Moving on From Group Commit: Autonomous Commit Enables High Throughput and Low Latency on NVMe SSDs

To appear at SIGMOD 2025

Lam-Duy Nguyen
lamduy.nguyen@tum.de
Technische Universität München
Germany

Adnan Alhomssi*
adnan.alhomssi@relational.ai
RelationalAI, Inc.
Germany

Tobias Ziegler
t.ziegler@tum.de
Technische Universität München
Germany

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

Abstract

Achieving both high throughput and low commit latency has long been a difficult challenge for Database Management Systems (DBMSs). As we show in this paper, existing commit processing protocols fail to fully leverage modern NVMe SSDs to deliver both high throughput and low-latency durable commits. We therefore propose *autonomous commit*, the first commit protocol that fully utilizes modern NVMe SSDs to achieve both objectives. Our approach exploits the high parallelism and low write latency of SSDs, enabling workers to explicitly write logs in smaller batches, thereby minimizing the impact of logging I/O on commit latency. Additionally, by parallelizing the acknowledgment procedure, where the DBMS iterates through a set of transactions to inspect their commit state, we mitigate excessive delays resulting from single-threaded commit operations in high-throughput workloads. Our experimental results show that autonomous commit achieves exceptional scalability and low-latency durable commits across a wide range of workloads.

CCS Concepts

• **Information systems → Database transaction processing**.

Keywords

Database Management Systems, Transaction Processing, Logging, Group Commit, Commit Processing, Latency

ACM Reference Format:
Lam-Duy Nguyen, Adnan Alhomssi, Tobias Ziegler, and Viktor Leis. 2025. Moving on From Group Commit: Autonomous Commit Enables High Throughput and Low Latency on NVMe SSDs: To appear at SIGMOD 2025. In *Proceedings of ACM on Management of Data 3 (SIGMOD '25)*. ACM, New York, NY, USA, 15 pages. https://doi.org/XXXXXXX.XXXXXXX

*Work done while at Friedrich-Alexander-Universität Erlangen-Nürnberg

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '25, June 22–27, 2025, Berlin, Germany
© 2025 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/XXXXXXX.XXXXXXX

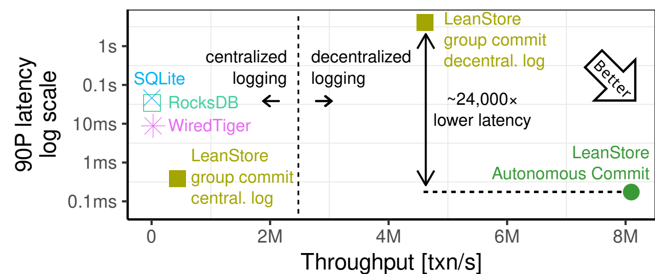


Figure 1: No existing DBMS or commit protocol can achieve both low-latency commits and high throughput. Our proposed solution, autonomous commit, accomplishes both.

1 Introduction

Logging on disk. Online Transaction Processing systems seek to optimize two critical but often conflicting goals: low latency and high throughput [14, 15, 36, 63]. Achieving both is challenging because of the durability requirements for write-ahead logging. Historically, logging operations were slow because magnetic disks could only provide millisecond-scale write latencies. To improve throughput on these slow devices, DBMSs implement group commit, which batches multiple transactions to amortize log writing costs [2, 3, 17, 21, 22, 30, 34], but this leads to higher commit latency [30, 55].

Logging on persistent memory. Persistent memory emerged as a promising solution, with prior research showing that achieving both objectives is possible with this new class of storage devices [28, 31, 54]. Unfortunately, Intel's Optane product has been discontinued [16, 26] and NVDIMMs are not widely available [32], which leaves the challenge unresolved.

SSDs to the rescue? The emergence of modern NVMe SSDs presents another opportunity to achieve both low latency and high throughput. Unlike magnetic disks, NVMe SSDs offer write latencies in the tens of microseconds and support high levels of parallelism. Moreover, they are cheap and widely available. This raises the question of whether existing commit protocols can effectively leverage low-latency NVMe SSDs to achieve both high throughput and low-latency durable commits?

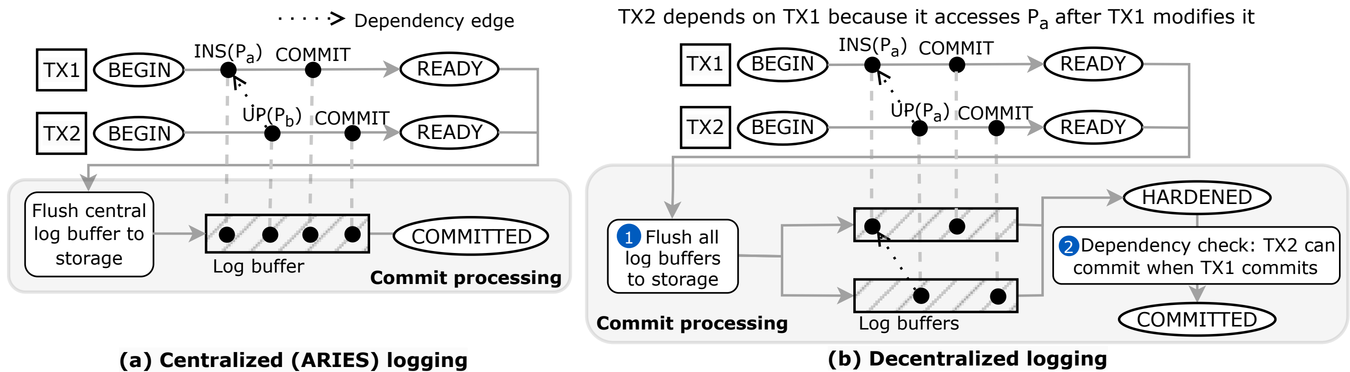


Figure 2: State machine of a transaction, in centralized (ARIES) logging (a) vs. decentralized logging (b).

