LeanStore: In-Memory Data Management Beyond Main Memory

Viktor Leis, Michael Haubenschild*, Alfons Kemper, Thomas Neumann

Technische Universität München
{leis,kemper,neumann}@in.tum.de

*Tableau Software**
mhaubenschild@tableau.com*

***Abstract*—Disk-based database systems use buffer managers in order to transparently manage data sets larger than main memory. This traditional approach is effective at minimizing the number of I/O operations, but is also the major source of overhead in comparison with in-memory systems. To avoid this overhead, in-memory database systems therefore abandon buffer management altogether, which makes handling data sets larger than main memory very difficult.**

In this work, we revisit this fundamental dichotomy and design a novel storage manager that is optimized for modern hardware. Our evaluation, which is based on TPC-C and micro benchmarks, shows that our approach has little overhead in comparison with a pure in-memory system when all data resides in main memory. At the same time, like a traditional buffer manager, it is fully transparent and can manage very large data sets effectively. Furthermore, due to low-overhead synchronization, our implementation is also highly scalable on multi-core CPUs.

I. Introduction

Managing large data sets has always been the raison d'être for database systems. Traditional systems cache pages using a buffer manager, which has complete knowledge of all page accesses and transparently loads/evicts pages from/to disk. By storing all data on fixed-size pages, arbitrary data structures, including database tables and indexes, can be handled uniformly and transparently.

While this design succeeds in minimizing the number of I/O operations, it incurs a large overhead for in-memory workloads, which are increasingly common. In the canonical buffer pool implementation [1], each page access requires a hash table lookup in order to translate a logical page identifier into an in-memory pointer. Even worse, in typical implementations the data structures involved are synchronized using multiple latches, which does not scale on modern multi-core CPUs. As Fig. 1 shows, traditional buffer manager implementations like BerkeleyDB or WiredTiger therefore only achieve a fraction of the TPC-C performance of an in-memory B-tree.

This is why main-memory database systems like H-Store [2], Hekaton [3], HANA [4], HyPer [5], or Silo [6] eschew buffer management altogether. Relations as well as indexes are directly stored in main memory and virtual memory pointers are used instead of page identifiers. This approach is certainly efficient. However, as data sizes grow, asking users to buy more RAM or throw away data is not a viable solution. Scaling-out an in-memory database can be an option, but has downsides including hardware and administration cost. For these reasons, at some point of any main-memory system's evolution, its designers have to implement support for very large data sets.

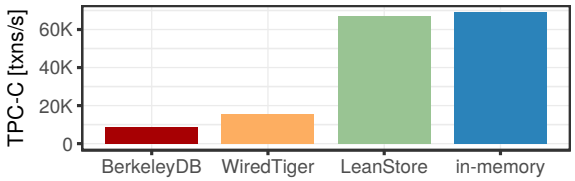

Fig. 1. Single-threaded in-memory TPC-C performance (100 warehouses).

Two representative proposals for efficiently managing larger-than-RAM data sets in main-memory systems are Anti-Caching [7] and Siberia [8], [9], [10]. In comparison with a traditional buffer manager, these approaches exhibit one major weakness: They are not capable of maintaining a replacement strategy over relational *and* index data. Either the indexes, which can constitute a significant fraction of the overall data size [11], must always reside in RAM, or they require a separate mechanism, which makes these techniques less general and less transparent than traditional buffer managers.

Another reason for reconsidering buffer managers are the increasingly common PCIe/M2-attached Solid State Drives (SSDs), which are block devices that require page-wise accesses. These devices can access multiple GB per second, as they are not limited by the relatively slow SATA interface. While modern SSDs are still at least 10 times slower than DRAM in terms of bandwidth, they are also cheaper than DRAM by a similar factor. Thus, for economic reasons [12] alone, buffer managers are becoming attractive again. Given the benefits of buffer managers, there remains only one question: *Is it possible to design an efficient buffer manager for modern hardware?*

In this work, we answer this question affirmatively by designing, implementing, and evaluating a highly efficient storage engine called *LeanStore*. Our design provides an abstraction of similar functionality as a traditional buffer manager, but without incurring its overhead. As Fig. 1 shows, LeanStore's performance is very close to that of an in-memory B-tree when executing TPC-C. The reason for this low overhead is that accessing an in-memory page merely involves a simple, well-predicted `if` statement rather than a costly hash table lookup. We also achieve excellent scalability on modern multi-core CPUs by avoiding fine-grained latching on the hot path. Overall, if the working set fits into RAM, our design achieves the same performance as state-of-the-art main-memory database systems. At the same time, our buffer manager can transparently manage very large data sets on background storage and, using modern SSDs, throughput degrades smoothly as the working set starts to exceed main memory.

II. Related Work

Buffer management is the foundational component in the traditional database architecture [13]. In the classical design, all data structures are stored on fixed-size pages in a translation-free manner (no marshalling/unmarshalling). The rest of the system uses a simple interface that hides the complexities of the I/O buffering mechanism and provides a global replacement strategy across all data structures. Runtime function calls to `pinPage`/`unpinPage` provide the information for deciding which pages need to be kept in RAM and which can be evicted to external memory (based on a replacement strategy like Least-Recently-Used or Second Chance). This elegant design is one of the pillars of classical database systems. It was also shown, however, that for transactional, fully memory-resident workloads a typical buffer manager is the biggest source of inefficiency [14].

One of the defining characteristics of main-memory databases is that they do not have a buffer manager. Memory is instead allocated in variable-sized chunks as needed and data structures use virtual memory pointers directly instead of page identifiers. To support data sets larger than RAM, some main-memory database systems implement a separate mechanism that classifies tuples as either "hot" or "cold". Hot tuples are kept in the efficient in-memory database, and cold tuples are stored on disk or SSD (usually in a completely different storage format). Ma et al. [15] and Zhang et al. [16] survey and evaluate some of the important design decisions. In the following we (briefly) describe the major approaches.

Anti-Caching [7] was proposed by DeBrabant et al. for H-Store. Accesses are tracked on a per-tuple instead of a per-page granularity. To implement this, each relation has an LRU list, which is embedded into each tuple resulting in 8 byte space overhead per tuple. Under memory pressure, some least-recently-used tuples are moved to cold storage (disk or SSD). Indexes cover both hot as well as cold tuples. Given that it is not uncommon for indexes to consume half of the total memory in OLTP databases [11], not being able to move indexes to cold storage is a major limitation. Thus, Anti-Caching merely eases memory pressure for applications that *almost* fit into main memory and is not a general solution.

Microsoft's **Siberia** [10] project is maybe the most comprehensive approach for managing large data sets in main-memory database systems. Tuples are classified as either hot or cold in an offline fashion using a tuple access log [8]. Another offline process migrates infrequently accessed tuples between a high-performance in-memory database and a separate cold storage in a transactional fashion [10]. Siberia's in-memory indexes only cover hot tuples [10] in order to keep these indexes small. In addition, Bloom filters, which require around 10 bits per key, and adaptive range filters [9] are kept in main memory to avoid having to access the cold storage on each tuple lookup. The Siberia approach, however, suffers from high complexity (multiple offline processes with many parameters, two independent storage managers), which may have prevented its widespread adoption.

A very different and seemingly promising alternative is to rely on the operating system's **swapping** (paging) functionality. Stoica and Ailamaki [17], for example, investigated this approach for H-Store. Swapping has the major advantage that hot accesses incur no overhead due to hardware support (the TLB and in-CPU page table walks). The disadvantage is that the database system loses control over page eviction, which virtually precludes in-place updates and full-blown ARIES-style recovery. The problems of letting the operating system decide when to evict pages have been discussed by Graefe et al. [18]. Another disadvantage is that the operating system does not have database-specific knowledge about access patterns (e.g., scans of large tables). In addition, experimental results (cf. [18] and Fig. 9) show that swapping on Linux, which is the most common server operating system, does not perform well for database workloads. We therefore believe that relying on the operating system's swapping/mmap mechanism is not a viable alternative to software-managed approaches. Indeed, main-memory database vendors recommend to carefully monitor main memory consumption to *avoid* swapping [19], [20]. An exception are OLAP-optimized systems like MonetDB, for which relying on the operating system works quite well.

