The Evolution of LeanStore

Adnan Alhomssi [1], Michael Haubenschild [2], Viktor Leis[3]

Abstract: LeanStore is a high-performance key-value storage engine optimized for many-core processors and NVMe SSDs. This paper provides the first full system overview of all LeanStore components, several of which have not yet been described. We also discuss crucial implementation details, and the evolution of the entire system towards a design that is both simple and efficient.

Keywords: storage engine; LeanStore; B-tree; caching; SSD; multi-core

1 Introduction

In-Memory DBMS. For more than a decade, main-memory database systems have been the focal point of research on high-performance transaction processing. Academically, this research program has been a tremendous success, introducing innovative techniques and achieving unprecedented performance results. However, the real-world adoption of pure in-memory database system has been limited. It is fair to say that even the most successful in-memory systems such as VoltDB [SW13], Hekaton [Di13], and HANA [Fä11] remain niche products. General-purpose transaction processing is still dominated by traditional (disk-based) database systems such as Oracle, SQL Server, and their cloud-native cousins Aurora [Ve17] and Socrates [An19].

Storage Cost Trends. The main-memory revolution was primarily fueled by rapidly shrinking DRAM prices. From 2000 to 2012 the price/byte for DRAM dropped by about 300× [HHL20]. This made it feasible to keep non-trivial databases in main memory. After 2012, however, DRAM prices have largely *stagnated* [HHL20]. And with Intel canceling Optane in 2022, persistent memory failed to achieve its promise as a replacement for traditional storage technologies.

Flash to the Rescue. In contrast to DRAM, the last decade has seen dramatic price reductions for flash. In 2005, the cost per byte for DRAM and for flash was comparable. Today, flash is about 20–50× cheaper [HHL20]. And flash SSDs have not just become cheap, with the introduction of NVMe they have also become very fast. For example, a Samsung PM1733 PCIe 4.0 SSD achieves up to 7 GB/s read throughput and 1.5 million random reads per second [Sa22a]. Commodity servers have dozens of PCIe lanes, which means that hardware

[1] Friedrich-Alexander-Universität Erlangen-Nürnberg, adnan.alhomssi@fau.de
[2] Salesforce, mhaubenschild@salesforce.com
[3] Technische Universität München, leis@in.tum.de

doi:10.18420/BTW2023-13

setups with 8 or 10 NVMe SSDs have become common (see Azure's Lsv2 [AW22b] and EC2's i3en [AW22a] instances). Consequently, servers with 10+ million I/O operations per second and an aggregated bandwidth rivaling DRAM are not just possible – but affordable and readily available. And the trend will continue with the upcoming PCIe 5.0 servers and SSDs [Sa22b], which will almost double I/O bandwidth once more.

NVMe-Optimized Systems. Neither in-memory nor disk-based systems even come close to being able of exploiting the capabilities of modern SSDs. Many of the design decisions of in-memory systems (e.g., small index nodes) are simply not suitable for storage on flash. Existing disk-based systems, on the other hand, look more conceptually promising (e.g., page-based data organization), but were developed when storage was slower by several orders of magnitude. As a result, a system such as PostgreSQL is completely *CPU*-bound on out-of-memory workloads on NVMe SSDs [HHL20]. Even though they are often treated that way, SSDs are not just fast disks – for good performance they require new DBMS designs.

LeanStore. The LeanStore [Le18] project started in 2016. The goal was to show that one can achieve performance close to in-memory systems without having to keep all data in memory. To be efficient and robust on modern hardware, LeanStore combines many of the modern in-memory optimizations (e.g., CPU and cache efficiency, lightweight synchronization [LHN19], contention avoidance [AL21]) with traditional decades-old DBMS techniques (e.g., buffer management [Le18], B-trees, paged storage, physiological logging, fuzzy checkpoints [Ha20]).

This Paper. While many of these novel techniques have already been published in separate papers [AL21; Ha20; Le18; LHN19] (or will be published in upcoming papers [AL23]), this paper provides the first overview of the full LeanStore system. Because research papers usually focus on a single contribution rather than the full system and the interaction of components, this includes many crucial unpublished aspects – such as the B-tree design, synchronization primitives, and logging optimizations. Doing so, we also describe the evolution of the system, which changed considerably over its lifetime. Many original design decisions had to be reconsidered in the light of hardware evolution, and over time we managed to radically simplify important implementation details. For example, the original paper [Le18] argues that a single latch for managing I/O operations is fast enough, and describes epoch-based memory reclamation. The former quickly proved wrong with fast SSDs, while the latter proved to be completely unnecessary.

Outline. The rest of the paper is organized as follows. Sect. 2 discusses related work. Sect. 3 then provides an overview of the functionality and key components of LeanStore. We then focus on four components, namely the B-tree in Sect. 4, the synchronization primitives in Sect. 5, the buffer manager in Sect. 6, and logging in Sect. 7. Sect. 8 experimentally compares LeanStore with two state-of-the-art storage engines for both in-memory and out-of-memory workloads. Finally, we summarize the paper in Sect. 9.

2 Related Work

Surprisingly, in the past decade research on designing flash-optimized database systems has been fairly sparse. Most of the new system developments focused on designs optimized for main memory (e.g., VoltDB [SW13], Hekaton [Di13], HANA [Fä11], HyPer [Ke12], Silo [Tu13]) or persistent memory. A notable exception is the Umbra [NF20] system, which adopted LeanStore's swizzling-based buffer management design, but extends it with support for variable-sized pages. There are two state-of-the-art open source storage engines optimized for flash that should be mentioned. WiredTiger [Mo22], which is the default storage engine of MongoDB, has several conceptual similarities with LeanStore, including reliance on B-trees and pointer swizzling. RocksDB [Do21], in contrast, is the most prominent LSM-based storage engine, optimizing for lower write amplification rather than read performance. We experimentally compare with both systems in Sect. 8 and descriptively in the remaining of this section.

Unlike LeanStore, **WiredTiger** uses a separate representation for on-disk and in-memory B-Tree nodes. In-memory nodes are dynamically allocated and not hosted on a fixed-size page like traditional buffer-managed systems do. On-disk nodes images are stored in immutable block that are grouped in files. In-memory node use prefix compression while on-disk blocks are mostly compressed with snappy algorithm. An index operation follows virtual memory pointers until it hits a non-resident (unswizzled) node. In this case, it reads the file and builds an in-memory node out of it. Updating a key in a leaf inserts a new entry in the node's per-key skip list. When the node is full and it is time to evict, WiredTiger reconciles, i.e., serializes, the node and merges all updates in the node to create the disk image. In LeanStore, we update nodes in-place and write the in-memory node as it is on SSDs without any serialization overhead.

To synchronize indexes, WiredTiger uses lock-free algorithms where LeanStore uses optimistic lock coupling. To make sure a thread is following a valid pointer, WiredTiger pushes the node's pointer to a hazard list before it accesses it. Nodes pointed by this list are excluded from memory reclamation or eviction. The processing thread removes the pointer once it leaves the node. Compared to LeanStore's optimistic lock coupling and pointer swizzling, hazard pointers in WiredTiger cost one more memory flush (fence) to publish the new hazard pointer for each node access.