Existing solutions. Figure 1 shows that existing DBMSs – specifically their commit processing protocols – fail to simultaneously achieve low-latency commits and high throughput with modern NVMe SSDs. The figure shows the results of a YCSB benchmark (details in Section 6.1) contrasting the two main approaches: (1) Group commit with centralized logging [46] (i.e., a shared log buffer between all threads) as used by WiredTiger, SQLite, and RocksDB, and (2) Group commit with decentralized logging (i.e., each thread has a designated log buffer) [28, 54, 57] as used by LeanStore [28]. As the figure shows, the centralized approach can achieve low 90th-percentile latency but offers limited throughput due to contention on the shared log buffer. Conversely, the decentralized design provides good scalability at the cost of high commit latency, with 90p latency exceeding four seconds.

Bottleneck. Our analysis (cf., Section 2.3) of LeanStore's decentralized group commit mechanism revealed that actual transaction execution accounts for only 0.001% of the total commit latency, with most of the latency stemming from the commit processing subsystem. Log flushing contributes just 8% of the total latency, while commit acknowledgment, particularly dependency checking, is a more significant bottleneck. The root cause of these inefficiencies is in the *unithreaded* design of group commit.

Autonomous commit. This work addresses these bottlenecks by designing a commit processing algorithm, *autonomous commit*, from the ground up, leveraging modern NVMe SSD characteristics. Modern SSDs support small, random, parallel, and durable writes with double-digit microsecond latency, enabling a fully concurrent and parallel commit processing protocol. Instead of batching large writes as in traditional group commit, autonomous commit parallelizes small log flushes across workers. We then decouple and parallelize the dependency checking from the flushing phase. We further introduce a series of optimizations to minimize latency and ensure robustness in realistic workloads.

Discussion. Our design incorporates a series of complementary techniques that, while simple, are remarkably effective, reducing latency by up to four orders of magnitude without sacrificing throughput. We attribute this surprisingly notable improvement to two factors. First, much of the recent research on transaction processing has focused primarily on throughput, often overlooking the critical role of latency. Second, modern SSDs offer extremely low-latency, durable writes, a capability that has received limited attention in the research literature. By carefully optimizing for commodity data center hardware, autonomous commit achieves sub-millisecond commit latency, even for high-throughput workloads.

2 Commit Processing and Its Bottlenecks

In this section, we provide an overview of commit processing, followed by an exploration of the characteristics of modern SSDs. We then analyze why state-of-the-art commit processing techniques lead to high commit latencies on SSDs.

2.1 Commit Processing

Commit processing is fundamental to ensuring transaction durability. In this section, we provide the necessary background to understand the stages of commit processing and its challenges.

Centralized logging: ARIES. ARIES has long been the standard logging and recovery mechanism for disk-based DBMSs. ARIES relies on *centralized logging*, meaning all transactions share a single log buffer where write-ahead log (WAL) entries are stored until they are flushed to storage.

ARIES transaction lifecycle. As illustrated in Figure 2(a), the life cycle of a transaction in a DBMS using ARIES-style logging follows several stages [46]. The transaction begins with a `BEGIN` command, followed by the execution of user queries and corresponding DBMS tasks, such as isolation validation. After execution, the transaction transitions to the `READY` state[1], indicating that it can be committed. At this point, its metadata is placed in a centralized buffer called the *pre-committed queue*. Transactions that have dependencies must be committed in a specific order to ensure consistency. Dependencies occur when one transaction reads or writes data that another transaction has accessed. For instance, in Figure 2(a), consider transaction `TX1`, which inserts a new tuple into page `Pa` but has not yet been completed. If transaction `TX2` subsequently updates another tuple in page `Pa`, a dependency is created between `TX2` and `TX1`.

How ARIES manages dependencies. To manage such dependencies, ARIES ensures that transactions are placed in the *pre-committed queue* in their execution order. This guarantees that dependencies are resolved correctly and that transactions can be

[1] `READY` is *ready to commit*, which is conceptually equivalent to `TRX_QUE_COMMITTING` in MySQL/InnoDB [2].

correctly recovered after a failure. A transaction is marked as COMMITTED and can only be acknowledged to the user once its log records have been durably written to storage.

Group commit with ARIES: Scalability challenges. In ARIES, the log buffer is flushed to storage with every transaction commit, resulting in a large number of I/O operations. To reduce this I/O overhead, DBMSs commonly use *group commit*, which defers I/O writes to enhance transaction throughput. Group commit works by delaying the writes of multiple READY transactions for a short period, then writing all of their logs to storage in a single I/O operation and acknowledging these transactions as COMMITTED together. This method reduces I/O costs by amortizing them across several transactions. However, with modern multi-core CPUs, the centralized log buffer of ARIES becomes a scalability bottleneck, ultimately limiting throughput [27, 34, 54, 57].