Funke et al. [21] proposed a **hardware-assisted access tracking** mechanism for separating hot and cold tuples in HyPer. By modifying the Linux kernel and utilizing the processor's Memory-Management Unit (MMU), page accesses can be tracked with very little overhead. Their implementation utilizes the same mechanism as the kernel to implement swapping without loosing control over page eviction decisions. It does not, however, handle index structures. Our approach, in contrast, is portable as it does not require modifying the operating system and handles indexes transparently. Combining our approach with hardware-assisted page access tracking would—at the price of giving up portability—improve performance further.

SAP HANA has always been marketed as an in-memory system and, for a long time, a column could only be moved to or from SSD in its entirety [22]. In a recent paper, Sherkat et al. [22] describe a new feature that enables **block-wise access to columnar data**. Indexes and the delta store, however, still reside in main memory, which is a major limitation for large transactional databases.

The fact that practically all in-memory systems have added some form of support for larger-than-RAM data sets clearly shows the importance of this feature. Yet we argue that the techniques discussed above—despite their originality—fall short of being robust and efficient solutions for the cold storage problem in transactional or Hybrid Transactional/Analytical Processing (HTAP) systems. Adding such a foundational feature to an existing in-memory system as an afterthought will not lead to the best possible design. We therefore now discuss a number of recent system proposals that were designed as storage managers from their very inception.

Bw-tree/LLAMA [23], [24] is a storage engine optimized for multi-core CPUs and SSDs. In contrast to our design, updates are performed out-of-place by creating new versions of modified entries ("deltas"). This design decision has benefits.

(a) traditional buffer manager

(b) swizzling-based buffer manager

Fig. 2. Tree structure consisting of a root page (P1) and three leaf pages (P2, P4, P3), two of which are in RAM (P2 and P4).

It enables latch-freedom and, for certain write-intensive workloads, it can reduce SSD write amplification. The downside is that creating and traversing deltas is a non-negligible overhead. Performance is therefore lower than that of in-memory systems that perform updates in place.

FOEDUS [25] is another recent storage manager proposal. It has very good scalability on multi-core CPUs as it avoids unnecessary latch acquisitions, and, like our design, it uses a buffer manager that caches fixed-size pages in main memory. However, FOEDUS is optimized for byte-addressable Storage Class Memories like Phase Change Memory, which have different properties than SSDs (i.e., byte-addressability).

Pointer swizzling, which is crucial in our design, is an old technique that has been extensively used in object-oriented database systems [26]. **Swizzling for buffer managers** has only recently been proposed by Graefe et al. [18], who implemented this idea in the page-based Shore-MT system. Their design shares important characteristics with our proposal. However, while their approach succeeds in reducing the page translation overhead, it does not address the multi-core scalability issues of page-level latches and mandatory page pinning. While latch implementations that reduce the cost of contention (e.g., MCS locks [27]) exist, these still reduce scalability because of cache invalidations during latch acquisition. Optimistic locks, in contrast, do not physically acquire the lock and therefore scale even better.

III. Building Blocks

This section introduces the high-level ideas behind LeanStore, a storage manager for modern hardware (i.e., large DRAM capacities, fast SSDs, and many cores).

A. Pointer Swizzling

In disk-based database systems, the most important data structures (e.g., heap files, B-tree indexes) are organized as fixed-size pages. To refer to other pages, data structures store logical page identifiers that need to be translated to memory addresses before each page access. As shown in Fig. 2a, this translation is typically done using a hash table that contains all pages currently cached in main memory. Today, database servers are equipped with main memory capacities large enough to store the working set of most workloads. As a result, I/O operations are becoming increasingly uncommon and traditional buffer managers often become the performance bottleneck [14].

In our design, pages that reside in main memory are directly referenced using virtual memory addresses (i.e., pointers)—accessing such pages does not require a hash table lookup. Pages that are currently on background storage, on the other hand, are still referenced by their page identifier. This is illustrated in Fig. 2b, which shows page references as circles that either contain a pointer or a page identifier. We use pointer tagging (one bit of the 8-byte reference) to distinguish between these two states. Consequently, the buffer management overhead of accessing a hot page merely consists of one conditional statement that checks this bit.

This technique is called *pointer swizzling* [28] and has been common in object databases [29], [26]. A reference containing an in-memory pointer is called *swizzled*, one that stores an on-disk page identifier is called *unswizzled*. Note that even swizzled pages have logical page identifiers, which are, however, only stored in their buffer frames instead of their references.

B. Efficient Page Replacement

Pointer swizzling is a simple and highly efficient approach for accessing hot pages. However, it does not solve the problem of deciding which pages should be evicted to persistent storage once all buffer pool pages are occupied. Traditional buffer managers use policies like Least Recently Used or Second Chance, which incur additional work for *every* page access. Moreover, for frequently accessed pages (e.g., B-tree roots), updating access tracking information (e.g., LRU lists, Second Chance bits) sometimes becomes a scalability bottleneck. Since our goal is to be competitive with in-memory systems, it is crucial to have a replacement strategy with very low overhead. This is particularly important with pointer swizzling as it does not suffer from expensive page translation.

We therefore deemed updating tracking information for each page access too expensive and avoid it in our design. Our replacement strategy reflects a change of perspective: Instead of tracking frequently accessed pages in order to avoid evicting them, our replacement strategy identifies *infrequently*-accessed pages. We argue that with the large buffer pool sizes that are common today, this is much more efficient as it avoids *any* additional work when accessing a hot page (except for the `if` statement mentioned above).

The main mechanism of our replacement strategy is to speculatively unswizzle a page reference, but without immediately evicting the corresponding page. If the system accidentally unswizzled a frequently-accessed page, this page will be quickly

swizzled again—without incurring any disk I/O. Thus, similar to the Second Chance replacement strategy, a speculatively unswizzled page will have a grace period before it is evicted. Because of this grace period, a very simple (and therefore low-overhead) strategy for picking candidate pages can be used: We simply pick a random page in the pool.

We call the state of pages that are unswizzled but are still in main memory *cooling*. At any point in time we keep a certain percentage of pages (e.g., 10%) in this state. The pages in the cooling state are organized in a FIFO queue. Over time, pages move further down the queue and are evicted if they reach the end of the queue. Accessing a page in the cooling state will, however, prevent eviction as it will cause the page to be removed from the FIFO queue and the page to be swizzled.

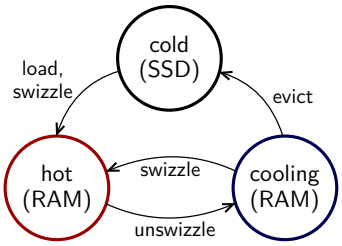

Fig. 3. The possible states of a page.

The "life cycle" of a page and its possible states are illustrated in Fig. 3. Once a page is loaded from SSD, it is swizzled and becomes hot. A hot page may be speculatively unswizzled and, as a result, transitions to the cooling state. A cooling page can become either hot (on access) or cold (on eviction).

To summarize, by speculatively unswizzling random pages, we identify infrequently-accessed pages without having to track each access. In addition, a FIFO queue serves as a probational cooling stage during which pages have a chance to be swizzled. Together, these techniques implement an effective replacement strategy at low cost.

C. Scalable Synchronization

The design described so far implements the basic functionality of a storage engine but does not support concurrency. Thread synchronization is tightly coupled with buffer management and therefore cannot be ignored. For example, before evicting a particular page, the buffer manager has to ensure that this page is not currently being accessed by some other thread.

In most buffer manager implementations, synchronization is implemented using latches. Every page currently in main memory is associated with a latch. Additional latches protect the hash table that maps page identifiers to buffer frames. A call to `pinPage` will first acquire (and, shortly thereafter, release) one or more latches before latching the page itself. The page remains latched—and, as a consequence, cannot be evicted—until `unpinPage` is called. Note that the page latches serve two distinct purposes: They prevent eviction and they are used to implement data structure specific synchronization protocols (e.g., lock coupling in B-trees).

