# LeanStore: A High-Performance Storage Engine for NVMe SSDs

Viktor Leis
Technische Universität München
Germany
leis@in.tum.de

## ABSTRACT

Neither traditional disk-based database systems nor modern in-memory database systems are capable of fully exploiting modern servers with multiple NVMe SSDs. LeanStore is a high-performance OLTP storage engine specifically optimized for NVMe SSDs and multi-core CPUs. The paper gives an overview of the architecture of LeanStore and describes all major components, covering caching, page replacement, I/O management, indexing, data structure synchronization, multi-version concurrency control, logging, checkpoints, and recovery. We also discuss some of the low-level implementation techniques necessary for achieving high performance on modern hardware.

**PVLDB Reference Format:**
Viktor Leis. LeanStore: A High-Performance Storage Engine for NVMe SSDs. PVLDB, 17(12): 4536-4545, 2024.
doi:10.14778/3685800.3685915

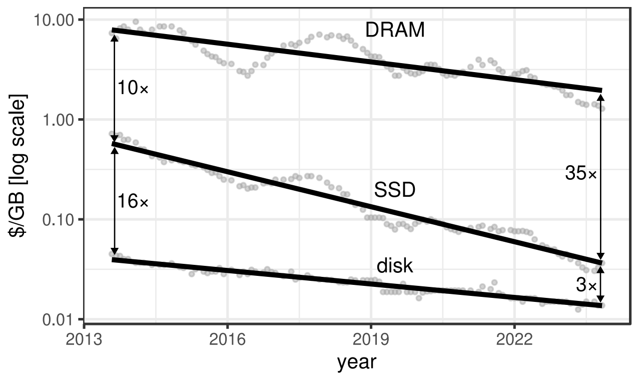

**Figure 1: DRAM, SSD, and disk prices from 2013 to 2023.**
data source: https://jcmit.net/memoryprice.htm

## 1 INTRODUCTION

**DRAM Stagnation.** Following Stonebraker's call to action [26, 64], main-memory database systems have been the focal point of research on high-performance database systems for more than a decade. Academically, this research program has been a tremendous success, introducing innovative systems and achieving unprecedented performance results. Surprisingly, however, the real-world adoption of pure in-memory transactional database systems has been limited. We believe it is fair to say that even the most successful in-memory systems remain niche products. The focus on main-memory systems was fueled by rapidly shrinking DRAM prices: from 2000 to 2012, the price/byte for DRAM dropped by about 300× [23]. For the first time, this made it feasible to keep databases of non-trivial size entirely in main memory. Since around 2012, however, DRAM prices have been decreasing at a much slower pace, which has limited the adoption of main-memory database systems. Consequently, general-purpose transaction processing is still dominated by traditional database systems optimized for secondary storage.

**Flash to the Rescue.** In contrast to DRAM, flash-based solid state drives (SSDs) have seen dramatic price reductions during the last 10 years. Whereas DRAM and flash cost per byte was comparable in 2005, today flash is about 20–50× cheaper[1] than DRAM [23]. As Figure 1 shows, SSDs are even closing the cost gap to disk. SSDs have not just become cheap, they have also become very fast. Internally, an SSD consists of many flash chips that can be accessed in parallel, which is why SSD bandwidth has historically been limited by the interface to the host system. The transition from SATA 3 (0.5 GB/s) to the NVMe/PCIe interface changed this by allowing users to exploit the quickly improving interconnect speeds of PCIe 3.0 (4 GB/s), PCIe 4.0 (8 GB/s), and PCIe 5.0 (16 GB/s). Thus, from 2017 to 2023, SSD bandwidth increased by 30 times – while price per byte dropped by one order of magnitude. With support for 10 or more SSDs in a single server, this means that a server with tens of millions of I/O operations per second and an aggregated bandwidth rivaling DRAM is not just possible – but affordable and readily available [24].

**SSD-Optimized Systems.** The LeanStore [45] project was motivated by the observation that neither in-memory nor disk-based systems come even close to being able to exploit the capabilities of modern SSDs. Many of the design decisions of in-memory systems such as the avoidance of a buffer pool and small index nodes are problematic for storage on flash, and augmenting in-memory architectures with out-of-memory support may result in overly complicated systems. Existing disk-based systems, on the other hand, appear more promising due to buffer pools and page-based storage, but were developed when storage was several orders of magnitude slower. As a result, with NVMe SSDs, a system like PostgreSQL is completely CPU-bound on out-of-memory OLTP workloads [23]. Even though they are often treated that way, SSDs

Proceedings of the VLDB Endowment, Vol. 17, No. 12 ISSN 2150-8097.
doi:10.14778/3685800.3685915

[1]In contrast, the price of Intel's persistent memory offering Optane remained close to DRAM – which is probably the main reason why the technology failed to achieve commercial success and was discontinued.

are not just faster disks – good performance requires novel DBMS designs.

**LeanStore.** The goal of LeanStore [45] is to build a storage engine that rivals the performance of in-memory systems without having to keep all data in main memory. To achieve this requires combining many of the modern in-memory optimizations (e.g., lightweight synchronization, CPU and cache efficiency) with techniques from disk-based systems (e.g., B-trees, paged storage, physiological logging, fuzzy checkpoints). The most important technical challenge is managing hardware parallelism at both the CPU and SSD level.

**Paper Overview.** The LeanStore project started in 2017 and many of the important components have been described in a series of papers [2–4, 24, 27, 28, 43, 45, 46, 58, 67]. These papers describe a particular component in isolation and are snapshots of our understanding at particular points in time. Since the start of the project, its goals stayed the same, but we revisited almost all ideas and techniques, leading to significant changes in the design. These changes have been motivated not just by trying to achieve ever higher performance, but also by the desire for simplicity, more performance robustness, and fewer configuration parameters. This paper discusses the key ideas behind all major LeanStore components, provides the underlying motivation behind our design decisions, and tries to distill what we learned over the history of the project.

## 2 LEANSTORE

**Functionality and Scope.** LeanStore is a storage engine optimized for NVMe SSDs and multi-core CPUs. Similar to other storage engines such as RocksDB [18], WiredTiger [55], Shore-MT [31], FASTER [13], and LMDB [65], it provides indexing and transactions, but no support for SQL or high-level query processing capabilities. LeanStore is an embeddable C++ library [2] offering APIs for index access (insert, update, delete, point lookup, range scan) and transaction management (begin, commit, rollback). Keys and payloads are opaque arrays of bytes, so it is up to the application to interpret the data appropriately [2].

### 2.1 Caching

**Hash-Table-Based Buffer Management.** Disk-based database systems cache pages from secondary storage in a buffer pool, which is usually implemented using a hash table that maps page identifiers to cached pages. For workloads where most page accesses are hits, this implementation can incur a significant performance overhead [26]. This has been one of the main motivations for main-memory database systems, most of which avoid the overhead by not implementing a buffer manager at all. Supporting larger-than-memory workloads without a buffer manager requires additional, fairly complicated, mechanisms [15, 17] for distinguishing hot and cold tuples. One major downside of these mechanisms is that they only deal with tuples, but cannot evict index structures. The conceptual beauty of a buffer manager is that it can transparently manage arbitrary data structures using a single replacement algorithm. Given the increasing performance and decreasing cost of NVMe storage, we argue that any modern database system requires a buffer manager.

