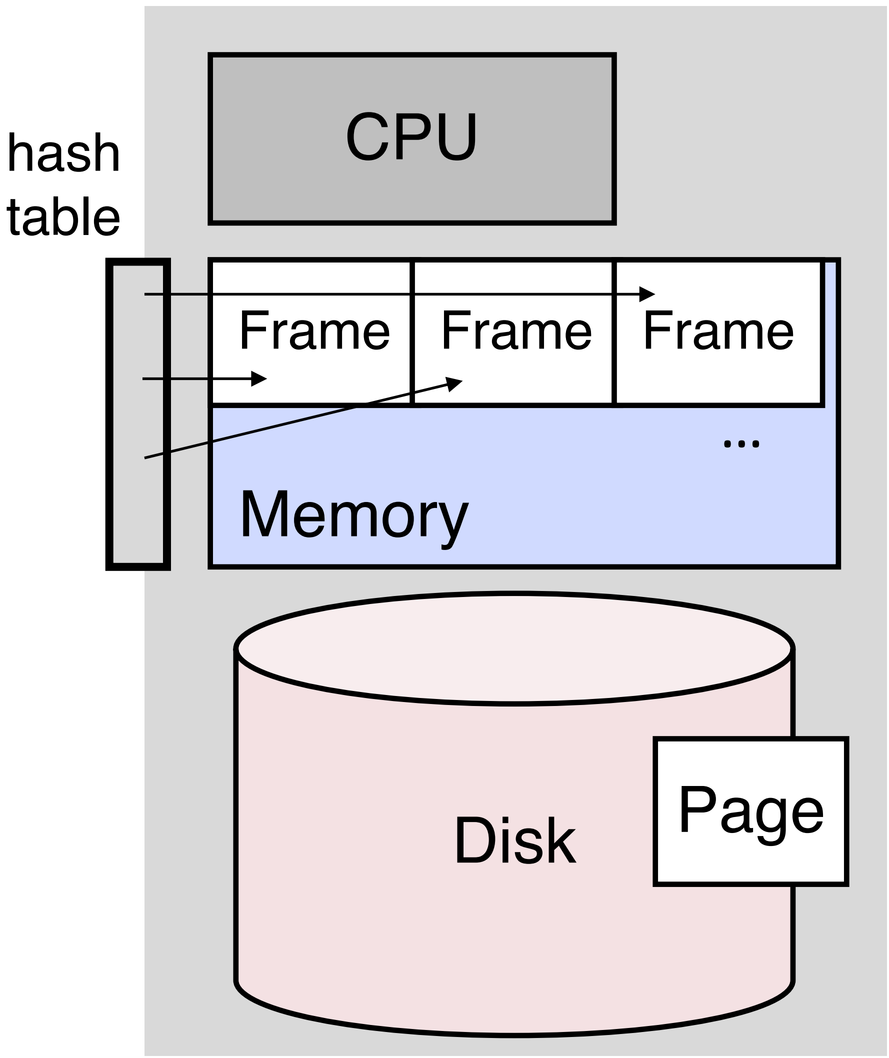
LeanStore design

- Pointer swizzling
- Page replacement
- Optimistic latching

Experiments

Fine-grained in-memory data management

Conventional DB Architecture



Page granularity: Data managed in page granularity

Indirection: Use page ID to lookup hash table to locate a page

Conventional DB Performance

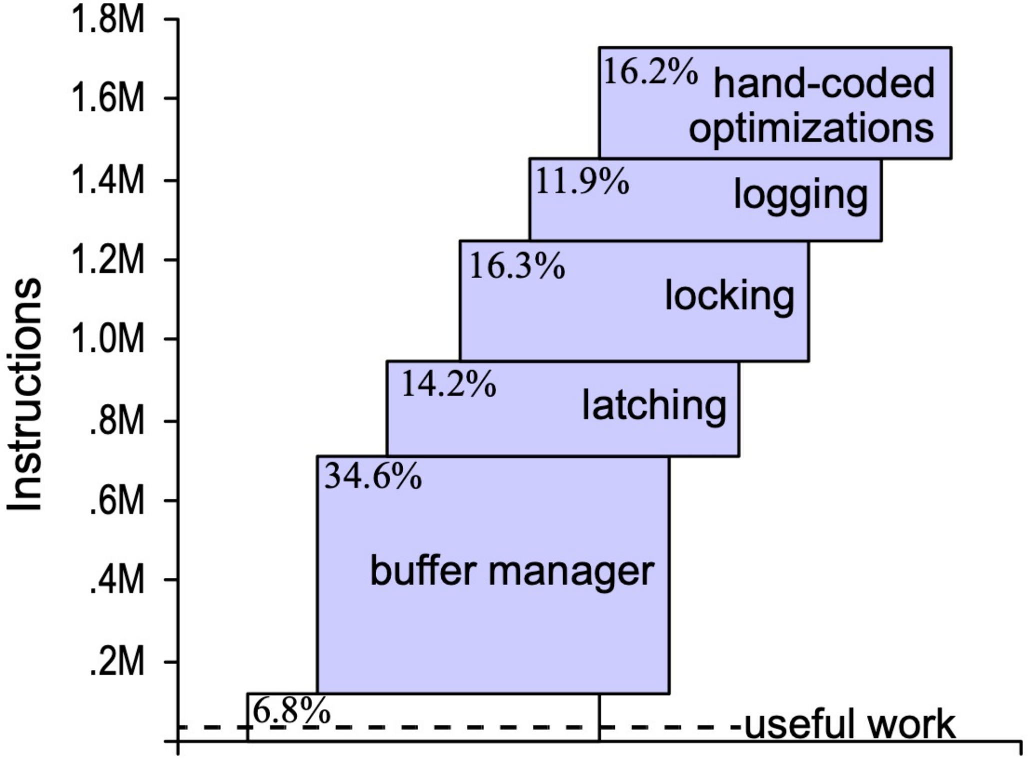


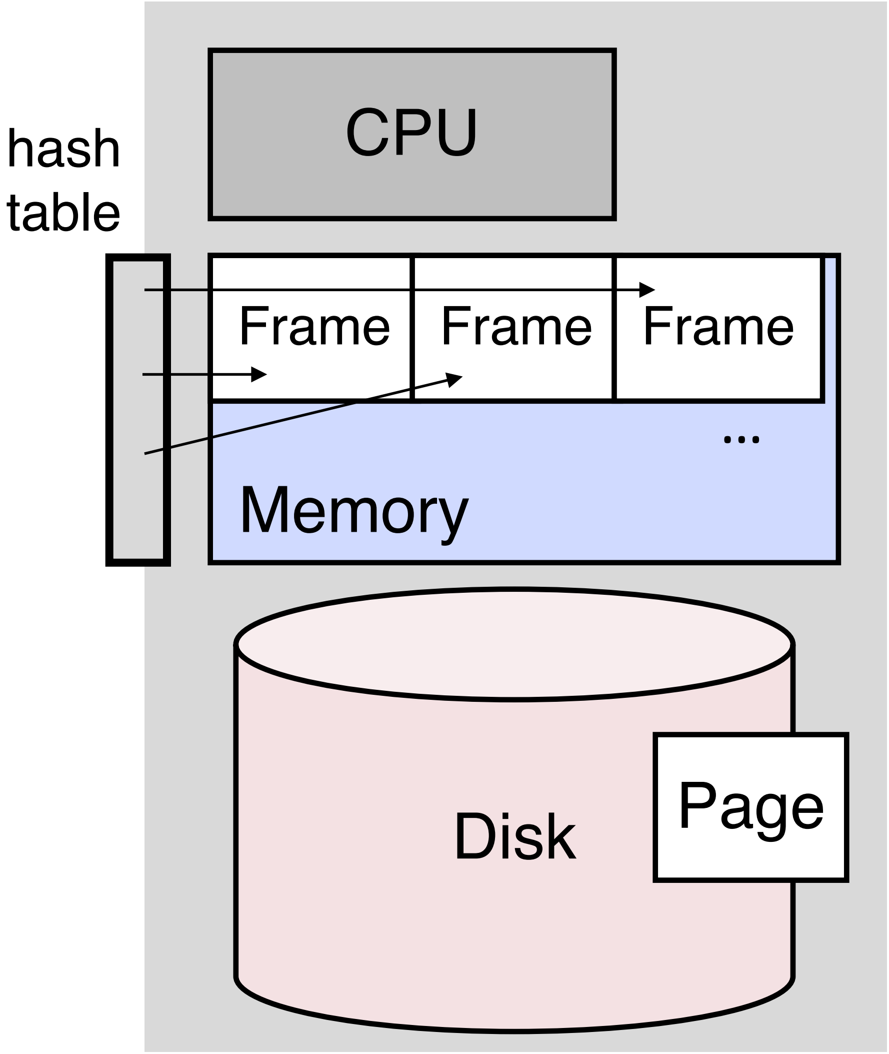
Figure 1. Breakdown of instruction count for various DBMS components for the New Order transaction from TPC-C. The top of the bar-graph is the original Shore performance with a main memory resident database and no thread contention. The bottom dashed line is the useful work, measured by executing the transaction on a no-overhead kernel.