Latch-based synchronization is one of the main sources of overhead in traditional buffer managers. One problem is the sheer number of latch acquisitions. In Shore-MT, for example, a single-row update transaction results in 15 latch acquisitions for the buffer pool and the page latches [30]. An even larger issue is that some latches are acquired frequently by different threads. For example, both the latch of a B-tree root and the global latch for the hash table that translates page identifiers are often critical points of contention. Due to the way cache coherency is implemented on typical multi-core CPUs, such "hot" latches will prevent good scalability ("cacheline ping-pong").

As a general rule, programs that frequently write to memory locations accessed by multiple threads do not scale. LeanStore is therefore carefully engineered to avoid such writes as much as possible by using three techniques: First, pointer swizzling avoids the overhead and scalability problems of latching the translation hash table. Second, instead of preventing page eviction by incrementing per-page pinning counters, we use an epoch-based technique that avoids writing to each accessed page. Third, LeanStore provides a set of optimistic, timestamp-based primitives [31], [32], [33] that can be used by buffer-managed data structures to radically reduce the number of latch acquisitions. Together, these techniques (described in more detail in Section IV-F and Section IV-G) form a general framework for efficiently synchronizing arbitrary buffer-managed data structures. In LeanStore, lookups on swizzled pages do not acquire any latches at all, while insert/update/delete operations usually only acquire a single latch on the leaf node (unless a split/merge occurs). As a result, performance-critical, in-memory operations are highly scalable.

IV. LeanStore

In this section, we describe how LeanStore is implemented using the building blocks presented in Section III.

A. Data Structure Overview

In a traditional buffer manager, the state of the buffer pool is represented by a hash table that maps page identifiers to buffer frames. Frames contain a variety of "housekeeping" information, including (1) the memory pointer to the content of the page, (2) the state required by the replacement strategy (e.g., the LRU list or the Second Chance bit), and (3) information regarding the state of the page on disk (e.g., dirty flag, whether the page is being loaded from disk). These points correspond to 3 different functions that have to be implemented by any buffer manager, namely (1) determining whether a page is in the buffer pool (and, if necessary, page translation), (2) deciding which pages should be in main memory, and (3) management of in-flight I/O operations. LeanStore requires similar information, but for performance reasons, it uses 3 separate, customized data structures. The upper half of Fig. 4 shows that a traditional page translation table is not needed because its state is embedded in the buffer-managed data structures itself. The information for implementing the replacement strategy is moved into a separate structure, which is called *cooling stage* and is illustrated in the lower-left corner of Fig. 4. Finally, in-flight I/O operations are managed in yet another structure shown in the lower-right corner of Fig. 4.

B. Swizzling Details

Pointer swizzling plays a crucial role in our design. We call the reference, i.e., the 8-byte memory location referring to a

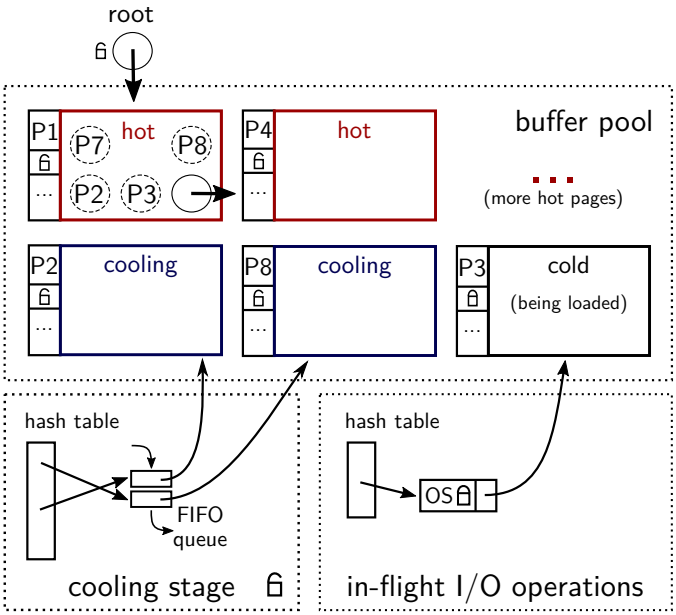

Fig. 4. Overview of LeanStore's data structures. Page P1 represents a root page (e.g., of a B-tree) with 5 child pages (P7, P8, P2, P3, P4). Pages P1 and P4 are hot (swizzled), while pages P2 and P8 are cooling (unswizzled). (In reality, the vast majority of in-memory pages will be classified as hot.) Pages P7 and P3 are on persistent storage with P3 currently being loaded.

page, a *swip*. A swip may either be swizzled (i.e., it stores an in-memory pointer) or unswizzled (i.e., it stores an on-disk page identifier). In Fig. 4, swips are shown as circles storing either a pointer (arrow) or a page identifier (P...). While most swips will typically reside on buffer-managed pages, some swips, for example the swips pointing to B-tree root pages, are stored in memory areas not managed by the buffer pool. In the figure, this is the case for the swip labeled as "root", which is stored outside of the buffer manager.

In a traditional buffer manager, the translation table that maps page identifiers to pointers is the single source of truth representing the state of the buffer pool. Pages are always accessed through this indirection and latches are used for synchronization. As a result, implementing operations like page eviction is fairly straightforward. In a pointer swizzling scheme, in contrast, the information of the translation table is decentralized and embedded in the buffer-managed data structure itself, which makes things more complicated. Consider, for example, a page Px that is referenced by the two pages Py and Pz. In this situation, Px can be referenced by one swizzled and one unswizzled swip at the same time. Maintaining consistency, in particular without using global latches, is very hard and inefficient. Therefore, in LeanStore, *each page has a single owning swip*, which avoids consistency issues when it is (un)swizzled. Consequently, each buffer-managed data structure becomes tree-like and the buffer pool in its entirety a forest of pages. Since each page in the buffer pool is referenced by exactly one swip, we also say a page can be (un)swizzled, depending on the state of the swip referencing it.

Another design decision is that we never unswizzle (and therefore never evict) a page that has swizzled children. A B-tree inner node, for example, can only be unswizzled (and eventually evicted) if all of its child nodes have been unswizzled. Otherwise pages containing memory pointers might be written out to disk, which would be a major problem because a pointer is only valid during the current program execution. To avoid this situation, buffer-managed data structures implement a special swip iteration mechanism that is described in Section IV-E. It ensures that, if an inner page happens to be selected for speculative unswizzling and at least one of its children is swizzled, one of these children is selected instead. While this situation is infrequent in normal operation (as there are typically many more leaf pages than inner pages), it needs to be handled for correctness. Also note that for tree-like data structures, preventing the eviction of a parent before its children is beneficial anyhow, as it reduces page faults.

C. Cooling Stage

The cooling stage is only used when the free pages in the buffer pool are running out. From that moment on, the buffer manager starts to keep a random subset of pages (e.g., 10% of the total pool size) in the cooling state. Most of the in-memory pages will remain in the hot state. Accessing them has very little overhead compared to a pure in-memory system, namely checking one bit of the swip.

As the lower left corner of Fig. 4 illustrates, cooling pages are organized in a FIFO queue. The queue maintains pointers to cooling pages in the order in which they were unswizzled, i.e., the most recently unswizzled pages are at the beginning of the queue and older pages at the end. When memory for a new page is needed, the least recently unswizzled page is used (after flushing it if it is dirty).

When an unswizzled swip is about to be accessed (e.g., swip P8 on page P1), it is necessary to check if the referenced page is in the cooling stage. In addition to the queue, the cooling stage therefore maintains a hash table that maps page identifiers to the corresponding queue entries. If the page is found in the hash table, it is removed from the hash table and from the FIFO queue before it is swizzled to make it accessible again.

Moving pages into the cooling stage could either be done (1) asynchronously by background threads or (2) synchronously by worker threads that access the buffer pool. We use the second option in order to avoid the risk of background threads being too slow. Whenever a thread requests a new, empty page or swizzles a page, it will check if the percentage of cooling pages is below a threshold and will unswizzle a page if necessary.

