Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines

Michael Haubenschild
mhaubenschild@tableau.com
Tableau Software

Caetano Sauer
csauer@tableau.com
Tableau Software

Thomas Neumann
neumann@in.tum.de
Technische Universität München

Viktor Leis
viktor.leis@uni-jena.de
Friedrich-Schiller-Universität Jena

ABSTRACT

For decades, ARIES has been the standard for logging and recovery in database systems. ARIES offers important features like support for arbitrary workloads, fuzzy checkpoints, and transparent index recovery. Nevertheless, many modern in-memory database systems use more lightweight approaches that have less overhead and better multi-core scalability but only work well for the in-memory setting. Recently, a new class of high-performance storage engines has emerged, which exploit fast SSDs to achieve performance close to pure in-memory systems but also allow out-of-memory workloads. For these systems, ARIES is too slow whereas in-memory logging proposals are not applicable.

In this work, we propose a new logging and recovery design that supports incremental and fuzzy checkpointing, index recovery, out-of-memory workloads, and low-latency transaction commits. Our continuous checkpointing algorithm guarantees bounded recovery time. Using per-thread logging with minimal synchronization, our implementation achieves near-linear scalability on multi-core CPUs. We implemented and evaluated these techniques in our LeanStore storage engine. For working sets that fit in main memory, we achieve performance close to that of an in-memory approach, even with logging, checkpointing, and dirty page writing enabled. For the out-of-memory scenario, we outperform a state-of-the-art ARIES implementation by a factor of two.

ACM Reference Format:
Michael Haubenschild, Caetano Sauer, Thomas Neumann, and Viktor Leis. 2020. Rethinking Logging, Checkpoints, and Recovery for High-Performance Storage Engines. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD'20), June 14–19, 2020, Portland, OR, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3318464.3389716

SIGMOD'20, June 14–19, 2020, Portland, OR, USA

ACM ISBN 978-1-4503-6735-6/20/06.
https://doi.org/10.1145/3318464.3389716

1 INTRODUCTION

Durability and recovery after failure are key features of database management systems. The design and implementation of recovery has implications across the entire database system architecture and affects overall performance. For decades, ARIES-style write-ahead logging (WAL) [38] has been the de facto standard for logging and recovery in disk-based database systems. This is due to the large feature set ARIES provides: it works with datasets and transaction footprints much larger than main memory, enables fast recovery in the presence of repeated crashes, provides native and transparent support for indexes and space management, allows page-based fuzzy checkpoints with low interference, and is able to recover from media failures.

Decades of rising DRAM capacities made it possible to keep many datasets in main memory rather than on disk. This hardware trend revealed major overheads in the traditional database system architecture. A study by Harizopoulos et al. [20] found that the Shore storage engine spends more than 50% of time on buffer management and logging. This has led to the development of in-memory database systems like Silo [49] and VoltDB [36], which avoid buffer management altogether by keeping all data in main memory and rely on lightweight logging techniques rather than full-blown ARIES. Modern in-memory database systems are therefore much more efficient (and scalable) than traditional disk-based implementations. Whereas Shore requires over 200k instructions per *neworder* TPC-C transaction only for logging [20], Silo executes the *entire* TPC-C transaction (including logging) using fewer than 100k instructions.

More recently, fast PCIe-attached solid-state drives (SSD) have emerged, changing the hardware landscape once again. During the last five years, DRAM prices and capacities have stagnated, while flash-based SSDs have become much cheaper and faster [18]. Currently, main memory costs about 20 times more per gigabyte than SSD. A modern SSD typically has a bandwidth of over 3 GB/s, and a single server has enough PCIe lanes to directly attach 8 or 16 SSDs. This results in a hitherto unprecedented secondary storage bandwidth. SSDs are also much faster at random access than disks, making them suitable for both OLAP and OLTP.

Because SSDs provide block-wise access, and data needs to be loaded into DRAM before it can be processed, a number of high-performance storage engines have recently been proposed to exploit fast SSDs [7, 31, 39, 50]. Using techniques like pointer swizzling [31] and scalable optimistic synchronization [32, 33], systems like LeanStore [31] offer transparent buffer management with very little overhead. This SSD-optimized approach is a new database architecture that significantly differs from both the disk-based and the in-memory designs.

Another potential alternative to SSDs is persistent memory. Unfortunately, the first generation of byte-addressable persistent memory ("Intel Optane DC Persistent Memory") is currently almost as expensive as DRAM, and is therefore not (yet) the ideal medium for primary data storage. However, its low write latency makes it a perfect technology for persisting the log. Therefore, for high-performance storage engines, an economical design is to store the data itself on SSD and use a small amount of persistent memory to store just the tail of the WAL. What currently remains unclear is how to best implement logging and recovery for this new class of database systems.

Neither of the existing approaches (ARIES or in-memory) is a good fit for SSD-optimized high-performance storage engines. Conceptually, traditional ARIES-based designs have all the required features and are designed for the page-wise storage access, but have too much overhead and do not scale well on modern multi-core CPUs due to the centralized logging component. State-of-the-art in-memory recovery techniques, as used by SiloR [55] and Hekaton [13, 30], have little overhead and good scalability, but are not optimized for data sets larger than memory.

In this work, we propose a logging and recovery design that is well suited for modern high-performance storage engines. It extends the scalable logging scheme proposed by Wang and Johnson [52] with a novel continuous checkpointing algorithm, an efficient page provisioning strategy, and an optimized cross-log commit protocol. We integrate all these components into LeanStore [31], a lightweight buffer manager that uses pointer swizzling and optimistic synchronization. The resulting system can sustain transaction rates close to that of a pure in-memory system when all data fits in DRAM, as well as handle out-of-memory workloads transparently. Our approach has low CPU overhead, scales well on multi-core CPUs, balances I/O overhead and recovery time, and exploits modern storage devices such as persistent memory for low-latency commits and PCIe-attached SSDs for larger-than-memory data sets. It offers many of the features typically associated with ARIES, including fuzzy checkpoints, index recovery, and transaction footprints larger than memory (*steal*), but has much lower overhead than ARIES implementations used in disk-based systems. Compared to over 200k instructions per TPC-C transaction spent for logging in Shore [20], our approach requires only 15k instructions, and scales well on multi-core CPUs.

The key contributions of this work are (1) a framework of practical techniques for all recovery- and durability-related components in flash-optimized high-performance storage engines, (2) a logging and commit protocol that uses *remote flush avoidance* for low-latency transactions on persistent memory, (3) a novel design for low-overhead, *continuous checkpointing* that bounds recovery time and smoothly spreads writes over time, (4) a thorough discussion of page provisioning in the context of high-performance buffer managers, and (5) an implementation and evaluation of our approach with real persistent memory against existing disk-based and in-memory designs, demonstrating its low instruction footprint, good scalability, and fast recovery.

2 BACKGROUND

This section reviews system designs that are relevant to the approach proposed in this paper.

2.1 ARIES

ARIES [37, 38] has become the standard logging and recovery mechanism of disk-based DBMSs. One of the cornerstones of the ARIES design is *physiological logging*, which assigns a page identifier to each log record but permits logical updates within a page. Logging in such a page-oriented fashion implies that pages can be recovered in parallel and provides a good trade-off between log volume and replay efficiency. ARIES also supports in-place updates on database storage with a *steal* policy, i.e., uncommitted changes can be written to disk. In order to remove such changes during recovery, an undo phase executes and logs *compensation* operations that are the logical inverse of the operations performed during forward processing or redo recovery. Such logical compensation steps enable record-level locking (i.e., concurrency control at the sub-page level) and system transactions that perform maintenance steps such as page splits.

Physiological logging and logical undo also provide two key benefits, the first of which is *fuzzy checkpointing*. This means that a dirty page can be written at any time, regardless of transaction activity, as long as no physical operations are in progress on that page—i.e., as long as appropriate latches are acquired. Fuzzy checkpointing introduces very low overhead on running transactions and permits constant propagation of updates to disk, which guarantees bounded recovery time, steady forward processing performance, and well-balanced I/O throughput. In addition, fuzzy checkpointing makes it much easier to produce both full and incremental backups, and thus provides support for efficient media recovery. The second benefit is the ability to log and recover

arbitrary page-based data structures, which implies that indexes can be recovered along with primary data and thus need not be fully rebuilt during recovery.

Despite all the advantages mentioned here, ARIES has been deemed obsolete by many recent proposals for its excessive forward-processing overhead and lack of scalability. Unfortunately, most of these approaches (as detailed below) also forego the features and advantages mentioned here. Therefore, a key design objective of this work is to maintain the features of ARIES-style write-ahead logging without abandoning performance and scalability.