RocksDB relies on the Log-Structured Merge-tree (LSM) data structure to reduce (random) write and space amplification and prolong the lifetime of SSDs which have limited endurance. By choosing B-trees as the foundational data structure, LeanStore is a more read-optimized design. Nevertheless, our design reduces write amplification by using 4 KiB pages and increases space utilization for B-Trees through the XMerge [AL21] technique that optimistically merges under-fill neighboring nodes.

The first layer "MemTable" in the LSM resides in memory and absorbs all write operations. Once it reaches a configured size, RocksDB flushes its content to an immutable Sorted

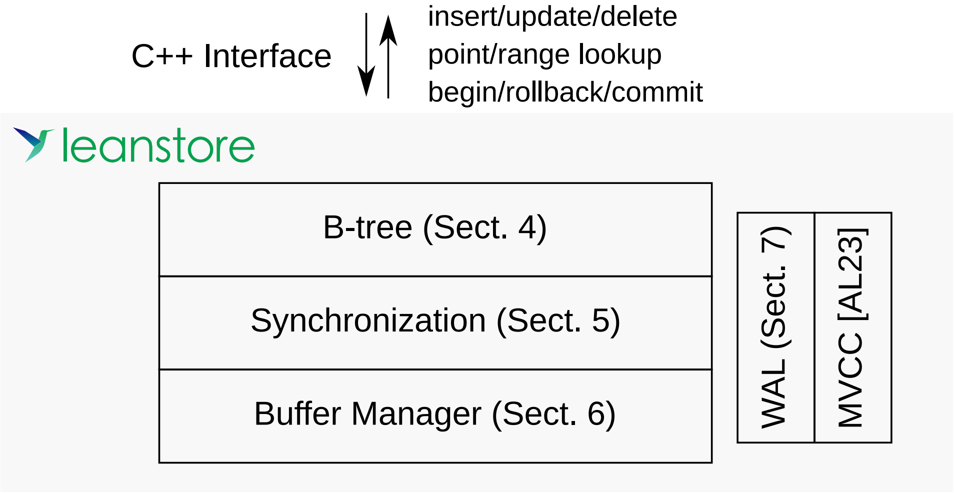

Fig. 1: Overview of main system components

String Table (SSTable). The earlier versions of RocksDB only supported single-writer to the MemTable. The newer ones allow multiple writers but only for the skip-based implementation. This signals how multi-core CPU support came a second thought where LeanStore is built up from the ground to scale on many-core with optimistic lock coupling.

To cache SST blocks, RocksDB relies on third-party allocator to manage its buffer pool instead from implementing its own. Concretely, RocksDB uses jemalloc to cache KV pairs from SSTs in variable-length blocks [Do21]. However, this reliance on jemalloc leads to fragmentation and performance issues that differ between workloads and instances. RocksDB user has to set two memory budgets. The first one for the size of the first LSM in-memory layer (called MemTable) – which also impacts the compaction strategy. The second one for caching which should account for the overhead by the allocator. The complex design and the resulting different configuration parameters made Tuning memory configuration is one of the LeanStore design does not suffer from this complication. We use a single buffer pool with exact capacity and fixed-size pages to host all kind of nodes.

3 LeanStore System Overview

Functionality and System Overview. LeanStore is a storage engine supporting basic key value operations (insert/update/delete, point/range lookup) and transactional semantics (begin, commit, rollback). This functionality is exposed through a C++ interface, i.e., as an embeddable library rather than a server process. Systems with comparable functionality include RocksDB and WiredTiger, which are often used as building blocks for full-blown data management systems. Fig. 1 illustrates the main components and the layering of the system. In the following, we provide a high-level overview of each component.

B-tree. The main data structure in LeanStore for both indexing and storing data is a B$^+$-tree. Keys and values are opaque byte sequences and the application is responsible for transforming and interpreting the data[4]. Although B-trees are not as fast [Wa18] as pure in-memory data structures such as ART [LKN13] or HOT [Bi22], they offer reasonable in-memory performance and work well in most out-of-memory situations. We implement several optimizations to narrow this gap to in-memory data structures as described in Sect. 4. In comparison with Log-Structured Merge (LSM) trees, which offer lower write amplification, B-trees are faster for reads and are simpler to implement while having far fewer parameters. Nevertheless, it is possible to offer LSM-trees in LeanStore as an additional option, and we already experimented with a prototype LSM implementation.

Synchronization Primitives. Modern CPUs have dozens of cores, which means that overall performance is often determined by scalability rather than single-threaded performance. In LeanStore, we implement an innovative synchronization framework that is scalable, has low overhead, and that is generic and easy to use. The synchronization primitives and their implementation is described in Sect. 5.

Buffer Manager. The first LeanStore paper [Le18] demonstrated that it is possible to implement buffer management with very little overhead in comparison with in-memory systems. The key ideas of that paper are pointer swizzling [Gr14] and a lightweight replacement strategy. With pointer swizzling, cached pages are directly accessible through pointers – avoiding any indirection data structure necessary in traditional buffer manager designs. An access to swizzled pages also does not incur any overhead for the page replacement algorithm. Once the system is running low on free memory, candidate pages for eviction are randomly unswizzled and put into a *cooling stage*. Two things can happen now: a candidate page is either fairly hot, in which case it will be swizzled again (without incurring I/O); or it is cold, in which case it is eventually evicted once it reaches the end of a FIFO list. Sect. 6 describes how we have significantly simplified page eviction and how we optimized I/O management to catch up with the increasing SSD performance.

Snapshot Isolation and MVCC. LeanStore supports snapshot isolation through an implementation of Multi-Version Concurrency Control (MVCC). Surprisingly, we found that the OLTP performance of state-of-the-art out-of-memory MVCC schemes collapses in the presence of long-running OLAP queries (and vice versa). In an upcoming paper [AL23], we propose a design that solves this problem. To do this, we combine (1) a novel out-of-memory commit protocol that enables efficient fine-granular garbage collection, (2) an auxiliary data

[4] LeanStore stores both keys and values/payloads as is. For payloads this is not a problem, but it presents an additional challenge for keys. For example, if we have a integer key and store it in the index as signed big endian, range scans will not yield the expected order. To solve this, it is necessary to transform keys (e.g., swap bytes for little endian). This is a widely-used technique that Graefe [Gr11] calls *normalized keys*.

Tab. 1: The motivation behind the design decisions and techniques behind LeanStore.

Technique	Motivation	Technique	Motivation
Pointer Swizzling [Le18]	Large Buffer	Partitioned I/O Stage	NVMe IOPS
Optimistic Locking [LHN19]	Scale Reads	Batched Async Writes	NVMe IOPS
Contention Split [AL21]	Scale Writes	Group Commit	NVMe IOPS
Distributed Logging [Ha20]	Scale TXs	XMerge [AL21]	Space Utilization
Tuple-Wise Depend. Tracking	Scale TXs	4 KiB Pages	Write Amplification

structure that moves logically-deleted tuples out of the way of operational transactions, and (3) an adaptive version storage scheme.

Logging. In disk-based database systems, transactional durability is guaranteed by a write-ahead log (WAL). Because the WAL is usually a centralized data structure, it becomes a point of contention that prevents multi-core scalability. In a SIGMOD paper from 2020 [Ha20], we propose a solution that uses one WAL per thread, and that ensures correctness through a lightweight page-based tracking mechanism called Remote Flush Avoidance (RFA). Since then, we refined this scheme to further reduce unnecessary log flushes as described in Sect. 7.