**Pointer Swizzling.** The original impetus for LeanStore came from the insight that a buffer manager with nearly zero overhead can

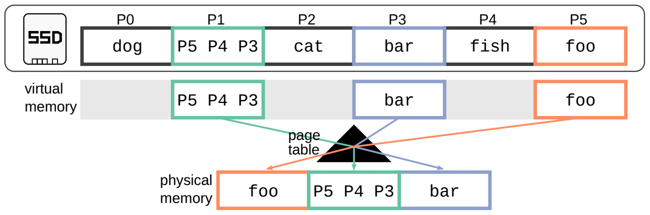

**Figure 2: Virtual-memory assisted buffer management. Cached pages are mapped into virtual memory at locations corresponding to offsets on SSD. In contrast to a file-backed memory mapping approach, the DBMS controls eviction and replacement.**

be implemented using pointer swizzling. Instead of translating the page identifier to a pointer on every page access, the main idea behind pointer swizzling is to change the reference to the pointer. In other words, a page reference can either be a pointer into the buffer pool or an offset on secondary storage. Pointer swizzling is an old idea that was heavily used in object-oriented database systems [35] and was later adapted to relational storage engines with page-based storage [22, 39, 56].

**Pointer Swizzling Downsides.** Although pointer swizzling is very fast, it also has significant downsides. First, it is an invasive technique that is not as transparent as a hash-table-based buffer manager. For example, before evicting a page, it is necessary to ensure that all references stored on that page are storage offsets rather than pointers [22]. To some extent, this requires the buffer manager to understand the content of buffer-managed pages. Second, pointer swizzling makes it difficult to support graph (non-tree) data structures. The reason is that before a page can be evicted, all incoming references to that page need to be unswizzled. Practical implementations therefore restrict themselves to tree data structures. Third, our experience shows that implementing pointer swizzling is subtle and challenging, particularly in terms of correct synchronization.

**Can We Make Translation Efficient?** Even though pointer swizzling was *the* foundational idea of the LeanStore project, its downsides led us to rethink our buffer manager design. Fundamentally, the problems of pointer swizzling stem from getting rid of the indirection between logical page identifier and pointer to cached memory. In other words, the traditional hash table indirection makes buffer managers flexible and simple, but comes at the cost of inefficient page access. One may wonder whether, instead of getting rid of the indirection table, there is a way to make translation efficient. Indeed, every modern CPU has hardware support (page walking in hardware, TLB) for translating virtual to physical memory addresses.

**Virtual-Memory-Assisted Buffer Management.** Why can the DBMS not just use the virtual memory hardware instead to make buffer managers both flexible and efficient? This question led us to the development of *vmcache*, a virtual-memory assisted buffer manager [43]. As Figure 2 illustrates, vmcache maps cached pages into virtual memory such that the virtual memory addresses correspond to the locations on disk. The virtual memory page table serves a similar indirection purpose as the hash table of traditional buffer managers. The result shown in Figure 2 could be achieved by mapping the storage device into virtual memory using the `mmap` system call in Unix. However, this would mean the operating system (OS) controls paging and eviction, causing major semantical and performance issues for database systems [14].

**The DBMS Is In Control.** Instead, in the vmcache design, the DBMS takes control using three widely-supported OS primitives:

- Lazy allocation of virtual memory:
  `mmap(NULL, ssdSize, ..., MAP_ANONYMOUS ...)`
- Page miss:
  `pread(fd, &virtMem[pid], pageSize, pid*pageSize)`
- Eviction:
  `madvise(&virtMem[pid], pageSize, MADV_DONTNEED)`

On startup, vmcache allocates a large amount of anonymous virtual memory, with its size corresponding to the total *storage* size (`ssdSize` in the code fragment). When a cache miss occurs for a page with the page identifier `pid`, the I/O system call (`pread` in the code fragment) uses the corresponding virtual memory address as the destination for the page. Before the physical memory capacity is exhausted, vmcache explicitly selects pages for eviction and then communicates this decision to the operating system (`madvise(...,MADV_DONTNEED)` in code fragment).

**Discussion.** The basic vmcache design offers fast (TLB-assisted) cache hits, works across all major operating systems, is fairly easy to implement, supports arbitrary graphs, and simplifies variable-size pages[2]. However, it is not without downsides. vmcache relies heavily on manipulating virtual memory through system calls: in steady state, every page miss will lead to one page fault and one page eviction. It turns out that today's storage devices are faster than today's OS virtual memory operations [43]. Unfortunately, fully exploiting fast storage devices therefore requires novel fast and scalable virtual memory primitives that can be implemented as a Linux kernel module [43] or within specialized unikernels [44]. Nevertheless, we believe that the qualitative benefits of virtual-memory assisted buffer management outweigh this downside, which is why we are in the process of transitioning LeanStore's buffer manager from pointer swizzling to the vmcache design.

## 2.2 Page Replacement

**Efficient Page Replacement.** Buffer managers need some algorithm for deciding which page to evict. Many disk-based systems rely on approximations of Least Recently Used (LRU) like Second Chance, often in conjunction with special handling for large table scans. The original LeanStore replacement algorithm [45] combines random candidate selection with a FIFO-based algorithm. Random pages are selected as candidates for eviction by unswizzling them and moving them to a FIFO list, which comprises a fixed percentage of the overall buffer pool (e.g., 10%). This gives candidate pages a grace period during which they can be re-promoted back to a swizzled page without incurring I/O. The motivation behind this two-stage algorithm is that accessing hot (swizzled) pages incurs absolutely no overhead, which is particularly beneficial for workloads where the working set fits into the buffer pool fully.

[2]Larger database pages simply consist of multiple contiguous virtual memory pages (usually 4KB). The page table prevents internal memory fragmentation by allowing the use of arbitrary (non-contiguous) physical pages.

**Effective Page Replacement.** Workloads with larger data sizes and a non-trivial miss rate, on the other hand, benefit from algorithms that invest more effort into selecting which page to evict. It is well known that more sophisticated algorithms such as ARC [52] and LRU-k [59] result in better hit rates for skewed, real-world workloads. However, these high-quality algorithms have been designed in a world where hardware looked very different than today, and would be too slow for modern hardware. Modern servers have in the order of one hundred CPU cores and support in the order of a million I/O operations per second. Flash storage is also asymmetric in the sense that writes are slower than reads. Therefore, modern flash storage engines require a replacement algorithm that is efficiently handling the high I/O rates, scales well with the number of CPU cores, and is aware that writes are more expensive than reads. This led us to develop the Write-Aware Timestamp Tracking (WATT) algorithm [67].

**Tracking Access History.** The main intuition behind WATT is that, to achieve a higher page hit rate, one needs to track more information about the page access history. For example, the Second Chance algorithm relies on a single bit per page, while LRU implicitly stores a rank for each page that is determined by the access history. To improve upon LRU in terms of hit rates, one needs more information about the page access history. One way to do that is to remember the entire access history of each page, which could be implemented by recording the timestamps of each page access. In practice, one has to limit the number of timestamps tracked to a small constant (e.g., 8).