2.2 In-memory Database Systems

In-memory database systems, in contrast to ARIES, employ lightweight logging techniques. While many variations of logical and physical logging exist (which we review in Section 5), the *value logging* approach used in Silo [49, 55] is the most relevant for this paper. With value logging, a log record does not contain a page identifier; rather, it simply stores a logical tuple identifier and a transaction ID, along with the modified tuple's contents. The main advantage of this approach is that it removes the dependency between log records of the same page, which in physiological logging is a source of additional thread synchronization during both normal processing and recovery.

To eliminate the bottleneck of a centralized log, Silo employs distributed logging among CPU cores [28, 49, 52]. This approach uses multiple log buffers, where each transaction is assigned to a log buffer (i.e., logs are partitioned by transaction) and each buffer is written to its own file without any coordination with other threads. Since durability is guided by *epochs* and a group commit protocol, recovery processing requires establishing the maximum epoch that is persisted in every log file. Then, each log file can be replayed in parallel and in an arbitrary order, thanks to the value logging mechanism and the use of monotonically-increasing transaction IDs (largest ID wins). Despite these advantages, value logging suffers from the following inherent disadvantages: it does not support index recovery, requires the entire data set to live in main memory, and lacks incremental checkpoints. Our approach combines the best of both worlds: the scalability and low overhead of value logging and the features of physiological logging.

2.3 Checkpoints

Checkpoints in a physiological logging system require writing dirty pages to persistent storage. Unfortunately, as we detail in Section 3.4, the state of the art has not been developed beyond the traditional disk-based techniques. Instead, recent research has mostly focused on in-memory databases that must write the entire database at a tuple granularity. This paper bridges this gap by proposing checkpointing and eviction techniques that are appropriate for the high transaction rates and I/O volume made possible by modern hardware.

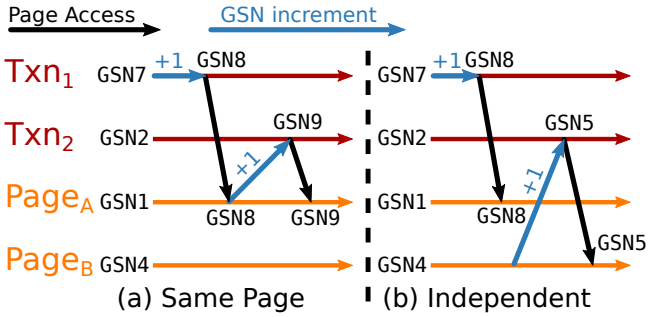

Figure 1: GSNs establish a partial order between log records sufficient for recovery. If two changes depend on each other, the second one will have a higher GSN (a). For independent changes, GSNs are unordered (b).

2.4 Scalable Logging

The obvious approach for improving the scalability of ARIES is to replace the single global log with multiple logs, and assign one or a few threads to each. Thus, when transactions write log records, they do not contend on a single global lock. However, on commit, a transaction has to persist the log records in other log partitions upon which their own changes depend. When the logs are stored on HDD, or even on SSD, flushing several logs becomes prohibitively expensive. Furthermore, distributed logs require additional measures for correct recovery: since each log assigns its own local LSN, the order in which to replay changes for a given page from different logs becomes nondeterministic.

Wang and Johnson [52] propose a scalable logging technique that tackles both issues by exploiting persistent memory and introducing the concept of *global sequence numbers (GSN)*. Persistent memory, which is by now commercially available (e.g., Intel Optane DC Persistent Memory), reduces the latency until changes are drained from CPU caches to persistent storage by more than an order of magnitude. Furthermore, these flushes can be done fully in parallel.

GSNs are a lightweight, decentralized mechanism which establishes a partial order between log records from different log partitions similar to distributed clocks [29] (also known as "Lamport timestamps"). In a distributed clock analogy, both transactions and pages act like processes in a distributed system, with log records being the events that need to be ordered. A transaction's `txnGSN` (timestamp) is set to `max(txnGSN,pageGSN)` whenever it accesses a page (synchronizes its local clock). New log records (events) are created with `txnGSN+1`. When two transactions access the same page (Figure 1a), the protocol ensures that the log record for the first change has the smaller GSN. On the other

hand, when two transactions access distinct pages, their GSNs are not synchronized, and the latter event can even have a smaller GSN (Figure 1b). Note that the full protocol also ensures GSN ordering of log records inside each log, which we omitted from Figure 1 for simplicity. During recovery, log records for a page are gathered from all individual logs, and sorted by GSN before they are applied.

For durability, Wang and Johnson propose *passive group commit*: When a transaction commits, it first flushes its own log and then waits in a group commit queue. Once all other logs are durable up to the transaction's commit record GSN, its commit is finally acknowledged. This design effectively solves the scalability issue for logging, and indeed we use it as the basis for logging in our system. However, its implementation in Shore-MT still suffers from high instruction overhead, has unnecessarily high transaction latencies and relies on a custom kernel module. For reference, the reported TPC-C throughput [52] with 40 threads is below what our system achieves with a single thread (26k vs. 41k txn/s).

3 LOW-LATENCY LOGGING AND ROBUST CHECKPOINTING WITH BOUNDED RECOVERY

In this paper, we propose a holistic approach for logging, checkpointing, and dirty page writing that combines the benefits of ARIES with those of lightweight logging techniques. In our approach, each worker thread is assigned its own log, which eliminates the single point of contention and improves scalability on multi-core CPUs. Log records consist of a type, page ID, transaction ID, GSN (see Section 2.4), and the before and after image of each change. Using a small persistent memory buffer, transactions commit immediately when they are finished, without being appended to a group commit queue first. We improve over existing distributed logging schemes with a new mechanism we call *Remote Flush Avoidance* (RFA), which can detect and skip unnecessary flushes in logs of other threads. Furthermore, our system uses a novel *continuous checkpointing* algorithm that smoothly spreads I/O over time and thus avoids write bursts and spikes in transaction latency. Compared to traditional ARIES implementations, the instruction overhead for creating log entries and writing them to persistent storage is much lower in our system. Finally, we support larger-than-memory workloads, since we build our logging framework on top of the state-of-the-art LeanStore engine.

3.1 Two-Stage Distributed Logging

The main scalability issue of ARIES is the synchronized access to the global log. Approaches such as Aether [22, 23], ELEDA [24], and Border-Collie [25] reduce contention on the log by minimizing the time spent in critical sections, but

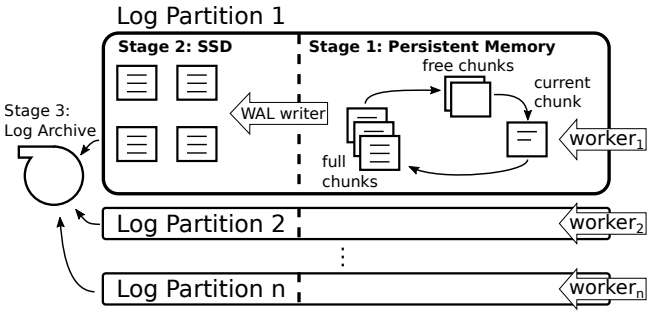

Figure 2: Overview of two-stage distributed logging.

with a large-enough core count, the single log still becomes the scalability bottleneck. Therefore, in our system, each worker thread has its own separate log as shown in Figure 2. Every transaction is pinned to a worker thread, so that its log records are all written to exactly one of those logs. Different transactions can still operate on the same page, so log records for a certain page can end up in different logs. This design is based on the transaction-level log partitioning approach of Wang and Johnson [52] explained in Section 2.4.

The log is organized into three stages. The first stage consists of a small number of log chunks organized in a circular list as shown in Figure 2. One of the chunks is always designated as the *current* chunk, which is where transactions append their log records. Whenever the current log chunk becomes full, it enters the *full* portion of the list. From there, chunks are picked up by a dedicated WAL writer thread—also one for each log—which flushes them into the second stage. After a chunk is successfully written, its buffer is zeroed out and placed into the *free* portion of the list, from which it will eventually be picked up as the current chunk to append new log records. Lastly, log files are archived into the third stage for media recovery—this is shown on the left side of Figure 2.

In the hardware configuration assumed throughout this paper, the first stage resides on persistent memory or battery-backed DRAM. Thus, transactions only need to flush CPU caches when they want to persist log records upon transaction commit, and do not have to wait for the staging of full log chunks to SSD. This allows for very low commit latency and high transaction throughput without the need for group commit. For the second stage, SSD storage is a good candidate because it is much cheaper than persistent memory. Furthermore, SSD bandwidth is sufficient for the amount of log volume that even the fastest transaction systems can produce.

Notwithstanding the benefits of logging to persistent memory, our design is not restricted to it. An alternative solution that keeps the first stage in DRAM and guarantees persistence once log chunks are flushed to SSD still achieves high throughput by employing an RFA-optimized version of group commit. This alternative is presented at the end of Section 3.2.