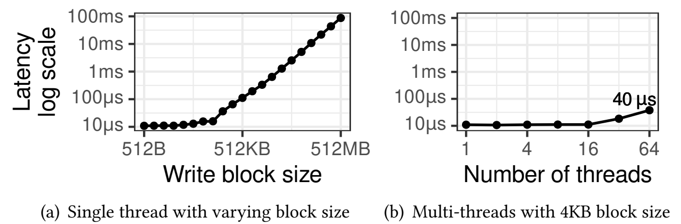
Decentralized logging: Scalability with a trade-off. To address the scalability issues of centralized logging, many works propose decentralized logging, where each worker is assigned its *own* log buffer [27, 28, 37, 54, 57]. This approach improves scalability by eliminating contention on a shared log buffer. At the same time, ensuring durability and consistency across all workers becomes more complex, making commit processing more difficult [28, 54].

Decentralized logging: Complex commit processing. In decentralized logging, commit processing becomes more complex due to transaction dependencies. Log records from dependent transactions may end up in separate buffers, as illustrated in Figure 2(b). Consider the same example as before: transaction TX1, which inserts a new tuple to page Pa but has not yet completed its execution. After that, transaction TX2 updates another tuple in page Pa, which creates a dependency edge from TX2 to TX1. Now, if the DBMS flushes the log buffer containing TX2 before TX1 is committed, TX2 will be marked as HARDENED (i.e., its log records are durable). However, because TX2 might have accessed data modified by TX1, it cannot be fully committed until TX1 is committed. This complicates commit processing in decentralized logging, as it requires ensuring the durability of log entries and checking that all dependencies are resolved before allowing transactions to commit.

Decentralized logging: Commit conditions. Figure 2 shows the transaction lifecycle in a decentralized logging-based DBMS. The commit processing subsystem must ensure that all dependencies are resolved when it transitions transactions through the stages from READY, to HARDENED (by writing the logs to storage) and finally to COMMITTED. We can formalize the *commit conditions* for a transaction in decentralized logging as follows:

1. **Log durability** (READY ⇒ HARDENED): Ensure all its modifications, i.e., log entries, are durable in the storage.
2. **Dependency checking** (HARDENED ⇒ COMMITTED): All dependencies must also be committed [17, 28, 54, 57].

Group commit with decentralized logging. Similar to centralized logging, group commit is used in decentralized logging to amortize I/O costs. However, unlike in centralized logging, where the group committer only has to flush the log buffer, the group committer in decentralized logging must ensure that the two commit conditions are satisfied. Group commit in decentralized logging operates as a sequence of *commit rounds*, each consisting of two stages: *log flush* and *commit acknowledgment*. At the start of every commit round, the DBMS iterates through all buffers to collect their READY transactions and associated logs. Then, it triggers *log flush*: writing all those log entries to the storage – usually with asynchronous I/O [9, 25, 33] – then moving the state of those transactions to HARDENED. Finally, the system calls *commit acknowledgment* to determine the new set of safe transactions, i.e., satisfying the above *commit conditions*, and acknowledges them as COMMITTED.



(a) Single thread with varying block size (b) Multi-threads with 4KB block size

Figure 3: Average random-write latency of an enterprise NVMe SSD, measured using microbenchmarks.

This two-stage process allows decentralized logging systems to overcome the scalability bottleneck of centralized logging, ensuring that transactions are durably and correctly committed. As a result, many modern DBMSs have adopted this model for commit processing. Unfortunately, while this approach delivers high performance, as shown in Figure 1, it suffers from high commit latency on modern SSDs. To understand the reasons behind this, we next explore the characteristics of modern NVMe SSDs and analyze the underlying bottlenecks.

2.2 Characteristics of NVMe SSDs

So far, we have focused on commit processing as it is traditionally implemented in disk-based systems. This section explores the key differences between conventional magnetic disks and modern NVMe SSDs. Understanding these differences will help explain why, despite the microsecond-level write capabilities of SSDs, commit latencies remain high in systems that use them.

Smaller writes (4-16KB). Enterprise NVMe SSDs offer low-latency writes [24, 25, 29, 36]. For example, as Figure 3(a) shows, our Kioxia SSD achieves random write latencies as low as $11\mu s$ for a 4KB write unit. Latency stays below $11\mu s$ for write units up to 64KB, but can reach up to 87ms for a 512MB write. Therefore, smaller writes (e.g., 4KB or 16KB) should be used to design a commit processing system capable of microsecond-level commit latencies.

Parallel writes. While smaller write sizes help minimize latency, high throughput can be maintained by leveraging the parallelism of NVMe SSDs. Unlike magnetic disks, which are limited by a single seek arm, SSDs can handle multiple writes concurrently. Furthermore, these writes do not need to be sequential; they can be random. As shown in Figure 3(b), NVMe SSDs maintain low-latency durable writes even with concurrent random writes. For example, write latency barely increases with a 4 KB write unit and 16 concurrent threads. Thus, we can balance high throughput and low latency by utilizing parallel small random writes.

Fsync for free. Up to this point, we have focused on write latencies. But historically, the most significant source of latency was making writes durable using `fsync()` [24]. This is because the physical flash

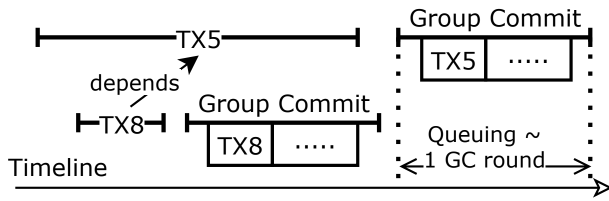


Figure 4: Queuing time is significant in group commit. TX8 can only be committed when its dependency, TX5, is committed.