**Page Value Calculation.** Given these timestamps, how does one select a page for eviction? WATT relies on the notion of a *page value*, and a page with a lower value is evicted more likely than a page with a higher value. When a page has been accessed at timestamps 8, 15, and 42, we get the following tracking history:

| $i$ | 1 | 2 | 3 |
|---|---|---|---|
| $t_i$ | 42 | 15 | 8 |

To compute the page value of a page at timestamp $t_{now}$, we first compute subfrequencies $SF_i := \frac{i}{t_{now}-t_i}$. For $t_{now} = 50$, we get the following results:

| $i$ | 1 | 2 | 3 |
|---|---|---|---|
| $SF_i$ | $1/(50-42) \approx 0.13$ | $2/(50-15) \approx 0.06$ | $3/(50-8) \approx 0.07$ |

Intuitively, the subfrequencies measure the average access frequency at different points in time. For example, for the $i = 2$ case, we know that we had 2 page accesses within a time frame of $50 - 15 = 35$ time units, resulting in an average access frequency of $\frac{2}{35} \approx 0.06$. To obtain the page value, WATT simply selects the largest subfrequency of that page: $PV := \max_i SF_i(t_{now})$. Given that the subfrequencies depend on the current time and therefore change constantly, it is not feasible to maintain accurate page values over the entire buffer pool. Instead, WATT relies on sampling: whenever free pages are needed, it computes page values for randomly-selected eviction candidates and then evicts, e.g., the 10% of pages with the lowest values.

**Write Awareness.** To make the algorithm write-aware, we track read and write accesses separately (using 8 read timestamps, and 4 write timestamps per page) and compute separate page values for reads and writes. These two page values are simply combined with a weighted sum. The weight parameter allows configuring the relative cost of reads and writes, e.g., based on the hardware specification of the storage device in use.

**Optimizations.** To make the timestamp tracking scalable on multi-core CPUs it is important to minimize cache line invalidations by avoiding to increment the timestamp counter too frequently. A good way to do this is to couple the number of buffer pool page allocations with the timestamp increments. In a buffer pool with 1,000,000 pages, for example, it would be enough to increment the timestamp counter every 100,000 page allocations to distinguish cold from hot pages. Another important practical consideration is to make eviction efficient using memory prefetching during the sampling phase and SIMD during page value calculation. This reduces the overhead of computing the value of a random page to approximately 100 cycles. Together, these optimizations result in an algorithm that not only achieves state-of-the-art replacement effectiveness [67], but is also efficient, scalable, and write-aware.

## 2.3 I/O Management

**Slow I/O, Fast I/O.** In the past, when disks achieved at most several thousands of I/O operations per second, what mattered for overall performance was the number of I/O operations. Consequently, disk-based database systems have been optimized for minimizing disk I/O operations. The recent performance explosion of NVMe SSDs has fundamentally changed these assumptions. The performance of modern PCIe 5.0 SSDs is approaching 3 million I/O operations per second (IOPS) for random 4 KB reads. Servers have enough PCIe lanes for ten or more such devices, which means that tens of millions of IOPS have become feasible.

**Tight CPU Budget.** In the original LeanStore paper, we argued that briefly acquiring a global lock before every I/O operation is not problematic [45]. This may have been the case in 2018. However, the assumption that any CPU work on the I/O path is negligible because I/O is slow anyway, is not true anymore. Consider a server with 100 cores at 3 GHz and 8 SSDs with 2.5 M IOPS each, which is similar to a server in our lab. Assuming very optimistically that all CPU cores of the server can be used and that there is no overhead from synchronization, this implies that for every I/O operation, we have a budget of at most 15,000 CPU cycles. For out-of-memory workloads, every transaction may very well cause a page miss and therefore an I/O operation. For any storage engine that wants to exploit the I/O capabilities, this means that it has a total of 15,000 cycles for the index lookup, concurrency control protocol, task management, page replacement, and for actually performing the I/O itself.

**Optimizing the I/O Stack.** Such a tight CPU budget implies that all internal overhead on the I/O path must be carefully engineered to be efficient and scale well. In LeanStore, we not only had to get rid of the global I/O lock, but rewrite the entire I/O path, for example, avoiding dynamic memory allocations for small objects used internally to keep track of outstanding I/O operations [24]. The I/O stack of the operating system is also a major source of overhead [24]. Another important key to unlocking the power of modern flash hardware is exploiting its tremendous internal parallelism. An SSD internally consists of many independent flash chips. Achieving high throughput on transactional out-of-memory workloads therefore requires scheduling and managing thousands of concurrent I/O operations [24]. This is challenging because the degree of parallelism that is beneficial for flash is higher than the number of hardware threads, requiring careful orchestration of asynchronous I/O interfaces and user-level tasks [24, 30]. Finally, achieving high I/O rates also requires disabling the OS page cache, the file system, and any software RAID [23, 24]. Even the remaining low-level block device layer of Linux can be a bottleneck, which can be avoided with user-space I/O NVMe stacks such as SPDK [24].

**Task Management.** In addition to the actual transaction work, a buffer managed system has additional tasks that have to be performed. For example, it is necessary to find eviction candidates, write back dirty pages, and perform polling for I/O operations. Earlier versions of LeanStore relied on separate background threads for these tasks [45]. Unfortunately, given the performance of modern I/O devices, a single thread for each of these tasks may not be sufficient. To avoid users having to configure the number of threads, we therefore switched [24] to an approach where we start exactly as many worker threads as there are hardware threads and where each worker thread is responsible for all tasks. We rely on a lightweight context switching mechanism, to switch between tasks at appropriate points, e.g., a page miss. Note that this effectively re-implements many of the I/O and scheduling responsibilities operating systems are supposed to handle, but is currently necessary to achieve the performance goals.

**What is the Best Page Size?** One thing disk and flash have in common is that they are both block devices with a relatively large access granularity. Typical database page sizes are 4 KB (DB2), 8 KB (PostgreSQL, SQL Server, Shore), 16 KB (MySQL), 32 KB (WiredTiger), and 64 KB (Umbra). In LeanStore, we originally used 16 KB [45], but later switched to 4 KB. 4 KB I/O operations not only have the lowest latency [24], but they also reduce I/O amplification for random workloads. Because enables higher I/O rates, a smaller page size also increases the performance pressure on many components of the DBMS. Nevertheless, we believe that the I/O amplification benefits are worth the engineering effort.

**Random vs. Sequential I/O.** For disk there is tremendous benefit from performing sequential rather than random I/O. Through a concept often called segments, disk-based systems therefore try to maintain spatial locality for related pages on disk, e.g., to allow table scans to result in sequential I/O. For flash, the benefit of sequential accesses is small, and placement decisions are under the control of the SSD rather than the software anyway. Flash-optimized storage engines can therefore avoid the complication of maintaining segments, significantly reducing complexity.