Our implementation uses a single latch to protect the data structures of the cooling stage. While global latches often become scalability bottlenecks, in this particular case, there is no performance problem. The latch is only required on the cold path, when I/O operations are necessary. Those are orders of magnitude more expensive than a latch acquisition and acquire coarse-grained OS-internal locks anyway. Thus the global latch is fine for both in-memory and I/O-dominated workloads.

D. Input/Output

Before a cold page can be accessed, it must be loaded from persistent storage. One potential issue is that multiple

threads can simultaneously try to load the same page. For correctness reasons, one must prevent the same page from appearing multiple times in the buffer pool. Also, it is obviously more efficient to schedule the I/O operation just once.

Like traditional buffer managers, we therefore manage and serialize in-flight I/O operations explicitly. As Fig. 4 (lower-right corner) illustrates, we maintain a hash table for all pages currently being loaded from disk (P3 in the figure). The hash table maps page identifiers to *I/O frames*, which contain an operating system mutex and a pointer to a newly allocated page. A thread that triggers a load first acquires a global latch, creates an I/O frame, and acquires its mutex. It then releases the global latch and loads the page using a blocking system call (e.g., `pread` on Unix). Other threads will find the existing I/O frame and block on its mutex until the first thread finishes the read operation.

We currently use the same latch to protect both the cooling stage and the I/O component. This simplifies the implementation considerably. It is important to note, however, that this latch is released before doing any I/O system calls. This enables concurrent I/O operations, which are crucial for good performance with SSDs. Also let us re-emphasize that this global latch is not a scalability bottleneck, because—even with fast SSDs—an I/O operation is still much more expensive than the latch acquisition.

E. Buffer-Managed Data Structures

The main feature of a buffer manager is its support for arbitrary data structures while maintaining a single replacement strategy across the entire buffer pool. Another advantage is that this is achieved without the need to understand the physical organization of pages (e.g., the layout of B-tree leaves), which allows implementing buffer-managed data structures without modifying the buffer manager code itself. In the following we describe how LeanStore similarly supports (almost) arbitrary data structures in a non-invasive way.

As mentioned in Section IV-B, our swizzling scheme requires that child pages are unswizzled before their parents. To implement this, the buffer manager must be able to iterate over all swips on a page. In order to avoid having to know the page layout, every buffer-managed data structure therefore implements an "iteration callback". When called for a leaf page, this callback simply does nothing, whereas for an inner page, it iterates over all swips stored on that page. Any buffer-managed data structure must implement this callback and register it with the buffer manager. In addition, pages must be self-describing, i.e., it needs to be possible to determine which callback function the page is associated with. For this reason, every page stores a marker that indicates the page structure (e.g., B-tree leaf or B-tree inner node).

Using the callback mechanism, the buffer manager can determine whether a page can be unswizzled. If a page has no children or all child pages are unswizzled, that page can be unswizzled. If a swizzled child swip is encountered, on the other hand, the buffer manager will abort unswizzling that node. Instead, it will try to unswizzle one of the encountered

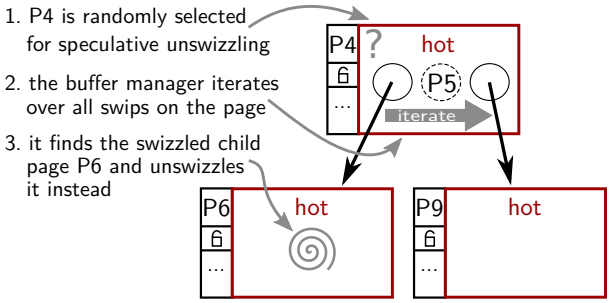

Fig. 5. Inner pages can only be unswizzled after all their child pages.

swizzled child pages (randomly picking one of these). Note that this mechanism, which is illustrated in Fig. 5, implicitly prioritizes inner pages over leaf pages during replacement.

In order to unswizzle a page, besides the callback mechanism, it is also necessary to find the parent page of the eviction candidate. Our implementation uses parent pointers, which are stored in the buffer frame. The parent pointers allow one to easily access the parent page when unswizzling and must be maintained by each data structure. Maintaining these parent pointers is not expensive in our design, because child nodes are always unswizzled before their parents and because parent pointers are not persisted.

As mentioned earlier, in our current implementation every page has one swip, i.e., one incoming reference. This design decision simplifies bookkeeping for unswizzling and was also made by Graefe et al.'s swizzling-based buffer manager [18]. However, this is not a fundamental limitation of our approach. An alternative implementation, in which a page can have multiple incoming pointers, would also be possible. One way to achieve this is to store multiple parent pointers in each frame, which would already be enough to implement inter-leaf pointers in B$^+$-trees. A more general approach would be to keep storing only one parent per page, but have two classes of swips. There would be normal swips that, as in our current design, are linked through parent pointers. In addition, one would have "fat" swips that contain both the page identifier and a pointer.

While our design does impose some constraints on data structures (fixed-size pages and parent pointers), the internal page layout as well as the structural organization are flexible. For example, in prior work [34], we developed a hash index that can directly be used on top of LeanStore: In this data structure, the fixed-size root page uses a number of hash bits to partition the key space (similar to Extendible Hashing). Each partition is then represented as a space-efficient hash table (again using fixed-size pages). Heaps can be implemented similar to B-trees using the tuple identifier as a key. Given that tuple identifiers are often (nearly) dense, one can use special node layouts to avoid the binary search used in B-trees and support very fast scans. The custom node layout also allows implementing columnar storage and compression.

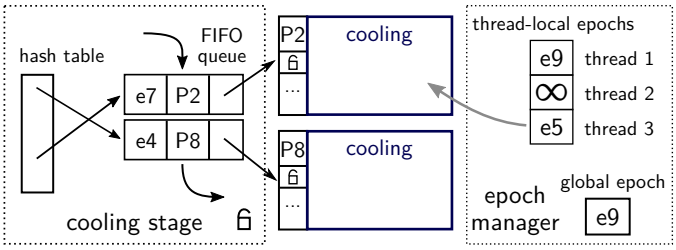

Fig. 6. Epoch-based reclamation.

F. Optimistic Latches

As Fig. 4 illustrates, each page in the buffer pool is associated with a latch. In principle, it would be possible to implement synchronization similar to that of a traditional buffer manager. By acquiring per-page latches on every page access, pages are "pinned" in the buffer pool, which prevents their eviction. Unfortunately, as discussed in Section III-C, requiring frequent latch acquisitions would prevent *any* buffer-managed data structure from being scalable on multi-core CPUs. To be competitive with pure in-memory systems (in terms of performance and scalability), for a buffer manager it is therefore crucial to offer means for efficient synchronization.

The first important technique is to replace the conventional per-page latches with *optimistic latches* [31], [32], [33]. Internally, these latches have an update counter that is incremented after every modification. Readers can proceed *without* acquiring any latches, but validate their reads using the version counters instead (similar to optimistic concurrency control). The actual synchronization protocol is data-structure specific and different variants can be implemented based on optimistic latches. One possible technique is Optimistic Lock Coupling [33], which ensures consistent reads in tree data structures without physically acquiring any latches during traversal. Optimistic Lock Coupling has been shown to be good at synchronizing the adaptive radix tree [35], [33] and B-tree [36]. In this scheme, writers usually only acquire one latch for the page that is modified, and only structure modification operations (e.g., splits) latch multiple pages.

G. Epoch-Based Reclamation

With optimistic latches, pages that are being read are neither latched nor pinned. This may cause problems if some thread wants to evict or delete a page while another thread is still reading the page. To solve this problem, we adopt an epoch-based mechanism, which is often used to implement memory reclamation in latch-free data structures [6], [32], [23].

The basic idea is to have one *global epoch* counter which grows periodically, and per-thread *local epoch* counters. These are shown at the right of Fig. 6. Before a thread accesses any buffer-managed data structure, it sets its local epoch to the current global epoch, which conceptually means that the thread has entered that epoch. In Fig. 6, thread 1 has entered epoch e9, while thread 3 is still in epoch e5. When a thread is finished accessing the data structure, it sets its local epoch to a special value (∞) that indicates that the thread does not access any data at the moment (thread 2 in the example).

