LeanStore: In-Memory Data Management Beyond Main Memory

Víctor Leis, Michael Hauschild*, Alfons Kemper, Thomas Neumann

Technical University Munich

Abstract—DBA-based database systems use buffer managers in order to transparently manage data sets larger than main memory. This traditional approach is inefficient in managing the number of I/O operations, but also the major source of workload in comparison with in-memory systems. To avoid this overhead, in-memory database systems therefore abandon buffer management altogether, while modern building data sets not larger than main memory.

In this work, we revisit the 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 high overhead in comparison with an in-memory database system; however, it can be regarded a reasonable trade-off. On the other hand, it is fully transparent and can manage very large data sets effectively. Furthermore, due to low-overhead configuration, our implementation is also highly scalable on multicore CPUs.

1. Introduction

Managing large data sets has always been the main issue for database systems. Traditional systems cache pages using a buffer manager, which has complete knowledge of all page accesses and transparently handle page faults. In main memory, data is stored on-disk, and the buffer manager is responsible for managing the data. However, in modern in-memory database systems, the buffer manager is not used, and the data is stored directly in main memory. This approach is efficient in terms of performance, but it requires a large amount of main memory.

In this work, we revisit the 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 high overhead in comparison with an in-memory database system; however, it can be regarded a reasonable trade-off. On the other hand, it is fully transparent and can manage very large data sets effectively. Furthermore, due to low-overhead configuration, our implementation is also highly scalable on multicore CPUs.

Another reason for considering buffer managers on the performance of in-memory database systems (i.e., the main memory) is that modern in-memory database systems (i.e., the main memory) is that modern in-memory database systems are used in a variety of scenarios, such as in-memory analytics, data warehousing, and big data processing.

In this work, we answer this question differently: by designing, implementing, and evaluating a highly efficient storage manager called LeanStore. Our design provides a balance of similar functionality as a traditional buffer manager but without increasing its overhead. As Fig. 1 shows, LeanStore’s performance is very close to that of an in-memory database when running TPC-C. The reason for this is that it is fast as an in-memory page read involves a single, well-predicted fault instead of a costly main memory read.

We also achieve excellent scalability on modern main memory CPUs by avoiding fine-grained locking on the file path. Overall, if the working set is within the RAM, our design achieves the same performance as modern in-memory database systems. At the same time, our buffer manager can transparently manage very large data sets at backgrounds using only modern SSDs, thereby degrading smoothly as the working set wants to exceed main memory.
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

Only a small fraction of instructions execute useful work

Significant instruction count dedicated to buffer management

---

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.

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
Pages that reside in main memory are directly referenced using virtual memory addresses (i.e., pointers)
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
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
**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
Challenge 2: pages containing memory pointers should not be written to disk

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

hot (RAM) → swizzle

load, swizzle

cold (SSD) → evict

cooling (RAM) → unswizzle
Discussion Question:
Is clock replacement necessarily worse than cooling replacement?

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

load, swizzle

cold (SSD)
evict

hot (RAM)
swizzle

cooling (RAM)
unswizzle
Agenda

Main-memory DB

LeanStore design
  – Pointer swizzling
  – Page replacement
  – Optimistic latching

Experiments

Fine-grained in-memory data management
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 Reclamration

**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

![Diagram](image)

**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

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
Submit review for

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