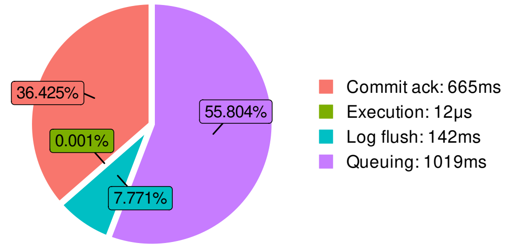


Figure 5: Latency breakdown of the group commit with decentralized logging in a multi-threaded YCSB benchmark.

write latency is several milliseconds. Consumer SSDs mitigate this by buffering writes in internal DRAM and finally writes to the non-volatile NAND flash chips when `fsync()` is triggered. Enterprise-grade NVMe SSDs, in contrast, eliminate this latency by ensuring that the DRAM buffer is durable, thanks to the built-in capacitors that supply backup power during system failures [7, 12, 23, 36]. As a result, `fsync()` is free with enterprise SSDs. Note that this assumes a setup bypassing the file system by using block devices, which improves performance significantly as proven in many existing works [24, 25, 40, 47, 48, 50, 58]. If a file system or the OS page cache is involved, `fsync()` is still required.

2.3 Bottleneck Analysis

As just discussed, commit processing on modern SSDs is most efficient when it performs small, parallel writes to fully utilize their capabilities. However, the fundamental principle of group commit – batching as many transactions as possible in a single thread to amortize I/O costs — directly conflicts with this approach.

Latency breakdown setup. To better understand the sources of latency, we measured the four key stages a transaction must pass through to be committed: (1) transaction processing, (2) queuing in the pre-committed queue, (3) log flushing, and (4) dependency checking for commit acknowledgment. For the measurements, we used the open-source storage engine LeanStore, which is optimized for high throughput on NVMe SSDs and implements a highly optimized version of decentralized logging with group commit [9, 28]. In terms of throughput, this implementation is capable of executing millions of transactions per second.

Breakdown results. Figure 5 shows the breakdown for the YCSB benchmark with an average transaction latency of 1.8s. Surprisingly, the primary bottleneck is not log flushing to SSD but rather *queuing*, where transactions wait to be committed. Log flushing is only the third largest contributor to latency, while commit acknowledgment (i.e., dependency checking) is the second major factor. Although log flushing accounts for only about 8% of the total latency, it still occurs on a millisecond scale. To achieve commit latencies in the microsecond range, all such bottlenecks must be addressed. The root cause of these issues is the *unithreaded* nature of group commit, causing three problems.

Problem 1: Flushing delays due to I/O spikes. In group commit, log flushing is handled by a single thread, which leads to large batches of logs being written at once, causing *I/O spikes* that hurt commit latency. This issue is particularly pronounced in highly parallel, high-throughput workloads, where workers fill their log buffers quickly. The single thread must iterate over tens or even hundreds of log buffers (workers), accumulating a large volume of logs in every commit round. In our experiments, the system completes 24 *commit rounds* over ten seconds, each causing an I/O spike of approximately 300MB in log writes. As Figure 3 shows, writing 300MB to SSD takes 45ms in a microbenchmark, significantly increasing the latency for all transactions in that round.

Problem 2: Single-threaded commit acknowledgment. In group commit, a single thread is responsible for iterating through all pre-committed transactions (of all workers) to verify the commit state of their dependencies, and then announces the new set of `COMMITTED` transactions. This worsens latency in decentralized logging systems with high throughput. This is evident in the latency breakdown in Figure 5 where *commit acknowledgment* accounts for 36% of the latency for the experiment presented in the introduction.

Problem 3: Queuing. In decentralized logging, the *queuing* overhead – the waiting time between transaction completion and being committed – spans at least one group commit round. This problem arises from the fact that group commit uses a *coupling* design: every commit round must trigger both *log flushing* and *commit acknowledgment*. We demonstrate this issue in Figure 4 with two transactions, `TX5` and `TX8`, where `TX8` depends on `TX5`. If the current round contains `TX8` while `TX5` has not yet completed, the DBMS still can not commit `TX8` until `TX5` is hardened. Consequently, the queuing time of `TX8` is at least one group commit round for `TX5`.

This explains why the *queuing* overhead is roughly the total time for both *log flush* and *commit acknowledgment*, as shown in Figure 5. In practice, dependency graphs are often more complex, leading to queuing overhead spanning *multiple* commit rounds instead of just one in the example.

3 Autonomous Commit

To address the inefficiencies outlined in the previous section, we introduce *autonomous commit*, a novel commit processing protocol consisting of two techniques: *autonomous log flush* and *autonomous commit acknowledgment*. In Section 3.1, we discuss *autonomous log flush*, which leverages the low-latency parallel writes of modern NVMe SSDs to resolve the *I/O spikes*. We describe the second technique, *autonomous commit acknowledgment*, in Section 3.2, and how this technique parallelizes the expensive commit acknowledgment to alleviate its impact on latency. In Section 3.3, we describe how the design of autonomous commit reduces *queuing* delays. Lastly, in Section 3.4 and Section 3.5, we discuss techniques to optimize further and improve the robustness of autonomous commit.

