CS 764: Topics in Database Management Systems
Lecture 5: Modern Buffer Management

Xiangyao Yu
9/21/2021
LeanStore: In-Memory Data Management Beyond Main Memory

Vitor Lisio, Michael Baechmann*, Allons Kemper, Thomas Neumann

LeanStore — In-Memory Data Management Beyond Main Memory

Abstract — In-memory database systems use buffer managers in order to transparently manage data sets that are larger than main memory. This traditional approach is effective at managing the number of IO operations, but it also the major source of overhead in comparison with memory-centric systems. In order to avoid this overhead, database systems frequently abandon buffer management altogether, which means building data sets too large for main memory. In this paper, we revisit this fundamental architecture and design a novel storage manager that is optimized for modern hardware.

1. 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 manages pages between disk and memory. By reading all data on-disk once per query, update data techniques, including database tables and indexes, can be handled completely and consistently.

While this design succeeds in minimizing the number of IO operations, it incurs a large overhead for on-disk secondary storage, which are increasingly common. In the wreaked buffer pool implementation [1], each page access requires a hard table lookup in order to translate a logical page identifier into an on-disk page. Even worse, in typical implementations the data structures involved are synchronized using multiple locks, which does not scale on modern multi-core CPUs. As Fig. 1 shows, traditional buffer managers implement like Hana [2] or WestEnd [3] therefore only achieve a fraction of the TPCC-C performance of an in-memory DBMS.

This is why in-memory database systems like Plume [3], Hikarion [3], Hana [4], HypeR [5], or Site [6] abandon buffer management altogether and scaled as so that data are directly stored in main memory and virtual memory pages are used instead of page identifiers. This approach is certainly efficient. However, in data sizes grow, asking users to buy more RAM than they are willing to pay for a single machine. Building an in-memory database can be expensive, but has drawbacks, including hardware and administration cost. For these reasons, at some point of any data-centric system’s evolution, its designers have to implement support for very large data sets.

LeanStore: In-Memory Data Management Beyond Main Memory

Beyond Main Memory

Fig. 1. Benchmark results on the TPCC-C benchmark (10X database).

Two representative proposals for efficiently managing large-RAM data sets in main-memory systems are Amazon’s RDS [7] and IBM’s DB2 [8,9,10]. In comparison with a traditional buffer manager, these approaches exhibit much more overhead. They are not capable of managing a replacement strategy over on-disk index and data. Either the indexes, which constitute a significant portion of the overall data, are not stored in RAM, or they represent a performance bottleneck, which makes these techniques less general and less transparent than traditional buffer managers.

Another reason for reconsidering buffer managers on the increasingly common NVMe-based high-end storage systems (SSDs), which are block devices, is that the page-size accesses to these devices may access multiple OS pages instead, as they are not limited by the relatively slow SATA interfaces. While modern SSDs are still at least 10 times faster than DRAM in terms of bandwidth, they are also cheaper than DRAM by a similar factor. Thus, for economic reasons [12], store buffer managers are becoming attractive again. Given the benefits of buffer managers, this 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 manager called LeanStore. Our design provides an alternative to 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 DBMS when using TPC-C. The reason for this is that LeanStore’s horizontal scalability allows fast accessing of an in-memory page merely involves a simple, non-prompted CPU commit rather than a costly hash table lookup. We also achieve excellent scalability on modern multi-core CPUs by avoiding lock-guarding overhead on the hot path. Overall, if the working set in RAM, our design achieves the same performance numbers on our in-memory database system. At the same time, our buffer managers can transparently manage very large data sets of background storage and, using modern SSDs, throughput degrades smoothly as the working set needs 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

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.

[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
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

---

<table>
<thead>
<tr>
<th>swip 1</th>
<th>8 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>(swizzled)</td>
<td></td>
</tr>
<tr>
<td>swip 2</td>
<td></td>
</tr>
<tr>
<td>(unswizzled)</td>
<td></td>
</tr>
</tbody>
</table>
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
**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 a 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

Load, swizzle

Evict

Swizzle

Unswizzle
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

load, swizzle

cold (SSD)
evict

hot (RAM)
swizzle
unswizzle

cooling (RAM)

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

Write \# of readers
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

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=NWxTOXJGSDhiekhwedXBOcG9qMjVkdz09
Submit review for

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