## 2.4 Indexing

**B-trees.** Disk-based database systems have historically relied on B-trees [20] as index structures, while in-memory systems use a variety of diverse data structures optimized for in-memory performance, such as tries [7, 47, 51]. While in-memory data structures are typically faster than B-trees, their smaller (and often variable) node sizes make them difficult to integrate into buffer-managed systems and potentially inefficient in out-of-memory workloads. For these reasons, LeanStore has always relied on B-trees. We support variable-size keys/payloads and employ a number of cache optimizations [2] to the traditional B-tree node layout to close much

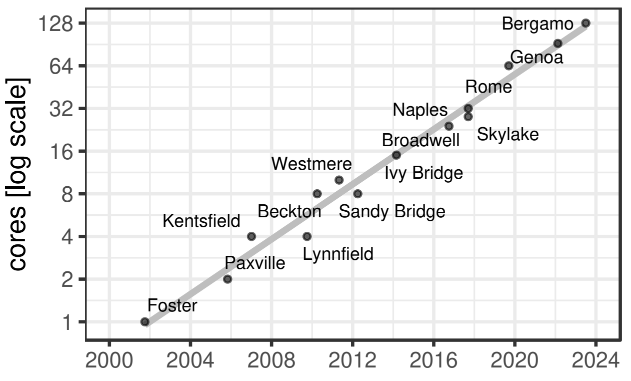

**Figure 3: Core count of the largest x86 server CPU over time.**

of the gap between high-performance in-memory data structures and B-trees.

**Contention Management in B-trees.** A downside of the relatively large node size of buffer-managed B-trees is that it may cause unnecessary lock contention when two or more frequently updated tuples happen to physically reside on the same page. To avoid unnecessary contention, we exploit a useful property that B-trees have: the same keys can be stored in B-trees with different node structures. Usually, this structure depends on the (accidental) insertion order, but nothing stops us from reorganizing it, e.g., to avoid contention. Specifically, the Contention Split [3] technique probabilistically detects unnecessary contention, and then forces a B-tree node split to spread the frequently accessed keys among separate nodes with separate locks.

**Space Management in B-trees.** Contention Split can result in underfull pages and therefore increase space consumption, which is why it should be combined with the XMerge technique [3]. XMerge optimizes memory consumption of B-trees by merging X neighboring nodes into X-1 nodes. A typical value for the X parameter is 5, with higher values resulting in more frequent merges and therefore higher average fill factors [3]. In LeanStore, XMerge is triggered during page replacement, i.e., when the system is about to run out of free pages. After all, it is just as good to free a page through merging as it is to evict a page. While XMerge is a useful technique in isolation, it also nicely complements Contention Split by preventing low space utilization, which could occur if the area of contention is changing continuously.

**Large Objects.** Most database systems support storing arbitrary data using the Binary Large OBject (BLOB) data type. Nevertheless, application developers generally prefer to store large objects in the file system. The split between database records describing the files and the file content causes major downsides, for example in terms of space management and the lack of transaction support. In recent work [58], we therefore proposed a number of optimizations that allow to manage large objects within the DBMS efficiently.

## 2.5 Low-Level Data Structure Synchronization

**Many Cores.** As Figure 3 shows, in the last two decades we went from having x86 server CPUs with a single core to over 100 cores – a growth rate of approximately 25% per year. High-end multi-socket (NUMA) systems and simultaneous multithreading ("hyper-threading" in Intel parlance) can bring the number of hardware threads to over 1,000. Synchronization is particularly challenging for transactional engines since transactions are often small (e.g., one single-tuple index lookup) and the synchronization overhead cannot be easily amortized. Exploiting the computational power of modern multi-core CPUs therefore requires efficient and scalable synchronization techniques for all internal data structures (such as index structures, work queues, database metadata).

**Locking.** The traditional way of synchronizing the internal data structures of a database system is to use fine-grained locking[3]. For example, each page in the buffer pool has a corresponding lock that can be acquired either in exclusive (for writes) or shared (for reads) mode on every page access. Unfortunately, locks do not scale well on modern multi-core CPUs not just due to their inherent pessimism, but also because each lock acquisition causes a physical write to memory – even for a shared lock acquisition. For example, consider the root node of a frequently accessed B-tree, which would benefit from being cached on all CPU cores. In a multi-threaded setting, each time the root is read, its lock needs to be acquired by physically writing to the lock, which causes the invalidation of the underlying cache line on all other cores.

**Partitioning.** Several synchronization approaches have been proposed to avoid the scalability problems of locking. Some in-memory systems, such as VoltDB and early versions of Hyper, physically partition the database according to a user-specified key. The data structures within a partition are completely independent, and different worker threads can be assigned to different partitions. As long as a transaction stays within one partition, this approach elegantly sidesteps low-level synchronization problems altogether. The downside of partitioning is that it makes cross-partition operations expensive. For example, suppose the customer table is partitioned by the customer identifier, and we have a secondary index on the customer name. A single logical lookup by name would have to perform lookups within each of the partitions.

**Lock-Free Data Structures.** Given that locks often fail to scale, it is natural to avoid locks altogether by relying on lock-free data structures, such as the Bw-tree [49], a lock-free B-tree variant, or the split-ordered list [63], a lock-free hash table. While lock-free structures usually scale well, in particular for read-heavy workloads, they are not necessarily the fastest data structures. Fundamentally, the problem is that lock-free data structures are based on a very limited set of hardware primitives (atomic loads, atomic stores, and compare-and-swap on 8-byte words). To implement a complex data structure operation such as a B-tree split, this limitation makes it necessary to introduce additional indirections such as mapping tables and delta records. This can result in substantial additional CPU overhead [69]. The synchronization protocols of lock-free data structures are also extremely complicated and therefore bug-prone,

[3]Following standard Computer Science terminology, we use the term *lock* for a low-level primitive providing mutual exclusion that the traditional database literature calls *latch*.

which is why we chose not to rely on lock-free data structures in LeanStore.

**Optimistic, Versioned Locks.** A less radical alternative to lock-free synchronization is to associate each lock with a version counter that is incremented on every data structure update. This version counter can be used by read operations to proceed optimistically and validate that a read is correct without physically acquiring any locks. We call such a lock, including a version counter, *optimistic lock*. Optimistic locks allow implementing lock-based data structures that scale just as well as lock-free data structures, while being faster and less complex. The idea was first proposed for synchronizing B-trees [12] and later used to synchronize the trie/B-tree hybrid Masstree [51] that became the index structure for the in-memory Silo system [66]. Based on optimistic locking, both data structures implement custom data-structure-specific synchronization protocols.

**Optimistic Lock Coupling.** The first LeanStore prototype used a custom protocol [12] to synchronize its B-tree indexes using optimistic versions. However, we quickly switched to an even simpler protocol, which we call Optimistic Lock Coupling (OLC) [46, 48]. We believe that the idea was first used in the context of a more complicated protocol for binary search trees [11]. As the name implies, the key idea behind Optimistic Lock Coupling is to combine Bayer and Schkolnick's [6] classic lock coupling idea with optimistic locks [12] by interleaving version validations. This idea radically simplifies the design of scalable, efficient, and correct data structures. OLC is used within LeanStore for synchronizing the B-tree indexes. LeanStore provides three locking modes for accessing buffer pool pages: exclusive, shared, and optimistic [2]. This is implemented by combining version counters with OS locks [9]. These synchronization abstractions are built into the buffer manager, and make it easy to implement additional scalable data structures on top [2].