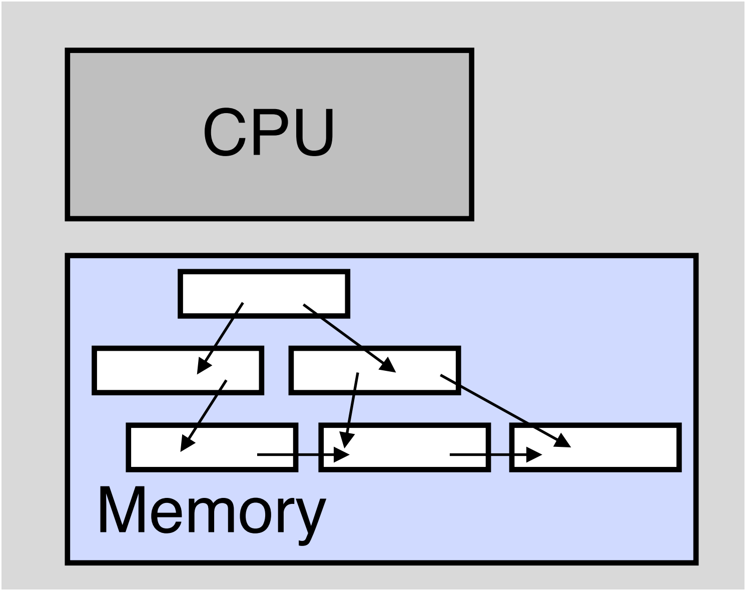
Only a small fraction of instructions execute useful work

Significant instruction count dedicated to buffer management

[1] Stavros Harizopoulos, et al., *OLTP Through the Looking Glass, and What We Found There*, SIGMOD 2008