When a page is unswizzled, it cannot be reused immediately (as other threads might still be reading from it). For this reason, we associate pages in the cooling stage with the global epoch at the time of unswizzling. (Note that only cooling pages need to be associated with an epoch, not hot pages.) In the example in Fig. 6, page P8 was put into the cooling stage during epoch e4, and page P2 in epoch e7. Right before a page is actually evicted (which happens when the page reaches the end of the FIFO queue), its associated epoch is compared with the minimum of all local epochs. Only if the minimum is larger than the associated epoch, the page can be safely reused since then it is ensured that no thread can read that page anymore. Since the minimum observed epoch of all threads in the example is e5, P8 can be safely evicted, while P2 is kept in the cooling stage until after thread 3 is finished with its current operation. It is unlikely that this check leads to further delay, since it takes quite some time for a page to reach the end of the FIFO queue. Checking this condition is, however, necessary for correctness.

Note that incrementing the global epoch very frequently may result in unnecessary cache invalidations in all cores, whereas very infrequent increments may prevent unswizzled pages from being reclaimed in a timely fashion. Therefore, the frequency of global epoch increments should be proportional to the number of pages deleted/evicted but should be lower by a constant factor (e.g., 100).

To make epochs robust, threads should exit their epoch frequently, because any thread that holds onto its epoch for too long can completely stop page eviction and reclamation. For example, it might be disastrous to perform a large full table scan while staying in the same local epoch. Therefore, we break large operations like table scans into smaller operations, for each of which we acquire and release the epoch. We also ensure that I/O operations, which may take a fairly long time, are never performed while holding on to an epoch. This is implemented as follows: If a page fault occurs, the faulting thread (1) unlocks all page locks, (2) exits the epoch, and (3) executes an I/O operation. Once the I/O request finishes, we trigger a restart of the current data structure operation by throwing a C++ exception. Each data structure operation installs an exception handler that restarts the operation from scratch (i.e., it enters the current global epoch and re-traverses the tree). This simple and robust approach works well for two reasons: First, in-memory operations are very cheap in comparison with I/O. Second, large logical operations (e.g., a large scan) are broken down into many small operations; after a restart only a small amount of work has to be repeated.

H. Memory Allocation and NUMA-Awareness

Besides enabling support for larger-than-RAM data sets, a buffer manager also implicitly provides functionality for memory management—similar to a memory allocator for an in-memory system. Memory management in a buffer pool is simplified, however, due to fixed-size pages (and therefore only a single allocation size). Buffer-managed systems also prevent

memory fragmentation, which often plagues long-running in-memory systems.

LeanStore performs one memory allocation for the desired buffer pool size. This allocation can either be provided physically by the operating system on demand (i.e., first access) or be pre-faulted on startup. LeanStore also supports Non-Uniform Memory Access (NUMA)-aware allocation in order to improve performance on multi-socket systems. In this mode, newly allocated pages are allocated on the same NUMA node as the requesting thread. NUMA-awareness is implemented by logically partitioning the buffer pool and the free lists into as many partitions as there are NUMA nodes. Besides the buffer pool memory itself, all other data structures (including the cooling stage) are centralized in order to maintain a global replacement strategy. NUMA-awareness is a best effort optimization: if no more local pages are available, a remote NUMA page will be assigned.

I. Implementation Details and Optimizations

Let us close this section by mentioning some implementation details and optimizations that are important for performance.

On top of LeanStore, we implemented a fairly straightforward **B$^+$-tree** (values are only stored in leaf nodes). The most interesting aspect of the implementation is the synchronization mechanism. Reads are validated using the version counters embedded into every latch and may need to restart if a conflict is detected. Range scans that span multiple leaf nodes are broken up into multiple individual lookups by using *fence keys* [37]. Together with the efficient synchronization protocol, breaking up range lookups obviates the need for the leaf links, which are common in traditional disk-based B$^+$-trees. An insert, delete, or update operation will first traverse the tree like a lookup, i.e., without acquiring any latches. Once the leaf page that needs to be modified is found, it is latched, which excludes any potential writers (and invalidates potential readers). If no structure-modifying operation (e.g., a node split) is needed (e.g., there is space to insert one more entry), the operation can directly be performed and the latch can be released. Structure-modifying operations release the leaf latch and restart from the root node—but this time latching all affected inner nodes.

In LeanStore, buffer **frames are physically interleaved** with the page content (Fig. 2b). This is in contrast with most traditional buffer managers that store a pointer to the page content in the buffer frame (Fig. 2a). Interleaving the buffer frames improves locality and therefore reduces the number of cache misses because the header of each page is very frequently accessed. A second, more subtle, advantage of interleaving is that it avoids performance problems caused by the limited CPU cache associativity: At typical power-of-two page sizes (and without interleaving), the headers of all pages fall into the same associativity set and only a small number of headers can reside in cache (e.g., 8 for the L1 caches of Intel CPUs).

Because we avoid pinning pages, a page cannot be reused immediately after unswizzling it. This applies not only to pages that are speculatively unswizzled but also to pages that are explicitly deleted (e.g., during a merge operation in a B-tree). One solution is to use the same mechanism described in Section IV-G, i.e., to put the deleted page into the cooling stage. In our implementation, each thread has a small **thread-local cache for deleted pages**. Just like in the cooling stage, the entries are associated with the epoch (of the moment of their deletion). When a thread requests a new, empty page, it consults its local cache and, if the global epoch has advanced sufficiently, chooses a page from the local cache instead of the cooling stage. As a result, deleted pages are reused fairly quickly, which improves cache locality and avoids negative effects on the replacement strategy.

Previously, we argued against asynchronous background processes, because they can cause unpredictable behavior and often have complex, hard-to-tune parameters. We make one exception for writing out dirty pages in the background. Like in traditional buffer managers, we use a separate **background writer** thread to hide the write latency that would otherwise be incurred by worker threads. Our background writer cyclically traverses the FIFO queue of the cooling stage, flushes dirty pages, and clears the dirty flag.

To enable fast scans, LeanStore implements **I/O prefetching**. Using the in-flight I/O component, scans can schedule multiple page requests without blocking. These I/Os are then executed in the background. Once a prefetching request is finished, the corresponding page becomes available to other threads through the cooling stage. Another optimization for large scans is **hinting**. Leaf pages accessed by scans can be classified as cooling instead of hot after being loaded from SSD. These pages are therefore likely to be evicted quickly. Also, since the cooling stage size is limited, no hot pages need to be unswizzled. As a consequence, only a small fraction of the buffer pool will be invalidated during the scan—avoiding thrashing.

V. In-Memory Evaluation

In this section, we investigate the performance and scalability of LeanStore using data sets that fit into main memory. Out-of-memory workloads are evaluated in Section VI.

A. Experimental Setup

The implementation of LeanStore and the B-tree (actually a B$^+$-tree) amounts to around 4 K lines of heavily-templated C++11 code (excluding the benchmarking code). Both the in-memory B-tree and the buffer-managed B-tree have the same page layout and synchronization protocol. This allows us to cleanly quantify the overhead of buffer management.

We use BerkeleyDB 6.0 and WiredTiger 2.9 to compare our design with traditional buffer managers. While BerkeleyDB was initially released in 1994, it is still widely used, for example, as the storage manager of the Oracle NoSQL Database. WiredTiger is a more recent development that has become the default storage engine of MongoDB. Both are standalone storage managers that implement a buffer manager and use fine-grained latches to support concurrency. We configured BerkeleyDB and WiredTiger to use B-trees (BerkeleyDB also supports hashing and WiredTiger supports LSM-trees). For

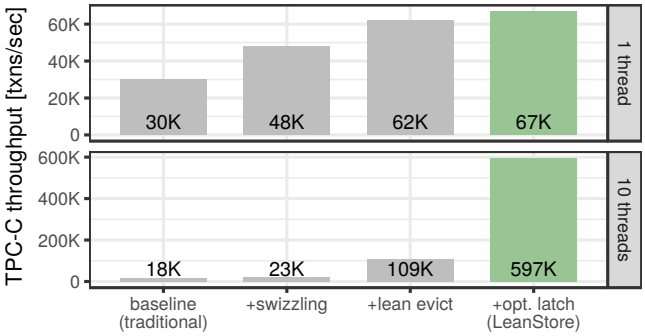

Fig. 7. Impact of the 3 main LeanStore features.

