CS 764: Topics in Database Management Systems

Lecture 13: Transaction Buffer Management

Xiangyao Yu

10/16/2025

Today's Papers: LeanStore and Two-Tree

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.

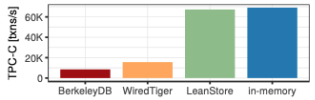


Fig. 1. Single-threaded in-memory TPC-C performance (100 warehouses).

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

Two is Better Than One: The Case for 2-Tree for Skewed Data Sets

Xinjing Zhou
MIT CSAIL
xinjing@mit.edu

Xiangyao Yu
University of Wisconsin-Madison
yxy@cs.wisc.edu

Goetz Graefe
Google
goetzg@google.com

Michael Stonebraker
MIT CSAIL
stonebraker@csail.mit.edu

ABSTRACT

Real-world data sets almost always exhibit skew, i.e., a majority of the accesses go to a minority of the records. Obviously, Smith and Brown are more popular names than Stonebraker and Graefe. Traditional block-oriented index structures such as B-trees are sub-optimal for skewed data because an index block often has a small number of hot records and a larger number of cold ones. This results in poor main memory utilization and increased cost.

To alleviate this problem, we propose a 2-Tree architecture, where hot index records are in one tree and cold ones are in a second tree. Hot tree blocks are frequently accessed and likely to remain in main memory, resulting in improved main memory utilization. Our core idea is to employ a lightweight general migration protocol to move records between trees in both directions when appropriate and to maintain access statistics at low cost.

In addition, the two trees can be configured separately for hardware differences. One tree can be optimized for main memory while the second exploits secondary storage. Obviously, the 2-Tree idea can also be generalized to multiple storage levels and/or devices. We show how the 2-Tree idea and record migration can be applied to both B+trees and LSM-trees to improve their memory utilization significantly (by 15× and 20× respectively) on a highly skewed workload. We also observed up to 1.7× throughput improvement on a Zipfian-skewed IO-bound workload compared to traditional single B+tree or LSM-tree using the same amount of main memory. Unlike existing solutions for improving memory utilization at the cost of inferior range scan performance, 2-Tree refuses to make such a compromise.

1 INTRODUCTION

Real-world keyed data is invariably highly skewed. A subset (the working set) of the data has a much higher access frequency than the rest [3, 6, 7, 33, 39]. For example, celebrities on social media get orders of magnitude more page views than average users. On the NYSE 40 stocks account for 60 percent of the daily transaction volume. Generally, the hot records are spread across the entire key space [8] rather than clustered in a few subranges. Lastly, this working set is often not static [4]. For example, trending tweets and breaking news change over time.

The traditional approach to indexing keyed data sets is to employ a homogeneous data structure such as B+tree with a main memory buffer pool. In this way, hot blocks are cached in the main memory buffer pool and cold blocks reside on secondary storage. This approach manages and migrates both main memory and disk-resident data at block granularity. On skewed data sets main memory blocks might have only one hot record in a block containing hundreds of cold records. This results in poor memory utilization and sub-optimal performance.

In this paper, we advocate migrating data at the record-level. We study a 2-Tree architecture in which there are two separate tree data structures, one (top tree) for the hot records and a second (bottom tree) for the cold ones. When a hot record becomes cold, it is migrated from one tree to the other. Likewise, records can move in the other direction. At the core of this architecture is a general-purpose migration protocol, which can accurately detect and maintain hot records at low cost. It adds 3 bits per hot record and works with any tree data structures. With this clustering of hot records, 2-Tree significantly increases memory utilization on skewed data.

Another advantage of this architecture is that the two data structures can be optimized separately for their underlying storage medium. The hot structure can be optimized for main memory, while the cold structure can be optimized for secondary storage. Therefore, 2-Tree can be used as an indexing architecture for main-memory database indexing that extends to workloads larger than memory [11, 13].

We can also generalize 2-Tree architecture to an N-Tree architecture to adapt to systems with more than two distinct storage levels and/or devices. While this is somewhat similar to a well-known multi-tree structure, LSM-tree [28], N-Tree moves data upwards upon read as well. Such upward migration can help improve memory utilization under read-heavy workloads for which LSM-tree is not optimized.

In this paper, we present three case studies of applying 2-Tree architecture and record-level migration. First, we study an application of 2-Tree for indexing in main-memory database in a larger-than-memory setting [11, 13] and show that it can outperform the Anti-Caching [11] approach significantly. Second, we show that two buffer-managed B+trees combined with record migration can significantly outperform a state-of-the-art single B+tree implementation [19] by improving buffer pool memory utilization using the same amount of main memory. Lastly, we show a preliminary N-Tree implementation by simply augmenting LSM-tree with record-level upward migration. This improves their performance significantly versus vanilla LSM-trees on read-heavy workloads.

We summarize our contributions as follows:

- We propose the 2-Tree architecture to address the limitations of existing approaches for managing larger-than-memory indexes.
- We propose an efficient record migration protocol that works between any two tree structures.

 13th Annual Conference on Innovative Data Systems Research (CIDR '23). January 8-11, 2023, Amsterdam, The Netherlands.