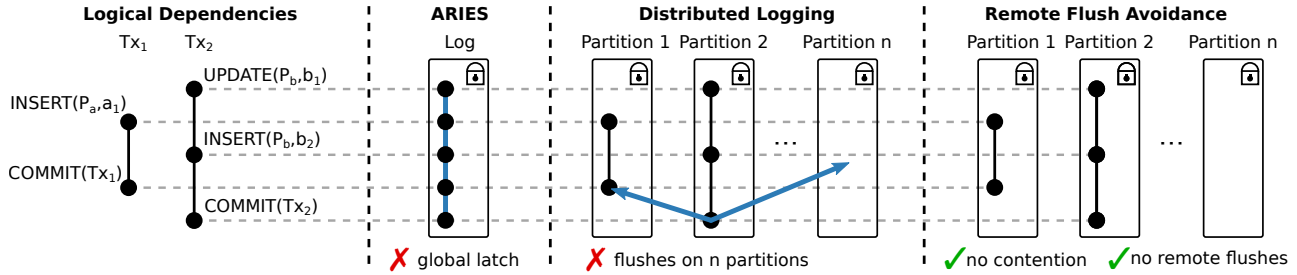

Figure 3: Two independent transactions and the synchronizing operations in different logging strategies.

3.2 Low-Latency Commit Through RFA

Splitting the log into multiple partitions and pinning each transaction to one of those partitions allows worker threads to create log entries without synchronization. However, to guarantee consistency *all* log partitions need to be flushed up to the transaction's current GSN as depicted for "Distributed Logging" in Figure 3.

Let us give an example of why this is necessary: suppose transaction Txn_1 deletes a tuple on page P_A and records this change with $GSN 12$ in log L_1, but does not yet commit. Then transaction Txn_2 also inserts a tuple on P_A and records this in its own log L_2. The GSN protocol mandates that this change will be recorded with a higher GSN, e.g., $GSN 13$. When Txn_2 commits, it has to ensure that all prior changes on this page have been persisted in the log. It therefore needs to flush all other log partitions up to $GSN 13$.

Profiling our system showed that these remote log flushes lead to significantly reduced scalability (see experiment in Section 4.1). Wang and Johnson's approach [52] works around this issue by using *passive group commit*, where transactions only flush their own log and are put in a commit queue. A group commit thread periodically checks the minimum flushed GSN of all logs, and sets those transactions to committed for which all necessary log records are persisted. Group commit is a good solution for workloads with a lot of concurrent transactions without low-latency requirements.

We propose a new technique called *Remote Flush Avoidance (RFA)*, which enables high throughput and scalability while still providing low-latency single transaction commits. The motivation for RFA comes from the observation that, for most transactions, there is neither logical conflict nor do they modify the same set of physical pages. A commit dependency is therefore not necessary. Consider the example shown in Figure 1b, in which Txn_1 pessimistically flushes the log of Txn_2 since $GSN_5 < GSN_8$, despite the fact that the two transactions modify different pages (P_A and P_B). The basic distributed logging approach fails to catch the independence of Txn_1 and Txn_2 as it projects the dependency graph onto a single time axis. In other words, it linearizes the partial ordering of GSNs back to a total ordering, and thereby sacrifices some of its scalability advantages. When two independent transactions create log records concurrently, one of them will unavoidably have the smaller GSN, and thus the other transaction needs to flush it.

RFA adds a few lightweight steps to the GSN protocol that enable it to skip most remote partition flushes:

(1) For each page, we remember L_{last}, the log that contains the record for the most recent modification of this page. This information does not need to be persisted, and can be stored in the buffer frame for that page.
(2) When a transaction starts, it determines $GSN_{flushed}$, the maximum GSN up to which all logs have been flushed at that point. Any log record created afterwards is guaranteed to have a higher GSN (see Section 2.4).
(3) Lastly, each transaction maintains a *needsRemoteFlush* flag, which is initially set to `false`.

Whenever a transaction accesses a page (either for read or write), it checks if the page GSN is below $GSN_{flushed}$. If that is the case, then all previous log records of that page have already been flushed and the page access can proceed. Otherwise, if the page GSN is above $GSN_{flushed}$, the algorithm further checks if the committing transaction's log is the same as L_{last}. If that is the case, then page access can also proceed, because it means that the last modification is from the same transaction and thus will be flushed anyway when the current transaction commits. Only if both checks fail, then the *needsRemoteFlush* flag is set to `true`, which causes all logs to be flushed when the transaction commits. In a nutshell, remote flushes are avoided whenever the log records of a transaction depend only on (1) guaranteed flushed changes or on (2) changes from its own log.

Figure 3 summarizes the benefits of RFA. Given two independent transactions, ARIES requires synchronized access for each log record, while standard distributed logging frequently requires synchronization on commit. In order to avoid this, one could explicitly track dependencies, but this is prohibitively expensive as noted previously [52]. Therefore, RFA detects such dependencies without expensive bookkeeping data structures, allowing the two transactions shown in Figure 3 to commit without any synchronization.

RFA and group commit are orthogonal optimizations that can be combined or used individually, yielding four designs.

When persistent memory is available, we argue that RFA without group commit is the best approach as it enables low-latency commits. However, even in the absence of persistent memory, RFA helps to reduce transaction latency when combined with group commit. Without RFA, group commit requires a transaction to wait on all logs to be persisted before its commit is acknowledged. With RFA, transactions with their *needsRemoteFlush* flag set to *false* can commit as soon as their own log is persisted. In this design, in addition to a global group commit queue, each log has its own queue for transactions that do not require a remote flush.

3.3 Challenges of Checkpointing

In a broad sense, a checkpoint is any technique that aims to bring the persistent state of a database up to date in order to reduce recovery time and recycle log space. The classic paper by Härder and Reuter [19] provides an abstract classification of checkpointing techniques and how they relate to the granularity of logging. Most in-memory systems write the entire contents of a database in a transaction-consistent manner when taking a checkpoint. To achieve that, these systems require shadow copies of individual tuples in main memory, usually in combination with multiversion concurrency control [49], or in some cases by duplicating the entire database [10]. These systems are usually slower while a checkpoint is being taken, because individual records must be versioned and transaction activity must halt—or at least be coordinated in multiple phases [45]—to establish a point of consistency. Given these restrictions, it is advisable to take checkpoints sparingly in such systems. However, this translates to longer recovery times, which is further worsened by the fact that log replay is substantially slower (see Section 5).

For the reasons given above, this paper focuses on page-based checkpointing to complement our page-based buffer management and logging schemes. Robust page-based checkpointing is a difficult challenge in practice because it requires finding an acceptable trade-off between opposing goals. In order to bound recovery time to an acceptable level, a system must flush dirty pages at a rate that is compatible with the rate of incoming transactions. If the flush rate is too high, this results in a waste of I/O resources and a needless increase in write amplification. On the other hand, if the flush rate is too low, the log size cannot be bounded, which can result in service outage when the log device fills up, and long recovery time in case of a failure. A low rate may also cause the buffer manager to be saturated with dirty pages, which can cause a drop in transaction throughput if insertions or queries cannot allocate memory. Even if an appropriate flush rate is found, the checkpointer must be smart about which pages to flush, since some dirty pages are worth flushing at a lower rate than others (e.g., hot pages with frequent updates). These issues have been largely ignored in recent research, which has focused mainly on alternative architectures that propagate changes at record (rather than page) granularity.

One good example of a checkpointing implementation in traditional WAL systems is PostgreSQL [48]. It uses two independent processes to write dirty pages: a *checkpointer* and a *background writer*. The former is responsible for bounding the size of the log and thus recovery time; it is triggered when the log reaches a certain size, or by a timeout (5 minutes by default). The checkpointer flushes *every dirty page* in the buffer pool (i.e., a *direct checkpoint* [19]). As such, the LSN of the checkpoint log record also establishes the starting point of redo recovery. One problem with this approach is that it incurs periodic bursts of high I/O activity and contention on the buffer manager data structures, leading to performance dips. To solve this problem, the background writer is triggered at fixed time intervals and writes a certain number of dirty pages calculated from configuration parameters. Besides PostgreSQL, other approaches argue that checkpointing should be done "periodically" [38], "every few seconds" [10], or "roughly 10 seconds after the previous checkpoint completed" [55]. The inherent problem of such time-based policies is that they fail to adapt to changes in the workload and shift the problem of setting reasonable values to the database administrator.

3.4 Continuous Checkpointing

Since the principal goal of checkpointing is to bound recovery time, which mainly depends on the amount of log that needs to be processed, it makes sense to couple the checkpointing policy to the amount of log written. Suppose we want to limit the size of the WAL to 20 GB. To achieve this, we need to write out certain (but not all) dirty pages. To be precise, all pages in the buffer pool that contain unflushed changes captured in log records that are more than 20 GB away from the tail of the log need to be written out before the log can be truncated to 20 GB. At the same time, we do not want the checkpointer to cause high CPU overhead and heavy write bursts by constantly scanning the whole buffer pool just to be able to truncate a few MB of WAL.