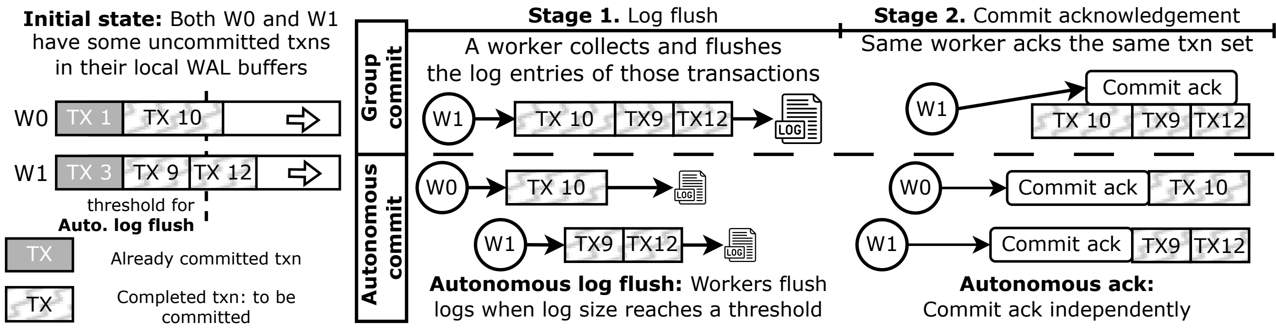


Figure 6: Traditional group commit vs. our proposed approach, autonomous commit.

3.1 Autonomous Log Flush

All workers should flush logs. As discussed in Section 2.3, group commit suffers from flushing delays caused by large log batches. During each group commit round, a significant log volume is written to storage all at once, adding hundreds of milliseconds to the commit latency. Instead, we propose that workers should *autonomously* flush their log buffers to the storage, hence the name *autonomous log flush*, to exploit the parallel low-latency writes of SSDs. Thereby, we can avoid the latency of large SSD writes.

How much should workers write? The next question is how frequently we write logs to storage. Most high-performance DBMSs bypass the OS page cache by using `O_DIRECT` [10, 25, 28, 44, 49, 56], which requires all writes to align with the storage block size (typically 4KB). Consequently, individual flushes for small transactions (i.e., tens or hundreds of bytes) would cause considerable write amplification and reduce system throughput significantly.

Proposal: Workers harden small batches. Instead, we propose buffering log entries in each worker's local log buffer until reaching a configurable threshold, termed the *log flush unit.* When a `COMMIT` log entry is generated, the worker evaluates if the size of the unflushed log entries meets this threshold. If so, the worker triggers *log flush* for its log buffer and then announces the hardened transaction set to other workers, preparing for the *commit acknowledgment* stage.

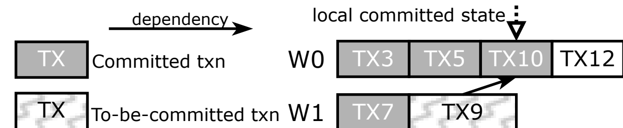
Figure 6 contrasts group commit (upper) with autonomous commit (lower). In both approaches, workers begin with their own buffers containing uncommitted transactions. In Stage 1, group commit relies on a single thread to collect and flush all uncommitted entries to the SSD. In contrast, in autonomous commit, each worker handles its own log flushing independently. For instance, the log record of TX10 is larger than TX9 and TX12 combined and reaches the log flush threshold on its own. The log flush threshold should align with the storage block size and remain relatively small (i.e., < 64KB) to ensure low-latency, durable writes in multi-threaded environments, as discussed in Section 2.2.

3.2 Autonomous Commit Acknowledgment

Autonomous log flush parallelizes log flushing and limits the log write size, exploiting modern SSDs' characteristics. This significantly reduces the latency of the log flushing stage, bringing it down from multiple milliseconds to tens of microseconds (cf. Figure 3). However, as Figure 5 illustrates, commit acknowledgment is a much bigger contributor to overall latency. We next describe how to parallelize commit acknowledgment.

Autonomous acknowledgment. In existing commit protocols, only *one* thread is responsible for running the commit acknowledgment process at any given time. Given that a decentralized logging system can complete millions of transactions per second, this single-threaded approach becomes a bottleneck. We propose to not only flush logs but also acknowledge transactions autonomously, i.e., every worker only triggers the *commit acknowledgment* on its own pre-committed transaction queue. The right part of Figure 6 (Stage 2) illustrates this approach and contrasts it with group commit.

Synchronization. Every acknowledgment round requires *synchronizing* the *committed state* of all workers. As the following figure illustrates, these states indicate up to which point all prior transactions local to each worker have been committed [28, 54, 57]:



The committed state of worker `W0` is `TX10`, meaning that all transactions prior to `TX10` belonging to `W0`, i.e., `TX3` and `TX5`, are also committed. Assume that in `W1`, `TX9` has just been hardened (its associated logs are flushed to the storage), and it depends on `TX10`. To determine that the dependencies of `TX9` are all committed, `W1` must retrieve the *committed state* of all other workers. In this example, after knowing that `TX10` belongs to `W0` and `W0` has committed all transactions until `TX10`, `W1` knows that `TX9`'s dependencies are all committed, hence acknowledges `TX9`.