CIDR 2023

Agenda

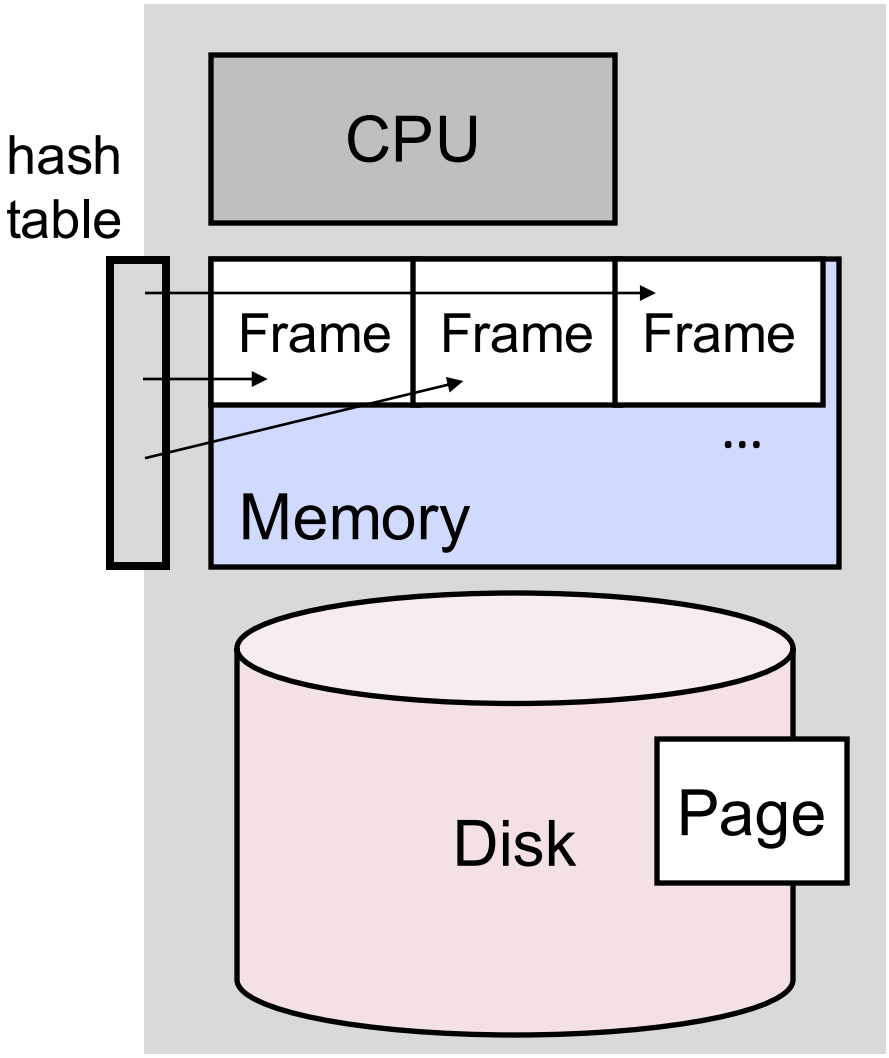
Main-memory DB

LeanStore design

- Pointer swizzling
- Page replacement
- Optimistic latching

Two-Tree

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

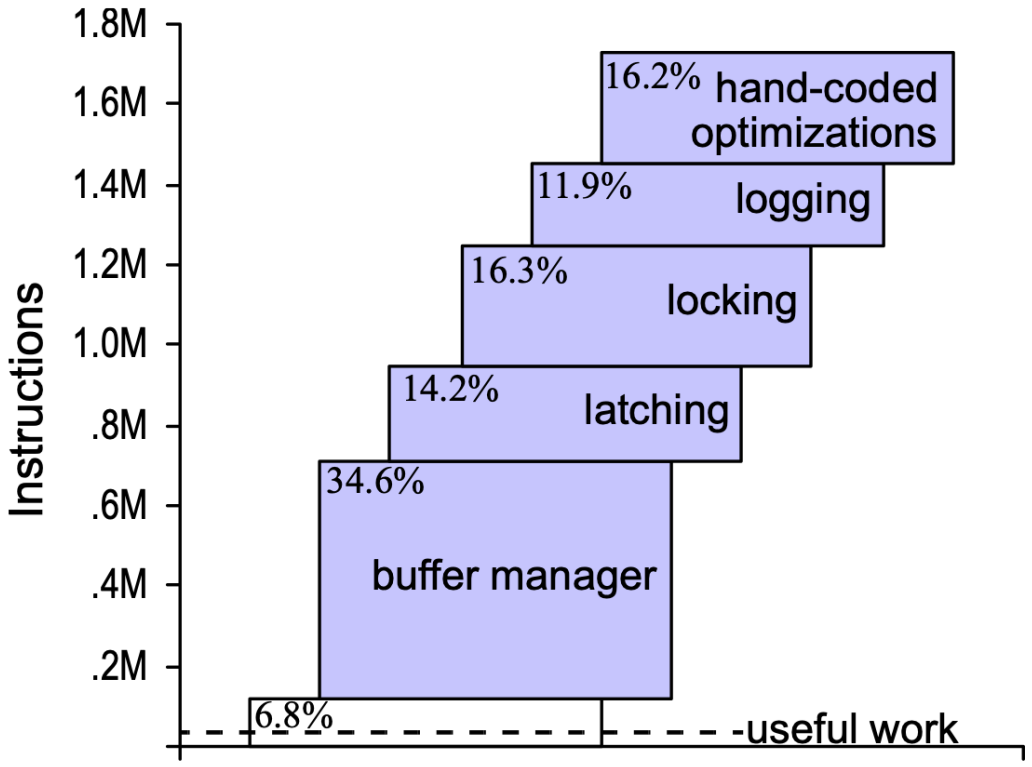


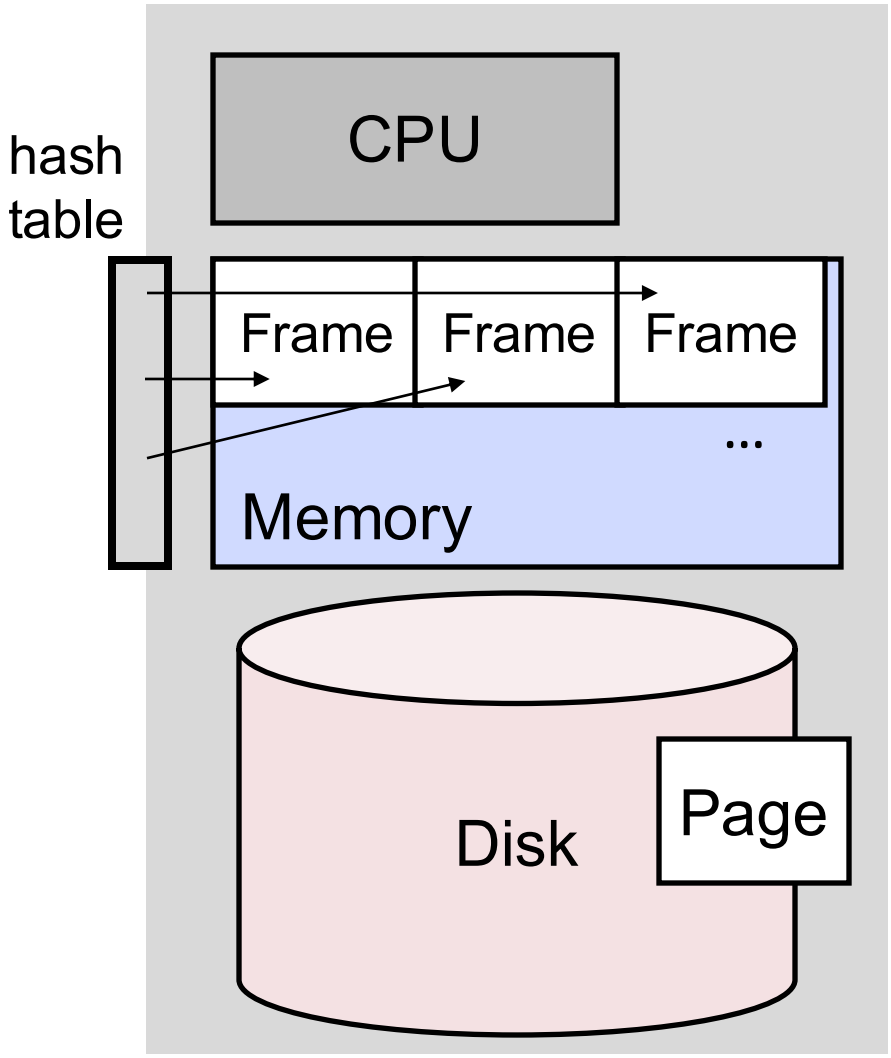
Figure 1. Breakdown of instruction count for various DBMS components for the New Order transaction from TPC-C. The top of the bar-graph is the original Shore performance with a main memory resident database and no thread contention. The bottom dashed line is the useful work, measured by executing the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

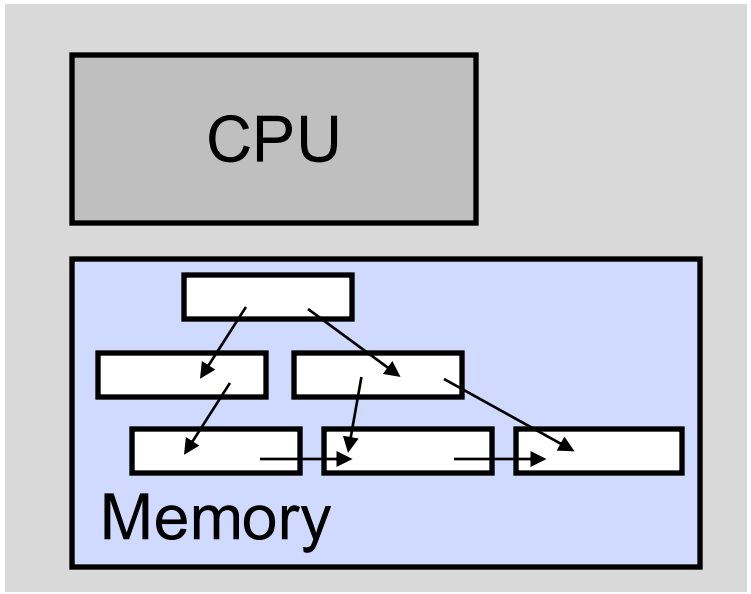
Only a small fraction of instructions execute useful work