To solve this problem, we propose a new technique called *continuous checkpointing*. Instead of taking a checkpoint over the whole buffer pool at once, we incrementally checkpoint fractions of it, which avoids bursts in disk writes and leads to a steadier performance. After each step, a small fraction of the WAL can be truncated.

To achieve this, we logically partition the buffer pool into S *buffer shards*. In each checkpointer invocation, which we call a *checkpoint increment*, the checkpointer picks a shard in a round-robin fashion, and writes out all dirty pages in it. An increment is triggered whenever $1/S$ of the configured log limit is moved to the second stage of the log (see Figure 2). For example, with a WAL limit of 20 GB, a buffer pool of

```
// Maintain the persisted GSN for each shard
GSN maxChkptedInShard[num_shards]; // Initially zero
unsigned current_incr = 0;
// Triggered by writing a certain amount of WAL
void checkpoint_increment() {
  min_current_gsn = logs.getMinCurrentGsn();
  shard = current_incr % num_shards;
  pages = bufferPool.getPagesInShard(shard);
  foreach (page in pages) {
    if (page.isDirty()) write(page);
  }
  maxChkptedInShard[shard] = min_current_gsn;
  // Get the minimum GSN up to which all
  // shards are checkpointed
  chkpted_gsn = min_val(maxChkptedInShard);
  min_tx_gsn = txManager.getMinActiveTxGsn();
  logs.prune(min(chkpted_gsn, min_tx_gsn));
  ++current_incr;
}
```

Figure 4: Pseudo code for continuous checkpointing.

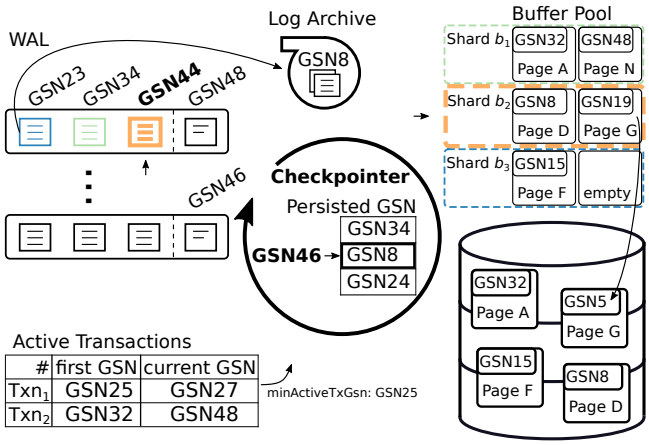

Figure 5: Checkpointing example. The frequency of checkpoint increments is coupled to WAL volume.

50 GB, and $S = 10$ shards, for every 2 GB of generated WAL, a checkpoint increment of a 5 GB shard is triggered.

The checkpointer maintains a table that, for each buffer shard, stores the GSN up to which all changes have been persisted in that shard. Before each increment, the checkpointer remembers the smallest current GSN among all logs, GSN_{min}. Any log records created afterwards are guaranteed to have a GSN larger than GSN_{min}. When the checkpointer finishes a shard, it updates that shard's table entry with GSN_{min}. The minimum value among all entries of this table is then used to determine the point up to which the log can be archived (as long as the GSN of the oldest active transaction also permits it—see Section 3.6). The algorithm is shown in Figure 4.

Figure 5 shows an example with a buffer pool of six pages and $S = 3$ shards. The most recent checkpoint increment has been on shard b_1, at which point GSN_{min} was $GSN34$ (table in the center). Since buffer shard b_2 is the next one to be checkpointed, it consequently has the oldest persisted GSN and limits log truncation to records with $< GSN8$. The staging of the orange WAL chunk now triggers the checkpoint increment of b_2. The only dirty page in b_2 is page G, which is written out. Next, the checkpointer updates the table entry for shard b_2 with $GSN46$, since that is the minimum current GSN across all logs when it started the increment. This makes $GSN24$ the new overall minimum log record needed for recovery. Taking into account the minimum GSN of all active transactions, which is $GSN25$, it can now move the blue log chunk to the log archive, since that chunk only contains log records up to $GSN23$.

This technique is the key to continuously making checkpoint progress (i.e., increase the maximum persisted GSN) without time-based configuration settings while still avoiding excessive write amplification. In steady state, for each new fraction of WAL staged, the oldest one will be pruned and the WAL volume will stay stable at its configured size. Furthermore, this design eliminates both the manual tuning knob as well as any write bursts that would occur in full checkpoints, and is effective in keeping the recovery time bounded. By increasing the number of shards to, e.g., $S = 128$, the smoothness of the writes can be increased and the absolute deviation from the configured WAL limit reduced.

3.5 Page Provisioning

Besides checkpointing, there is another reason to persist dirty pages in a buffer-managed system: when pages have to be read from disk, but the buffer pool is already full, then some other pages need to be evicted. If these pages are dirty, they need to be written out to disk first. Database systems usually have a dedicated background thread for this (e.g., the background writer of PostgreSQL [48] mentioned earlier), so worker threads do not have to issue synchronous writes before reading a page. However, research so far has looked at the dirty page writer mostly in isolation.

We take a holistic view on the different page flows that happen inside the storage manager and how they interact with each other. LeanStore partitions pages into a hot and a cool area for page eviction [31]. Pages in the hot area can be directly referenced by pointers, i.e., they are swizzled, while cool pages must be looked up with their page ID. As Figure 6 shows, we consider a storage manager to be a closed dynamic system in which buffer pages flow between different states. Assuming the workload is in steady state, worker threads continuously swap buffer pages into the swizzled (hot) state at a certain rate, e.g., 1,000 pages per second. They do so by allocating a new page, accessing a page on disk, or (this is specific to our storage manager design) accessing a page that is currently in the unswizzled (cool) state.

Since LeanStore achieves several hundred thousand transactions per second, even a moderate amount of additional

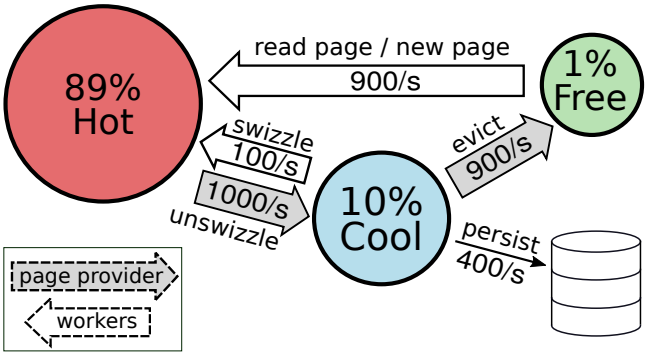

Figure 6: The division of LeanStore's buffer pool into hot (swizzled), cool (unswizzled) and free pages. Actions in the system cause pages to transition between the states. In steady state, worker threads request exactly as many free pages as the page provider supplies.

work, such as consulting the page replacement strategy, can lead to a significant drop in throughput. Consequently, we exclude all other transitions (shaded arrows in Figure 6) from the critical path, so worker threads can focus solely on transaction processing. This means that in steady state, a certain number of pages is in the "free" state. These pages are kept in a separate list, allowing workers to request new pages with very little overhead, in particular without consulting global eviction data structures. The percentage of pages in the free list can be low, e.g., 1% as shown in Figure 6. It must only suffice to bridge the gap of short bursts of page requests.

For all support tasks, we introduce a dedicated *page provider* thread that does all of the extra housekeeping work:

(1) It unswizzles pages and puts them into the cool area, which is organized as a FIFO queue. From there they get re-swizzled if worker threads access them again in a timely manner. Access to the queue is sped up with a hash table lookup.
(2) Pages from the older end of the queue are evicted and put into the free list.
(3) Before a dirty page can be evicted, it is persisted to disk first.

A classical dirty page writer, as employed by PostgreSQL for example, is only concerned with the third point. Our page provisioner, in contrast, is responsible for keeping the whole buffer pool in an equilibrium. Worker threads wake up the page provisioner whenever the buffer pool deviates from the desired composition. It then runs as many rounds as necessary until the target sizes of the FIFO queue and the free list are restored. In each round, it first unswizzles a fixed number of pages, e.g., 256, and inserts them at the front of the queue. Second, it iterates over the queue from the back and unlinks clean pages, buffering them in a local list. Once that list reaches a certain size, it is appended to the free list with a single latch acquisition. Dirty pages, on the other hand, are simply marked as *writeBack* and copied into a local writeback buffer. Pages in the *writeBack* state can still be modified. They can even transition between the swizzled and unswizzled state. However, they cannot be evicted, as otherwise it cannot be guaranteed that a subsequent read from disk will return the latest version. When the writeback buffer fills up, it is written out in one go and the disk cache is flushed. Note that the page provisioner gives up all latches during the I/O operation, so afterwards it starts again at the back of the queue. There it finds the (hopefully still clean) pages that it just wrote out and can move them to the free list. Thus, clean pages in the FIFO queue are accessed exactly once, while dirty pages are normally accessed twice.