**Memory Reclamation.** Lock-free and optimistic synchronization protocols usually require additional mechanisms when freeing memory. The problem is that unless all operations lock, an operation deleting a node from a data structure cannot be sure when it is safe to reclaim that node's memory for other purposes (because an optimistic read might still be ongoing). Lock-free data structures therefore generally have to be combined with techniques such as hazard pointers [53] or an epoch-based approach [66] for delaying memory reclamation. The original version of LeanStore relied on epoch-based memory reclamation. Surprisingly, we later found out that for buffer-managed systems like LeanStore it is possible to forgo a delayed memory reclamation mechanism altogether [27] as long as two conditions are fulfilled: (1) the buffer manager never returns memory to the operating system, and (2) version counters always increase monotonically. Both conditions are easy to ensure in LeanStore, which is why we were able to remove the code implementing the epoch-based approach. This not only simplifies the implementation, it also makes the system more robust because it avoids the potential of misbehaving threads stopping system-wide memory reclamation [27].

## 2.6 Multi-Version Concurrency Control

**MVCC and Snapshot Isolation.** Besides low-level data synchronization, database systems also require a high-level concurrency control mechanism to isolate concurrent transactions logically from each other. Today, most widely-used systems rely on some variant of Multi-Version Concurrency Control (MVCC). In an MVCC system, when a tuple is changed, a new version of the tuple is created while the old version remains available. A key attraction of MVCC is that it allows a long-running transaction to operate on an older snapshot of the database without having to lock all tuples and therefore without directly interfering with current updates. MVCC therefore implicitly promises Hybrid Transaction/Analytical Processing (HTAP) in one system.

**MVCC Tradeoffs.** Broadly, there are two major types of MVCC designs: those for disk-based systems such as PostgreSQL and MySQL/InnoDB, and those for in-memory systems such as Microsoft Hekaton [40] and Hyper [57]. Disk-optimized protocols efficiently support transactions of arbitrary size (*steal*) at the cost of expensive and unscalable snapshot creation and visibility checks that do not scale well on many-core CPUs. Protocols optimized for in-memory systems are generally more efficient, but require revisiting the entire write set on commit [4], which can be prohibitively expensive for large, out-of-memory transactions.

**Fast Commit and Fast Visibility Through OSIC.** The Ordered Snapshot Instant Commit (OSIC) [4] is a commit protocol that tries to combine the best of both worlds. Like in-memory systems, OSIC enables instant and scalable snapshot creation and fast visibility checks, but avoids the need to revisit the write set on commit. The key idea is to exploit a *transitive commit invariant*, which states that if some transaction was committed at a particular timestamp on some worker, then all transactions on that worker with an earlier timestamp were also committed. Each worker maintains a *commit log*, implemented as a fixed-size array, that is used during visibility checks by other workers. The transitive commit invariant allows each worker to cache the commit log entries from other workers, which makes visibility checks incremental and efficient. Tuples are annotated with a version and the worker identifier that created that version. When a worker encounters a tuple written by another worker, it will first check its own worker's cache before consulting (and caching) the commit log entry of the other worker.

**LeanStore MVCC Overview.** LeanStore implements snapshot isolation using the OSIC commit protocol, which provides cheap snapshot creation that scales with the number of workers and enables transactions of arbitrary size. The implementation uses the first-writer-wins rule, i.e., a transaction attempting to update a tuple for which the most recent version is not visible is aborted [4]. This results in recoverable schedules under snapshot isolation without the need for a validation phase. Besides the commit protocol and visibility checking logic, the efficiency and robustness of an MVCC implementation depend on how versions are stored and indexed. LeanStore chains tuple versions in newest-to-oldest order, favoring OLTP over OLAP, and stores delta entries for updates rather than creating a full copy of an updated tuple.

**Better Performance Robustness for Hybrid Workloads Using Graveyard Index.** MVCC implicitly promises that long-running read transactions can coexist with update-heavy OLTP transactions. Surprisingly, we found that OLTP performance collapses on all tested MVCC systems in the presence of a single read-only snapshot [4]. The reason for this is fairly subtle: indexes accumulate logically-deleted versions of frequently-updated tuples that cannot be garbage collected due to the old snapshot. Note that even fully precise garbage collection does not avoid this problem. The only solution is to physically move these tuple versions out of the main index into a separate data structure, which we call the *graveyard index*. Latency-critical OLTP transactions thereby avoid the large performance drop, while long-running snapshots can still retrieve these old versions from the graveyard index.

**Adaptive Storage and Garbage Collection.** Another important aspect of an efficient MVCC implementation is the question of where versions are stored and how garbage is collected. For some workloads, it is better to store old versions in a separate data structure (off-row storage). Other workloads benefit from older versions staying physically next to the newest one (in-row storage). LeanStore implements both (off-row: Delta Index and in-row: FatTuple) and adaptively chooses between them depending on the per-tuple access pattern [4]. By default, old versions are stored in the Delta Index. The Delta Index makes cheap bulk garbage collection possible by effectively serving as a per-thread garbage collection todo list. The second format, FatTuple, is employed for frequently-updated tuples, which reduces the risk of precise garbage collection causing random I/O. We found that achieving performance robustness for challenging hybrid workloads requires several kinds of garbage collection (GC) [4]. In many situations, the Delta Index allows very cheap GC through high watermarks and cheap range deletes. Other cases require precise garbage collection, e.g., traversing long version chains and checking for each one if it may safely be garbage collected. Finally, LeanStore also performs GC when evicting pages containing FatTuple entries, as these are not tracked by the Delta Index.

**Discussion.** LeanStore's MVCC implementation is efficient, scalable on multi-core CPUs, supports transactions of arbitrary size, and provides robust performance for complex heterogeneous workloads. Arguably, the main downside of LeanStore's approach is its complexity. Most of this complexity (graveyard index, different tuple storage layouts, several kinds of GC) stems from the desire for performance robustness. The performance overhead of these additional techniques is moderate, and we argue that the robustness gains are certainly worth it. It is a compelling open question whether these robustness goals can be achieved with a simpler design.

## 2.7 Logging, Checkpoints, and Recovery

**Logging in In-Memory Systems.** The durability of committed transactions is typically guaranteed using some form of write-ahead logging. In-memory systems generally log only tuple changes but do not log the index data structures themselves [16, 50]. Instead, the recovery process rebuilds indexes from scratch using the recovered tuple data. This approach is simple and efficient as long as index structures fit into main memory, but becomes infeasible for larger-than-memory indexes. Unfortunately, indexes constitute a significant fraction of the overall space consumption [73] in OLTP use cases.

**Logging in Disk-Based Systems.** Most disk-based systems, in contrast, log the changes to all pages, including the indexes, and are therefore able to recover the entire database state. The classical solution for disk-based systems is ARIES-style logging [54], which supports incremental recovery and fuzzy checkpoints. ARIES relies on a single centralized Write Ahead Log (WAL) that orders all log records according to a single monotonically growing Log Sequence Number (LSN). On multi-core CPUs, the centralized WAL is a scalability bottleneck for write-intensive workloads. Thus, neither lightweight schemes nor ARIES satisfy all desired properties, which means that SSD storage engines require a different approach.