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

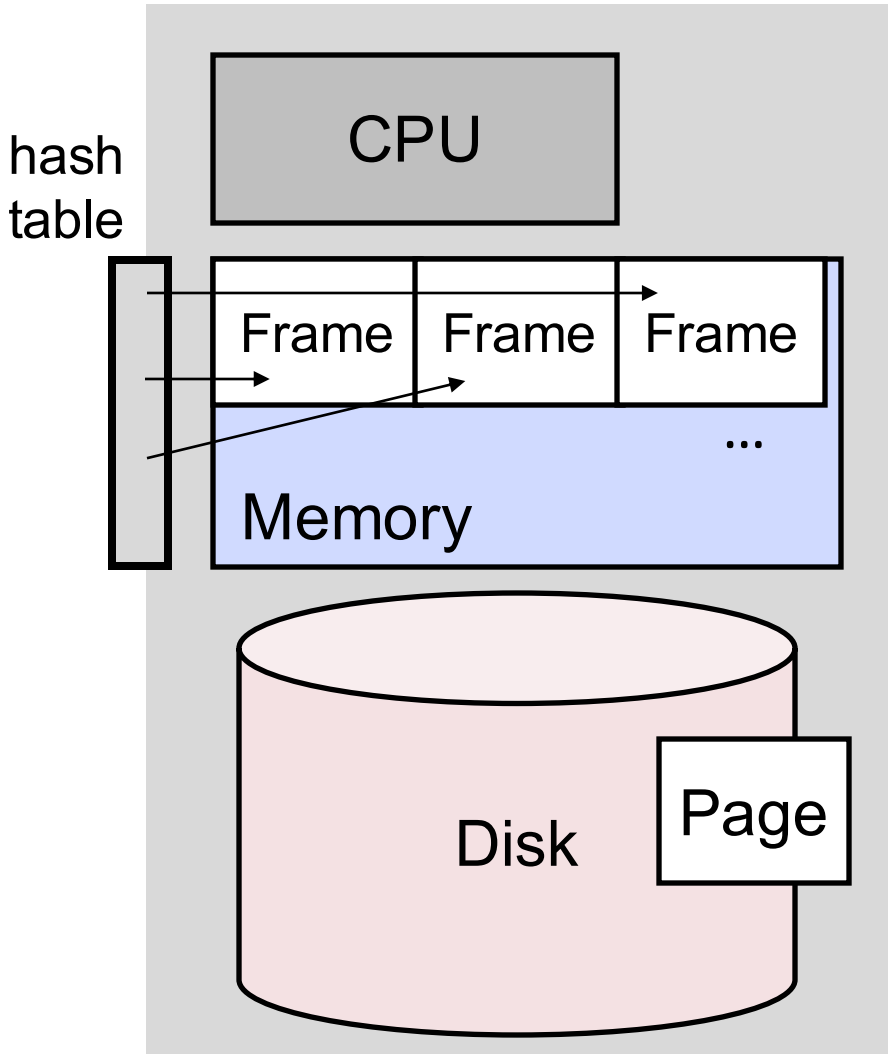


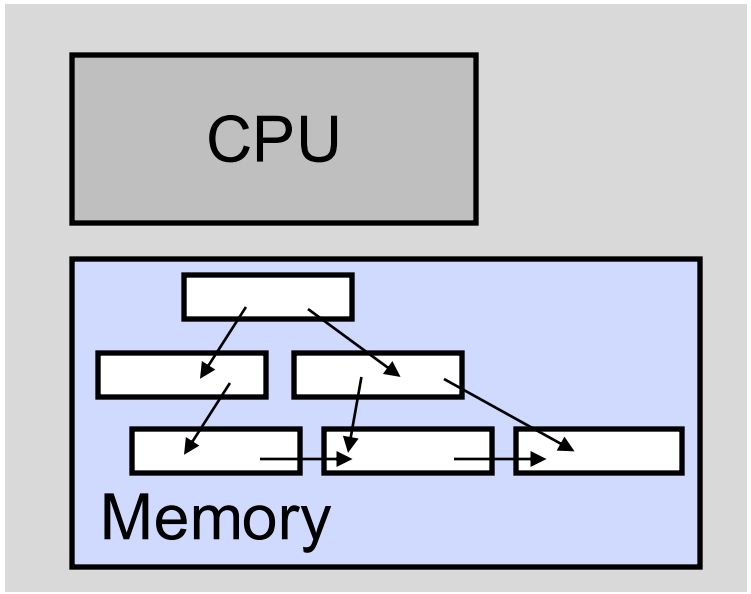


Fine-granularity: Fine-grained (e.g., tuple-level) data management

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 LeanStore**

Agenda

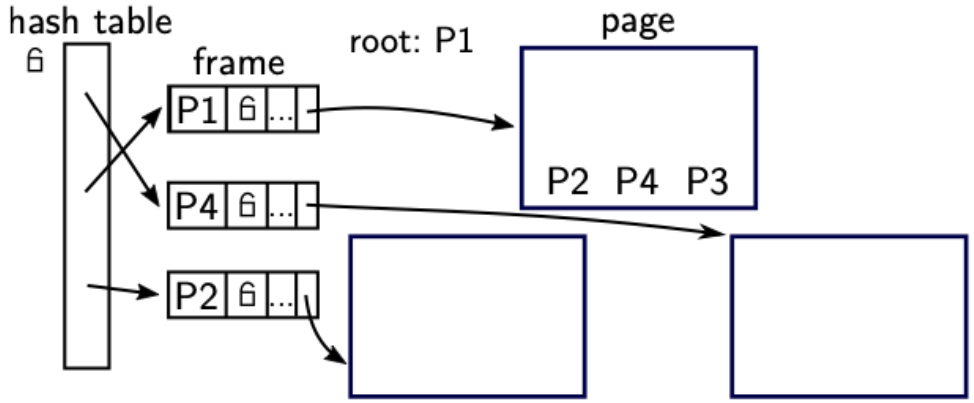
Main-memory DB

LeanStore design

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

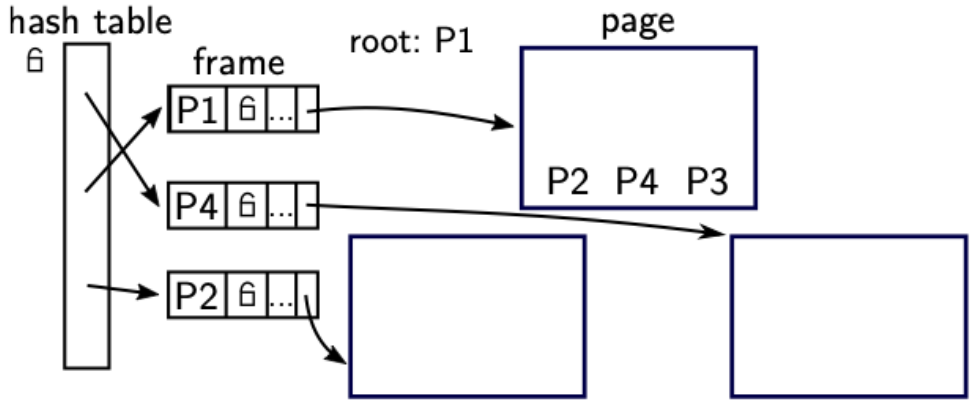
Two-Tree

Pointer Swizzling

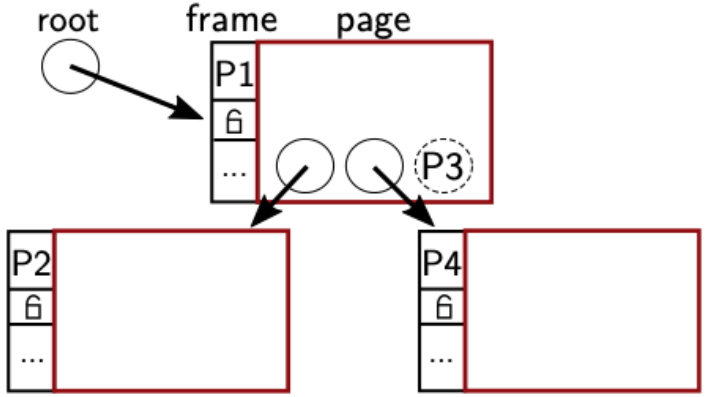


(a) traditional buffer manager

Pointer Swizzling



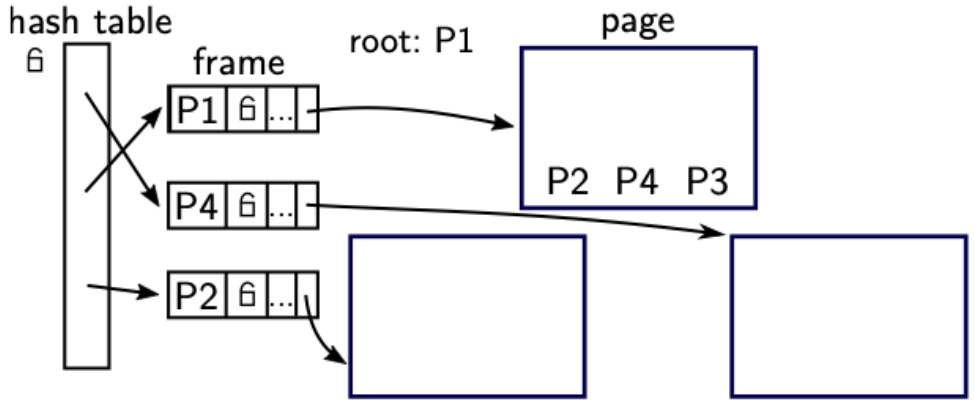
(a) traditional buffer manager



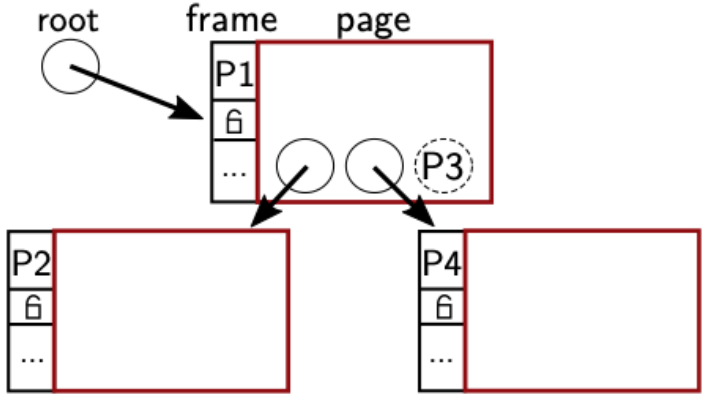
(b) swizzling-based buffer manager

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

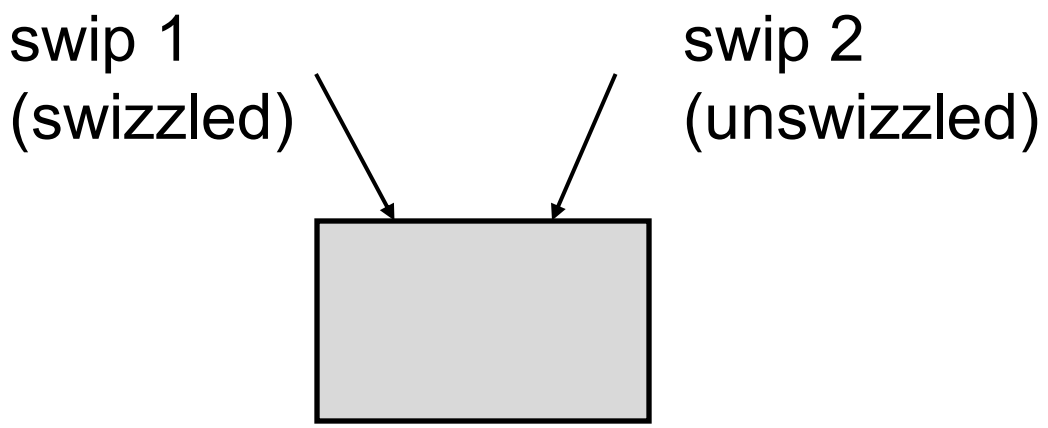
Pages that reside in main memory are directly referenced using virtual memory addresses (i.e., pointers)