Main-Memory DB Architecture

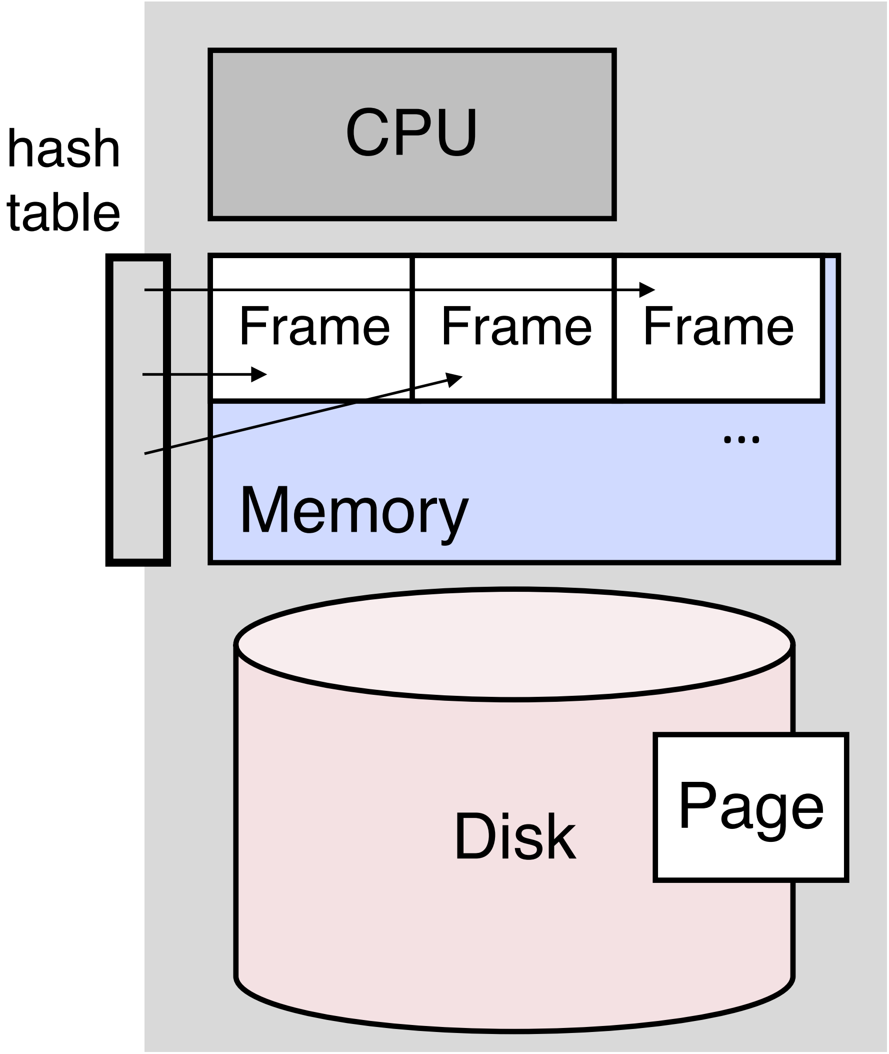


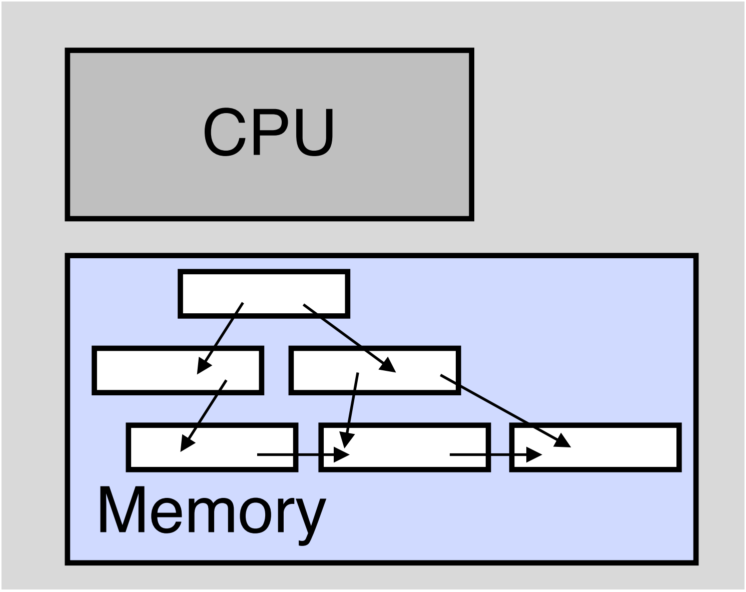


Fine-granularity: Fine-grained (e.g., tuple-level) data management

No Indirection: reference data following pointers

Main-Memory DB Architecture





Fine-granularity: Fine-grained (e.g., tuple-level) data management

No Indirection: reference data following pointers

⇒ **Focus of this paper**

Agenda

Main-memory DB

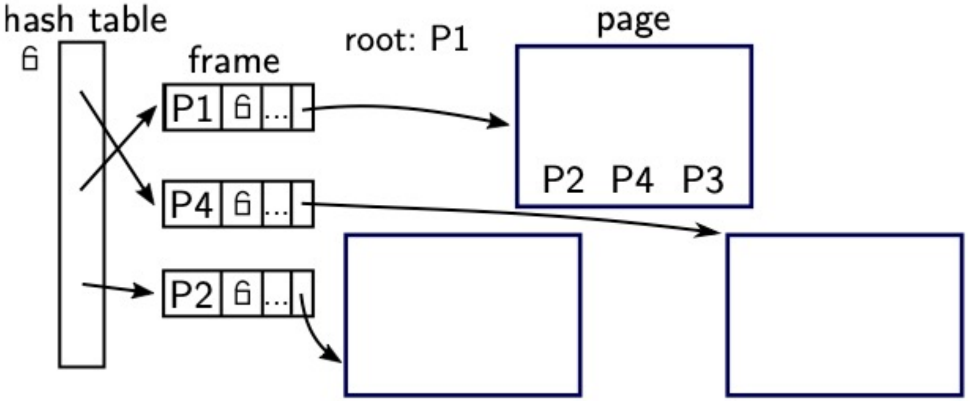
LeanStore design

- **Pointer swizzling**
- Page replacement
- Optimistic latching

Experiments

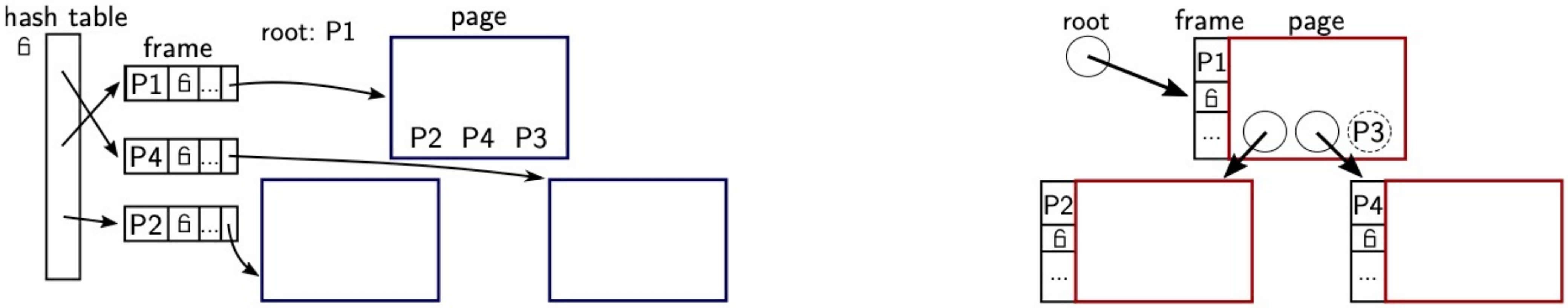
Fine-grained in-memory data management

Pointer Swizzling



(a) traditional buffer manager

Pointer Swizzling

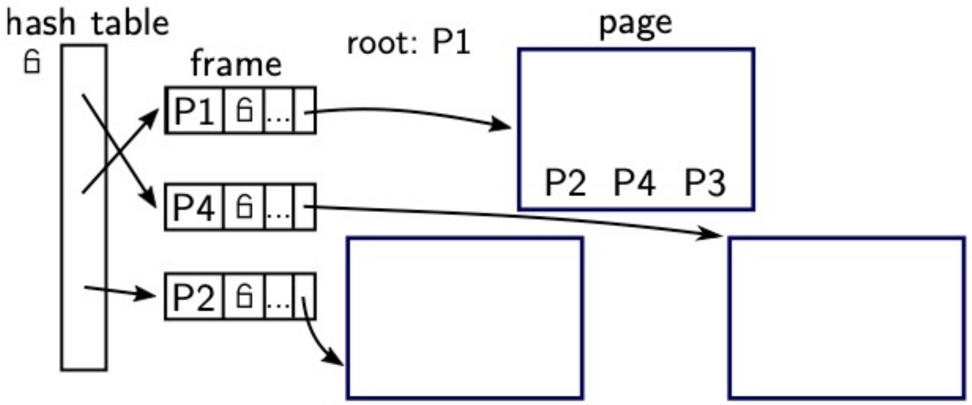


(a) traditional buffer manager

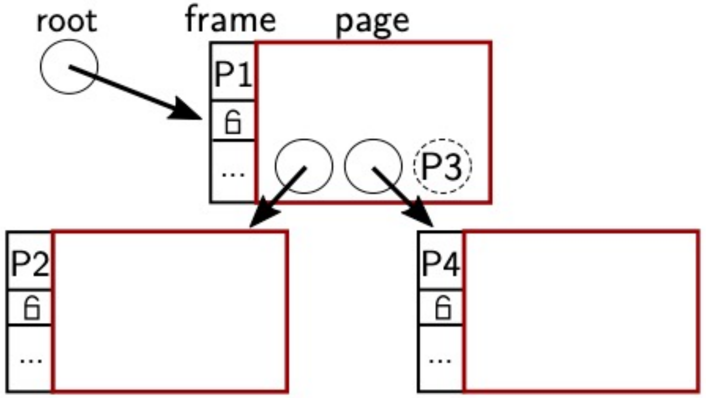
(b) swizzling-based buffer manager

Pages that reside in main memory are directly referenced using virtual memory addresses (i.e., pointers)

Pointer Swizzling



(a) traditional buffer manager



(b) swizzling-based buffer manager

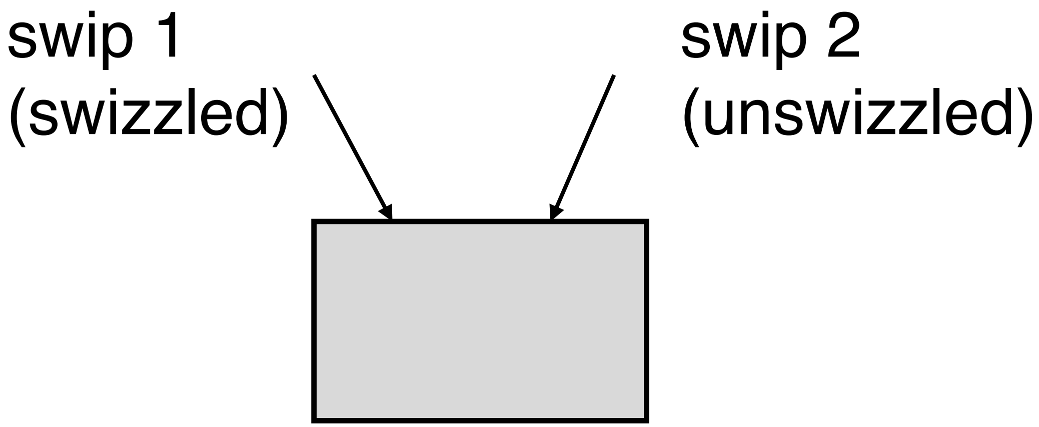
Pages that reside in main memory are directly referenced using virtual memory addresses (i.e., pointers)

Swip: the 8-byte memory location referring to a page

Pointer Swizzling Design Constraints

Challenge 1: concurrency problem if a page is referrenced by multiple swips

- All references must be identified and changed atomically if the page is swizzled or unswizzled



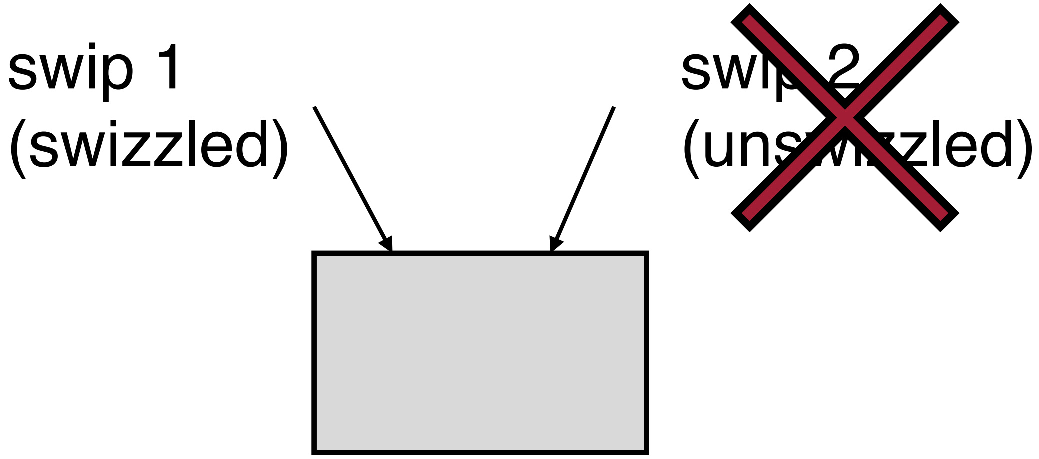
Pointer Swizzling Design Constraints

Challenge 1: concurrency problem if a page is referrenced by multiple swips

- All references must be identified and changed atomically if the page is swizzled or unswizzled

Solution: each page has a single owning swip

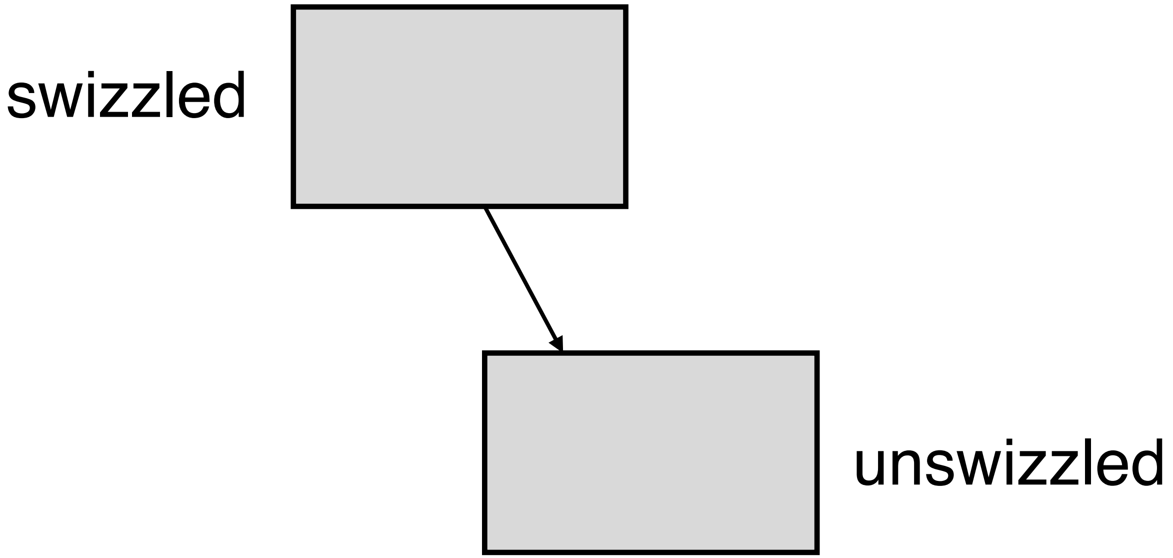
- In-memory data structures must be trees or forests