**Decentralized Logging.** LeanStore's logging approach [28], like ARIES, is based on physiological logging: log entries reference physical page identifiers and contain logical per-page redo and undo information. Unlike ARIES, LeanStore supports per-thread logs using distributed clocks (or Lamport timestamps) [28, 68]. The approach relies on the Global Sequence Number (GSN) concept, which is a decentralized way for maintaining a partial order between log records. Every transaction, page, and log entry have an associated GSN [28]. When two transactions access the same pages, an explicit order between them is established through the GSN mechanism, while transactions that access distinct sets of pages will remain unordered and therefore unsynchronized [28].

**Commit Acknowledgment.** One challenge with decentralized logging is knowing when it is safe to acknowledge that a committed transaction is durable. It is not enough to simply flush the log containing the log records of one particular transaction, as that transaction might depend on unflushed log records with lower GSNs in other logs. One solution is to rely on a group commit process that periodically flushes all logs, computes the minimum GSN across these logs, and then acknowledges transactions above that minimum GSN. While this approach can achieve very high throughput, it can also lead to a high per-transaction commit latency. To be able to flush logs independently and therefore optimize commit latency requires a tracking mechanism at either (1) page granularity, such as Remote Flush Avoidance [28], or (2) at transaction granularity [2]. We currently employ transaction-level tracking [2], which integrates with our MVCC algorithm and allows the use of the Early Lock Release optimization [32].

**Continuous Checkpointing and Parallel Bounded Recovery.** Our logging and recovery scheme supports fuzzy checkpoints and bounds the recovery volume in terms of how much data has to be recovered. Say, we want to bound the recovery log volume to 100 GB. To achieve this, we perform incremental checkpoints: whenever we accumulate a certain fraction (e.g., 1 GB) of log volume, we checkpoint 1% of the buffer pool in a round-robin fashion. This ensures that the entire buffer pool is periodically checkpointed in a continuous manner without causing significant I/O spikes, and that recovery log volume is bounded at the configured limit. The recovery process is similar to ARIES with analysis, redo, and undo phases, except that all phases can be parallelized [28]. Overall, this results in a robust and scalable scheme for logging, checkpointing, and recovery.

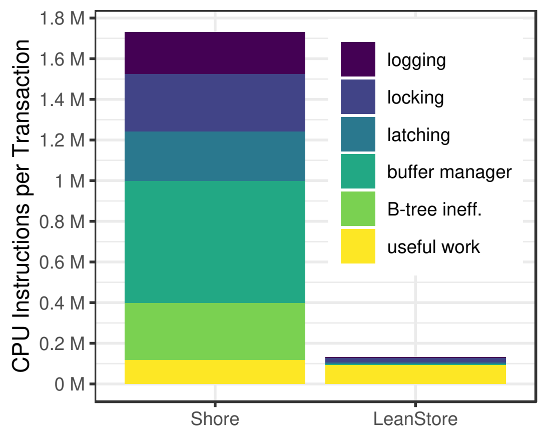

**Figure 4: Machine instructions (x86) per TPC-C Neworder transaction (1 thread, in-memory). Shore is a traditional disk-based storage engine.**

## 2.8 Related Work

LeanStore is one of the few transactional systems from academia specifically optimized for NVMe SSDs rather than persistent memory or DRAM. There is extensive work on optimizing disk-based systems for SSDs [5, 41, 60] and managing several types of storage technologies [25, 29, 75]. We were also somewhat influenced by recent work on logging [21, 33, 34, 37, 62], MVCC [19, 71], and garbage collection [10, 36, 38, 42, 61, 74].

## 3 DISCUSSION

**Architecture.** LeanStore shares many of the architectural features of traditional disk-based systems, such as page-based storage, buffer management, B-tree indexing, physiological redo/undo logging, fuzzy checkpointing, and a concurrency control implementation that supports arbitrarily-large transactions. A decade ago, this functionality was thought to come at a high cost in terms of in-memory CPU overhead. For example, Harizopoulos et al.'s [26] careful experimental study shows that in a traditional storage engine such as Shore, the internal overhead for logging, concurrency control, and buffer management dominates the overall CPU time. What LeanStore shows is that these components can be implemented efficiently such that useful work becomes dominant. CPU efficiency and scalability is not just important for in-memory workloads but also in out-of-memory situations [24].

**OLTP Trough The Looking Glass Revisited.** To compare Harizopoulos et al.'s [26] results with LeanStore, we followed the same methodology of removing as many components as possible while running the TPC-C *Neworder* transaction in-memory using a single thread [1]. Figure 4 shows the x86 instructions counts across these two systems[4]. We see that in LeanStore, *useful work*, rather than internal overhead, dominates the overall CPU time. As a result, LeanStore's in-memory performance is higher by one order of magnitude. Also note that LeanStore scales much better across CPU cores [4] and is capable of achieving similar performance in out-of-memory workloads [24]. These performance gains are achieved through careful optimization of all system components for highly parallel hardware and techniques such as optimistic data structure synchronization.

**LeanStore Evolution.** Over the course of the project, LeanStore was rewritten from scratch several times. The first prototype was developed by Michael Haubenschild and demonstrated that low-overhead caching is possible [45]. The second version, which is available as open source, was primarily implemented by Adnan Alhomssi and added support for multi-version concurrency control [4]. We are happy that this version is used by other research groups as a prototyping platform for novel ideas [72, 76]. A third implementation is under development and is based on virtual-memory-assisted buffer management [43]. Each implementation benefited from the experience of the previous one, and led to improvements and simplifications. While re-writing the system several times may not be the fastest way to a production-grade system, it allows for radical, experience-based simplifications that, in the long run, result in a much simpler design and implementation. Thus, we very much agree with Wirth's plea for lean software [70]:

> *"Reducing complexity and size must be the goal in every step – in system specification, design, and in detailed programming."* Niklaus Wirth

## 4 FUTURE WORK

The LeanStore project is ongoing and we are currently working on several topics. One is making commit processing on SSDs more efficient, in particular in terms of latency. Another topic, often ignored by the database community, is the frontend of the system that manages and schedules incoming network requests. We are also working on optimizing SSD writes, both at the data structure and I/O level, including the exploitation of novel SSD interfaces such as ZNS [8] and FDP. Finally, let us mention that many of the low-level I/O and scheduling optimizations we had to implement have historically been thought of as the job of the operating system rather than the DBMS. We are therefore also exploring opportunities for DB/OS co-design, in particular in the context of unikernels [44].

## ACKNOWLEDGMENTS

LeanStore is primarily the work of Michael Haubenschild, Adnan Alhomssi, Gabriel Haas, Demian Vöhringer, Lam-Duy Nguyen, Marcus Müller, and Bohyun Lee. I would also like to thank my mentors and collaborators Thomas Neumann, Alfons Kemper, and Peter Boncz for their longstanding and ongoing support. Michael Stonebraker has been an inspiration and role model for how to do ambitious and impactful systems research in academia. LeanStore was funded by Technische Universität München, Friedrich-Alexander-Universität Erlangen-Nürnberg, Friedrich-Schiller-Universität Jena, and Deutsche Forschungsgemeinschaft.