One might wonder whether it would be better to introduce separate threads for unswizzling, page writing, and page eviction. But all three actions depend on each other: Pages are written out at the latest possible moment before they get evicted, thus avoiding write amplification from writing out pages multiple times. Also, in order to evict the correct pages, they need to migrate through the FIFO queue first. Thus, the queue must always stay appropriately filled. Doing all of this in the same thread proves to be a simple and robust design, where pages move continuously, and no single action can outrun the others, which would introduce imbalance.

3.6 Transaction Abort

Traditional systems implement the *steal* paradigm, i.e., uncommitted changes may be written back to the persisted database, whereas for in-memory systems only a *no-steal* policy makes sense. In terms of CPU instructions, there is not much difference whether the undo images are stored in the log or in a dedicated in-memory rollback segment. For high-performance storage engines, in principle both approaches would be possible. With current DRAM sizes, one could consider a system that pins all pages with uncommitted changes in the buffer pool. The advantages of such a system are the omission of undo recovery and therefore less WAL volume. On the other hand, a system that implements *steal* can handle larger-than-RAM write transactions. Also, tracking all pages with uncommitted changes incurs overhead and can hold back the checkpointer from forcing out the necessary pages for truncating the log, which violates our design goals.

For these reasons, we use a *steal* architecture and write before-images in the WAL for each data modification. Transaction aborts are done logically, i.e., the regular access path is used to execute the reverse operation (e.g., delete for insert and vice versa), including writing new log records. Aborts are efficient, as all log records for a transaction can be found in a single log. After all changes are undone, an end-of-transaction record is inserted into the log. The final log flush required for successful transactions can be omitted.

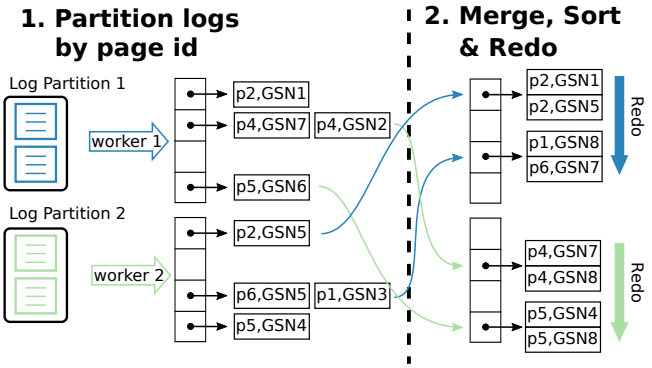

Figure 7: Overview of the first two recovery phases. The final undo phase is not shown.

Most transaction aborts are fast because recent log records still reside in persistent memory. Therefore, only old or large transactions need to read from SSD. We found that the additional space required in the WAL to write undo-images is often small (e.g., only 20%, or 2230 Byte vs. 1850 Byte on average for a TPC-C transaction). The reason is that insert log records do not require a before-image at all, update records only contain the before and after image of the changed attributes, and deletes are infrequent in common workloads.

3.7 Recovery

Our checkpointing strategy ensures a tight bound on the log volume that needs to be recovered after a failure. However, the concrete value should be set in proportion to the buffer pool size, so that the ratio between the amount of WAL written and the corresponding checkpoint work stays constant. Thus, for large buffer pools, e.g., several hundred gigabytes, the amount of work during recovery is not trivial.

Similar to ARIES, recovery has three phases: log analysis, redo, and undo. In order to still be able to recover in an adequate time frame, we parallelize each phase. As Figure 7 shows, in the first phase, each thread scans all chunks of a log partition, including those residing in persistent memory, to separate winner and loser transactions. For winner transactions, it partitions the log records by page ID into a thread-local redo table. Log records for loser transactions are put into a separate undo list.

In the redo phase, each thread is assigned a range of page IDs. For each range, it merges the log records from all redo tables from the previous phase. Afterwards, it sorts them by $(pageId, GSN)$, so that all log records for a page end up next to each other in GSN order. Then, redo recovery can proceed page by page. All log records for a page, including those of loser transactions, are applied at this step. Merge, sort, and redo can happen interleaved and without synchronization, since threads operate on distinct ranges.

Finally, in the undo phase, all changes from loser transactions are reverted by iterating over the log records in the undo bucket. Each change is undone logically as explained for transaction aborts in Section 3.6.

3.8 Implementation Details

This section discusses implementation details and performance optimizations that we implemented in our system.

Persistent memory can be accessed either in *memory mode*, or in *app direct mode*. We use app direct mode, which allows us to create an `ext4` file system. A log chunk is allocated in persistent memory by creating a file of the desired size, which is then memory mapped with the `dax` option, which bypasses OS buffers and allows direct access with cache line granularity. We avoid thrashing the CPU cache when writing log records by using non-temporal store instructions, which are conveniently accessible by the Intel Persistent Memory Development Kit (PMDK) via `pmem_memcpy`.

Finding the last valid log record in persistent memory is not trivial, as log records can still leave the CPU in arbitrary order. Thus, one could consider storing the end offset within each chunk explicitly. However, this would require an expensive cache flush per log entry, as the data needs to be flushed before offset to ensure correct ordering. Additionally, as Renen et al. [51] observed, repeated updates of the same persistent memory location are unreasonably expensive. They propose to use the `popcnt` instruction to compute a lightweight checksum of each record. During recovery, these checksums are used to determine the last fully written log entry. We implemented this approach in our system, and found that it does not introduce any measurable overhead. Partially staged log chunks are not problematic, as in that case the intact copy in persistent memory gets precedence.

The **Writeback Buffer** mentioned in Section 3.5 accumulates pages locally before writing them out all at once. In order to saturate the bandwidth of high-speed NVMe SSDs, we use the asynchronous I/O API of the Linux Kernel. Additionally, we open the database file with `O_DIRECT` to avoid OS caching effects and reduce CPU overhead. Thus, the only cache that remains to be flushed is on the device itself. We use the `fdatasync` call for that instead of the typical `fsync`, which provides enough guarantees but allows for a slightly faster implementation. We issue 1024 page writes at a time to keep the device queue filled. After a batch of pages has been written out and flushed, we update the $GSN_{persisted}$ field in their buffer frames to the GSN of the copy of the page. It is important that the GSNs are only updated after the writes have been flushed, as otherwise the checkpointer might prune the log too early. The writeback buffer is used by both the checkpointer and the page provisioner.

Log Compression reduces the amount of data that has to be transferred to both persistent memory and SSD. We do not employ a general-purpose compression scheme such as LZ4, as it would incur a non-negligible CPU overhead,

which would outweigh its benefits. We instead reuse information from former log entries of the same category. For example, we maintain the ID of the last inserted page, and the transaction that did so. We apply this optimization for insert and update entries, as they make up the majority of log volume in common workloads. If we create another insert entry, we check if it can reuse that information. Note that we reset the cached IDs on chunk boundaries, so that log chunks stay independent, and can be processed in parallel. Furthermore, update records only contain the before and after image of changed attributes together with a bitmask with the modified attributes set to 1. We observe that these two techniques already reduce log volume by 30% in TPC-C.

For performance, it is vital that the **checkpointer thread** does not latch all pages in a shard at the same time, which can be tens of thousands of pages for realistic buffer sizes. Similar to the page provisioner (see Section 3.5), it only latches each page briefly to mark it as *writeBack* and copy it into a local writeback buffer before proceeding to the next page. The copy is necessary anyway in our design, because the in-memory representation of a page contains swizzled pointers, which must not get serialized to persisted storage. So before a page gets written out, all pointers in the copy are replaced by their corresponding page IDs [17]. This design minimizes the impact of checkpointing, as only one page is latched at a time, and no page latches are held during I/O. To keep up with the worker threads on machines with many cores, our implementation has multiple checkpointer threads, which can work on distinct checkpoint increments in parallel. This does not affect correctness, as the minimum GSN in the table determines how far the log can be truncated. For the results reported in Section 4, we use two checkpointer threads.

4 EVALUATION

We implemented our logging and recovery approach in the high-performance LeanStore storage engine, which is written in C++17. LeanStore uses pointer swizzling and partitions the buffer pool into a hot/cold area. For the benchmarks, we use TPC-C with all five transaction types. The benchmark driver is directly linked into the engine, so the results are unimpeded by network overhead. Relations and indexes are stored in B$^+$-trees with a page size of 16 KB. Because our system does not yet implement full transaction isolation, we effectively run all experiments in read uncommitted mode.