Pointer Swizzling Design Constraints

Challenge 2: pages containing memory pointers should not be written to disk

- The pointers would not make sense if the system restarts

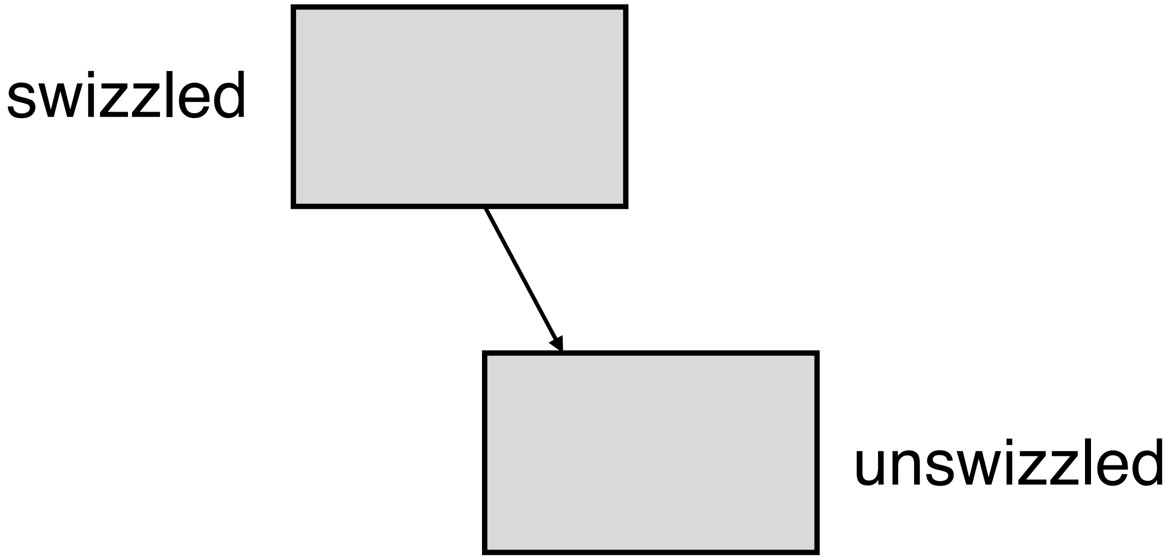


Pointer Swizzling Design Constraints

Challenge 2: pages containing memory pointers should not be written to disk

- – The pointers would not make sense if the system restarts

Solution: Never unswizzle a page that has swizzled children

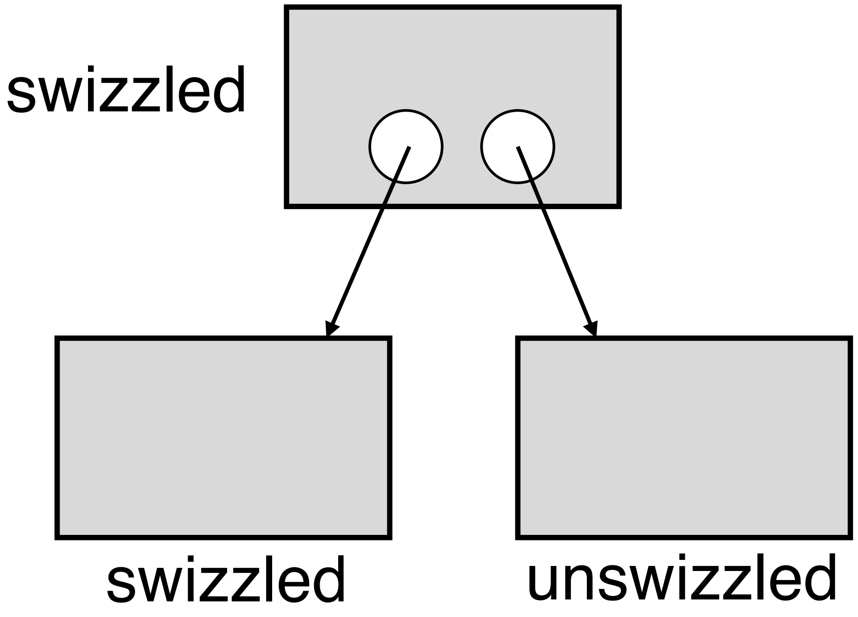


Pointer Swizzling Design Constraints

Constraint 1: each page has a single owning swip

Constraint 2: Never unswizzle a page that has swizzled children

⇒Must be able to iterate over all swips on a page

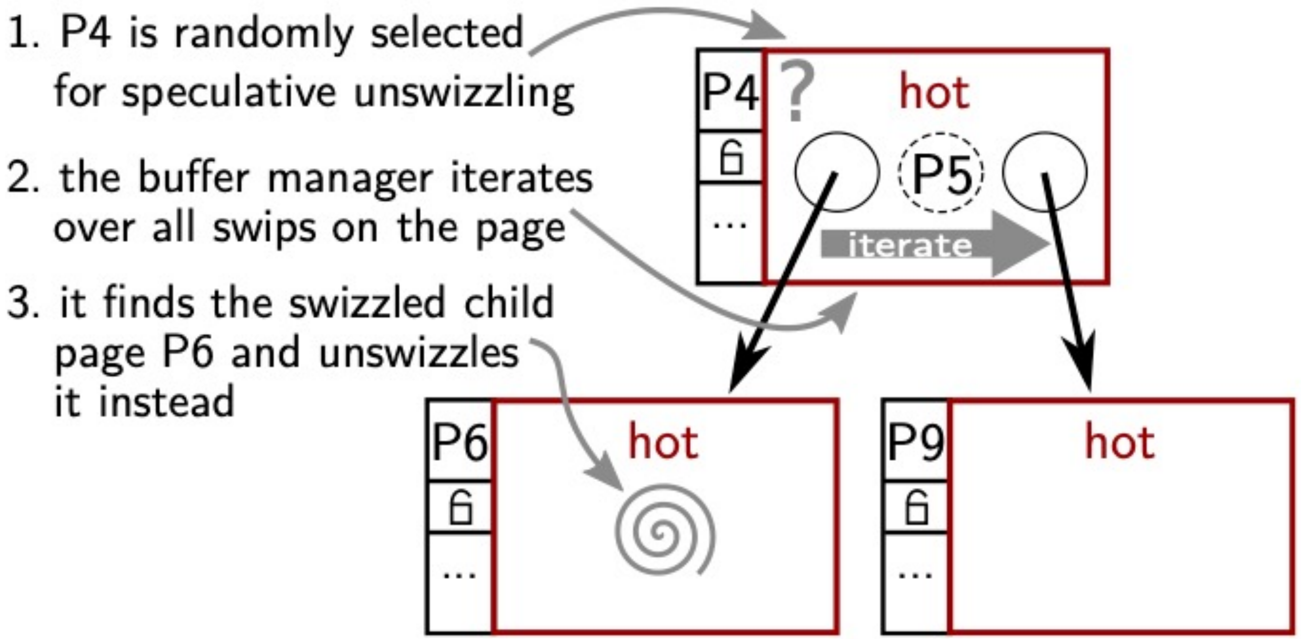


Pointer Swizzling Design Constraints

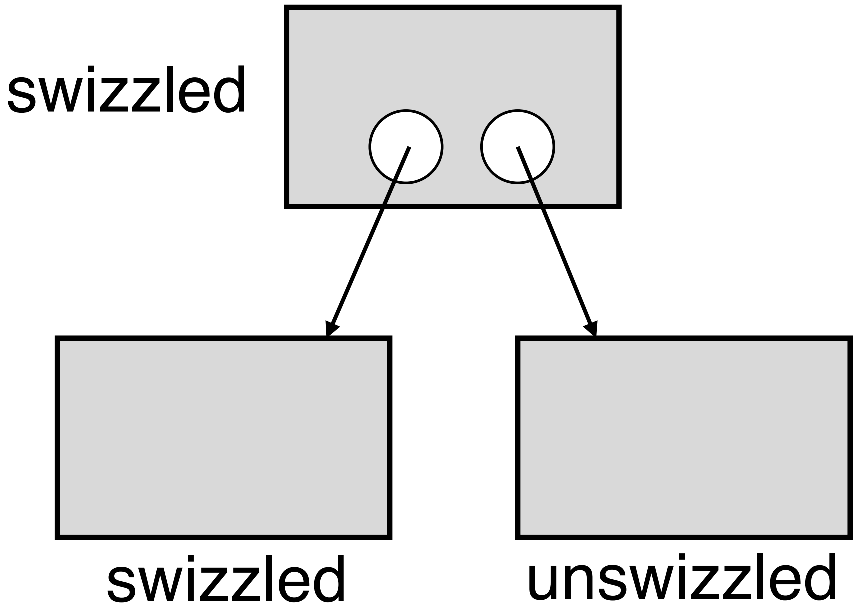
Constraint 1: each page has a single owning swip

Constraint 2: Never unswizzle a page that has swizzled children

⇒Must be able to iterate over all swips on a page



1. P4 is randomly selected for speculative unswizzling
2. the buffer manager iterates over all swips on the page
3. it finds the swizzled child page P6 and unswizzles it instead