Optimization: Acknowledgment in small groups. On large servers, the state synchronization may involve gathering the *committed state* of hundreds of workers, each protected by a separate latch. Assuming that there are 100 workers and worker `W` wants to acknowledge its pre-committed queue of only *one* transaction. In this case, worker `W` has to acquire 100 latches to verify the dependencies of a single transaction, which is unnecessarily excessive. Multiple pre-committed queues should be processed per acknowledgment round to amortize this synchronization overhead.

One way to do that is to divide the workers into *acknowledgment groups* of configurable size, where any worker can acknowledge transactions of all workers belonging to the same group. Doing so allows a worker to trigger *autonomous acknowledgment* on all

the workers of the same group. This parallelizes the expensive acknowledgment while reducing state synchronization frequency.

3.3 Mitigating *Queuing* Overhead

In decentralized logging, a transaction may be durable but remains uncommitted if its dependencies have not yet been hardened. Group commit *couples* log flushing and commit acknowledgment, which can lead to *queuing* delays as dependencies wait to be committed throughout *multiple* rounds of both steps. This delay occurs because the single group committer must iterate through all worker log buffers, hardening them and checking dependencies. If a dependency is not yet fulfilled, the committer moves on to the next worker, leaving the transaction to wait until the next round. This explains why dependency-related queuing is the biggest overhead of group commit, as shown in Figure 5.

Decoupling I/O from acknowledgment. By splitting the commit operations into two parts – *autonomous log flush* and *autonomous acknowledgment* – and executing them independently, we eliminate log flushing delays from the *queuing* overhead. That is, workers can perform *autonomous commit acknowledgment* to verify the commit state of their remote dependencies (i.e., dependencies located in other workers) without needing an additional log flush or waiting for a group committer to perform the check on their behalf. In the example presented in Figure 4, the *log flush* for TX5 can be interleaved with the commit round containing TX8. As a result, the *queuing* time of TX8 no longer includes any delay from log flushing.

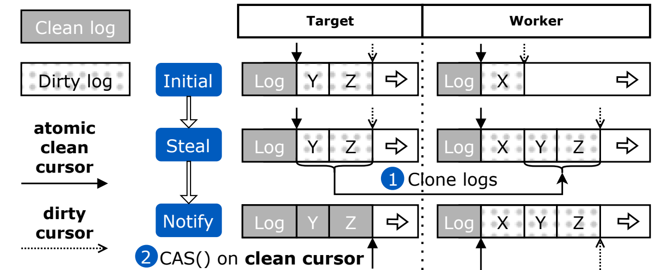
Tackling *queuing*: Frequent acknowledgment. Transactions with remote dependencies require workers to trigger commit acknowledgment *frequently* such that the worker owning those dependencies will perform *log flush* to harden them. However, performing commit acknowledgment too often (e.g., after every transaction completes) can waste CPU cycles. As the number of workers increases, the synchronization cost of retrieving the *committed states* rises, commit acknowledgment becomes more expensive and hence should be triggered less frequently. Whenever a worker completes a transaction, it triggers acknowledgment probabilistically based on the worker count, as the following code shows:

```
if (rand(log2(# workers)) == 0) CommitAcknowledge();
```

3.4 Optimization: Log Stealing

Autonomous log flush mitigates I/O spikes and reduces the write latency. However, transactions still face delays while waiting for logs to accumulate to the flush threshold, especially for small log records. For instance, with a 16 KB flush threshold and an average log size of 160 B, each flush hardens about 100 transactions. If a transaction takes $6\mu s$ to execute, the average delay before being hardened is $300\mu s$, which is excessive. Lowering the flush threshold (e.g., to 4 KB) reduces latency by hardening fewer transactions per flush but does not fully resolve the issue. Forcing flushes earlier can reduce delays but leads to I/O amplification if the batches are not full, consuming SSD IOPS needed for other tasks like checkpointing and buffer management.

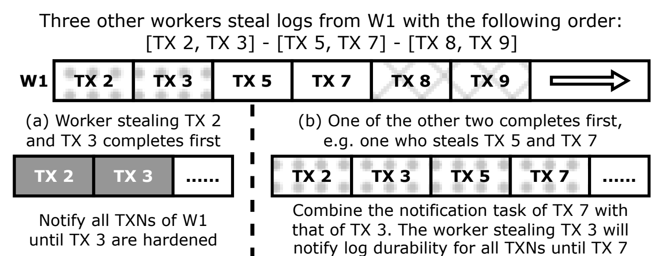
Solution: Log stealing. Instead, we introduce log stealing as a solution to mitigate excessive delays caused by workers accumulating log entries. We trigger stealing operations only at the end of a transaction's execution. Upon the end of a transaction's execution, the worker loops through all target workers and executes the log stealing operation as depicted in the following figure:



The figure illustrates the three main steps involved in the log stealing operation (from top to bottom), where the current worker (right) attempts to steal log entries from the target worker (left). Initially, the current worker only contains the log X, and it is trying to steal Y and Z entries from the target worker. First, (1) the worker calls memcpy() to clone the target's dirty logs Y and Z, i.e., from *atomic clean cursor* to *dirty cursor*, into its log buffer. Because the *dirty cursor* is coupled within the *committed state* of the targeted worker, which is already protected by a *latch* (cf., Section 3.2), it does not need to be atomic. Next, (2), the worker performs a compare_and_swap() (CAS() in the figure) which serves two purposes (1) it ensures that no other worker has stolen the same logs otherwise the CAS() would fail and (2) if it succeeds it notifies to other workers that it has just stolen those logs. Finally, the worker submits an I/O write request to persist the log buffer and notifies everyone that the log flush, which contains logs of transactions across several workers, is completed.