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

Pointer Swizzling Design Constraints

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

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



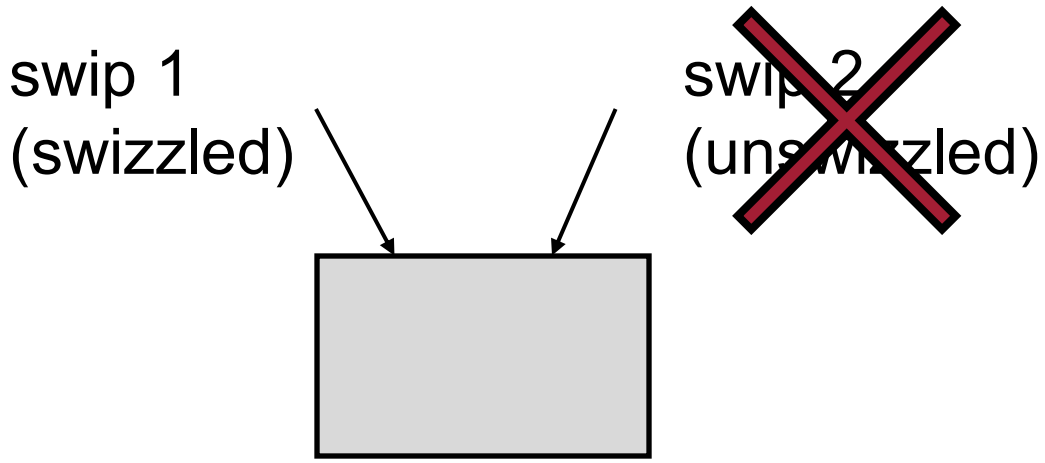
Pointer Swizzling Design Constraints

Challenge 1: concurrency problem if a page is referenced 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

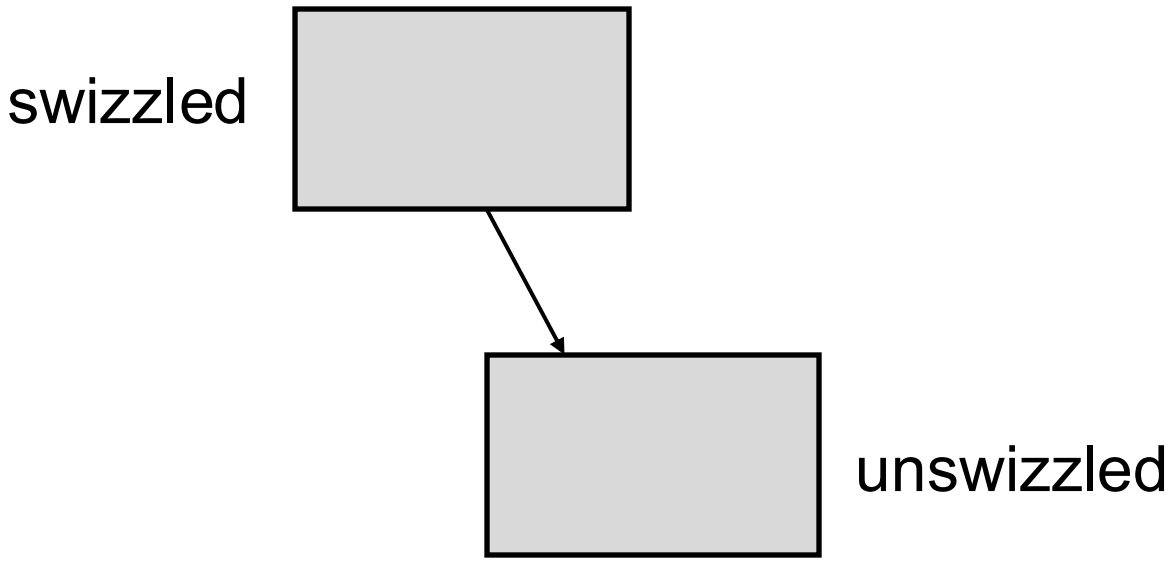
- In-memory data structures must be trees or forests