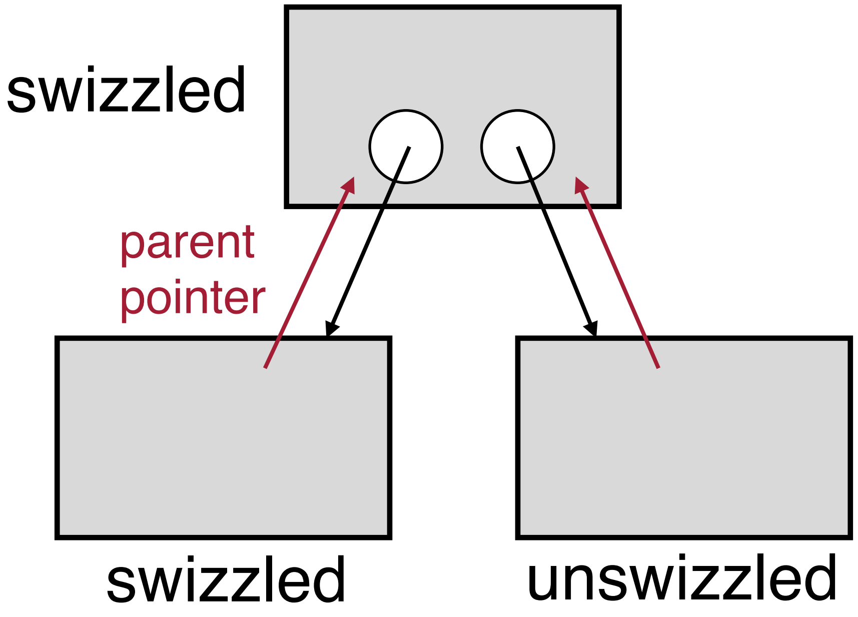
Pointer Swizzling Design Constraints

Constraint 1: each page has a single owning swip

Constraint 2: Never unswizzle a page that has swizzled children

⇒Must be able to iterate over all swips on a page

⇒Must be able to identify parent swip



Pointer Swizzling Design Constraints

Constraint 1: each page has a single owning swip

Constraint 2: Never unswizzle a page that has swizzled children

⇒Must be able to iterate over all swips on a page

⇒Must be able to identify parent swip

For example: B+-trees cannot have
link pointer

Agenda

- Main-memory DB
- LeanStore design
  - Pointer swizzling
  - **Page replacement**
  - Optimistic latching
- Experiments
- Fine-grained in-memory data management

Page Replacement Background

Least Recent Used (LRU)

Clock replacement (aka second chance)

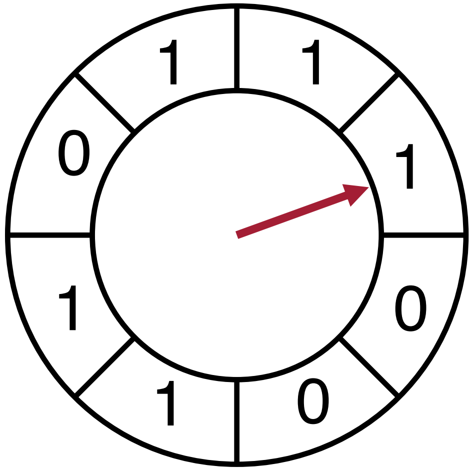
- An approximation of LRU

Page Replacement Background

Least Recent Used (LRU)

Clock replacement (aka second chance)

- An approximation of LRU

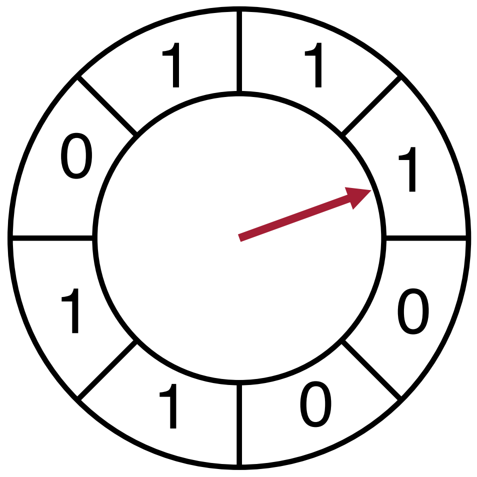


Page Replacement Background

Least Recent Used (LRU)

Clock replacement (aka second chance)

- An approximation of LRU



Look for page to replace

    If the bit = 0: evict
    If the bit = 1: set to 0 and move to next entry

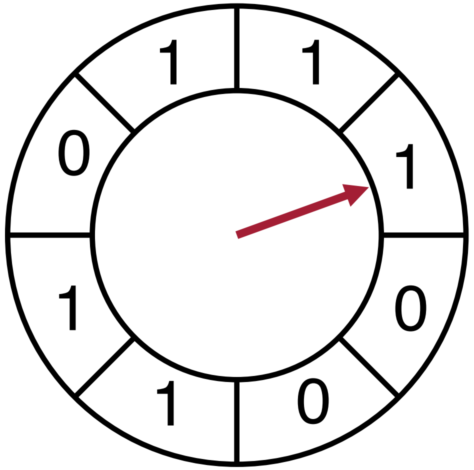
When a page is accessed, set bit to 1

Page Replacement Background

Least Recent Used (LRU)

Clock replacement (aka second chance)

– An approximation of LRU



Look for page to replace

    If the bit = 0: evict

    If the bit = 1: set to 0 and move to next entry

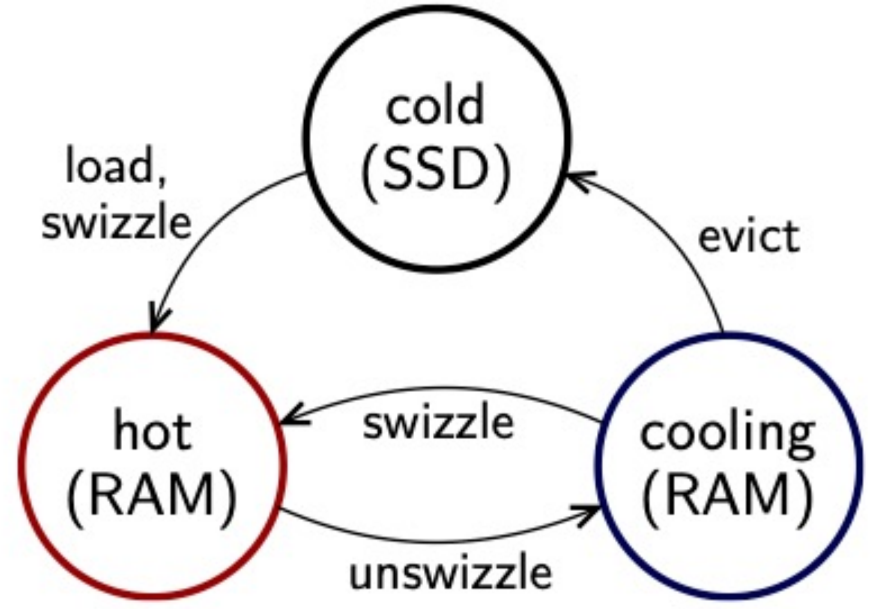
When a page is accessed, set bit to 1

Updating tracking information for each page access is too expensive

Page Replacement — Cooling

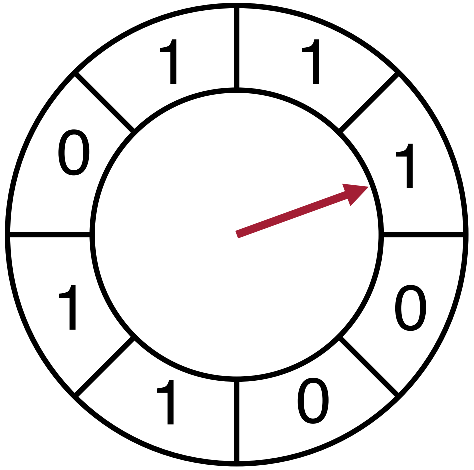
Randomly add pages to cooling stage

- Cooling pages are unswizzled but not replaced
- Cooling pages enter a FIFO queue; a page is replaced if it reaches the end of the queue
- Upon an access, a cooling page is swizzled