Who to steal logs? Unrestricted log stealing incurs inter-die communication, which is expensive on modern CPUs[2]. To mitigate this, we pin the workers to specific CPU cores and restrict them to only steal within their topology group. For example, an AMD EPYC processor with multiple CCDs, each has eight CPU cores sharing the same L3 cache. In this setup, we can use a group size of eight threads. Consequently, a worker only steals from others through the shared L3 cache, reducing expensive memory access.

Synchronization of concurrent stealing operations. Log stealing significantly reduces the queueing time. However, it may suffer from a flush race condition, i.e., multiple workers steal from the same worker and complete their log flush in a different order than the stealing sequence. One solution is to employ a *parking lot*-like technique [13, 43], as is illustrated in the following figure:



[2]Modern server CPUs usually comprise multiple dies [1, 5], in which inter-die communications are slower than intra-die communications.

There are two possible scenarios: either the first worker (e.g., stealing TX2 and TX3), completes its log flush first, or any of the other two completes first. In the first scenario, that worker notifies that all transactions until TX3 of worker one are durable. For the second scenario, the notification task for the stolen transactions is merged with the preceding transactions. As in the example, the worker who completes TX7 will merge the notification task for TX7 with that for TX3 and then not notify anything. After that, the worker who steals TX3 will now notify TX7, which inherently includes three older transactions. This approach ensures the correct transactional durability despite potential out-of-order log flushes.

3.5 Managing Latency Under Low Load

Problem. Combining all techniques described so far achieves low-latency commits in a steady state where transactions contiguously execute one after another. This differs from realistic workloads with frequent idle periods between transactions, e.g., where users pause to think before making new requests [8, 18, 51]. This inactivity presents a challenge for autonomous commit, as workers produce minimal or no logs during these times and thus cannot reach the flush threshold quickly. In these situations, transactions may remain volatile for an extended period because the log buffers take longer to fill up, delaying log flush operations and leading to higher commit latency.

Force commit. One solution is to force commit (i.e., trigger both *autonomous log flush* and *autonomous acknowledgment*) if the worker remains idle for a reasonable duration. That is, every worker predicts the next idle time; if there is enough free time, it will flush its log buffer and subsequently invoke commit acknowledgment. A simple strategy to predict the next idle time is to average a few last idle periods. Under specific conditions, e.g., when the average idle time exceeds the write I/O latency, the worker will write its dirty log buffer to the storage and call commit acknowledgment.

When to force commit? If we naively trigger force commit if $next_idle_time \leq force_commit_time$ (the average time taken for *force commit*), the worker may not trigger force commit at all. For instance, assuming that the average force commit time is $50us$ and there is a sequence of idle time: $[5\mu s, 10\mu s, 16\mu s, 12\mu s]$. Based on this sequence, one may predict that the $next_idle_time \leq 20\mu s$, which is smaller than the average force commit time. Therefore, the naive comparison approach may fail to trigger force commits, resulting in high commit latency. Instead, we use a probabilistic method for the trigger condition:

```
// Max idle time should be small, e.g., 50 microseconds
if (avg_idle_time ≥ rand(max_idle_time)) ForceCommit();
```

4 Lock-Free Transaction Queue

Up to this point, we have introduced the core concepts of autonomous commit. This protocol efficiently parallelizes log flushing and dependency checking by aggressively stealing and acknowledging transactions from other workers. While this approach enables low-latency durable commits, it heavily relies on a highly concurrent data structure to effectively share transaction objects[3] across multiple threads. Notably, other commit processing protocols, such as group commit, also require an efficient data structure to manage transaction objects across threads. To address this need, in this section, we present an optimized lock-free transaction queue that is a crucial building block for autonomous commit and also complements all other commit processing protocols.

```
1  struct Queue:
2    atomic<int> head // Read objects from buffer[head]
3    atomic<int> tail // Push new objects to buffer[tail]
4    u8 *buffer = malloc(CAPACITY)
5  void Queue::Enqueue(Transaction &txn):
6    // (1) Evaluate object size after being serialized
7    int size = txn.SerializedSize()
8    // (2) Wait until having enough space for new obj
9    int r_head = head.load()
10   int w_tail = tail.load()
11   while(ContiguousFreeBytes(r_head, w_tail) < size)
12     r_head = head.load()
13   // (3) Serialize the txn and enforce the write order
14   new (&buffer[w_tail]) SerializedTxn(txn)
15   tail.store(w_tail + size)
16 void Queue::BatchDequeue(u64 no_txns, lambda &commit_ack):
17   int ptr = head.load()
18   // (4) Commit the pre-committed queue
19   for (int tx_i : Range(0, no_txns))
20     SerializedTxn txn = &buffer_[ptr]
21     if (!commit_ack(txn)) { break }
22     ptr += txn->Size()
23   head.store(ptr)
```

Listing 1: Lock-free queue of serialized transactions.

Problems of latch-protected transaction queue. Previous approaches usually employ a data structure (e.g., transaction pool [2] or queue [9]) protected by a latch to manage transaction objects. In this approach, the workers must acquire a latch before allocating a new transaction object. However, another worker may be consuming transactions from the same queue, e.g., iterating through the queue to verify the transaction commit state, which also requires acquiring the same latch. This leads to potential contention as workers periodically halt progress, lowering the throughput and increasing latency significantly.