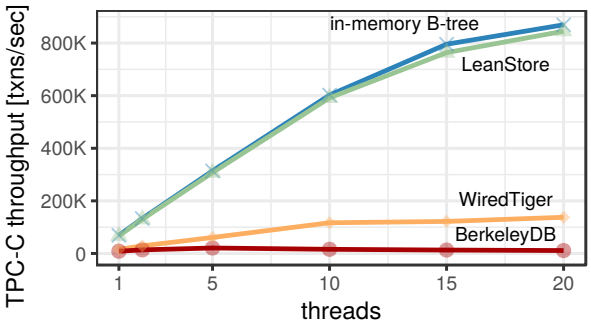

Fig. 8. Multi-threaded, in-memory TPC-C on 10-core system.

both storage managers, we disabled transactions, logging, compaction, and compression.

The storage managers are directly linked/compiled into our test driver, which implements TPC-C (without "think" times) as well as a number of micro benchmarks. Note that even without transactional semantics, TPC-C, which consists of inserts, updates, deletes and range scans, is a fairly complex workload for any storage manager. In terms of access patterns, there are tables with high locality as well as randomly accessed tables and the benchmark is very insert-heavy. We use 16 KB pages and each relation is represented as a single B-tree (no horizontal partitioning into separate trees).

The experiments were performed on a Linux 4.8 system with an Intel Xeon E5-2687W v3 CPU (3.1 GHz, Haswell EP, 10 cores, 20 hardware threads) and 64 GB of main memory.

B. TPC-C

Let us begin with a number of in-memory experiments using TPC-C. We use 100 warehouses, which amounts to (initially) 10 GB, and a buffer pool large enough to encompass all of it. As Fig. 1 (on page 1) shows, with a single thread, LeanStore achieves 67 K transactions per second. The in-memory B-tree is slightly faster at 69 K tps. BerkeleyDB and WiredTiger achieve only 10 K and 16 K tps, respectively.

To better understand which of LeanStore's design decisions are most important, we disabled some of its crucial features. Specifically, we replaced pointer swizzling with traditional hash table lookups, implemented LRU instead of our lean eviction strategy, and used traditional latches instead of optimistic latches. The performance of the resulting system, which resembles a traditional buffer manager, is shown in Fig. 7 (labeled as "baseline"). The figure also shows the performance when swizzling, our replacement strategy, and optimistic latches are enabled step by step. In the single-threaded case, pointer swizzling and our replacement strategy (labeled as "lean evict") are the largest factors, which together result in around 2× performance gain. The optimistic latches are only slightly faster than traditional latches, as single-threaded execution does not suffer from any latch contention. The multi-threaded results show a very different picture. First, the performance differences are much larger. Second, all three features are crucial for obtaining good performance as any scalability bottleneck will result in dramatically lower performance. Using a global hash table for page translation *or* a single LRU list *or* traditional latches immediately prohibits scaling with 10 threads. These results highlight the huge overhead of traditional buffer managers, in particular in a multi-threaded setting, but also show that it is indeed possible to implement a buffer manager with very little overhead.

The performance difference between traditional storage engines and LeanStore becomes even larger when multiple threads are used. As Fig. 8 shows, BerkeleyDB barely scales at all. Its peak performance is 20 K transactions per second with 5 threads (speedup 2.4×). WiredTiger scales better than BerkeleyDB and achieves a speedup of 8.8× with 20 threads. The in-memory B-tree and LeanStore both use the same optimistic synchronization protocol, which enables excellent scalability. Also, there is no contention on the cooling stage latch as it is unused for in-memory workloads. LeanStore achieves a speedup of 8.8× with 10 threads, and 12.6× with HyperThreading.

C. System Comparison

Let us now look at other systems besides BerkeleyDB and WiredTiger. One relevant comparison is to the swizzling-based Shore-MT variant [18]. In a very similar setup as our experiments (in-memory TPC-C with 100 warehouses, 12 clients, no locking, and no logging), it achieves around 70K transactions per second (cf. Figure 14 in [18]), which is 10x less than LeanStore. We believe that this is due to Shore-MT having been designed for a disk-based setting (and therefore having higher CPU overhead) and the scalability issues of fine-grained latches.

Silo [6] and FOEDUS [25] are more recent systems. While they cannot be configured to disable transactions and use different TPC-C implementations, a comparison with these systems is still interesting. Silo achieves 805K and FOEDUS 1,109K TPC-C transactions per second on our machine. LeanStore's performance lies in between with 845K txns/sec. FOEDUS performs better than LeanStore and Silo because of workload-specific access path optimizations (e.g., non-growing tables are stored as fixed-size arrays instead of trees, tables are vertically partitioned into static and dynamic attributes, one table is implemented as an append-only structure). While we used general-purpose B-tree, custom structures can be integrated into LeanStore too (cf. Section IV-E).

TABLE I
LEANSTORE SCALABILITY RUNNING TPC-C ON 60-CORE NUMA SYSTEM.

	txns/sec	speedup	remote accesses
1 thread	45K	1.0×	7%
60 threads: baseline	1,500K	33.3×	77%
+ warehouse affinity	2,270K	50.4×	77%
+ pre-fault memory	2,370K	52.7×	75%
+ NUMA awareness	2,560K	56.9×	14%

D. Scalability on Many-Core Systems

Besides the 10-core system, we also investigated LeanStore's scalability on a 60-core NUMA system, which consists of four 15-core Intel Xeon E7-4870 v2 CPUs (Ivy Bridge, 2.3GHz). Achieving good scalability on such systems is challenging and requires additional optimizations because of the (1) large number of cores, (2) high synchronization cost due to the lack of shared L3 cache, and (3) NUMA. Table I shows the TPC-C performance and speedup with 60 threads and 60 warehouses. The baseline speedup of 33.3× can be increased to 50.4× by assigning each worker thread a *local warehouse*. This well-known, TPC-C specific optimization [6] reduces contention in the workload. Another optimization, which increases the speedup to 52.7×, is to pre-fault the buffer pool memory to avoid scalability issues in the operating system. The final change, explicit NUMA-awareness, reduces the percentage of NUMA-remote DRAM accesses from 75% to 14% and improves performance by 8%. With all optimizations enabled and 60 threads, LeanStore achieves a speedup of 56.9×.

VI. OUT-OF-MEMORY EVALUATION

Let us now focus on experiments with data sets that are larger than main memory, which are, after all, the reason for buffer managers. For storage, the system has a PCIe-attached 400 GB Intel DC P3700 SSD, which provides 2,700/1,080 MB sequential read/write throughput per second—and, like most SSDs, can achieve similar performance with random I/O. Such high-performance SSDs have been commercially available for some time now and are quite affordable. The recently-launched Samsung 960 PRO SSD, for example, offers performance similar to our device for less than $0.7 per GB.

With our high-end SSD, which uses the recently-introduced NVMe interface, we ran into a number of Linux scalability issues, in particular when performing many random reads and writes *simultaneously*. I/O operations are therefore performed using direct I/O (`O_DIRECT`), which prevents operating system caching and makes performance more stable. We also found the file system (ext4) to be a scalability bottleneck, even though our database is organized as a single large file. In addition to using direct I/O, we therefore access the SSD directly as a block device avoiding file system overhead altogether.

A. TPC-C

In the first out-of-memory experiment, the data initially fits into main memory, but then, because of the high write rates of TPC-C, quickly grows to more than twice the size of the buffer pool. For LeanStore, BerkeleyDB, and WiredTiger, the buffer pool size is set to 20 GB. We also measured the performance of the in-memory B-tree when its memory is restricted to 20 GB and the SSD as a swapping device.