Page Replacement Comparison

Clock replacement



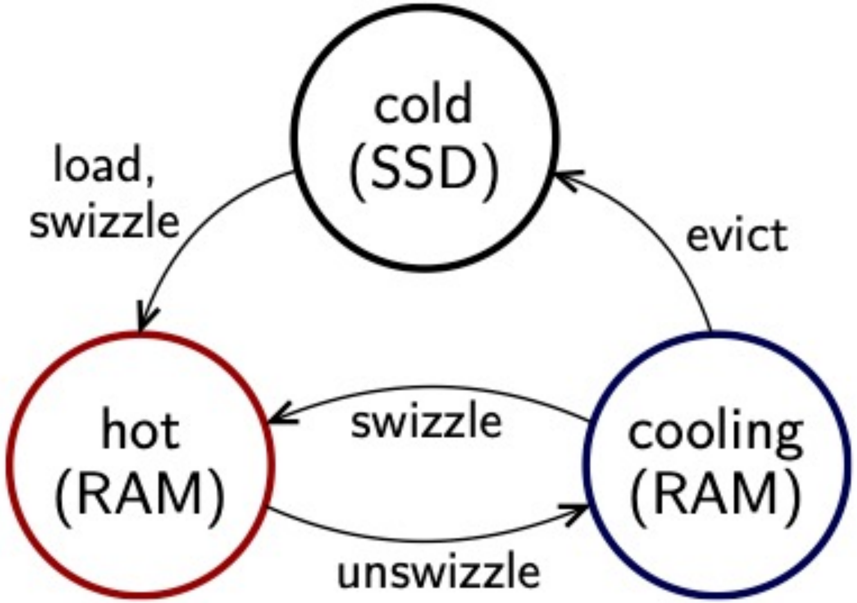
Look for page to replace

If the bit = 0: evict

If the bit = 1: set to 0 and move to next entry

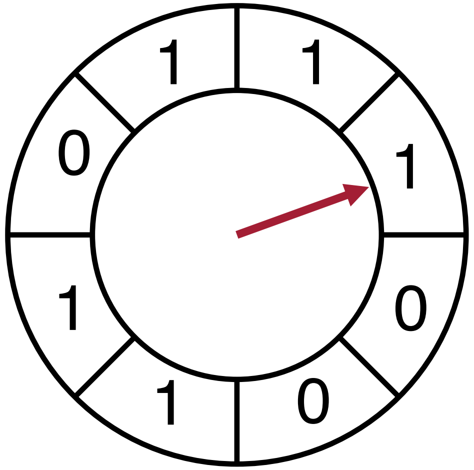
When a page is accessed, set bit to 1

LeanStore replacement



Page Replacement Comparison

Clock replacement



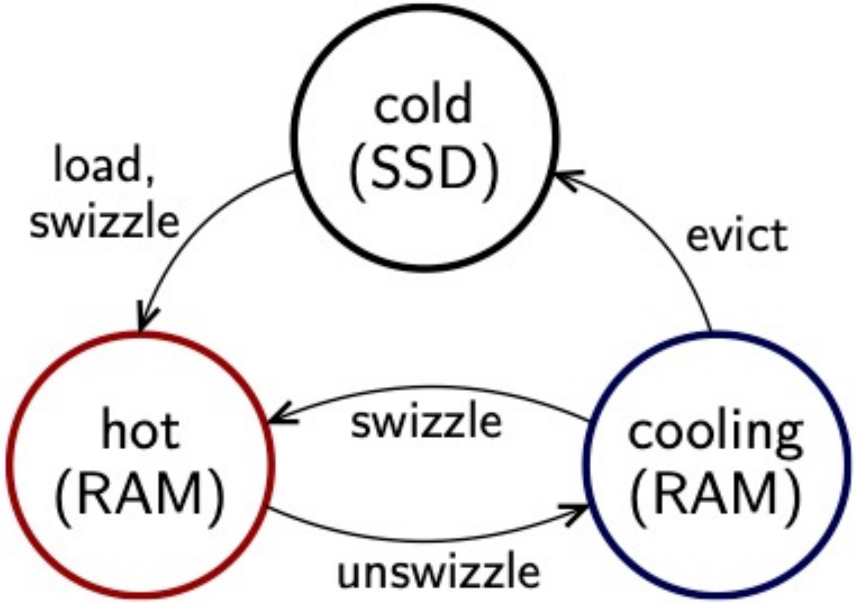
Look for page to replace

If the bit = 0: evict

If the bit = 1: set to 0 and move to next entry

When a page is accessed, set bit to 1

LeanStore replacement



Discussion Question:

Is clock replacement necessarily worse than cooling replacement?

Agenda

Main-memory DB

LeanStore design

- Pointer swizzling
- Page replacement
- **Optimistic latching**

Experiments

Fine-grained in-memory data management

Latching is Expensive

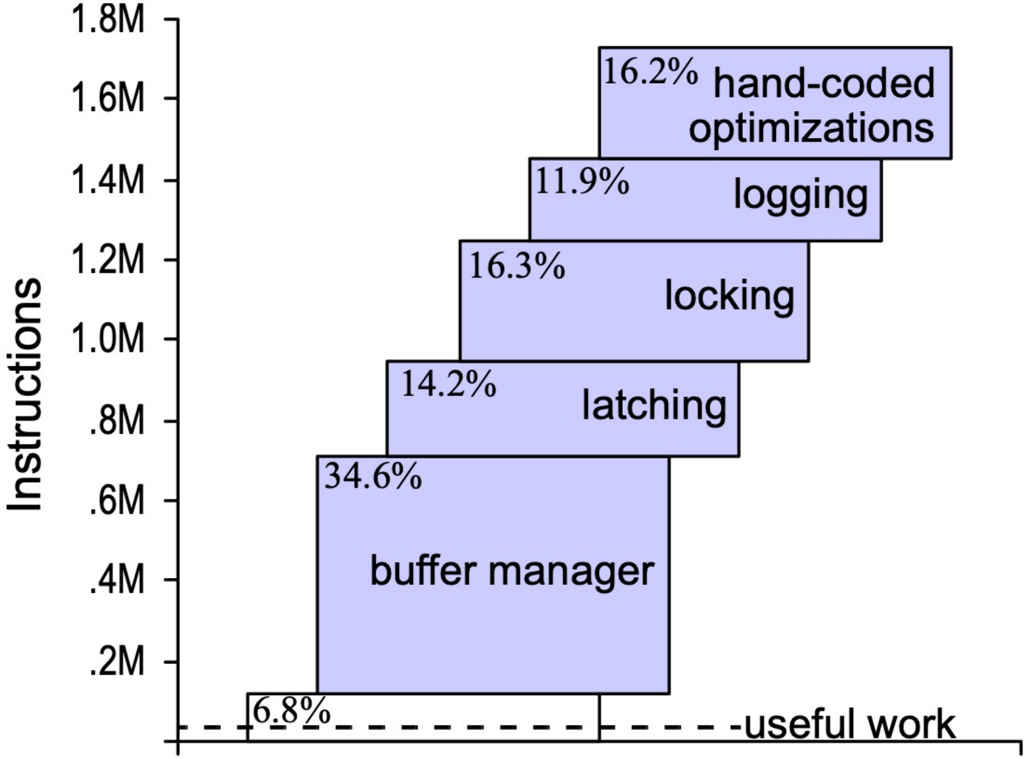


Figure 1. Breakdown of instruction count for various DBMS components for the New Order transaction from TPC-C. The top of the bar-graph is the original Shore performance with a main memory resident database and no thread contention. The bottom dashed line is the useful work, measured by executing the transaction on a no-overhead kernel.

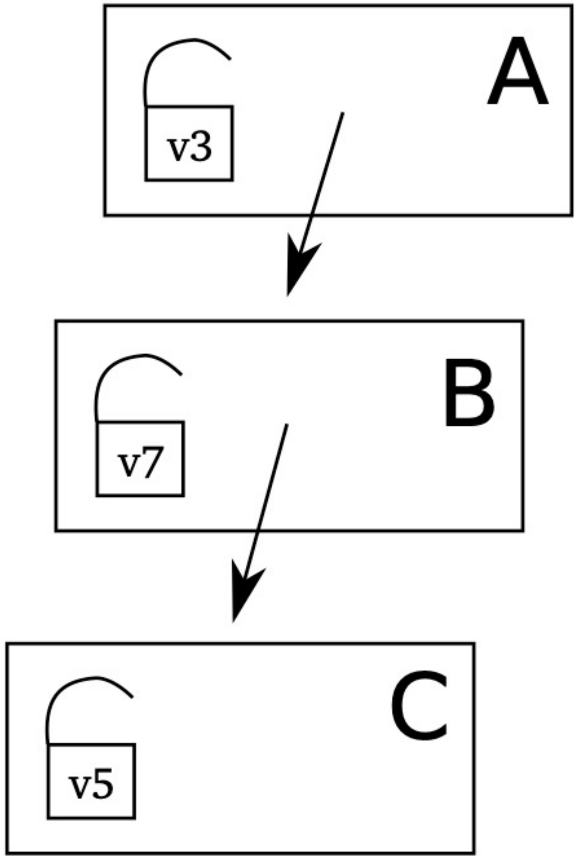
Lock Coupling

traditional

1. lock node A
2. access node A

3. lock node B
4. unlock node A
5. access node B

6. lock node C
7. unlock node B
8. access node C
9. unlock node C



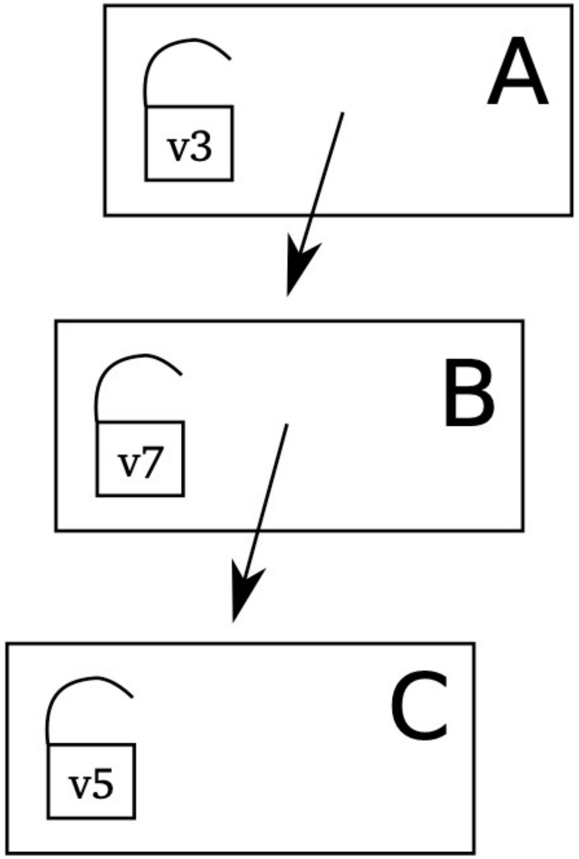
Optimistic Lock Coupling

traditional

1. lock node A
2. access node A

3. lock node B
4. unlock node A
5. access node B

6. lock node C
7. unlock node B
8. access node C
9. unlock node C



optimistic

1. read version v3
2. access node A

3. read version v7
4. validate version v3
5. access node B

6. read version v5
7. validate version v7
8. access node C
9. validate version v5