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

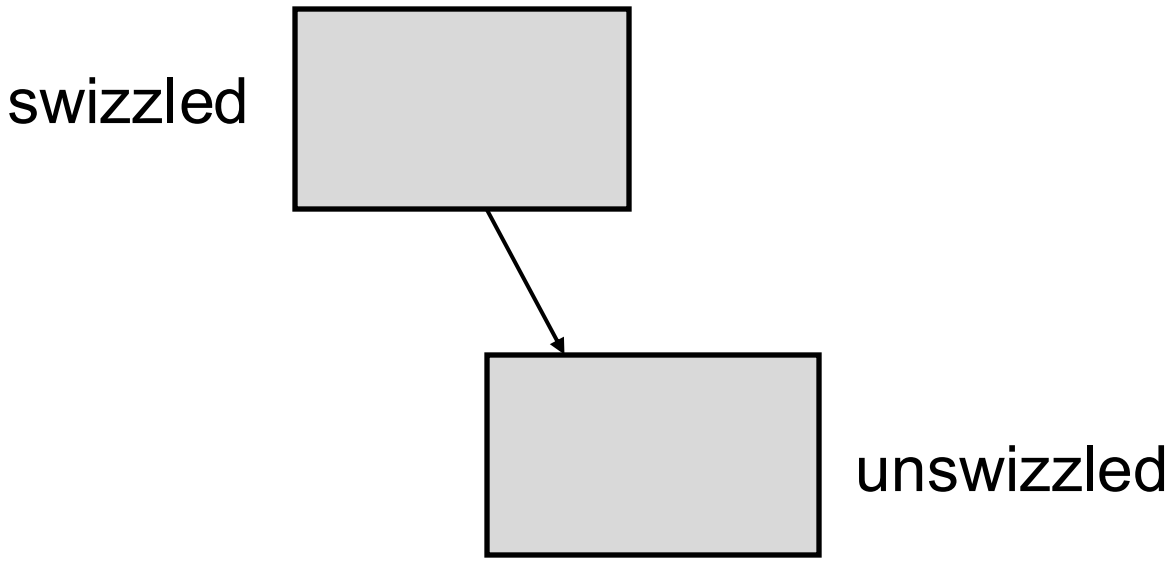


Pointer Swizzling Design Constraints

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

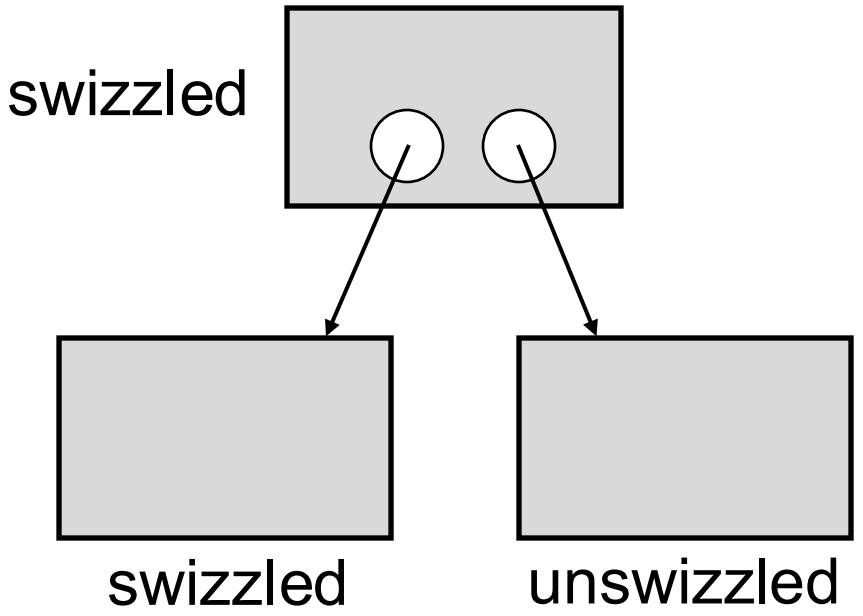


Pointer Swizzling Design Constraints

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

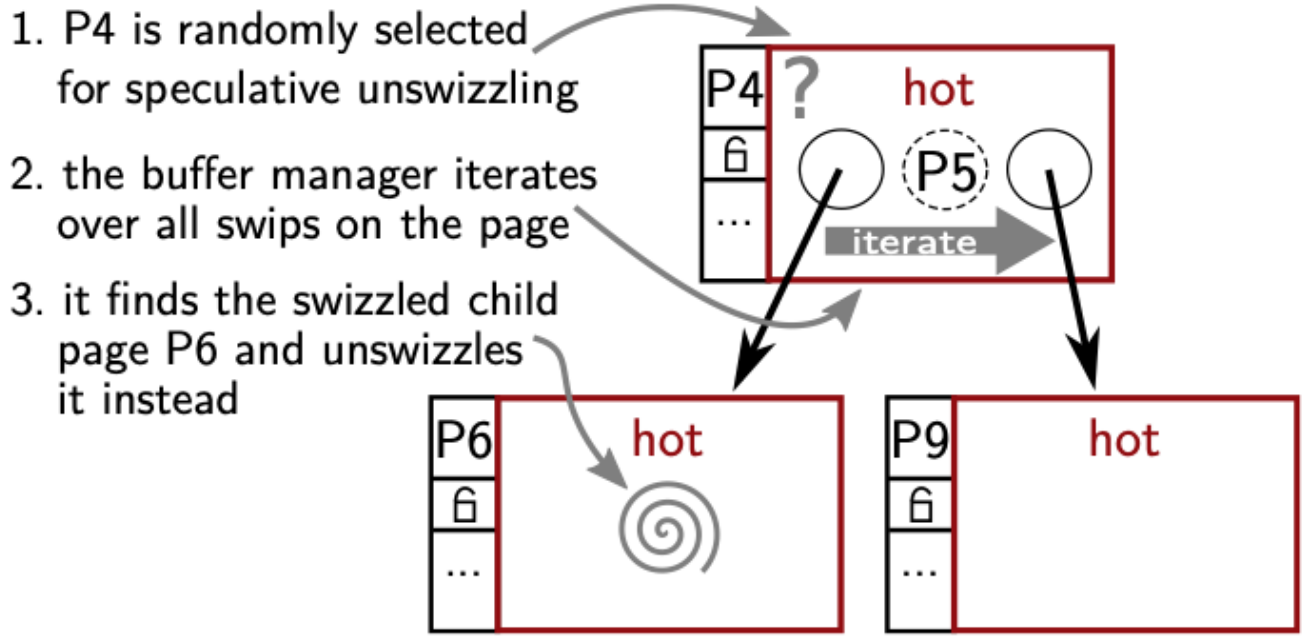


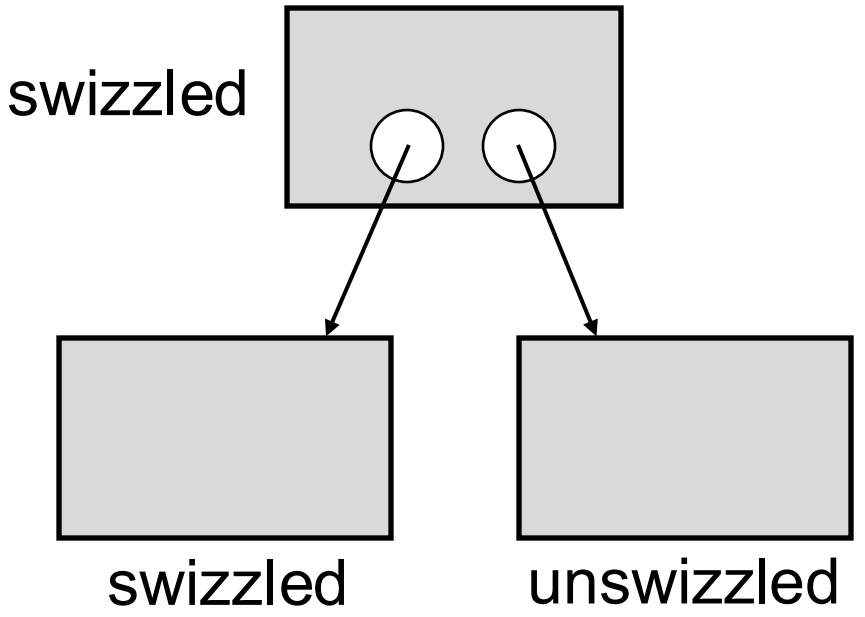
Pointer Swizzling Design Constraints

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





Pointer Swizzling Design Constraints

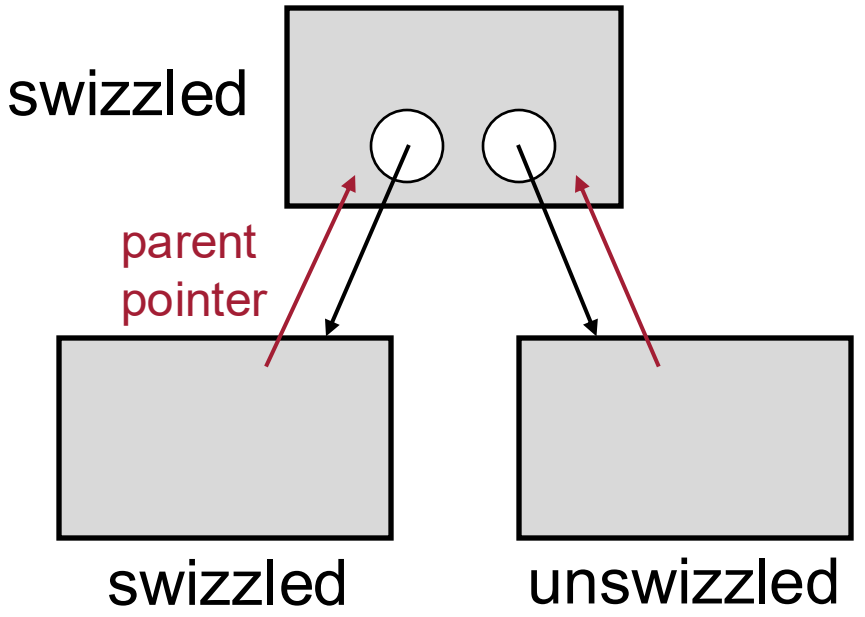
Constraint 1: each page has a single owning swip

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

(to update parent pointer when unswizzling)



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

Two-Tree

Page Replacement Background

Least Recent Used (LRU)

Clock replacement (aka second chance)

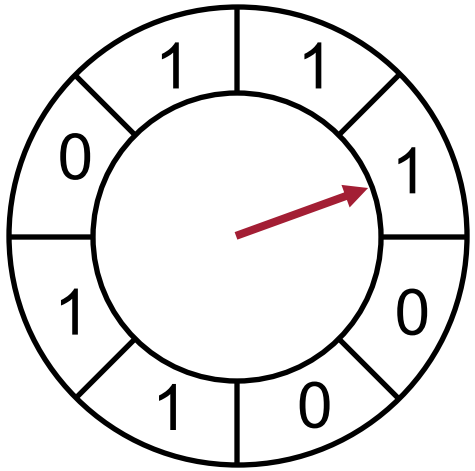
- An approximation of LRU

Page Replacement Background

Least Recent Used (LRU)

Clock replacement (aka second chance)