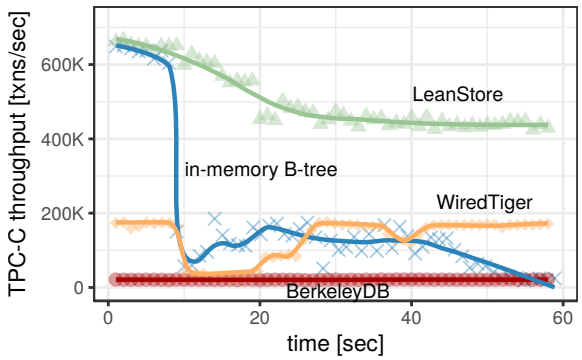

Fig. 9. TPC-C with 20 GB buffer pool (100 warehouses, 20 threads). The data grows from 10 GB to 50 GB—exceeding the buffer pool.

The results in Fig. 9 show that the performance of LeanStore stays close to the in-memory performance although around 500 MB/sec are written out to the SSD in the background. Performance stays very high because the replacement strategy is successful in identifying the working set, which stays in the buffer pool. With swapping the situation looks very different; performance drops severely and is highly unstable, which shows that current implementations of swapping are not a viable alternative. The performance of BerkeleyDB is generally very low and it takes over 10 minutes (not fully shown in the graph) before the buffer pool is exhausted. WiredTiger performs much better than BerkeleyDB, but is still slower by more than 2x compared to LeanStore.

As another I/O-based experiment, we measured the "ramp-up time" from cold cache (e.g., starting the database after a clean shutdown) for a 10 GB database. Besides the PCIe SSD (Intel DC P3700 SSD), we used a low-end consumer SATA SSD (Crucial m4), and a magnetic disk (Western Digital Red) to measure the influence of different storage devices. With the PCIe SSD, peak performance is reached after around 8 seconds whereas the SATA SSD takes longer with 35 seconds. The disk, on the other hand, only achieves around 10 (!) transactions per second for around 15 minutes, because the working set of TPC-C is loaded using a random access pattern. The disk can only read in pages into the buffer pool at around 5 MB/sec, whereas the random access pattern does not impede the performance of SSDs. The large performance gap between magnetic disks and main memory is likely one of the reasons for why disk-based systems have fallen out of fashion. Once I/O becomes necessary, performance often "drops off a cliff". With modern SSDs, on the other hand, performance degrades much more gracefully. High-performance storage engines like LeanStore therefore should only be used with fast storage devices such as SSDs—not disks.

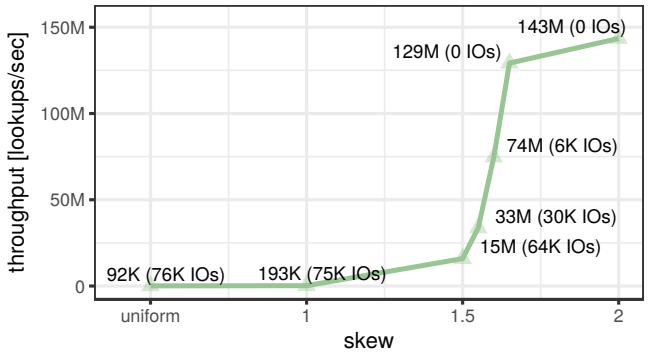

Fig. 10. Lookup performance and number of I/O operations per second (20 threads, 5 GB data set, 1 GB buffer pool).

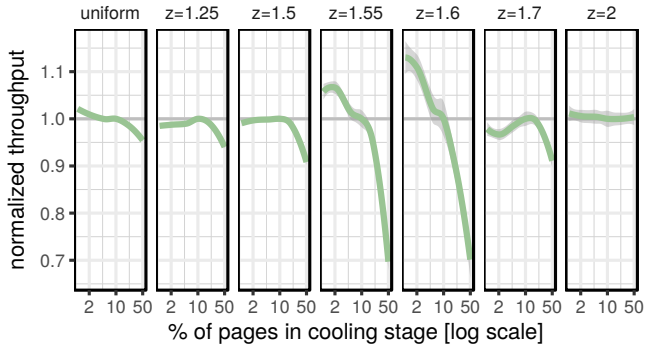

Fig. 11. Effect of cooling stage size on throughput. The throughput is normalized by the 10% cooling pages setting.

B. Point Lookups

All experiments so far were based on TPC-C, which—while certainly being challenging—may not be representative of many real-world workloads. In particular, TPC-C is very insert-heavy (it has a low select to insert ratio), has a number of large relations (`stock`, `customer`) that are accessed in a completely random fashion, the largest relation (`orderline`) has a very peculiar access pattern. As a result, the working set is a significant fraction of the total data size.

To model common workloads, we therefore implemented a read-only micro benchmark similar to YCSB workload C. Our data set has 5 GB and consists of one B-tree storing 41 M key/value pairs (the keys are 8 bytes, the values are 120 bytes) and the buffer pool size is 1 GB. We perform lookups with 20 threads by drawing keys from a uniform or Zipf distribution and use a cooling stage size of 10%. Fig. 10 shows the lookup performance and the number of read I/O operations per second under varying degrees of skew. In low skew settings, the majority of all lookups lead to page faults and the throughput is therefore close to the number of I/O operations. As the skew increases and the number of page faults decreases, much higher performance is achieved. These results show that our replacement strategy effectively identifies the working set.

Our replacement strategy's only parameter is the percentage of pages in the cooling stage. To find a good setting, we varied the size of the cooling stage from 1% to 50%. Fig. 11 shows the results, which are normalized using the 10% setting, for different levels of skew. The performance is very stable under different cooling stage sizes—in particular for "reasonable" settings between 5% and 20% percent. Under low skew (below 1.55), performance decreases with larger cooling stage sizes, since there is basically no working set to be identified and wasting any memory for the cooling stage is unnecessary. Under very high skew (above 1.7), on the other hand, the working set is tiny, and can quickly be identified—with a small as well as a large cooling stage.

Only for very specific skew settings (around 1.6), one can observe performance that is more than 10% off from the optimum. For these skew settings, the working set is close to the buffer pool size. If the cooling stage size is too large, performance is affected by frequently (and unnecessarily) swizzling and unswizzling hot pages. In other settings (e.g., 1.7), the working set cannot effectively be identified if the cooling stage size parameter is too small. We can generally recommend a value of 10% for the cooling stage size as a default setting, because this offers a good tradeoff between the factors described above.

Besides executing the workload in LeanStore directly, we also traced all page accesses and simulated different replacement strategies. The page hit rates for a 5 GB data set, 1 GB buffer pool, and a Zipf factor of 1.0 are as follows:

		LeanEvict						
Random	FIFO	5%	10%	20%	50%	LRU	2Q	OPT
92.5%	92.5%	92.7%	92.8%	92.9%	93.0%	93.1%	93.8%	96.3%

These results show that the page hit rate of our replacement strategy ("LeanEvict") lies between that of very simple strategies (random and FIFO) and that of more elaborate techniques (LRU, 2Q). However, with the exception of the optimal replacement strategy ("OPT"), which is only of theoretical interest, all approaches are fairly close. Also note that the page hit rates do not directly translate into performance, as more complex strategies like LRU and 2Q would also result in a higher runtime overhead that is not captured in page hit rates. We thus argue that in the modern hardware landscape, simple strategies are often preferable due to lower runtime overhead.

C. Scans

So far, the evaluation focused on index-heavy workloads, which are quite challenging for storage engines. Since we designed LeanStore as a general-purpose storage engine, we also experimented with full table scans. To do this, we execute two full table scans using two threads. One thread continuously scans a 10 GB *orderline* table, and another thread continuously scans a 0.7 GB *order* table (we use TPC-C data with 400 warehouses). We vary the buffer pool size from 2 GB to 12 GB and enable I/O prefetching. Fig. 12 (top) shows the scan speed of both threads. The smaller scan, which is shown as the dotted black line, is unaffected by the different buffer pool sizes because the replacement strategy always successfully keeps its pages in the buffer pool. The performance of the larger scan, on the other hand, depends on the size of the buffer pool, as is shown by the four green lines of Fig. 12.