All experiments are run on a system with an Intel Xeon Gold 6212U CPU, which has 24 cores and 48 Hyper-Threads. The system is equipped with 192 GB DRAM and 768 GB Intel Optane DC Persistent Memory. The operating system (Ubuntu 19.10) is installed on one SSD, while the database file resides on another PCIe-attached NVMe SSD. We put the log on the persistent memory device, which is able to absorb the almost 2 GB/s of log volume we generate.

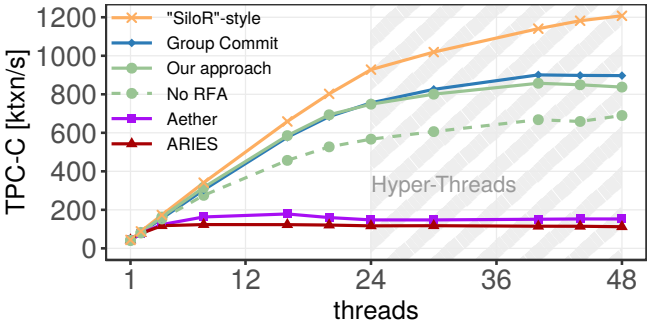

Figure 8: TPC-C (500 warehouses, 100 GB buffer pool).

Each thread gets allocated 5 WAL chunks of 20 MB each, which has shown to be sufficient for the background staging thread to offload the WAL. We fix the clock frequency of the CPU at 2.6 GHz, which is the highest value it can sustain when all threads are active. This reduces noise and leads to more comparable benchmark results.

The main goal of this section is to evaluate the different logging approaches for a high-performance storage engine. Since Silo is an in-memory system (and its recovery implementation SiloR is not publicly available), we decided to implement all competitors in our own system. This allows us to quantify the exact impact of logging and checkpointing as all other system parameters, i.e., the benchmark code, the storage engine, and the data structures, stay unchanged. The ARIES-style configuration uses a single global log. We also implemented a state-of-the-art variant that contains the optimizations proposed by Aether [22], namely *consolidation arrays*, *flush pipelining* (group commit), and *decoupled buffer fill*. The SiloR-style approach uses per-thread logs and employs epoch-based group commit. It is the only competitor which allocates log chunks in DRAM first. It does not write GSNs, page IDs or undo images into the WAL. Thus, it does not support out-of-memory workloads that persist pages with uncommitted changes to disk. All competitors take full checkpoints each time the log exceeds its limit, during which SiloR persists the whole database, while ARIES/Aether writes out dirty pages.

4.1 Scalability

In our first experiment, we compare transaction throughput and scalability of our approach against Aether, ARIES, and SiloR-style logging when the workload fits into memory. We also compare the scalability of RFA against passive group commit [52] and a variant that flushes all logs synchronously upon commit (labeled "No RFA"). For that, we run TPC-C with 500 warehouses (which amounts to 50 GB of initial data) and a memory budget of 100 GB. Figure 8 shows that our approach is able to scale very well. It achieves 41k TPC-C

transactions per second with one thread and scales to 750k txn/s with 24 threads. Beyond that, it further benefits from hyperthreading and reaches its peak performance at 857k txn/s with 40 threads. The only competitor that performs significantly better is the in-memory group commit "SiloR" approach, which achieves 43k txn/s single threaded and 1.2M txn/s with 48 threads. We investigated why its scalability is better than the other competitors, even in comparison with the non-blocking passive group commit variant. We found that the persistent memory device is the culprit. By putting the first stage of the WAL into DRAM, and leaving all other parameters the same, our approach achieves 960k txn/s instead of 857k txn/s with 40 threads (scalability 22.6x instead of 20.9x), indicating that our first generation persistent memory platform is oversaturated when many threads write into persistent memory concurrently.

ARIES-style logging is limited by the global log, so it reaches its peak performance of 123k txn/s already with 4 threads. Aether performs slightly better and achieves up to 180k txn/s with 16 threads, but eventually also runs into contention. This is consistent with previous reported results [52], and amplified in our system as our base performance is an order of magnitude higher, resulting in the single log becoming the bottleneck much sooner. The dashed line in Figure 8 shows the throughput of our system when RFA is disabled. It performs better than ARIES and Aether as it does not have to synchronize threads on the creation of log entries. However, because it still has to synchronize all per-thread logs upon *each* transaction commit, it starts to scale worse than RFA or group commit beyond 8 threads, reaching its peak of 690k txn/s with 48 threads.

As the scalability results show, RFA manages to avoid most remote log flushes, resulting in a throughput close to that of a group commit protocol which does not flush other logs at all. However, depending on the contention in the workload, remote log flushes may be inevitable. To demonstrate this effect, we varied the number of warehouses (which indirectly controls transaction interference in TPC-C) with 40 worker threads and measured the percentage of transactions requiring a remote flush:

	w=1	w=5	w=10	w=50	w=100	w=500
rem. flushes	92.0%	91.8%	90.8%	15.5%	14.7%	8.1%
perf [txn/s]	596k	707k	721k	929k	861k	857k

As these results indicate, the more logical independence the workload has, the more likely it becomes that remote flushes can be avoided.

4.2 Log Volume and Checkpointing

Among the main features of our approach are the continuous checkpointing algorithm and its capability to bound

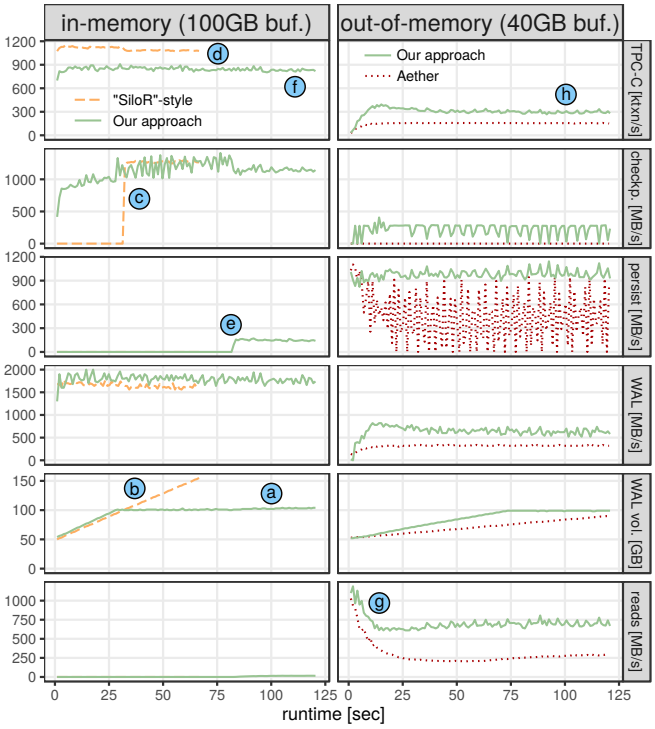

Figure 9: TPC-C performance over time (500 warehouses, 100GB WAL limit, 40 worker threads).

the log volume. To understand the behavior of our approach, we run TPC-C for two minutes with 40 worker threads and 500 warehouses (50 GB of initial data). The left column of Figure 9 shows that with a buffer pool of 100 GB, our system achieves a sustained transaction rate of 850k txn/s while constantly writing out 1.7 GB/s of WAL, and over 1 GB/s of pages necessary for checkpointing. After 32 seconds the WAL volume reaches its configured limit and stays there (a), which validates that our continuous checkpointing approach can keep the WAL bounded. In contrast, the checkpointer in the SiloR approach cannot keep up with the worker threads (b) resulting in a growing WAL volume. It has to write out the entire database (c)—which grows quickly—for each checkpoint. After 66 seconds, when the memory budget of 100GB is exhausted, the SiloR system stops processing (d) because it runs out of memory. In our system, in contrast, the page provisioner starts supporting the checkpointer with writing out pages at 140 MB/s to free up buffer space (e). Since this is done in the background, and the page replacement strategy is identifying correct eviction candidates, processing continues with no observable performance drop (f).

The right column of Figure 9 shows system behavior when the buffer pool does not even fit the initial data set. For that, we set the buffer pool size to 40 GB, while keeping the number of warehouses at 500. At the beginning of the benchmark, both Aether and our approach need a few seconds to load

Table 1: TPC-C transaction rates and CPU instructions (500 warehouses, 40 worker threads). We enable all necessary logging components step-by-step.

#	Component	txn/s	instr. per txn
1	no logging	1.439k	47k
2	+ create WAL records	1.110k	59k
3	+ stage WAL records	934k	61k
4	+ remote log flushes	666k	78k
5	+ RFA	854k	65k
6	+ checkpointing	850k	66k

the working set into RAM (g), starting at over 1 GB/s. Afterwards, both are able to keep a stable performance level, but, even in this out-of-memory scenario, Aether's performance is still a factor of two lower due to log synchronization. This underlines that the classical disk-based database architecture is unsuited for fast SSDs. In steady state, our system writes 2 GB/s (checkpointing: 280 MB/s, page persisting to free up buffer space: 1 GB/s, WAL writing: 650 MB/s) and reads 700 MB/s (k), while sustaining 300k txn/s (h) and bounding the WAL to 100 GB.