[4]Comparing instruction counts of similar CPU architectures (32-bit x86 vs. 64-bit x86) allows for some abstraction from hardware differences, though not completely.

## REFERENCES

[1] Adnan Alhomssi. 2024. *Concurrency Control for High-Performance Storage Engines.* Ph.D. Dissertation. University of Erlangen-Nuremberg, Germany.

[2] Adnan Alhomssi, Michael Haubenschild, and Viktor Leis. 2023. The Evolution of LeanStore. In *BTW (LNI)*, Vol. P-331. 259–281.

[3] Adnan Alhomssi and Viktor Leis. 2021. Contention and Space Management in B-Trees. In *CIDR*.

[4] Adnan Alhomssi and Viktor Leis. 2023. Scalable and Robust Snapshot Isolation for High-Performance Storage Engines. *PVLDB* 16, 6 (2023), 1426–1438.

[5] Mijin An, In-Yeong Song, Yong Ho Song, and Sang-Won Lee. 2022. Avoiding Read Stalls on Flash Storage. In *SIGMOD*. 1404–1417.

[6] Rudolf Bayer and Mario Schkolnick. 1977. Concurrency of Operations on B-Trees. *Acta Informatica* 9 (1977).

[7] Robert Binna, Eva Zangerle, Martin Pichl, Günther Specht, and Viktor Leis. 2022. Height Optimized Tries. *ACM Trans. Database Syst.* 47, 1 (2022), 3:1–3:46.

[8] Matias Bjørling, Abutalib Aghayev, Hans Holmberg, Aravind Ramesh, Damien Le Moal, Gregory R. Ganger, and George Amvrosiadis. 2021. ZNS: Avoiding the Block Interface Tax for Flash-based SSDs. In *USENIX ATC*. 689–703.

[9] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and robust latches for database systems. In *DaMoN*.

[10] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable Garbage Collection for In-Memory MVCC Systems. *PVLDB* 13, 2 (2019), 128–141.

[11] Nathan Grasso Bronson, Jared Casper, Hassan Chafi, and Kunle Olukotun. 2010. A practical concurrent binary search tree. In *PPoPP*. 257–268.

[12] Sang Kyun Cha, Sangyong Hwang, Kihong Kim, and Keunjoo Kwon. 2001. Cache-Conscious Concurrency Control of Main-Memory Indexes on Shared-Memory Multiprocessor Systems. In *VLDB*. 181–190.

[13] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: An Embedded Concurrent Key-Value Store for State Management. *PVLDB* 11, 12 (2018), 1930–1933.

[14] Andrew Crotty, Viktor Leis, and Andrew Pavlo. 2022. Are You Sure You Want to Use MMAP in Your Database Management System?. In *CIDR*.

[15] Justin DeBrabant, Andrew Pavlo, Stephen Tu, Michael Stonebraker, and Stanley B. Zdonik. 2013. Anti-Caching: A New Approach to Database Management System Architecture. *PVLDB* 6, 14 (2013).

[16] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Åke Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL server's memory-optimized OLTP engine. In *SIGMOD*. 1243–1254.

[17] Ahmed Eldawy, Justin J. Levandoski, and Per-Åke Larson. 2014. Trekking Through Siberia: Managing Cold Data in a Memory-Optimized Database. *PVLDB* 7, 11 (2014).

[18] Facebook. 2024. RocksDB | A persistent key-value store. http://rocksdb.org/.

[19] Michael J. Freitag, Alfons Kemper, and Thomas Neumann. 2022. Memory-Optimized Multi-Version Concurrency Control for Disk-Based Database Systems. *PVLDB* 15, 11 (2022), 2797–2810.

[20] Goetz Graefe. 2011. Modern B-Tree Techniques. *Foundations and Trends in Databases* 3, 4 (2011), 203–402.

[21] Goetz Graefe, Wey Guy, and Caetano Sauer. 2014. *Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, and Media Restore.* Morgan & Claypool Publishers.

[22] Goetz Graefe, Haris Volos, Hideaki Kimura, Harumi A. Kuno, Joseph Tucek, Mark Lillibridge, and Alistair C. Veitch. 2014. In-Memory Performance for Big Data. *PVLDB* 8, 1 (2014), 37–48.

[23] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*.

[24] Gabriel Haas and Viktor Leis. 2023. What Modern NVMe Storage Can Do, And How To Exploit It: High-Performance I/O for High-Performance Storage Engines. *PVLDB* 16, 9 (2023), 2090–2102.

[25] Xiangpeng Hao, Xinjing Zhou, Xiangyao Yu, and Michael Stonebraker. 2024. Towards Buffer Management with Tiered Main Memory. *PACMMOD* 2, 1 (2024), 31:1–31:26.

[26] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the looking glass, and what we found there. In *SIGMOD*.

[27] Michael Haubenschild and Viktor Leis. 2023. Lock-Free Buffer Managers Do Not Require Delayed Memory Reclamation. In *SiMoD@SIGMOD*. 1–3.

[28] Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *SIGMOD*. 877–892.

[29] Kaisong Huang, Darien Imai, Tianzheng Wang, and Dong Xie. 2022. SSDs Striking Back: The Storage Jungle and Its Implications to Persistent Indexes. In *CIDR*.

[30] Kaisong Huang, Tianzheng Wang, Qingqing Zhou, and Qingzhong Meng. 2023. The Art of Latency Hiding in Modern Database Engines. *PVLDB* 17, 3 (2023), 577–590.

[31] Ryan Johnson, Ippokratis Pandis, Nikos Hardavellas, Anastasia Ailamaki, and Babak Falsafi. 2009. Shore-MT: a scalable storage manager for the multicore era. In *EDBT*. 24–35.

[32] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: A Scalable Approach to Logging. *PVLDB* 3, 1 (2010).

[33] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2012. Scalability of write-ahead logging on multicore and multi-socket hardware. *VLDB Journal* 21, 2 (2012).

[34] Hyungsoo Jung, Hyuck Han, and Sooyong Kang. 2017. Scalable Database Logging for Multicores. *PVLDB* 11, 2 (2017).

[35] Alfons Kemper and Donald Kossmann. 1995. Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis. *VLDB Journal* 4, 3 (1995).

[36] Jong-Bin Kim, Hyunsoo Cho, Kihwang Kim, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2020. Long-lived Transactions Made Less Harmful. In *SIGMOD*.

[37] Jong-Bin Kim, Hyeongwon Jang, Seohui Son, Hyuck Han, Sooyong Kang, and Hyungsoo Jung. 2019. Border-Collie: A Wait-free, Read-optimal Algorithm for Database Logging on Multicore Hardware. In *SIGMOD*.

[38] Jong-Bin Kim, Kihwang Kim, Hyunsoo Cho, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2021. Rethink the Scan in MVCC Databases. In *SIGMOD*.