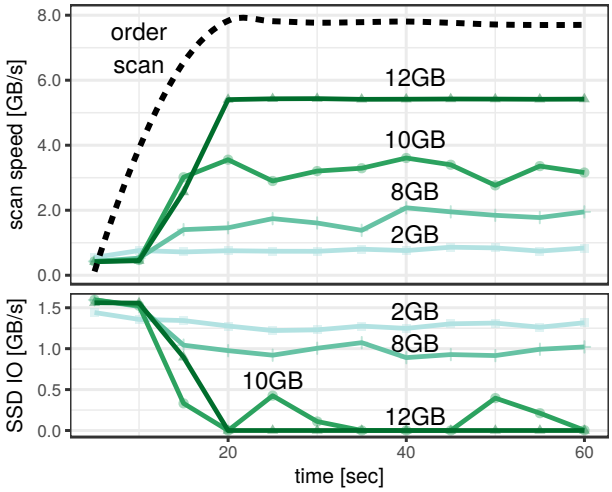

Fig. 12. Concurrent scan of the 0.7 GB order table and the 10 GB orderline table using buffer pool sizes between 2 GB and 12 GB.

For pool sizes below 12 GB, the larger table cannot fully be cached. Therefore, parts of it must be continually read from SSD, as can be observed in Fig. 12 (bottom), which shows how much data is read from SSD. One interesting case is the 10 GB setting, where the buffer pool is slightly smaller than the size of both tables combined. Since some data has to be read from SSD in each iteration of the scan, we see a cyclical I/O pattern (at 25 and 50 seconds). Despite the occasional I/O, the scan performance remains high because most of the data is cached.

VII. Summary

We have presented LeanStore, a novel storage manager that is based on pointer swizzling, a low-overhead replacement strategy, and a scalable synchronization framework. Our experiments show that this design has negligible overhead in comparison with in-memory data structures, while supporting the same functionality as a conventional buffer manager.

References

[1] W. Effelsberg and T. Härder, "Principles of database buffer management," *ACM Trans. Database Syst.*, vol. 9, no. 4, 1984.

[2] A. Pavlo, "On scalable transaction execution in partitioned main memory database management systems," Ph.D. dissertation, Brown University, 2013.

[3] C. Diaconu, C. Freedman, E. Ismert, P. Larson, P. Mittal, R. Stonecipher, N. Verma, and M. Zwilling, "Hekaton: SQL server's memory-optimized OLTP engine," in *SIGMOD*, 2013.

[4] F. Färber, N. May, W. Lehner, P. Große, I. Müller, H. Rauhe, and J. Dees, "The SAP HANA database – an architecture overview," *IEEE Data Eng. Bull.*, vol. 35, no. 1, 2012.

[5] A. Kemper and T. Neumann, "HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots," in *ICDE*, 2011.

[6] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," in *SOSP*, 2013.

[7] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. B. Zdonik, "Anti-Caching: A new approach to database management system architecture," *PVLDB*, vol. 6, no. 14, 2013.

[8] J. J. Levandoski, P. Larson, and R. Stoica, "Identifying hot and cold data in main-memory databases," in *ICDE*, 2013.

[9] K. Alexiou, D. Kossmann, and P. Larson, "Adaptive range filters for cold data: Avoiding trips to Siberia," *PVLDB*, vol. 6, no. 14, 2013.

[10] A. Eldawy, J. J. Levandoski, and P. Larson, "Trekking through Siberia: Managing cold data in a memory-optimized database," *PVLDB*, vol. 7, no. 11, 2014.

[11] H. Zhang, D. G. Andersen, A. Pavlo, M. Kaminsky, L. Ma, and R. Shen, "Reducing the storage overhead of main-memory OLTP databases with hybrid indexes," in *SIGMOD*, 2016.

[12] J. Gray and G. Graefe, "The five-minute rule ten years later, and other computer storage rules of thumb," *SIGMOD Record*, vol. 26, no. 4, 1997.

[13] J. M. Hellerstein, M. Stonebraker, and J. R. Hamilton, "Architecture of a database system," *Foundations and Trends in Databases*, vol. 1, no. 2, 2007.

[14] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker, "OLTP through the looking glass, and what we found there," in *SIGMOD*, 2008.

[15] L. Ma, J. Arulraj, Z. Zhao, A. Pavlo, S. Dulloor, M. Giardino, J. Parkhurst, J. Gardner, K. Doshi, and S. Zdonik, "Larger-than-memory data management on modern storage hardware for in-memory OLTP database systems," in *DaMoN*, 2016.

[16] H. Zhang, G. Chen, B. C. Ooi, W. Wong, S. Wu, and Y. Xia, ""Anti-Caching"-based elastic memory management for Big Data," in *ICDE*, 2015.

[17] R. Stoica and A. Ailamaki, "Enabling efficient OS paging for main-memory OLTP databases," in *DaMoN*, 2013.

[18] G. Graefe, H. Volos, H. Kimura, H. A. Kuno, J. Tucek, M. Lillibridge, and A. C. Veitch, "In-memory performance for Big Data," *PVLDB*, vol. 8, no. 1, 2014.

[19] "SAP HANA memory usage explained," https://hcp.sap.com/content/dam/website/saphana/en_us/Technology%20Documents/HANA_Memory_Usage_SPS8.pdf.

[20] "MemSQL memory management," https://docs.memsql.com/docs/memory-management.

[21] F. Funke, A. Kemper, and T. Neumann, "Compacting transactional data in hybrid OLTP&OLAP databases," *PVLDB*, vol. 5, no. 11, 2012.

[22] R. Sherkat, C. Florendo, M. Andrei, A. K. Goel, A. Nica, P. Bumbulis, I. Schreter, G. Radestock, C. Bensberg, D. Booss, and H. Gerwens, "Page as you go: Piecewise columnar access in SAP HANA," in *SIGMOD*, 2016.

[23] J. J. Levandoski, D. B. Lomet, and S. Sengupta, "The Bw-Tree: A B-tree for new hardware platforms," in *ICDE*, 2013.

[24] J. Levandoski, D. Lomet, and S. Sengupta, "LLAMA: a cache/storage subsystem for modern hardware," *PVLDB*, vol. 6, no. 10, 2013.

[25] H. Kimura, "FOEDUS: OLTP engine for a thousand cores and NVRAM," in *SIGMOD*, 2015.

[26] A. Kemper and D. Kossmann, "Adaptable pointer swizzling strategies in object bases: Design, realization, and quantitative analysis," *VLDB Journal*, vol. 4, no. 3, 1995.

[27] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, 1991.

[28] H. Garcia-Molina, J. D. Ullman, and J. Widom, *Database systems - the complete book (2. ed.)*. Pearson Education, 2009.

[29] S. J. White and D. J. DeWitt, "QuickStore: A high performance mapped object store," in *SIGMOD*, 1994.

[30] R. Johnson, I. Pandis, and A. Ailamaki, "Eliminating unscalable communication in transaction processing," *VLDB Journal*, vol. 23, no. 1, 2014.

[31] S. K. Cha, S. Hwang, K. Kim, and K. Kwon, "Cache-conscious concurrency control of main-memory indexes on shared-memory multiprocessor systems," in *PVLDB*, 2001.

[32] Y. Mao, E. Kohler, and R. T. Morris, "Cache craftiness for fast multicore key-value storage," in *EuroSys*, 2012.

[33] V. Leis, F. Scheibner, A. Kemper, and T. Neumann, "The ART of practical synchronization," in *DaMoN*, 2016.

[34] R. J. Barber, V. Leis, G. M. Lohman, V. Raman, and R. S. Sidle, "Fast multi-tier indexing supporting dynamic update," US Patent 20 160 283 538, 2015.

[35] V. Leis, A. Kemper, and T. Neumann, "The adaptive radix tree: ARTful indexing for main-memory databases," in *ICDE*, 2013.

[36] Z. Wang, A. Pavlo, H. Lim, V. Leis, H. Zhang, M. Kaminsky, and D. Andersen, "Building a bw-tree takes more than just buzz words," in *SIGMOD*, 2018.

[37] G. Graefe, "A survey of B-tree locking techniques," *ACM Trans. Database Syst.*, vol. 35, no. 3, 2010.