– An approximation of LRU

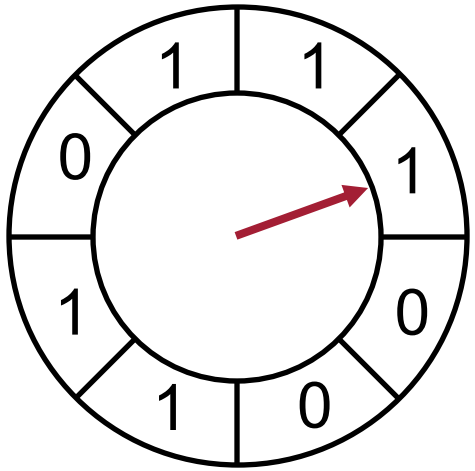


Page Replacement Background

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

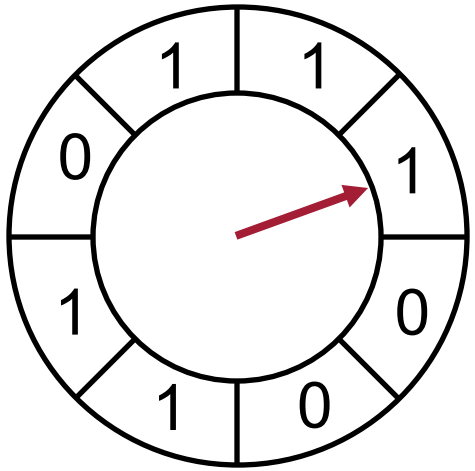
When a page is accessed, set bit to 1

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

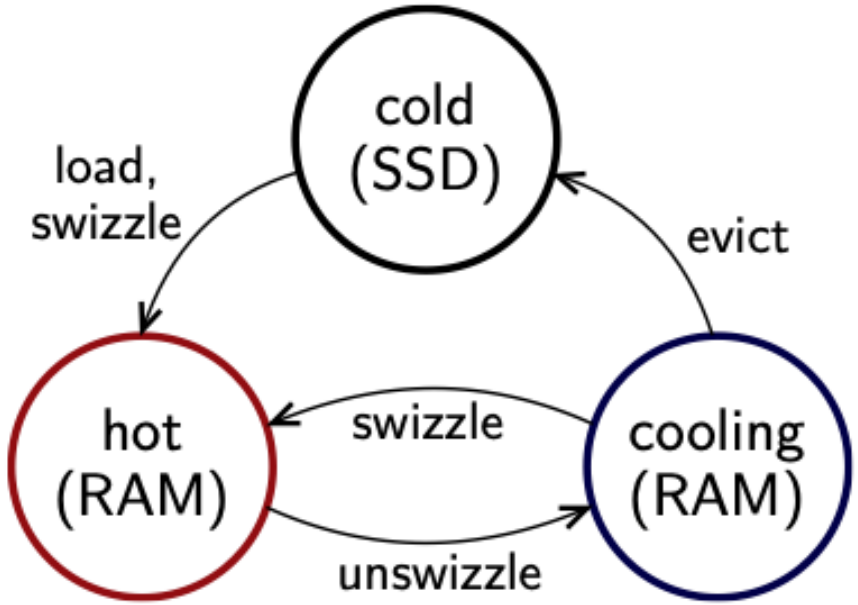
When a page is accessed, set bit to 1

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



Agenda

Main-memory DB

LeanStore design

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

Two-Tree

Latching is Expensive

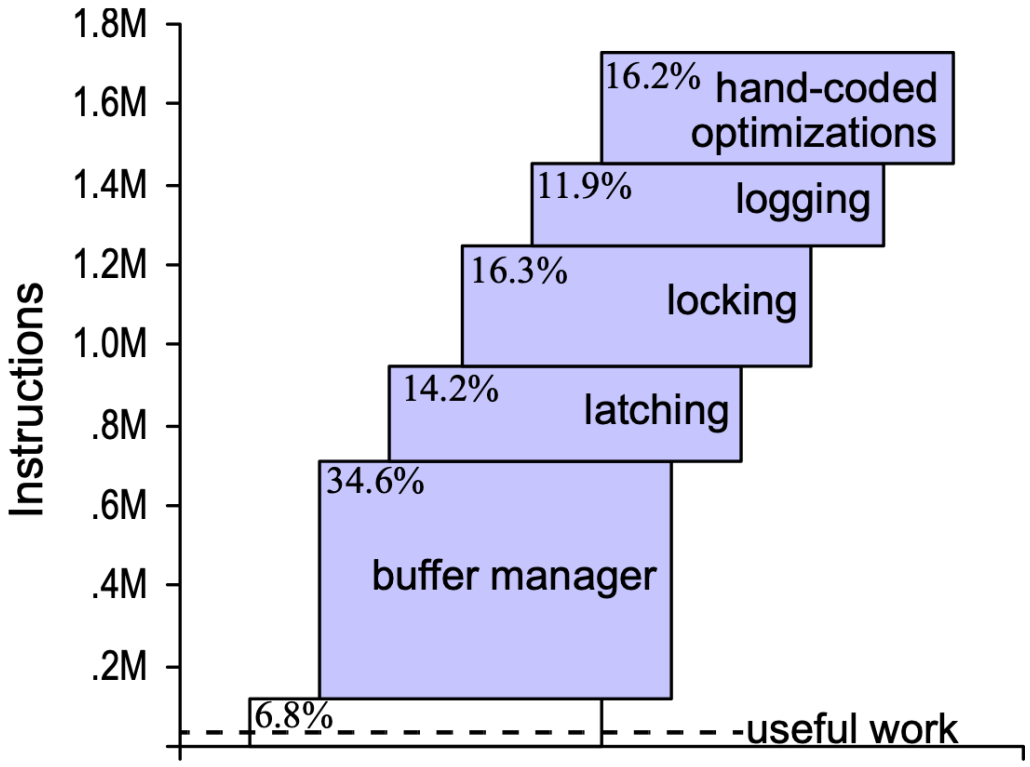
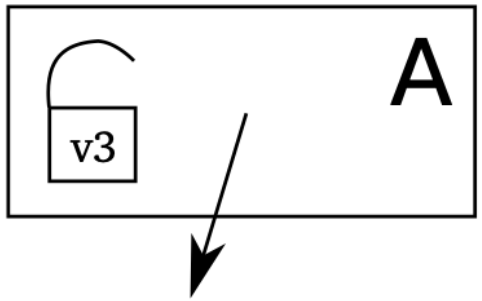


Figure 1. Breakdown of instruction count for various DBMS components for the New Order transaction from TPC-C. The top of the bar-graph is the original Shore performance with a main memory resident database and no thread contention. The bottom dashed line is the useful work, measured by executing the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

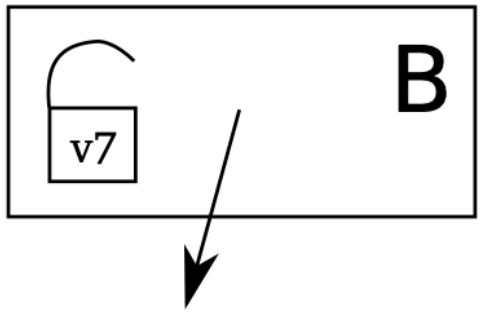
Lock Coupling

traditional

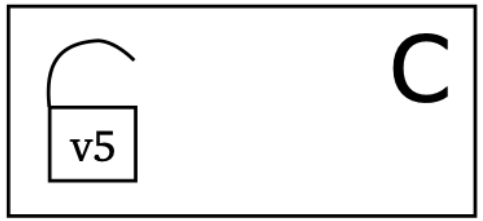
1. lock node A
2. access node A



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



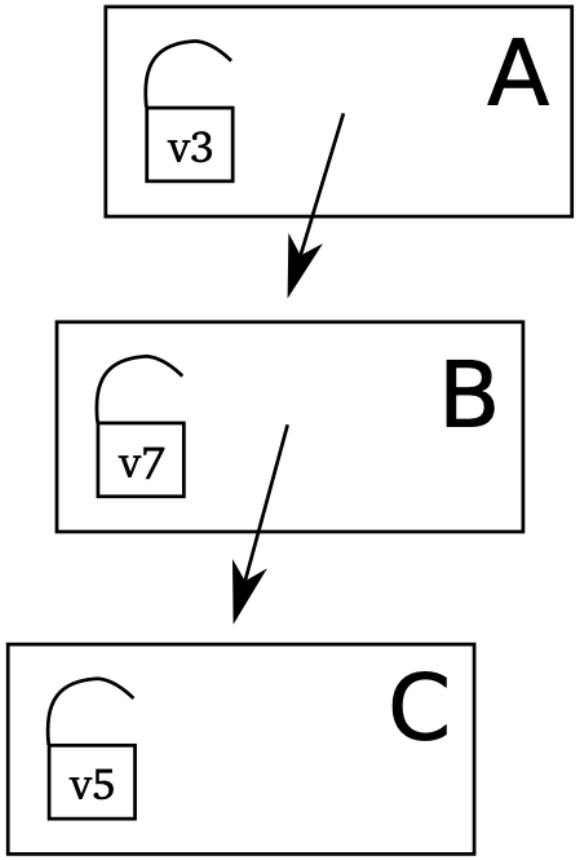
6. lock node C
7. unlock node B
8. access node C
9. unlock node C



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

Agenda

Main-memory DB

LeanStore design

- Pointer swizzling
- Page replacement
- Optimistic latching

Experiments

Fine-grained in-memory data management

Experiments

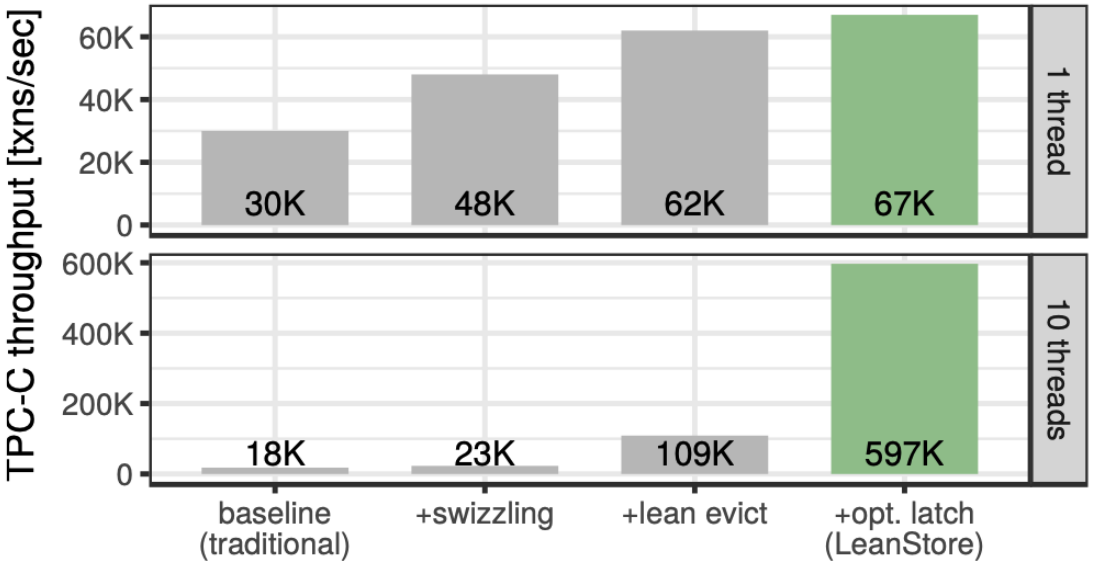


Fig. 7. Impact of the 3 main LeanStore features.