In Tab. 1, we list the design decisions and techniques we have implemented in LeanStore as a response to the different hardware trends like large DRAM buffers, NVMe SSDs and many-core CPUs. The ones without citations are presented in this paper.

4 B$^+$-Tree

Based on a classic slotted page B-tree node layout, we implement a number of B-tree optimizations that have been proposed in the literature.

Node Layout. Fig. 2 shows the layout of a leaf node storing three keys (“http://fau.de”, “http://tum.de”, and “http://uni-jena.de”). The slots at the front of the page contain the offset and length of the key/payload stored at the end of the page. An inner node looks the same except that it stores child pointers as a payload instead of values.

Fence Keys and Prefix Compression. Fence keys [Gr04] simplify the implementation of prefix compression, and they are stored in every node. They are immutable and determined on node initialization, i.e., after a split or merge. The lower and upper fence values are the exclusive lower and the inclusive upper bound for the keys that could be potentially stored in that node. When a new B-tree is created, the initial leaf page conceptually has its fence keys set to special values $-\infty$ and ∞. All other fence keys are given automatically by the split (or

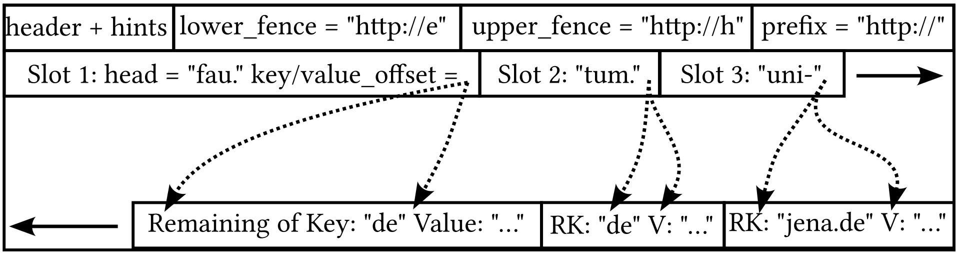

Fig. 2: Our variable-length keys and values B$^+$-tree features fences, prefix compression, heads extraction and hints to accelerate binary search and save space

"separator") keys during node splits. Using the fence keys, it becomes straightforward to implement the well known key prefix compression technique [BU77]. The prefix is derived from the fence keys by taking the longest common prefix of both fence keys.

Speeding Up Binary Search with Heads. Although the basic slotted page layout enables binary search, it does so quite inefficiently because every key comparison will require dereferencing a pointer to obtain the key – leading to many cache misses. To reduce the number of cache misses, Graefe and Larson proposed "poor man normalized keys" [GL01], which are a fixed-size prefix of the key. In each key slot, we therefore store the first 4 bytes of the key in the equivalent unsigned integer representation and call it *head*. By default, all keys are stored as strings and we use lexicographical order to compare keys. That means on little-endian systems (the majority of today's CPU architectures), we fold integer and compound keys using byte swap operations to maintain the same sort order for their serialized string representation. This allows efficient less-than comparisons based on the integer representation. With the heads available, we can first perform binary search solely on the heads in the slots array using cheap integer comparison. Only for heads that equal the lookup key, it becomes necessary to retrieve the full key and perform a byte-wise comparison.

Avoiding Binary Search with Hints. To accelerate binary search in B-Tree nodes, we further implement the *hints* optimization, which can be considered a form of in-page micro-index [Lo01]. Instead of starting the binary search over the whole range, i.e., [0, #slots], we try to first shrink the range using cache efficient search over hints. These hints are stored in a dense array with a fixed number of entries (we use *#hints* = 16). The distance between keys from which the heads are taken is constant and defined as `distance = #slots` ÷ (`#hints` + 1). The original position of each hint key is determined using a simple formula. As shown in List. 1, after finding the candidate range in hints, we can translate it to the corresponding range in the node keys' slots array. We also store just a copy of the first four bytes (*hint_size*) instead of the whole key in the hints array. Because all hints fit in a single cache line, we can perform a quick binary search using integer comparisons to minimize the candidate range (lower, upper) for the key we are looking for. We also found updating the hints structure to have negligible overhead.

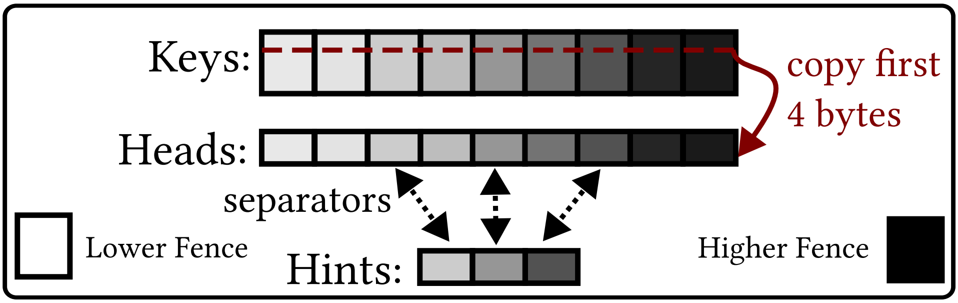

Fig. 3: Hints optimization: the heads of equally-distanced keys are stored as 4-byte integers, inner node binary search starts by narrowing the range using quick binary search over hints

```
updateHints() { // hints_count = 16, hint_size = 4 -> 64 Bytes
   u32 dist = slots_count / (hints_count + 1);
   for (u16 i = 0; i < hints_count; i++)
      hint[i] = key[dist * (i + 1)].substr(0, hint_size);
}

searchHints(u8* key, u16 key_length) { // Binary search starts with returned range
   u32 key_head = head(key, key_length); // first hint_size bytes of key
   u32 pos, pos2;
   for (pos = 0; pos < hint_count; pos++) // skip smaller hints
      if (hint[pos] >= key_head)
         break;
   for (pos2 = pos; pos2 < hint_count; pos2++) // find equal hints
      if (hint[pos2] != key_head)
         break;
   // convert pos and pos2 to full key range
   u32 dist = slots_count / (hints_count + 1);
   u32 lower = pos * dist;
   u32 upper = (pos2 < hint_count) ? ((pos2 + 1) * dist) : slots_count;
   return {lower, upper};
}
```

List. 1: Narrowing binary search range using efficient integer search over hints

Contention Split and XMerge. In contrast to deterministic data structures such as tries, the same set of keys may result in different B-tree structures (usually only depending on the insertion order). We exploit this observation by dynamically and adaptively optimizing the B-tree structure using the *Contention Split* and *XMerge* optimizations. Contention Split reduces unnecessary latch contention (i.e., where a page contains more than one hot tuple), by splitting the node – even though the node may not be full. The second optimization, increases the fill factor of B-tree nodes by opportunistically merging X neighboring nodes into X-1 nodes. Both optimizations have been described in detail previously [AL21].

5 Practical and Scalable Synchronization

5.1 Optimistic Lock Coupling and Buffer Management in C++

With optimistic latching, we eliminated unnecessary read contention for short page reads [Le16]. The original implementation relied on a single atomic version counter and a spin lock implementation. This simple design is not robust enough to handle the variety of workloads that a database has to deal with [Bö20]. Reverse CPU priority and unfairness are examples for anomalies that such spins locks exhibit.