Epoch-Based Reclamation

Problem: reads do not block writes in optimistic locking

- A page is evicted or deleted while another thread is reading the page

Epoch-Based Reclamation

Problem: reads do not block writes in optimistic locking

– A page is evicted or deleted while another thread is reading the page

Solution: Epoch-based reclamation

– Reclaim a page only if all threads have finished reading it

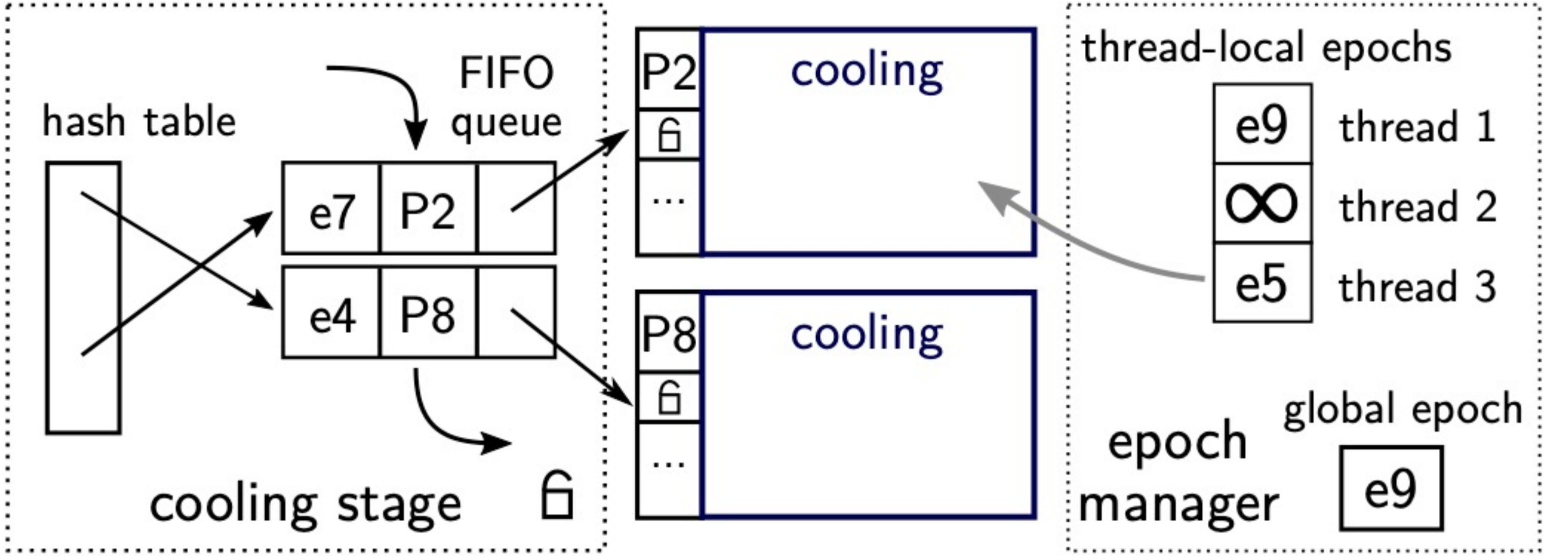


Fig. 6. Epoch-based reclamation.

Agenda

Main-memory DB

LeanStore design

- Pointer swizzling
- Page replacement
- Optimistic latching

Experiments

Fine-grained in-memory data management

Experiments

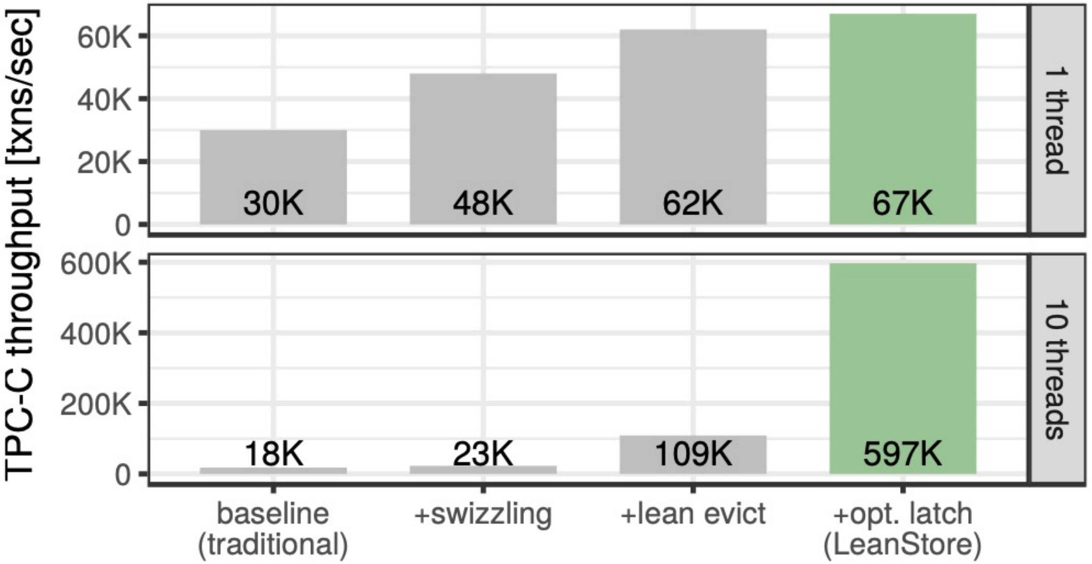


Fig. 7. Impact of the 3 main LeanStore features.

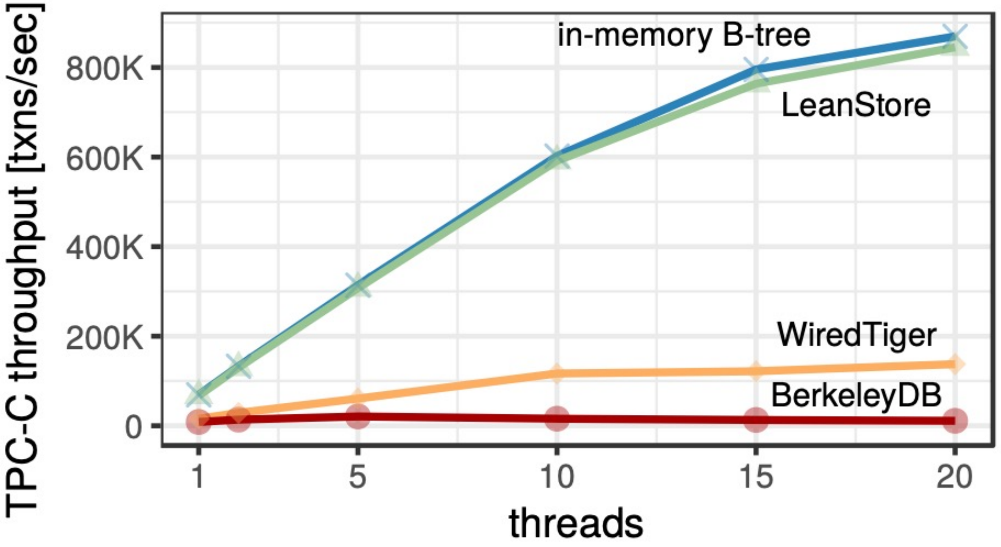


Fig. 8. Multi-threaded, in-memory TPC-C on 10-core system.