4.3 Dissecting the Features

In the next experiment, we dissect the performance impact of the different logging components using TPC-C (40 threads, 500 warehouses, 100 GB buffer pool). Table 1 shows that the basic storage manager, without any logging, checkpointing or page cleaning, is able to achieve 1.4M TPC-C transactions per second (#1). Next, the worker threads allocate, write, and checksum the log records in persistent memory (#2). This incurs the highest increase in CPU instructions, from 47k to 59k per transaction, and decreases performance to 1.1M txn/s. Staging the log records to SSD (#3) is almost purely an I/O operation. Thus, the instructions per transaction do not increase much, but performance drops slightly to 934k txn/s since staging is done by a background thread that has to access the log. However, once transactions flush all logs during commit (#4), performance drops to 666k txn/s. By introducing RFA (#5), performance increases to 854k txn/s. Finally, the fully functioning system including checkpointing (#6) achieves 850k txn/s. Thus, the total amount of work for the recovery component is 19k instructions per transaction in our approach.

4.4 RFA and Contention

To analyze the impact of RFA in more detail, we also run a YCSB experiment with a fixed table size of 500M records, each consisting of an 8 Byte key and a 64 Byte value. The resulting database has a size of 36 GB. We run the benchmark with 24 threads, each executing a transaction consisting of a single-tuple update. This stresses log synchronization to

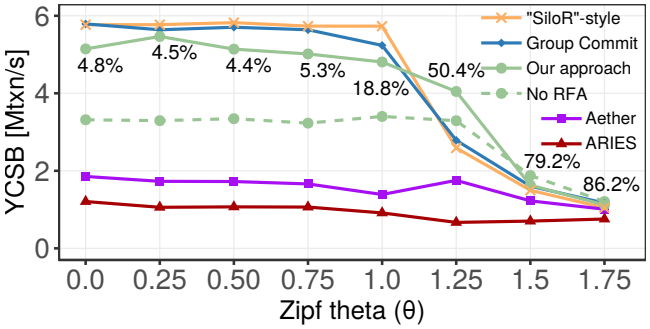

Figure 10: YCSB-style workload (100% single-tuple updates). Our approach is annotated with the percentage of remote flushes.

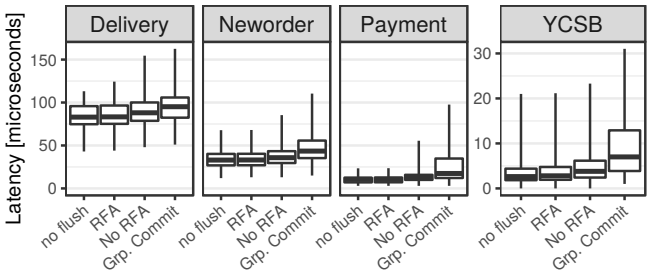

Figure 11: TPC-C & YCSB transaction latencies for different commit flush strategies. RFA is close to the optimum, while group commit has higher latencies.

the maximum, as much of the work consists of creating log records. We vary the Zipf theta, resulting in different levels of skew. As Figure 10 shows, with $\theta = 0$ (i.e., a uniform distribution) RFA is most effective and only 4.8% of all transactions need to synchronize their commit with other logs. Theta values up to 0.75 do not increase remote flushes significantly. An even higher θ increases the number of remote flushes, but performance stays competitive compared to group commit and Silo-R. With $\theta = 1.25$, contention in the workload starts to dominate, and with even higher values all competitors converge to a similar performance.

4.5 Transaction Latencies

In this experiment, we compare optimal transaction latencies, i.e., those that could be achieved by not flushing log records in other logs at all, to those with and without RFA, and group commit. Transactions enter the system based on a Poisson process where the time between two transactions is exponentially distributed.

Figure 11 shows the execution latencies for the three write transactions in TPC-C for a transaction rate of 100k txn/s. Without flushing other logs, a single delivery transaction has a median execution time of 83 microseconds. With RFA, the median does not increase, and the 99th percentile only slightly increases from 113 μs to 124 μs. The median when

Figure 12: WiredTiger performance for TPC-C over time (40GB buffer pool, 20 threads). We incrementally disable checkpointing and logging.

always flushing all other logs is 88 μs, while with group commit it rises to 95 μs. Neworder and payment transactions show a similar behavior, while the general trend is that for short transactions, the added latency of group commit is more significant. In YCSB, the median latency for group commit is 7 μs, which is more than double the latency of RFA, which has a median latency of 2.8 μs.

For comparison, the epoch-based group commit system SiloR has transaction latencies in the order of milliseconds, which is three orders of magnitude higher than in our system.

4.6 Recovery

As the experiment in Section 4.2 shows, our approach is effective in bounding the log volume that needs to be processed during recovery. With a WAL limit of 100 GB, we measure the different phases of recovery time. Using 40 threads, total recovery time is 38 seconds, of which 14 seconds are spent on partitioning the log chunks by page ID, and the remaining 24 seconds are used for merging, sorting and applying the log records (undo time is negligible). This corresponds to 2.6 GB of WAL recovered per second. Within one second after recovery finished, TPC-C transaction performance approaches the value it had before the crash because recovery implicitly pre-warmed the buffer cache.

For comparison, Zheng et al. [55] report 0.93 GB/s for processing the recovery log in the state-of-the-art SiloR system using 32 cores. However, because their checkpointing approach fails to put a tight upper bound on the log, it grows up to 180 GB in their TPC-C experiment. This leads to a much higher recovery time of 211 seconds (which includes loading time for the last checkpoint). Therefore, despite frequent checkpointing, their recovery time is more than five times higher than in our system.

4.7 System Comparison

To the best of our knowledge, WiredTiger is one of the most efficient open-source storage engines, which we found to be faster than LevelDB, RocksDB, BerkeleyDB, and PostgreSQL on TPC-C. Furthermore, WiredTiger implements a typical recovery approach close to textbook ARIES and allows one to disable logging and checkpointing individually. We integrated WiredTiger 2.9.3 into our TPC-C benchmark driver. To make for a fair comparison with our persistent-memory-based approach, we deactivate `fsync` in WiredTiger.

As in previous experiments, we run TPC-C with 500 warehouses. Figure 12 shows the performance of WiredTiger over time, once with a buffer pool of 100 GB where the workload fits in memory, and once with 40 GB where the workload exceeds it. One apparent observation is that WiredTiger has very high variability, especially when checkpoints are enabled. This shows that checkpoints cause major drops in performance, and are thus an issue in existing systems. The performance of our system, in contrast, is not only much more stable, but also one order of magnitude faster in the out-of-memory case.

5 RELATED WORK

This section reviews alternative designs for logging, checkpointing, and recovery as well as complementary techniques.

In-memory database systems make various trade-offs in order to improve forward-processing performance at the expense of recovery performance. Broadly speaking, there are two classes of systems that represent these kinds of trade-offs, namely *physical* and *logical* logging. In the previous sections, we reviewed Silo as a representative for physical logging systems; in this section, we discuss the latter class.

Command logging [36] is a logical logging approach in which every transaction must run as a stored procedure, and it is logged as a single log record containing the procedure ID and its arguments. This approach has three major limitations: first, checkpoints must not only scan and write the entire database, but they must also be transaction-consistent, which incurs additional overhead. Second, recovery requires re-executing transactions serially, leading to prohibitively long outages in case of system failures. Adaptive logging [54] is a hybrid approach that attempts to improve recovery time of command logging by tracking dependencies among transactions and selectively producing physical log records that eliminate such dependencies, thus speeding up log replay by permitting a higher degree of parallelism.

Hekaton [13] has a unique approach among in-memory systems. It takes checkpoints by sorting and merging log files (with value logging), essentially behaving like a log-structured merge tree [40]. As such, it permits incremental checkpoints, i.e., writing only recent modifications rather than the entire database. FineLine [46] employs a similar concept but it relies on physiological logging instead of value logging. FOEDUS [28] is a persistent-memory system that propagates changes from volatile images of pages into their persistent representation by means of log replay, which can

be seen as a single-level merge tree. In contrast, the staging approach used in HANA [47]—with its *L1* and *L2 deltas*—can be seen as a two-level merge tree. However, in HANA changes are propagated at the granularity of whole columns instead of pages. Such log-structured approaches have two fundamental caveats: first, merging log files potentially increases I/O load and write amplification; second, they forbid evicting uncommitted changes to disk (i.e., they are *no-steal*).