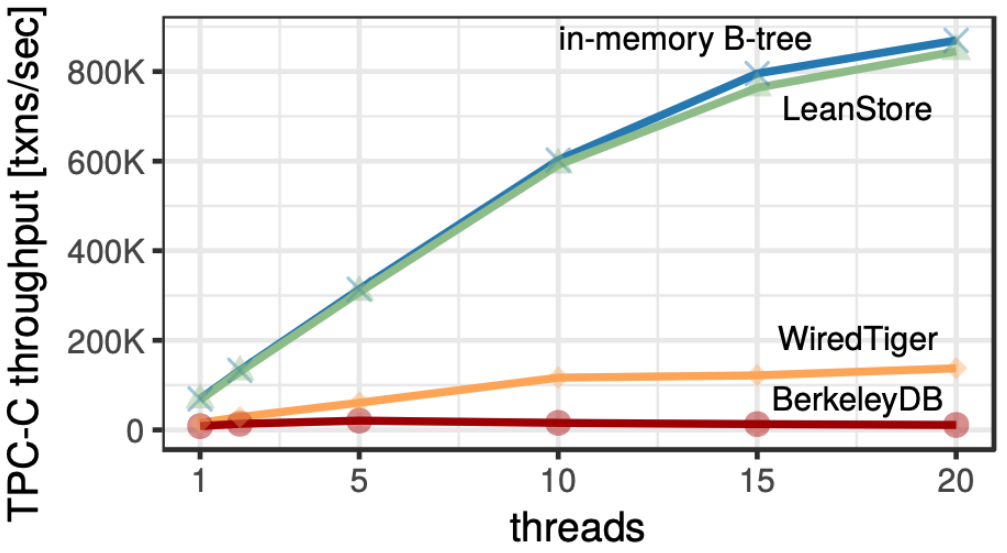


Fig. 8. Multi-threaded, in-memory TPC-C on 10-core system.

Agenda

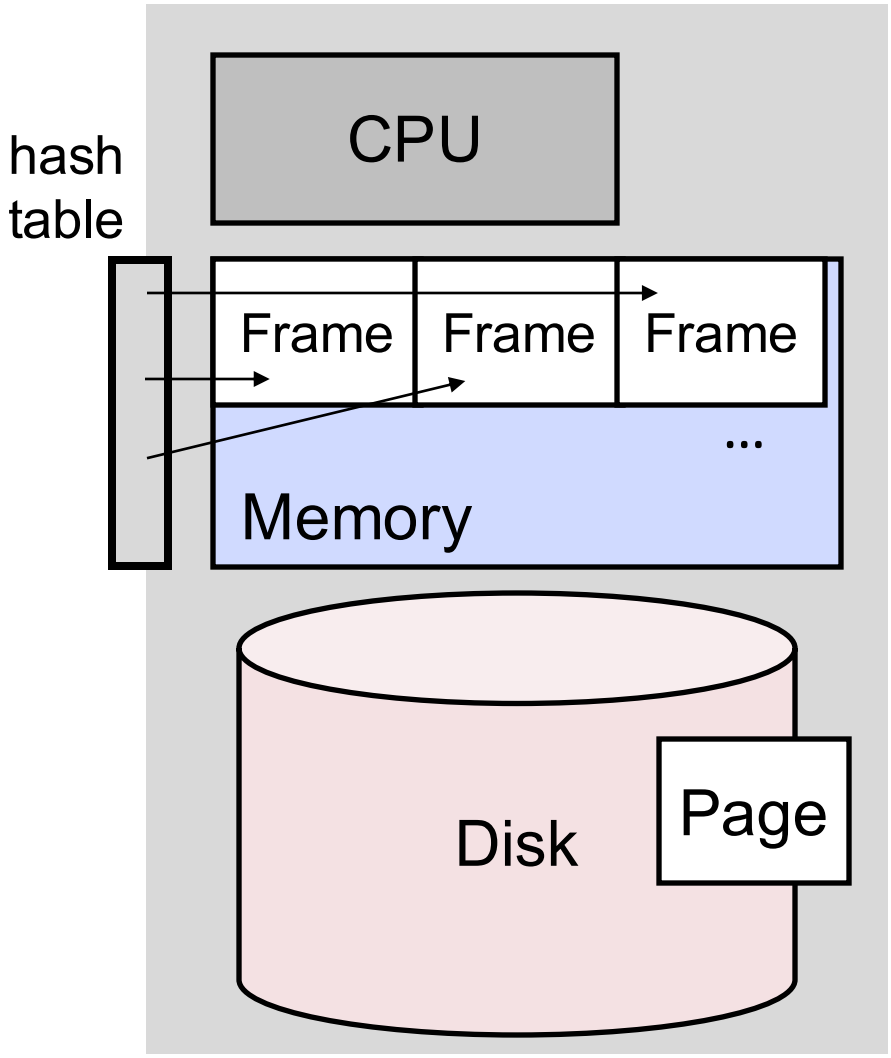
Main-memory DB

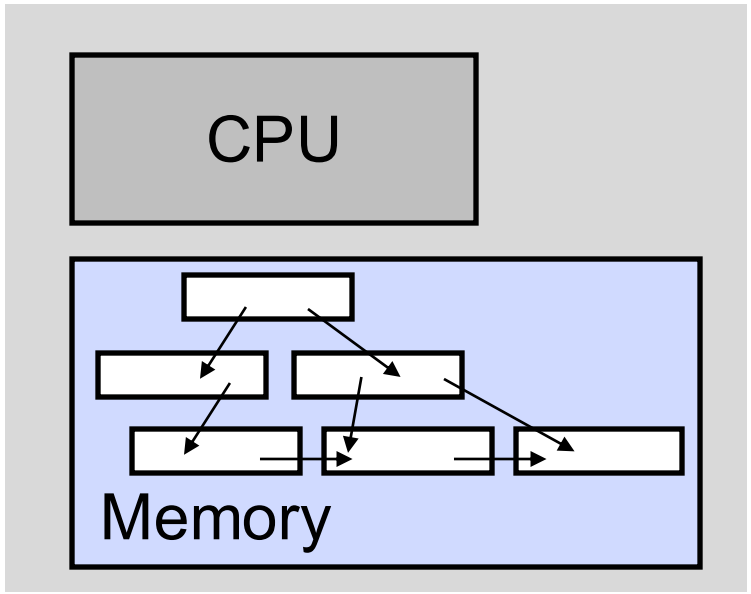
LeanStore design

- Pointer swizzling
- Page replacement
- Optimistic latching

Two-Tree

Main-Memory DB Architecture





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

No Indirection: reference data following pointers

Fine-Grained Buffer Management

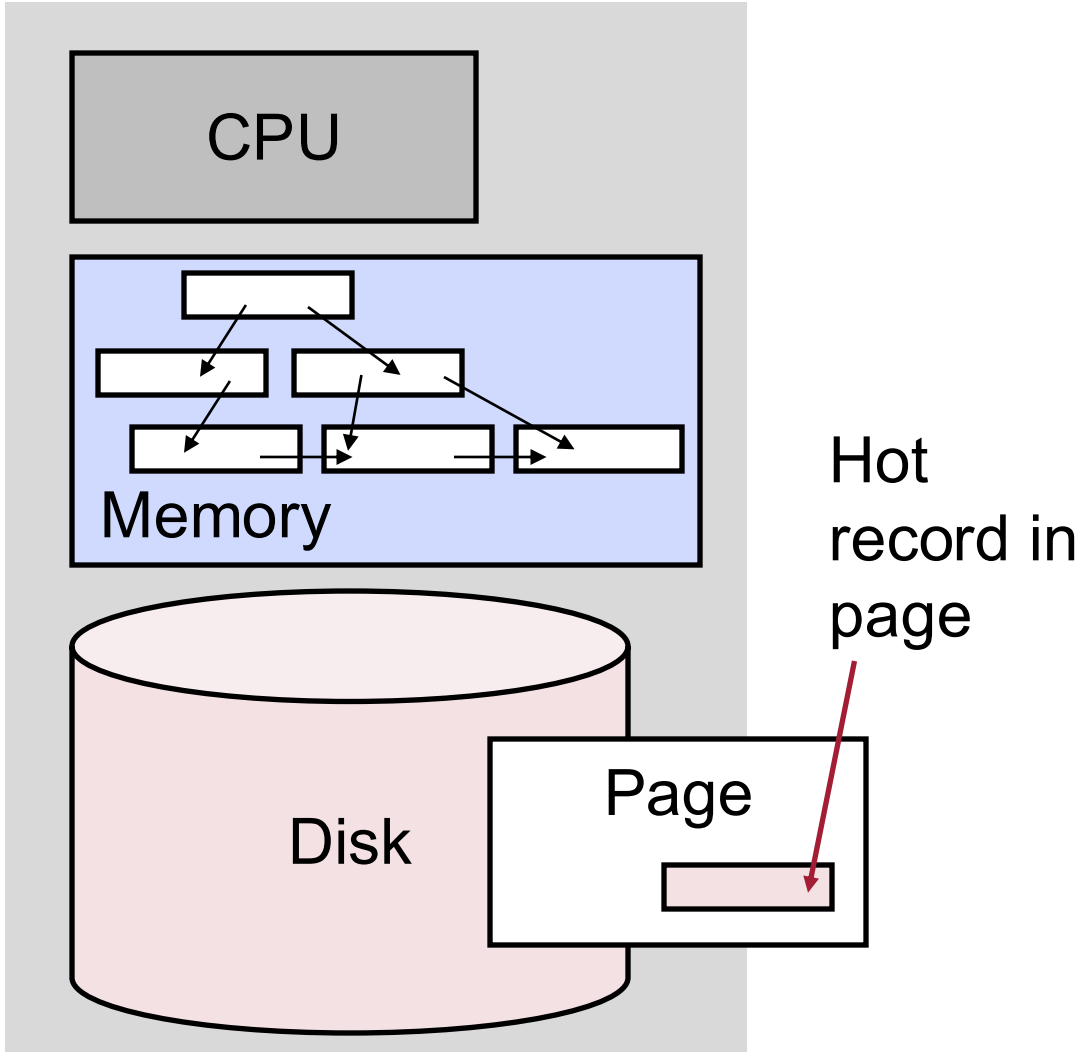
Data is often skewed

- Few hot records and many cold records in a page
- Inefficient memory usage and IO