[39] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *SIGMOD*. 691–706.

[40] Per-Åke Larson, Spyros Blanas, Cristian Diaconu, Craig Freedman, Jignesh M. Patel, and Mike Zwilling. 2011. High-Performance Concurrency Control Mechanisms for Main-Memory Databases. *PVLDB* 5, 4 (2011), 298–309.

[41] Bo-Hyun Lee, Mijin An, and Sang-Won Lee. 2023. LRU-C: Parallelizing Database I/Os for Flash SSDs. *PVLDB* 16, 9 (2023), 2364–2376.

[42] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. 2016. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *SIGMOD*.

[43] Viktor Leis, Adnan Alhomssi, Tobias Ziegler, Yannick Loeck, and Christian Dietrich. 2023. Virtual-Memory Assisted Buffer Management. *PACMMOD* 1, 1 (2023), 7:1–7:25.

[44] Viktor Leis and Christian Dietrich. 2024. Cloud-Native Database Systems and Unikernels: Reimagining OS Abstractions for Modern Hardware. *PVLDB* 17, 8 (2024), 2115–2122.

[45] Viktor Leis, Michael Haubenschild, Alfons Kemper, and Thomas Neumann. 2018. LeanStore: In-Memory Data Management beyond Main Memory. In *ICDE*. 185–196.

[46] Viktor Leis, Michael Haubenschild, and Thomas Neumann. 2019. Optimistic Lock Coupling: A Scalable and Efficient General-Purpose Synchronization Method. *IEEE Data Eng. Bull.* 42, 1 (2019), 73–84.

[47] Viktor Leis, Alfons Kemper, and Thomas Neumann. 2013. The adaptive radix tree: ARTful indexing for main-memory databases. In *ICDE*. 38–49.

[48] Viktor Leis, Florian Scheibner, Alfons Kemper, and Thomas Neumann. 2016. The ART of practical synchronization. In *DaMoN*.

[49] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for new hardware platforms. In *ICDE*. 302–313.

[50] Nirmesh Malviya, Ariel Weisberg, Samuel Madden, and Michael Stonebraker. 2014. Rethinking main memory OLTP recovery. In *ICDE*.

[51] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache craftiness for fast multicore key-value storage. In *EuroSys*. 183–196.

[52] Nimrod Megiddo and Dharmendra S. Modha. 2003. ARC: A Self-Tuning, Low Overhead Replacement Cache. In *FAST*.

[53] Maged M. Michael. 2004. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Trans. Parallel Distrib. Syst.* 15, 6 (2004).

[54] C. Mohan, Don Haderle, Bruce G. Lindsay, Hamid Pirahesh, and Peter M. Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS* (1992).

[55] MongoDB. 2024. WiredTiger Storage Engine. https://docs.mongodb.com/manual/core/wiredtiger/.

[56] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*.

[57] Thomas Neumann, Tobias Mühlbauer, and Alfons Kemper. 2015. Fast Serializable Multi-Version Concurrency Control for Main-Memory Database Systems. In *SIGMOD*.

[58] Lam-Duy Nguyen and Viktor Leis. 2024. Why Files If You Have a DBMS?. In *ICDE*.

[59] Elizabeth J. O'Neil, Patrick E. O'Neil, and Gerhard Weikum. 1993. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *SIGMOD*. 297–306.

[60] Tarikul Islam Papon and Manos Athanassoulis. 2023. ACEing the Bufferpool Management Paradigm for Modern Storage Devices. In *ICDE*. 1326–1339.

[61] Aunn Raza, Periklis Chrysogelos, Angelos-Christos G. Anadiotis, and Anastasia Ailamaki. 2023. One-shot Garbage Collection for In-memory OLTP through Temporality-aware Version Storage. *PACMMOD* 1, 1 (2023), 19:1–19:25.

[62] Caetano Sauer, Goetz Graefe, and Theo Härder. 2018. FineLine: log-structured transactional storage and recovery. *PVLDB* 11, 13 (2018), 2249–2262.

[63] Ori Shalev and Nir Shavit. 2003. Split-ordered lists: lock-free extensible hash tables. In *PODC*. 102–111.

[64] Michael Stonebraker, Samuel Madden, Daniel J. Abadi, Stavros Harizopoulos, Nabil Hachem, and Pat Helland. 2007. The End of an Architectural Era (It's Time for a Complete Rewrite). In *VLDB*. 1150–1160.

[65] Symas. 2024. Lightning Memory-Mapped Database Manager (LMDB). http://www.lmdb.tech/doc/.

[66] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOPS*. 18–32.

[67] Demian E. Vöhringer and Viktor Leis. 2023. Write-Aware Timestamp Tracking: Effective and Efficient Page Replacement for Modern Hardware. *PVLDB* 16, 11 (2023), 3323–3334.

[68] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging through Emerging Non-Volatile Memory. *PVLDB* 7, 10 (2014), 865–876.

[69] Ziqi Wang, Andrew Pavlo, Hyeontaek Lim, Viktor Leis, Huanchen Zhang, Michael Kaminsky, and David G. Andersen. 2018. Building a Bw-Tree Takes More Than Just Buzz Words. In *SIGMOD*. 473–488.

[70] Niklaus Wirth. 1995. A Plea for Lean Software. *Computer* 28, 2 (1995), 64–68.

[71] Yingjun Wu, Joy Arulraj, Jiexi Lin, Ran Xian, and Andrew Pavlo. 2017. An Empirical Evaluation of In-Memory Multi-Version Concurrency Control. *PVLDB* 10, 7 (2017), 781–792.

[72] Geoffrey X. Yu, Markos Markakis, Andreas Kipf, Per-Åke Larson, Umar Farooq Minhas, and Tim Kraska. 2022. TreeLine: An Update-In-Place Key-Value Store for Modern Storage. *PVLDB* 16, 1 (2022), 99–112.

[73] Huanchen Zhang, David G. Andersen, Andrew Pavlo, Michael Kaminsky, Lin Ma, and Rui Shen. 2016. Reducing the Storage Overhead of Main-Memory OLTP Databases with Hybrid Indexes. In *SIGMOD*. 1567–1581.

[74] Ling Zhang, Matthew Butrovich, Tianyu Li, Andrew Pavlo, Yash Nannapaneni, John Rollinson, Huanchen Zhang, Ambarish Balakumar, Daniel Biales, Ziqi Dong, Emmanuel J. Eppinger, Jordi E. Gonzalez, Wan Shen Lim, Jianqiao Liu, Lin Ma, Prashanth Menon, Soumil Mukherjee, Tanuj Nayak, Amadou Ngom, Dong Niu, Deepayan Patra, Poojita Raj, Stephanie Wang, Wuwen Wang, Yao Yu, and William Zhang. 2021. Everything is a Transaction: Unifying Logical Concurrency Control and Physical Data Structure Maintenance in Database Management Systems. In *CIDR*.

[75] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David E. Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *SIGMOD*. 2195–2207.

[76] Xinjing Zhou, Xiangyao Yu, Goetz Graefe, and Michael Stonebraker. 2023. Two is Better Than One: The Case for 2-Tree for Skewed Data Sets. In *CIDR*.