The first designs proposed for in-memory databases had the limitation that the entire dataset had to fit into main memory. Given the advent of fast SSDs and persistent memory, this limitation can be a "deal breaker", especially from the economic perspective [4]. Thus, some techniques such as Anti-Caching [12] and Siberia [2, 34] were proposed to allow an in-memory database to grow larger than main memory without relying on a page-based buffer manager. Instead, they identify less-frequently used tuples and migrate them to separate data structures on persistent storage. To support reads, additional lookups in such data structures are required, which can be optimized with techniques like Bloom filtering. A buffer manager with pointer swizzling performs better when data is larger than main memory, since it does not require such lookups or tuple-level tracking of accesses.

Persistent memory provides byte addressability and low latency. It has been explored in multiple logging design proposals [11, 14, 21, 27]. These approaches aim to reduce commit latency while still relying on a centralized log at some point, thus overlooking the scalability issue. Our design not only addresses scalability but also considers the wider context of both logging and checkpointing.

Some designs for persistent-memory database systems also consider the radical approach of eliminating redo logging altogether (i.e., *force*) by performing out-of-place updates directly on the persisted database [8, 15, 41, 43, 44], in some cases delegating all storage to specialized search structures [5, 42]. However, this fundamental design tradeoff presents some major limitations. First, since durability requires random persistent writes, commit latency is unavoidably higher than in logging approaches, as one flush operation is required per tuple modified. Second, data must not be updated in-place, which results in lower performance for some workloads [6, 44], and introduces garbage collection overhead as well as issues with indexing and space management. Third, since a system without redo logging cannot support media recovery, replication to a stand-by secondary server is required [53] to handle device failures.

Instant recovery [16] is a family of algorithms that allow new transactions to execute immediately after a system or even media failure, including localized single-page corruptions. A complementary technique is *constant-time recovery* [3], which exploits multi-version storage to allow instantaneous undo of arbitrarily long user transactions. Since both techniques rely on physiological logging, they can be combined with our approach to achieve efficient logging and checkpointing as well as faster recovery.

ERMIA [26] is a physiological logging system that improves the scalability of centralized logging by leveraging multi-version storage, which is used both for concurrency control as well as recovery purposes. Our approach, in contrast, is orthogonal to concurrency control and also works with single-version storage.

The general problem of **synchronizing transaction activity in the presence of multiple logs** has also been explored by Bernstein and Das [9], who propose a mechanism to parallelize the validation phase of an optimistic concurrency control algorithm (OCC) by partitioning the database. The key difference is that our approach considers a distributed log partitioned by transaction and not by database contents (i.e., by page), and that the focus here is not on OCC validation but on guaranteeing durability regardless of the employed concurrency control protocol.

Other concurrency-control techniques of both the pessimistic [1] and optimistic [35] kind require tracking dependencies among transactions to enforce commit-time ordering, but such tracking happens at the granularity of individual records. The RFA technique proposed in this paper, on the other hand, only tracks dependencies at the page level, and once detected, a dependency neither has any influence on the serializability order or the decision to commit or abort, nor does it cause any delay in the logical execution of other transactions—it just triggers the flush of different log buffers.

6 SUMMARY

Neither ARIES nor in-memory logging approaches work well for high-performance storage managers. We present a design that covers all aspects of logging, checkpointing, and page provisioning. Using continuous checkpointing, we eliminate spikes in transaction performance and I/O. Remote Flush Avoidance enables low-latency transaction commit and makes our approach scalable on multi-core CPUs. Overall, our implementation inside LeanStore achieves not only the rich feature set and robustness of a classical disk-based database architecture, but also the performance and scalability of state-of-the-art in-memory systems.

REFERENCES

[1] Divyakant Agrawal, Amr El Abbadi, Richard Jeffers, and Lijing Lin. 1995. Ordered Shared Locks for Real-Time Databases. *VLDB J.* 4, 1 (1995).

[2] Karolina Alexiou, Donald Kossmann, and Per-Åke Larson. 2013. Adaptive Range Filters for Cold Data: Avoiding Trips to Siberia. *PVLDB* 6, 14 (2013).

[3] Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and

Girish Mittur Venkataramanappa. 2019. Constant Time Recovery in Azure SQL Database. *PVLDB* 12, 12 (2019).

[4] Raja Appuswamy, Renata Borovica-Gajic, Goetz Graefe, and Anastasia Ailamaki. 2017. The Five-minute Rule Thirty Years Later and its Impact on the Storage Hierarchy. In *ADMS*.

[5] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. 2018. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *PVLDB* 11, 5 (2018).

[6] Joy Arulraj, Andrew Pavlo, and Subramanya Dulloor. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *SIGMOD*.

[7] Joy Arulraj, Andy Pavlo, and Krishna Teja Malladi. 2019. Multi-Tier Buffer Management and Storage System Design for Non-Volatile Memory. *CoRR* (2019).

[8] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *PVLDB* 10, 4 (2016).

[9] Philip A. Bernstein and Sudipto Das. 2015. Scaling Optimistic Concurrency Control by Approximately Partitioning the Certifier and Log. *IEEE Data Eng. Bull.* 38, 1 (2015).

[10] Tuan Cao, Marcos Antonio Vaz Salles, Benjamin Sowell, Yao Yue, Alan J. Demers, Johannes Gehrke, and Walker M. White. 2011. Fast checkpoint recovery algorithms for frequently consistent applications. In *SIGMOD*.

[11] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. 2013. From ARIES to MARS: transaction support for next-generation, solid-state drives. In *SOSP*.

[12] 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).

[13] 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*.

[14] Ru Fang, Hui-I Hsiao, Bin He, C. Mohan, and Yun Wang. 2011. High performance database logging using storage class memory.

[15] Shen Gao, Jianliang Xu, Theo Härder, Bingsheng He, Byron Choi, and Haibo Hu. 2015. PCMLogging: Optimizing Transaction Logging and Recovery Performance with PCM. *IEEE Trans. Knowl. Data Eng.* 27, 12 (2015).

[16] Goetz Graefe, Wey Guy, and Caetano Sauer. 2016. *Instant Recovery with Write-Ahead Logging: Page Repair, System Restart, Media Restore, and System Failover, Second Edition.* Morgan & Claypool Publishers. https://doi.org/10.2200/S00710ED2V01Y201603DTM044

[17] 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).

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

[19] Theo Härder and Andreas Reuter. 1983. Principles of Transaction-Oriented Database Recovery. *ACM Comput. Surv.* 15, 4 (1983).

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

[21] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. *PVLDB* 8, 4 (2014).

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

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

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

[25] 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*.

[26] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *SIGMOD*.

[27] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *ASPLOS*.

[28] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *SIGMOD*.

[29] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (1978).

[30] Per-Åke Larson, Mike Zwilling, and Kevin Farlee. 2013. The Hekaton Memory-Optimized OLTP Engine. *IEEE Data Eng. Bull.* 36, 2 (2013).

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

[32] 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).

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

[34] Justin J. Levandoski, Per-Åke Larson, and Radu Stoica. 2013. Identifying hot and cold data in main-memory databases. In *ICDE*.

[35] David B. Lomet, Alan Fekete, Rui Wang, and Peter Ward. 2012. Multi-version Concurrency via Timestamp Range Conflict Management. In *ICDE*.

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

[37] C. Mohan. 1999. Repeating History Beyond ARIES. In *VLDB*.

[38] 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).

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

[40] Patrick E. O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth J. O'Neil. 1996. The Log-Structured Merge-Tree (LSM-Tree). *Acta Inf.* 33, 4 (1996).

[41] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: a hybrid SCM-DRAM storage engine for fast data recovery. In *DaMoN*.

[42] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*.

[43] Ismail Oukid, Wolfgang Lehner, Thomas Kissinger, Thomas Willhalm, and Peter Bumbulis. 2015. Instant Recovery for Main Memory Databases. In *CIDR*.

[44] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. 2013. Storage Management in the NVRAM Era. *PVLDB* 7, 2 (2013).

[45] Kun Ren, Thaddeus Diamond, Daniel J. Abadi, and Alexander Thomson. 2016. Low-Overhead Asynchronous Checkpointing in Main-Memory Database Systems. In *SIGMOD*.

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

[47] Vishal Sikka, Franz Färber, Wolfgang Lehner, Sang Kyun Cha, Thomas Peh, and Christof Bornhövd. 2012. Efficient transaction processing in SAP HANA database: the end of a column store myth. In *SIGMOD*.

[48] Hironobu Suzuki. 2016. *The Internals of PostgreSQL. Self-published.* http://www.interdb.jp/pg/

[49] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SIGOPS*.

[50] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *SIGMOD*.

[51] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent Memory I/O Primitives. In *DaMoN*.

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

[53] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2017. Query Fresh: Log Shipping on Steroids. *PVLDB* 11, 4 (2017).

[54] Chang Yao, Divyakant Agrawal, Gang Chen, Beng Chin Ooi, and Sai Wu. 2016. Adaptive Logging: Optimizing Logging and Recovery Costs in Distributed In-memory Databases. In *SIGMOD*.

[55] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *OSDI*.