Hybrid Latches. In the latest iteration of the LeanStore design, we use hybrid latches with OS support that allow a "pessimistic shared" and an "exclusive" mode besides the original optimistic mode. We implemented them using a std::shared_mutex next to a std::atomic<uint64_t> which serves as the version counter. With the three options in place, we can use the most suitable one for each operation: For long page reads, e.g., in scans, we directly use the pessimistic shared mode, which also simplifies the scan logic compared to our original implementation. Before, we had to worry about concurrent modifications could lead to restarts while scanning a page. For short reads in leaf nodes and for inner node traversal, we first try to latch optimistically and immediately fallback to the pessimistic shared mode if we find the latch is already locked. Page modification operations, i.e., insert, update, and delete, latch the page in exclusive mode.

Page Guard. To help with the complexity of buffer management and the different latching modes, we use Page Guards to abstract some of the complexity away from the data structure implementation. Page Guards are a C++ template class that handle latching and buffer management and provide a simple interface to implement a data structure with. They follow the known Resource acquisition is initialization (RAII) programming idiom. Once a page guard is created on the stack in certain latch mode, it keeps the page latched in that mode until the guard object goes out of scope. This protects against leaking latches for pages that we forget to unlock in Code. We also use the page guard to access the underlying data structure by overloading the "->" operator, which is helpful to make sure we are accessing a latched page. In List. 2, we show how we use page guards to implement the B-tree insert method. For tree traversal, we use the optimistic guard to avoid contention on inner nodes. Once we locate the required leaf node, we upgrade to exclusive mode by constructing an ExclusivePageGuard from the moved OptimisticPageGuard. The overloaded arrow operator (->) returns the underlying buffer managed object which is the BTreeNode in our case.

5.2 Implementing Restart in C++

One consequence of our optimistic latch implementation and the fact that we do not hold a latch on a parent page while reading one of its children from storage is that we have to make data structure operations restartable. Unlike optimistic concurrency control, where

```
void insert(key, value) {
   while(true) { // restart until success
      jumpmuTry() {
         // Traverse inner nodes optimistically
         OptimisticPageGuard<BTreeNode> p_opt_guard;
         // Swizzle the root_swip (load the page) if necessary and
         // optimistically latch the page (load the version).
         // It can trigger restart if root_latch version has changed in-between
         OptimisticPageGuard<BTreeNode> c_opt_guard(root_swip, root_latch);
         while(c_opt_guard->is_inner) {
            // Binary search to find the child swip
            Swip &c_swip = c_opt_guard->searchInner(key);
            p_opt_guard = std::move(c_opt_guard); // reassigns without releasing latch
            // Checks if p_opt_guard has been modified before following c_swip
            c_opt_guard = OptimisticPageGuard(p_opt_guard, c_swip);
         }
         // Latch the leaf in pessimistic exclusive mode
         ExclusivePageGuard c_ex_guard(std::move(c_opt_guard));
         p_opt_guard.release(); // parent is not needed anymore
         if(c_ex_guard->canInsert(key,value)) {
            c_ex_guard->insert(key, value);
            jumpmu_return; // success
         } else { // Leaf is full. We can't immediately split because
           // we released the parent. Thus, we copy the inclusive
           // upper bound and release all latches.
           upper_bound = c_ex_guard->upper_bound;
           c_ex_guard.release();
           // Then find and split the leaf that covers given upper bound
           // starting from the root
           trySplit(upper_bound); // Next iteration will find space in leaf
         }
      } jumpmuCatch() {} // Jumps are handled simply by restarting from root
   }
}
```

List. 2: B-tree Insert Code Illustrating Page Guard Usage we potentially restarts the whole transaction at the end, low-level optimistic latching can force the index operation to restart any point in the code. This makes optimistic latching more challenging to adopt. In the following, we discuss the options in C and C++ that one could use to implement such restarts. Then we discuss our custom solution *JumpMU* that allows us to implement scalable and efficient restarts while maintaining readable code.

Goto and Labels. The *goto* statement immediately transfers control to another statement in the same function. When we detect a concurrent write to the page we optimistically read from, we can *goto* back to the first statement in our data structure operation. However, this works only in the same stack frame (same function call in C++). Storage engines are usually complex and are composed out of several components like a buffer manager and a B-tree with several functions calling each other. With goto, a function call has to inform its caller if it returned successfully or faced a synchronization error that forces a restart. Integrating these checks everywhere in the code wherever an optimistic read could occur is very wearisome, error-prone, and makes the code hard to read.

C++ Exceptions. Another native option in C++ is using *exceptions*. Exceptions at first sight look like a very good fit. They support jumping up several stack levels to the place where the data structure started its operation. Also, they automatically unwind the stack and call the destructors along the path to the exception handler. However, the standard exceptions implementation does not scale with many CPU cores, because common implementations (GCC and LLVM) acquire a global lock every time an exception is thrown. Although the scalability issue can be mitigated by avoiding the global lock that protects the list of dynamically loaded objects, the CPU cost of handling an exception remains high. A C++ exception is handled in two phases. The first phase traverses the stack up until it finds a landing pad that handles the thrown exception. The second phase starts from scratch and unwinds each call stack on the way up to the exception handler. This process costs around 10K instructions and lies on the hot path of B-tree accesses in our engine. For instance, if the exclusive lock acquisition for a child node failed after we managed to lock the parent, then the restart process will be triggered after around 10K instructions. During this time, the parent remains exclusively locked which also affects scalability besides being inefficient.

Long Jump in C. The C standard library provides a mechanism to change control flow back to a certain checkpoint set by the user across multiple levels of call stacks. The function *setjmp* sets this checkpoint which can be used later as a landing pad by the complementary function *longjmp*. For storage engines written in C, this mechanism is all what one could need to implement restarts efficiently. However, the correct stack unwinding that C++ takes care of automatically for the programmer does not happen with longjmp. This means that none of the destructors of the objects allocated on the stack between longjmp and setjmp will be called, which can lead to resource leaks and many other problems. Thus, this mechanism alone is no viable option either.

JumpMU: Long Jump with Manual Stack Unwinding. To implement restarts efficiently, we propose a solution that combines the benefits of C++ exceptions and the C longjmp function at the cost of additional destructor tracking that could be also automated. We call our solution *JumpMU* where MU stands for manual unwinding. JumpMU builds on setjmp and longjmp and provides a C++ safe variant by manually calling all non-trivial destructors that the C longjmp alone would. Concretely, just before longjump is called, JumpMU destructs all statically created objects on the stack since the last setjmp – effectively unwinding the stack. Objects without non-trivial destructors are irrelevant for JumpMU. To achieve this efficiently, we split the work between compile-time and run-time. At run-time, we maintain a stack of two pointers: one to the object we destruct when we jump and one to its destructor function. The order in which the pointers pair is inserted corresponds to the order of objects construction on the thread stack. An object constructor pushes the needed pointers to destruct it into JumpMU stack and its destructor pops it from the stack. At compile-time, we augment the constructors and destructors with the necessary instructions to keep the JumpMU stack in sync, i.e., push and pop pointers.