Idea: cache data in record granularity

Challenges:

- Buffer management



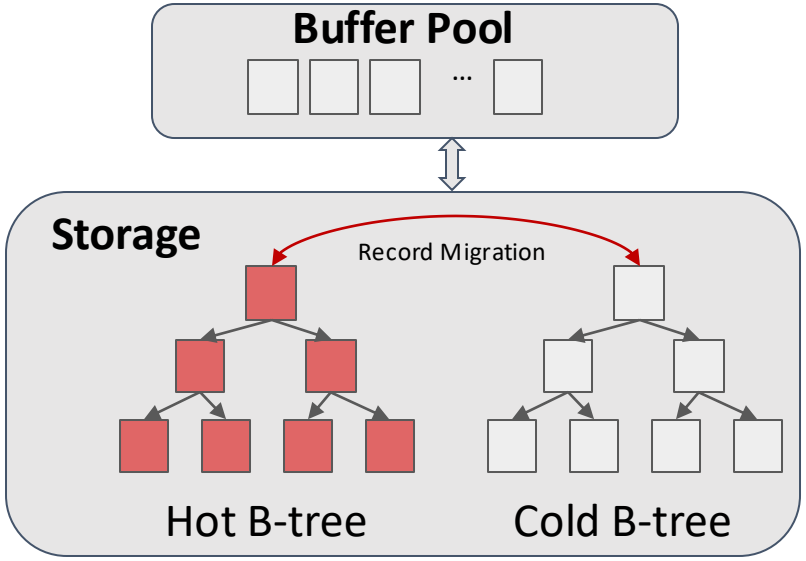
Two-Tree Architecture

A logical tree is implemented as two physical trees, each with its own buffer management

Move data between trees at record granularity

Different trees can be optimized independently

- Hot tree can fit in memory
- Hot tree can use memory-optimized structures



Record Migration

Migration of a record = **insertion** to a tree and/or **deletion** from a tree

- The tree structure is adjusted accordingly

Downward migration

- Clock replacement to find eviction candidate

Upward migration

- Probabilistic with a sampling rate D ($0 < D \leq 1$)

Inclusive versus exclusive

- Removal from bottom tree when inserting to the top tree?
- By default choose inclusive

Two-Tree Operations

- Point lookup
  - Search the top tree, if found, return
  - Otherwise search the bottom tree
  - Handle migration logic

- Update
  - If found in top tree, set dirty bit, update record (record will be written back to bottom tree during eviction)
  - Otherwise update in the bottom tree
  - Handle migration logic

Two-Tree Operations

- Insert
  - Perform Lookup, if does not exist, insert to top tree and set dirty bit
- Delete
  - Search top tree, if found, set delete bit
  - Otherwise insert placeholder record: empty payload + delete bit set
  - Delete applied to bottom tree when the record is evicted
- Range scan
  - Scan both trees
  - If a record exists in both trees, use the one in the top tree

Durability and Recovery

System transactions for record migration

- Lightweight, do not force log record to storage

Discussion question: do we need to log for migration at all?

- Two physical trees representing a single logical tree
- Maybe sufficient to recover only the logical tree

Evaluation — 1-Tree vs. 2-Tree

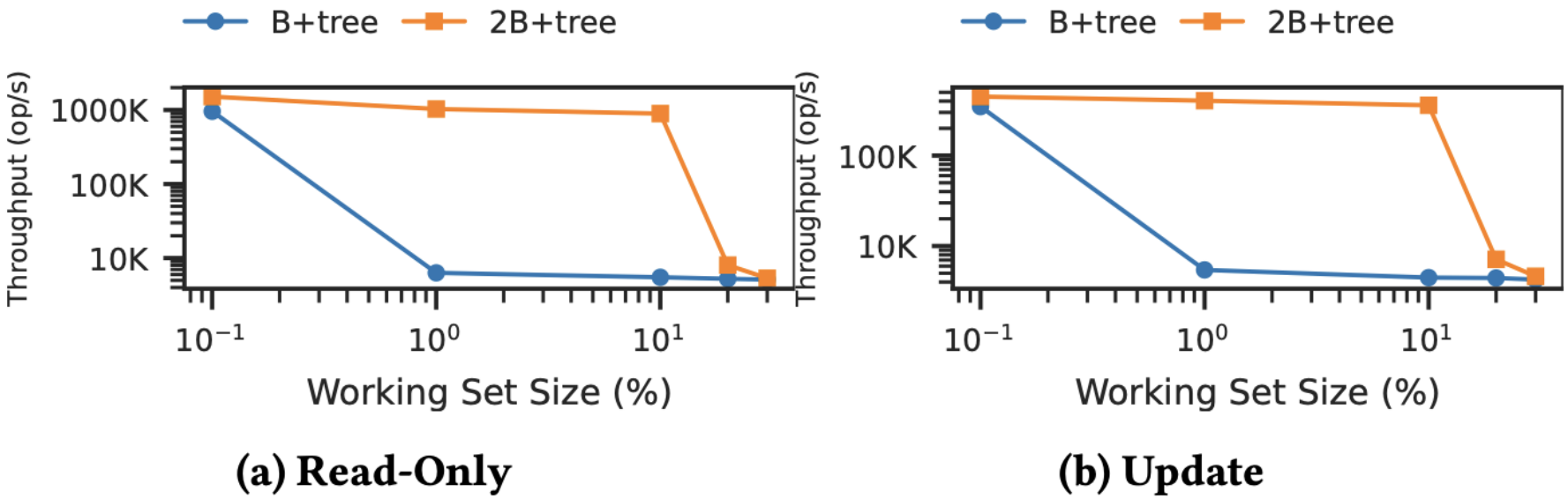


Figure 7: Throughput of 2B+tree and Single B+tree on Hotspot Request Distribution with Varying Working Set Size.

Benefits of 2-tree design

- More hot records can be cached

Evaluation — Top-Tree Implementation

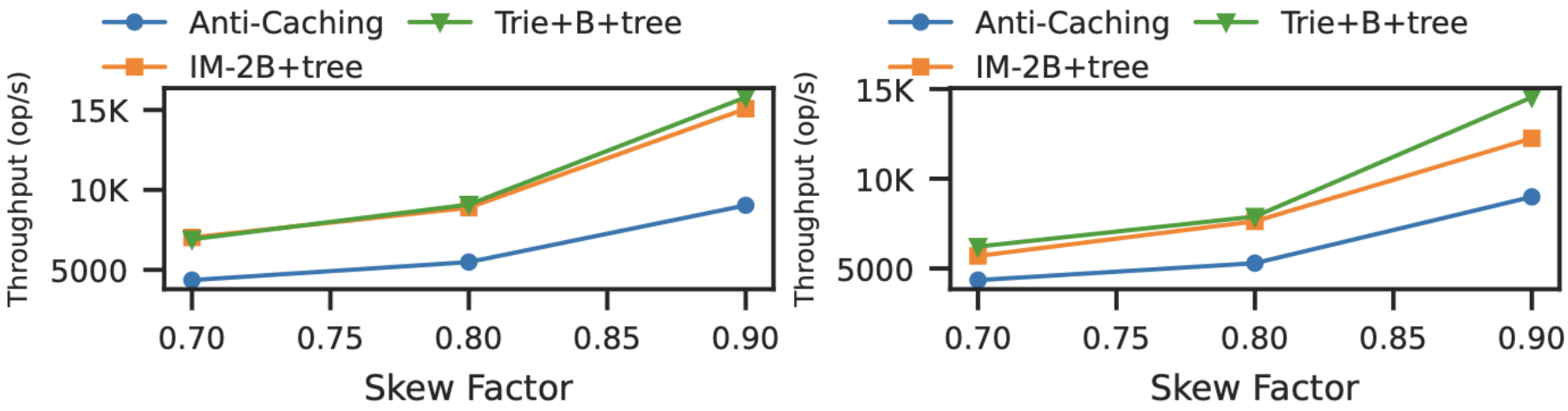


Figure 12: Throughput for 2-Tree Variations and Anti-Caching on Zipfian Request Distribution

Memory-optimized top-tree leads to better performance

Discussion

2-Tree might perform worse than 1-Tree for the following operations:

- **Scan**: always need to scan bottom tree
- **Negative lookup**: always need to search bottom tree

Q/A – Transaction Buffer Management

Adaptive and learning-based promotion?

Why sampling-based upward migration? Better alternatives?

Scale to multiple threads?

High migration cost when hot set changes rapidly?

Scale to a distributed, sharded environment?

Extend to non-tree structures (hash table)?

Next Lecture

Submit review for

Philip Lehman, S. Bing Yao, Efficient Locking for Concurrent Operations on B-Trees. ACM Transactions on Database Systems, 1981