Agenda

Main-memory DB

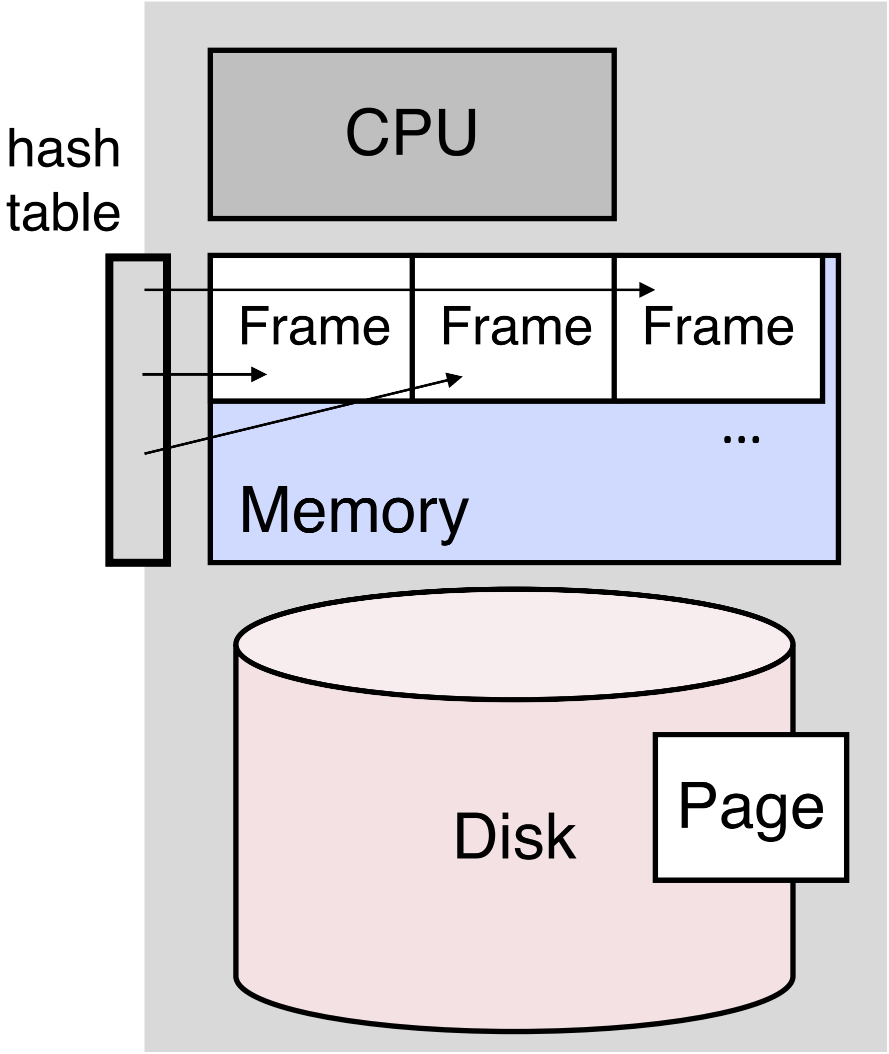
LeanStore design

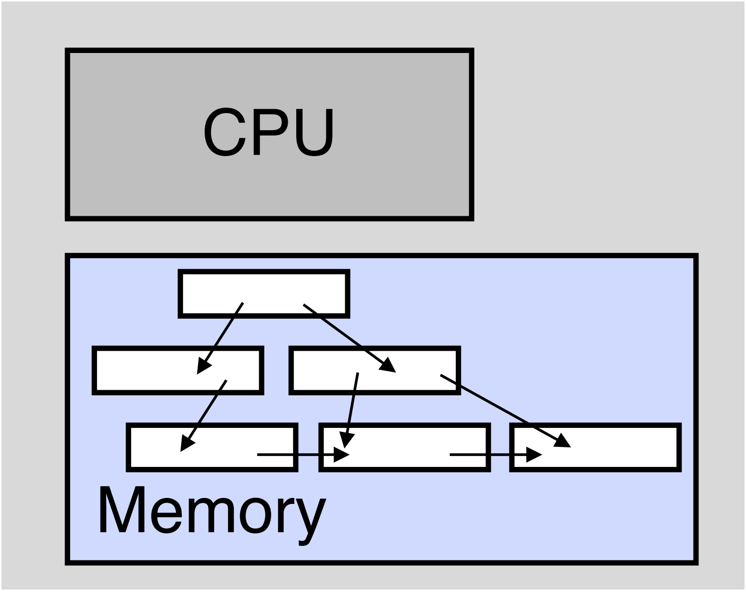
- Pointer swizzling
- Page replacement
- Optimistic latching

Experiments

Fine-grained in-memory data management

Main-Memory DB Architecture





Fine-granularity: Fine-grained (e.g., tuple-level) data management

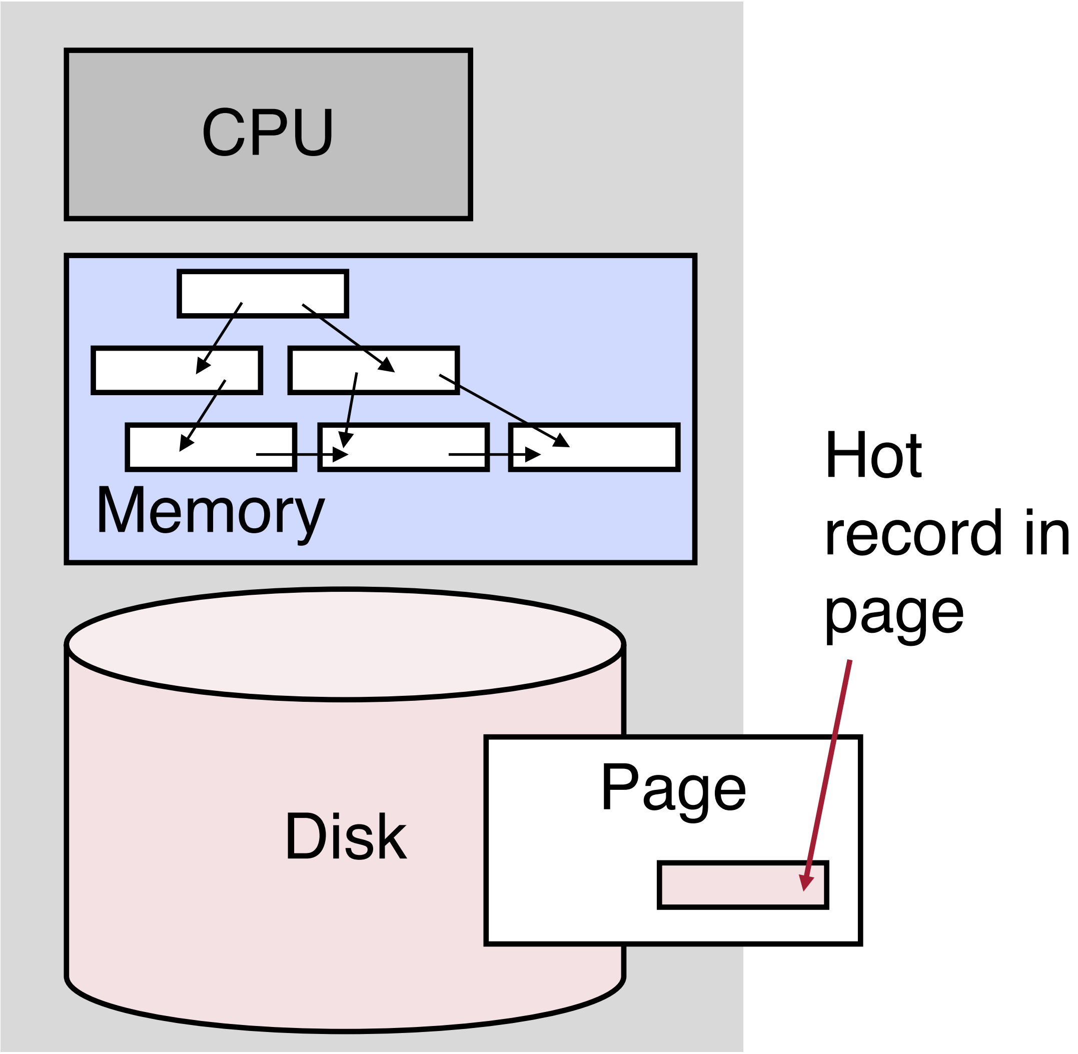
No Indirection: reference data following pointers

Fine-Grained Buffer Management

Migrate tuples, instead of pages, between memory and disk

Challenges

- Tracking all data in the system
- Avoid random writes to disk
- Identifying hot/cold data



[1] Justin DeBrabant, et al., Anti-Caching: A New Approach to Database Management System Architecture. VLDB, 2013
[2] Ahmed Eldawy, et al., Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database. VLDB 2014

Q/A – LeanStore

Drawbacks of LeanStore?

- A hot page is constantly unswizzled?

Is scaling out bad in cloud environment?

What recovery guarantees does buffer management provide?

Concurrency control in this paper?

Predict pages for cooling instead of randomly picking?

Why does latching have high overhead?

Is the hash table a bottleneck?

Wisconsin DB Affiliates Workshop

Time: **Thursday, 8:30am–4pm**

Location: **Northwoods (Union South 3rd Floor)**

Workshop contents

- Research highlight talk from faculty member
- Research talks from PhD students
- Pitch talks from industry
- Poster session
- Discussion with industry partners including AWS, Databricks, Google, MatrixOrigin, Microsoft, Oracle, Snowflake, TiDB

Can also attend on zoom:

https://uwmadison.zoom.us/j/95526978682?pwd=NWxTOXJGSDhiekhwdXBOcG9qMjVKdz09

Before Next Lecture

Submit review for

Patricia G. Selinger, et al., Access Path Selection in a Relational Database Management System. SIGMOD, 1979