JumpMU: Interface. We provide a similar programming experience to C++ exceptions by using C macros as shown in the example in List. 2. Similar to the C++ *try* keyword, we use the *jumpmutry()* to define a code block that can trigger a restart. Implementation wise, jumpmuTry calls setjmp and saves the current environment in a thread local array that holds multiple environment structs so we can nest setjmps. Once a jump is triggered – by calling jumpmu::jump(), the control is transferred to the block defined in *jumpmuCatch()*.

JumpMU: Current Caveats. When we force the control flow out of a jumpmuTry block, we have to care of destroying the setjmp environment that was created by the jumpmuTry block. For now, we use extra macros for every keyword that could move the control flow out of the surrounding try block. This includes jumpmu_continue, jumpmu_break and jumpmu_return. Moreover, the current implementation requires the user to manually annotate all objects definition that lie in a jump range with JumpMU macros. All of the above can be automated using a compiler plugin, which we leave for a future work.

6 Buffer Manager

In this section, we describe the changes that our buffer manager design witnessed since its introduction in the original LeanStore paper [Le18]. The changes are geared towards simplification and better scalability. Although our initial designed delivers very competitive in-memory performance that is similar to an in-memory system, its implementation remained complicated. We revised our design and managed to simplify it significantly as we show in the following.

6.1 Memory Reclamation is Unnecessary

In the original LeanStore design, we implemented an epoch-based memory reclamation. The epoch-based approach was not only invasive and complex to maintain, it also lacked

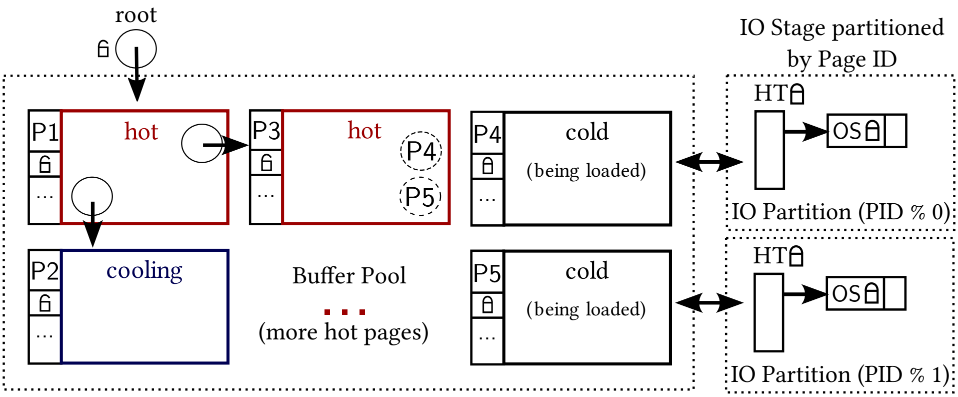

Fig. 4: The revised buffer manager design obliterates the need for a cooling stage and partitions the I/O stage by the page id to avoid contention on the hash table latch

robustness because a single slow thread could impede the advancement of the current global epoch which hinders the eviction of pages. Surprisingly, although memory reclamation seems mandatory for optimistic and lock-free synchronization, we eventually realized that this component is not needed after all. Thus, we could simplify our system by ensuring that 1) we do not reset or change the address of the atomic version counter of any buffer frame and 2) we do not release the memory allocated to the buffer back to the operating system. Condition 1 can easily be satisfied, and condition 2 is generally true in buffer managers. Thus, memory reclamation is completely unnecessary. We consequently we removed the code and got rid of the robustness issues of epoch-based reclamation.

6.2 Page Replacement

Lightweight Replacement Strategy Without a Cooling Stage. To achieve near in-memory performance, we aim at removing any overhead from the hot path, i.e., from read-only access to frequently visited pages. To that end, we designed our replacement strategy right from the start to work differently than in a traditional buffer manager. Instead of tracking the hot set of accessed pages, we identify the cold set of unused pages in a background fashion. Our first proposal *LeanEvict* follows this design principle [Le18], but did so using an unnecessarily complicated *cooling stage*, which was also vulnerable to contention. In this paper, we propose replacing LeanEvict with the well-known Second Chance replacement strategy. Second Chance does not add any overhead to the hot path and works in the same background fashion as LeanEvict. At the same time experimental results show that it does not lead to worse eviction candidates than the much more elaborate LeanEvict. In the following we describe our implementation of Second Chance in LeanStore.

Swizzling-Based Second Chance Implementation. Whenever free pages are needed, we pick a set of 64 random buffer frames and load their status. Then, we iterate over them and "cool the hot" (swizzled) buffer frames by 1) setting the most significant bit in the parent's swip that points to them 2) changing their buffer frame status to "cool". Any page that is already cool gets written back to disk if dirty, or evicted immediately otherwise. Note that with second-chance, the whole cooling stage, which includes a hash table and FIFO queue, is not needed anymore. Moreover, a cool swip holds a virtual memory pointer also when it is cool but the pointer is tagged to mark the page it points to as cool. This is in contrast to our previous design where a swip to a cool page gets unswizzled and replaced by the page identifier. Accessing a cool page does not require any access to the cooling hash table as is the case in LeanEvict. A hot page access does not require any extra work with our Second Chance implementation because we encode the status of the child node in the swip that points to it. If the cooling bit is not set, then it follows the swip pointer as usual. Otherwise, in the cold page access case, the bit is unset in the swip and the buffer frame status is changed back to hot.

No Parent Pointers with Top Down Tree Traversals. In order to evict a page, LeanStore has to unswizzle it by replacing the virtual memory pointer to the page in its parent node with the page identifier. This means we need to be able to get the parent node of each node we want to evict. The initial LeanStore implementation [Le18] achieves this by storing a parent pointer in each buffer frame. Although this seems an efficient way to reach the parent, it imposes several restrictions and complicates the system. Concretely, whenever we split a B-tree node, we have to update the parent pointer in the buffer frames of half of its child nodes. Also, care needs to be taken when latching the parent, as in general lock coupling in B-trees is based on the invariant that latches may only be acquired top-down. In our current design, we refrain from storing parent pointers and use a generic *findParent* method that returns a handler to the parent by traversing the data structure via its usual access path from the root down to the requested node. The same mechanism is also used when splitting or merging a node in the B-tree, as these also require access to the parent node.

6.3 Scaling to Multiple NVMe SSDs

For efficiency and correctness reasons, the system must synchronize I/O requests to prevent different threads from loading the same page into possibly different locations in the buffer pool simultaneously. To this end, we use an I/O stage to explicitly serialize I/O requests by page identifier (PID) like traditional buffer managers. The I/O stage uses a hash table that maps each PID to its state and a waiting queue that other threads sleep in until the page I/O finishes. With one SSD, the single I/O stage with the global mutex in the original LeanStore design from 2018 was enough to saturate the SSD bandwidth. With today's PCIe 4.0, we can attach an array of, e.g., ten NVMe SSDs [HHL20], where each drive can deliver up to 7000 MB/s read bandwidth and 3800 MB/s write bandwidth. In such a system, we need high concurrency to issue a large number of parallel I/O requests to fully utilize all SSDs. The global mutex that protects the I/O hash table[5] would quickly become a scalability bottleneck and prevent us from reaching exploiting the maximum bandwidth that modern flash drives can deliver.

Partitioned I/O Stage. We partition our I/O stage according to the page identifier (PID) as shown in Fig. 4. The least significant bits of a Page ID determines the I/O stage partition that we will use to handle the I/O operation. Page IDs are assigned randomly and as a result, the I/O operations are evenly distributed among the different partitions regardless of the workload characteristics. This reduces the contention on the mutex that protects the I/O stage significantly and allows more worker threads to read pages in parallel. The number of partitions is a freely configurable parameter. As a heuristic, we set the number of I/O partitions at least equal to the number of threads in the system to maximize bandwidth. Note that the buffer pool is still not partitioned and a buffer frame can host a page that is assigned to any of the I/O partitions.

Background Page Provider Threads. In the original design, foreground worker threads cool and evict the pages. This adds latency to transactions that suddenly get paused to evict the needed free pages. We deviate from this design and implement a page provider thread routine that cool and evict pages in the background [Ha20]. To fully utilize the available IOPS, multiple page provider threads can be launched in parallel. Moreover, to write back dirty pages, each thread submits *batches* of write commands to the kernel using a native asynchronous interface like libaio or io_uring. As we have removed the cooling stage, synchronizing the background threads become simpler.

7 Logging

In our first logging paper [Ha20], we describe a scalable distributed logging scheme that uses persistent memory (pmem) as a first landing stage for WAL records. However, the slow adoption of pmem and its total abandonment by Intel encouraged us to implement a logging scheme that is optimized for flash SSD characteristics solely. LeanStore now also implements Early Lock Release (ELR) [De84; Jo10] which reduces transaction abort rates by releasing write locks without having to wait for flushing the log. Both ELR and distributed logging amplify the cost of cross-worker logging dependencies. Therefore, we introduce a new fine-granular dependency tracking mechanism to reduce the false positive dependencies. In the following, we assume that transactions run in Read Committed or Snapshot Isolation level.

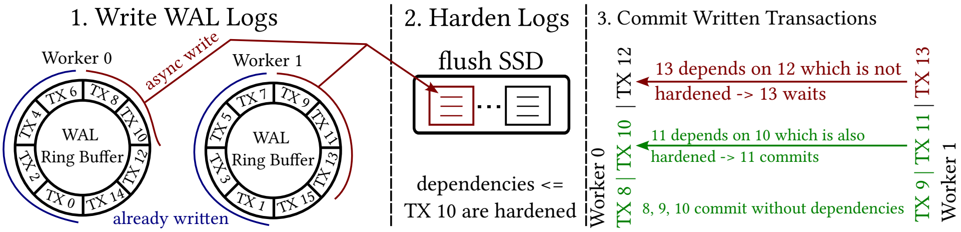

Fig. 5: Group Commit: Each round collects and writes WAL records on SSD before calling fsync(). Consequently, it signals all safe-to-commit transactions if their dependencies are also hardened

7.1 Group Commit

Despite being a widely implemented technique in database systems, few research papers describe SSD-optimized group commit. In Fig. 5, we illustrate our group commit implementation. Group commit works in rounds of 3 steps each. Most of the work is done by a background Group Committer Thread (GCT). Worker threads only push the *pre-committed* transactions, which have already passed the concurrency control validation phase, into a ready-to-commit queue and the GCT takes on from this point. The first step in a group commit round is to write WAL records from every worker thread on SSD. On Linux, we use the asynchronous IO interface from libaio to batch all log writes and submit them using a single system call. Once the writes are done, we flush the block device with *fsync* to make sure that the log records we have just written are durable. Consequently, we calculate the new safe set of transactions that are hardened and ready to signal their commit to the client. With this information in hand, we can commit the *pre-committed* transactions in each worker that have their own log and their dependencies hardened.

7.2 Dependency Tracking

With distributed logging [Ha20] and Early Lock Release (ELR), transaction commit dependencies to other transactions must be determined. With ELR, a transaction T1 releases the write locks in its pre-commit phase before having its WAL records hardened on SSD. This means that a transaction T2 that reads from T1 cannot commit before T1. This dependency must be determined and hardened with the WAL commit log to preserve this dependency at recovery. Dependencies across transactions processed by the same worker thread have no impact on performance as they are processed serially. Cross-worker thread dependencies, on the other hand, impact latencies negatively as the system has to wait until all of the transaction's dependencies are hardened. Coarse-grained dependency tracking using the Global Sequence Number (GSN) on each page leads to many false positive dependencies.

[5] The I/O hash table tracks in-flight I/O operations and is required to avoid race conditions which could otherwise occur when the same uncached page is accessed multiple times [Le18].

For example, if T1 and T2 concurrently update different tuples on the same page, GSN synchronization will lead to a false dependency between the two transactions.

System vs. User Transactions Before we define the dependency rule, let us mention the two types of transactions in our storage engine and how they differ from each other. *User TXs* execute the commands of the storage engine client. They have a start and commit timestamp that are drawn from the same global atomic counter. Their side effects are only visible after commit and only to user transactions that started from the commit. *System TXs* are triggered by the storage engine itself [Gr11]. For example, they are used to split or merge B-tree pages, manipulate the free space inventory and their side effects are immediately visible to all other transactions. They have do not change the data written by the user but only impact the physical representation. A system transaction exclusively locks all the pages it wants to modify before applying the changes. The start and commit timestamp of a system TX are thus always equal and drawn from another global atomic counter separate to the user transactions.

Commit Condition. Because read committed is the lowest sensible isolation level, user transaction dependencies can be tracked at transaction granularity instead of log sequence numbers like LSNs or GSNs. A user transaction depends on previous user and system transactions. For system transactions, it is enough to track the maximum system transaction ID it has witnessed while reading pages. Each page stores the maximum system transaction that had written on it. For the user one, we can either use the user TX start timestamp in case of snapshot isolation level or the start timestamp for the most-recent (pre-)committed user transaction it has read from. The Group Commit Thread calculates the two watermarks: user_tx_hardened_up_to and system_tx_hardened_up_to of all workers during each round and commits the pre-committed transactions that are below them.

8 Evaluation

In this section, we first experimentally compare LeanStore with WiredTiger and RocksDB using TPC-C and YCSB. Both benchmarks do not involve complex analytical operations like join or aggregation and are fully supported in all of the competing transactional key-value storage engines. We then show the effectiveness of the B-tree optimizations described in Sect. 4 and of the commit dependency tracking discussed in Sect. 7. All benchmarks are implemented as a C++ client linked with the storage engine library. Unless explicitly stated, all engines have been configured to run under snapshot isolation. WiredTiger supports the lower isolation mode read uncommitted (RU) only for read-only transactions which we denote as WiredTiger (RU) and evaluate to measure the impact of snapshot construction. All experiments were performed on a single-socket server with an AMD EPYC 7713 64-Core CPU (128 hardware threads) with 512 GB of DRAM running Linux. For storage, we use a RAID-0 of ten 3.8 TB Samsung PM1733 SSDs using XFS as file system.

8.1 In-Memory Scalability

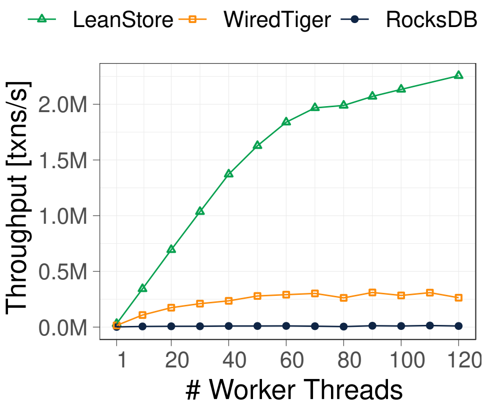

Fig. 6: TPC-C in-memory scalability

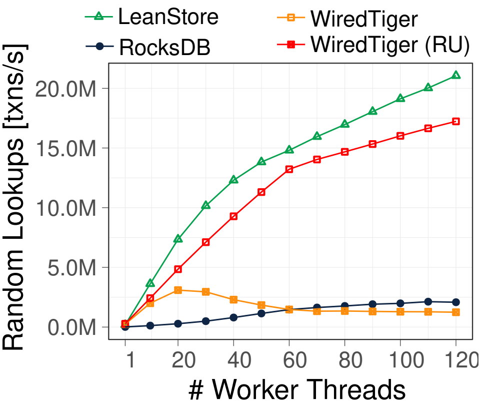

Fig. 7: YCSB read-only in-memory scalability

Let us first investigate in-memory scalability. We compare the performance of LeanStore with its competitors on TPC-C (Fig. 6) (with warehouse affinity) and a YCSB-like random lookup workload (Fig. 7). In both experiments, the buffer pool is configured to be large enough to fit the data sets.

On TPC-C, we see that LeanStore performs much better than the other two systems at higher thread counts. Note that, as expected, when more than 64 worker threads are used, the scalability is slightly worse due to hyperthreading. The absolute performance of over 2 million TPC-C transactions per second shows that LeanStore can compete with the fastest in-memory systems (despite also supporting out-of-memory workloads).

On the read-only random lookup workload, only WiredTiger under the lower isolation mode read uncommitted (RU) comes close to LeanStore snapshot isolation scalability. Despite running on higher isolation level, LeanStore remains around 30% faster. In the absence of concurrency control (RU), hazard pointers and free-lock techniques employed in WiredTiger make the index lookup scales similar to LeanStore's pointer swizzling and optimistic lock coupling. This result highlights the importance of scalability primitives on modern hardware. In WiredTiger, the cost of constructing a snapshot is proportional to the number of active threads in the systems. At higher threads count, snapshot construction costs more the actual lookup which leads to the performance deterioration beyond 20 threads. With our novel's MVCC implementation [AL23], LeanStore scales linearly without hard coding any optimization for single-statement or read-only transactions.

8.2 Out-Of-Memory Scalability

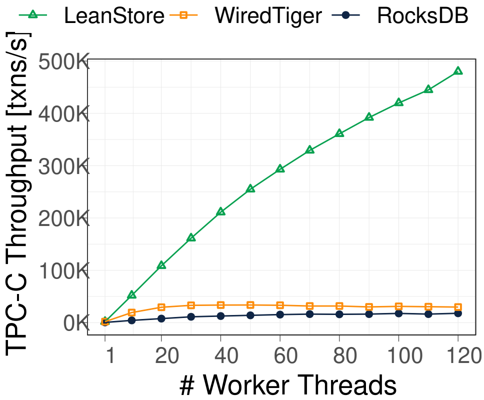

Fig. 8: TPC-C out-of-memory scalability

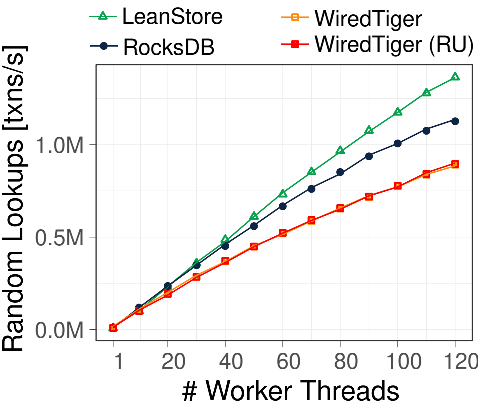

Fig. 9: YCSB read-only out-of-memory scalability

For the next two experiments we configured the buffer pool to be smaller than the data set. For TPC-C we use a 10 GB buffer pool for a 24 GB data set, and for YCSB we use a 10 GB buffer pool for a 100 GB data set. Fig. 8 shows that only LeanStore scales well, reaching 500 thousand TPC-C transactions per second in an out-of-memory setting. Interestingly, on the read-only YCSB benchmark shown in Fig. 9, the gap to the other systems is smaller. The impact of snapshot construction in WiredTiger diminishes because of the dominance of I/O cost. Apparently, these systems can handle read-only workloads reasonably well, but not write-intensive workloads like TPC-C. The observed performance with 120 threads of over 1 million random reads per seconds can be explained by being entirely I/O latency bound: the latency of a single read I/O is about 100 microseconds [HHL20], which corresponds to 10K per second per thread. Our hardware setup could theoretically support an order of magnitude higher I/O rates, but this would require more concurrent operations.

8.3 B-tree Optimizations

Tab. 2: Performance impact of B-Tree optimizations on single-threaded lookup performance in a B-tree of 10 million 8-byte integers or 6.4 million strings with an average length of 63 bytes

	Integers			Strings		
	operations/s	instr./op.	L1-miss/op.	operations/s	intr./op.	L1-miss/op.
baseline	1,094,092	1,255	71	914,693	1,266	99
+ prefix	1,127,396	1,261	72	1,013,265	1,252	83
+ heads	1,811,594	590	36	1,296,897	1,067	58
+ hints	2,427,184	567	17	1,383,918	1,137	45

Let us next evaluate the impact of the B-tree performance optimizations. We look at the single-threaded in-memory lookup performance. As keys, we use 10 million dense random 8-byte integers (plus 8-byte payload), and 6.4 million real-world URLs with an average length of 63 bytes. Tab. 2 shows the performance, CPU instructions and L1 cache misses for both workloads. The baseline is the basic slotted page layout without any optimizations. We then cumulatively enable the prefix, heads, and hints optimizations. As the table shows, each optimization improved performance significantly for both workloads. What is interesting is that in the baseline case, the integer workload is only 20% faster than the string workload, while with all optimizations the gap increases to 76%. This is because for string data set and its long keys (63 bytes), many of the cache misses are hard to avoid. In the integer case, on the other hand, the optimizations are extremely effective, reducing the number of L1 misses by from 71 to 17 and improving overall performance by 2.2×.

8.4 Commit Dependency Tracking

Tab. 3: Percentage of cross thread transaction dependencies (lower is better)

Tracking Granularity	Warehouse Affinity	Cross Warehouse
Page Wise	52.0%	95.5%
Tuple Wise	3.9%	91.9%

In our final experiment, we demonstrate the effectiveness of Commit Dependency Tracking by measuring the ratio of remote flushes necessary. We use TPC-C benchmark with 120 worker threads and 120 warehouses, which corresponds to about 15GB of data. In [Ha20] we proposed a lightweight page wise tracking scheme, which we now compare with tuple-wise tracking described in Sect. 7. Tab. 3 shows that tuple-wise tracking reduces the number of log flushes in both TPC-C settings. In the "Cross Warehouse" setting, many conflicts are unavoidable, but even here the tuple-wise scheme is more precise. With "Warehouse Affinity", there are few logical conflicts, and tuple-wise tracking reduces the flush rate from 52% to 4%.

9 Summary

The goal of LeanStore is to build a high-performance storage engine optimized for multi-core CPUs and NVMe SSDs. In this paper, we describe some important LeanStore components for the first time. Seemingly intricate implementation details of the B-tree, synchronization, buffer management, and logging are crucial for overall performance, scalability, and code maintainability. While the goal of LeanStore has stayed the same since the start of the project, many internals have changed and we expect this evolution to continue. For example, we have recently designed an OS-assisted buffer manager [Le23], which we are now considering for LeanStore.

Acknowledgments. This work was funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) – 447457559.

References

[AL21] Alhomssi, A.; Leis, V.: Contention and Space Management in B-Trees. In: CIDR. 2021.

[AL23] Alhomssi, A.; Leis, V.: Scalable and Robust Snapshot Isolation for High-Performance Storage Engines. In: Under Submission. 2023.

[An19] Antonopoulos, P.; Budovski, A.; Diaconu, C.; Saenz, A. H.; Hu, J.; Kodavalla, H.; Kossmann, D.; Lingam, S.; Minhas, U. F.; Prakash, N.; Purohit, V.; Qu, H.; Ravella, C. S.; Reisteter, K.; Shrotri, S.; Tang, D.; Wakade, V.: Socrates: The New SQL Server in the Cloud. In: SIGMOD. 2019.

[AW22a] AWS: Amazon EC2 I3en Instances, `https://aws.amazon.com/ec2/instance-types/i3en/`, 2022.

[AW22b] AWS: Lsv2-series, `https://learn.microsoft.com/en-us/azure/virtual-machines/lsv2-series`, 2022.

[Bi22] Binna, R.; Zangerle, E.; Pichl, M.; Specht, G.; Leis, V.: Height Optimized Tries. ACM Trans. Database Syst. 47/1, 3:1–3:46, 2022.

[Bö20] Böttcher, J.; Leis, V.; Giceva, J.; Neumann, T.; Kemper, A.: Scalable and robust latches for database systems. In: DaMoN. 2020.

[BU77] Bayer, R.; Unterauer, K.: Prefix B-Trees. ACM Trans. Database Syst. 2/1, pp. 11–26, 1977.

[De84] DeWitt, D. J.; Katz, R. H.; Olken, F.; Shapiro, L. D.; Stonebraker, M.; Wood, D. A.: Implementation Techniques for Main Memory Database Systems. In: SIGMOD. 1984.

[Di13] Diaconu, C.; Freedman, C.; Ismert, E.; Larson, P.; Mittal, P.; Stonecipher, R.; Verma, N.; Zwilling, M.: Hekaton: SQL server's memory-optimized OLTP engine. In: SIGMOD. Pp. 1243–1254, 2013.

[Do21] Dong, S.; Kryczka, A.; Jin, Y.; Stumm, M.: RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. ACM Trans. Storage/, 2021.

[Fä11] Färber, F.; Cha, S. K.; Primsch, J.; Bornhövd, C.; Sigg, S.; Lehner, W.: SAP HANA database: data management for modern business applications. SIGMOD Rec. 40/4, pp. 45–51, 2011.

[GL01] Graefe, G.; Larson, P.: B-Tree Indexes and CPU Caches. In: ICDE. 2001.

[Gr04] Graefe, G.: Write-Optimized B-Trees. In: VLDB. Pp. 672–683, 2004.

[Gr11] Graefe, G.: Modern B-Tree Techniques. Found. Trends Databases 3/4, pp. 203–402, 2011.

[Gr14] Graefe, G.; Volos, H.; Kimura, H.; Kuno, H. A.; Tucek, J.; Lillibridge, M.; Veitch, A. C.: In-Memory Performance for Big Data. PVLDB 8/1, pp. 37–48, 2014.

[Ha20] Haubenschild, M.; Sauer, C.; Neumann, T.; Leis, V.: Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In: SIGMOD. Pp. 877–892, 2020.

[HHL20] Haas, G.; Haubenschild, M.; Leis, V.: Exploiting Directly-Attached NVMe Arrays in DBMS. In: CIDR. 2020.

[Jo10] Johnson, R.; Pandis, I.; Stoica, R.; Athanassoulis, M.; Ailamaki, A.: Aether: A Scalable Approach to Logging. PVLDB 3/1, pp. 681–692, 2010.

[Ke12] Kemper, A.; Neumann, T.; Funke, F.; Leis, V.; Mühe, H.: HyPer: Adapting Columnar Main-Memory Data Management for Transactional AND Query Processing. IEEE Data Eng. Bull. 35/1, pp. 46–51, 2012.

[Le16] Leis, V.; Scheibner, F.; Kemper, A.; Neumann, T.: The ART of practical synchronization. In: DaMoN. 2016.

[Le18] Leis, V.; Haubenschild, M.; Kemper, A.; Neumann, T.: LeanStore: In-Memory Data Management beyond Main Memory. In: ICDE. Pp. 185–196, 2018.

[Le23] Leis, V.; Alhomssi, A.; Ziegler, T.; Loeck, Y.; Dietrich, C.: Virtual-Memory Assisted Buffer Management. In: SIGMOD. 2023.

[LHN19] Leis, V.; Haubenschild, M.; Neumann, T.: Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. IEEE Data Eng. Bull. 42/1, pp. 73–84, 2019.

[LKN13] Leis, V.; Kemper, A.; Neumann, T.: The adaptive radix tree: ARTful indexing for main-memory databases. In: ICDE. Pp. 38–49, 2013.

[Lo01] Lomet, D. B.: The Evolution of Effective B-tree: Page Organization and Techniques: A Personal Account. SIGMOD Rec. 30/3, pp. 64–69, 2001.

[Mo22] MongoDB: WiredTiger Storage Engine, `https://docs.mongodb.com/manual/core/wiredtiger/`, 2022.

[NF20] Neumann, T.; Freitag, M. J.: Umbra: A Disk-Based System with In-Memory Performance. In: CIDR. 2020.

[Sa22a] Samsung: PCIe Gen 4-enabled PM1733 SSD, `https://semiconductor.samsung.com/ssd/enterprise-ssd/pm1733-pm1735/mzwlj3t8hbls-00007/`, 2022.

[Sa22b] Samsung: PM1743, `https://semiconductor.samsung.com/ssd/enterprise-ssd/pm1743/`, 2022.

[SW13] Stonebraker, M.; Weisberg, A.: The VoltDB Main Memory DBMS. IEEE Data Eng. Bull. 36/2, pp. 21–27, 2013.

[Tu13] Tu, S.; Zheng, W.; Kohler, E.; Liskov, B.; Madden, S.: Speedy transactions in multicore in-memory databases. In: SIGOPS. Pp. 18–32, 2013.

[Ve17] Verbitski, A.; Gupta, A.; Saha, D.; Brahmadesam, M.; Gupta, K.; Mittal, R.; Krishnamurthy, S.; Maurice, S.; Kharatishvili, T.; Bao, X.: Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In: SIGMOD. 2017.

[Wa18] Wang, Z.; Pavlo, A.; Lim, H.; Leis, V.; Zhang, H.; Kaminsky, M.; Andersen, D. G.: Building a Bw-Tree Takes More Than Just Buzz Words. In: SIGMOD Conference. ACM, pp. 473–488, 2018.