Problems of private transaction queue. Huang et al. [33] propose to deallocate transaction objects lazily by the workers. Therefore, the transaction set can be thread-local, eliminating the latch protecting the transaction set. However, this approach does not allow sharing transaction queues between threads, which are important in several scenarios, e.g., for asynchronous BLOB logging [48] or autonomous commit acknowledgment as described in Section 3.2.

Lock-free queue. We advocate for the use of a single-producer, single-consumer lock-free queue. This is because every worker maintains its transaction queue (i.e., single producer), and during every commit round, only one thread processes a queue (i.e., single

[3] Transaction objects, or their in-memory representations, store metadata about transactions, including start and commit timestamps, isolation levels, transaction states, and more. These objects serve as references to the corresponding transactions during DBMS operations and are typically processed in a first-in-first-out (FIFO) order.

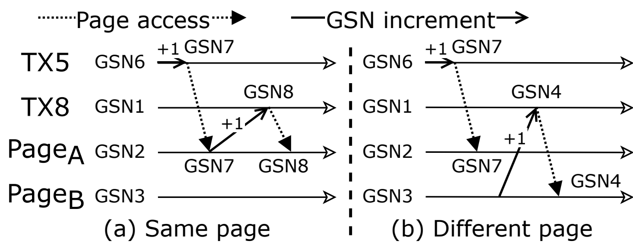


Figure 7: Dependency tracking with GSNs.

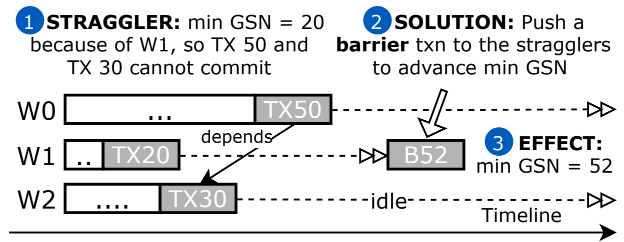


Figure 8: Straggler problem in GSN-based approaches.

consumer). We recommend the circular queue because it is simpler and more efficient than an unbounded queue. Additionally, because autonomous commit releases transactions aggressively, the number of queued transactions per worker is bounded.

Serialized transaction. With millions of transactions per second, transaction deallocation introduces significant overhead, worsening both throughput and commit latency. To address this, we *serialize* every transaction object into a variable-sized byte array. Consequently, multiple transaction objects can be placed contiguously within a large memory buffer, making deallocation very efficient: Large buffers can be released with `madvise(MADV_DONTNEED)`, while smaller ones are kept allocated for reuse.

Implementation. Listing 1 shows our queue implementation. For simplicity, we do not include the code for the queue operations. First, 1 we ensure that all serialized objects are aligned to the CPU cache line to prevent false sharing. Next, 2 the system waits until there is enough space to serialize a new object into the queue. After that, 3 we serialize the object to the queue buffer and notify other threads about the new enqueued transaction. This makes retrieving the pre-committed queues trivial by simply *load()* the number of queued items (omitted from Listing 1). With the number of queued transactions, 4 the system can iterate through the completed transaction set efficiently to verify their commit state. This batching dequeue mechanism also reduces cache coherence traffic [59], thus improving performance.

5 Robust Dependency Tracking for Stragglers

So far, we have glossed over how dependency tracking works, i.e., how we determine that one transaction depends on another.In practice, this plays a crucial role to system performance: Different tracking mechanisms (1) require varying amounts of information to be tracked [54, 57] and (2) dictate how workers assess the commit state of remote dependencies. These differences significantly influence the efficiency of the commit processing subsystem.

In this section, we will begin by reviewing existing dependency tracking strategies. We will explain why *Global Sequence Numbers* [54] stands out as the most practical design among all decentralized logging variants, followed by a discussion about the trade-offs for its advantages: the *straggler* problem. We will then introduce an incremental optimization that, as demonstrated in our experiments (cf., Section 6.2), effectively mitigates the *straggler* problem and thus enhances commit latency in workloads where stragglers may occur.

5.1 Existing Approaches

Precise causality-tracking is memory-intensive. Existing techniques for tracking dependencies fall into two main categories: *precise causality-tracking* (PCT) and *total ordering*. There are two variants under the first category: DistDGCC [60] and Taurus [57]. DistDGCC [60] includes the whole dependency graph in log records, incurring considerable runtime and storage overheads. Taurus [57] achieves the same effect by using *Log Sequence Number vectors* (LVs) to represent dependencies. However, with Taurus, every transaction and log entry stores an LV sized by the number of workers. Assume 100 workers and LVs are represented using 8-byte integers; every transaction and log would need at least 800 B of metadata, far exceeding the typical transaction and log record size.

Advantages of total ordering. Wang et al. proposed using *Global Sequence Numbers* (GSNs), based on Lamport clocks, in decentralized logging to achieve a global total order of transactions [54]. In this approach, a GSN is assigned to each transaction and page, establishing a partial order of transactions based on the sequence of page accesses, ultimately providing a total order for log entries associated with any page or transaction.

As Figure 7 (left) illustrates, with GSNs, a transaction synchronizes its local GSN to $max(localGSN, pageGSN)$ whenever it accesses a page (for both reads and writes). When a new log record is generated, the transaction advances both its local GSN and the GSN of the page associated with that log record. This technique was further refined with *Remote Flush Avoidance* (RFA) [9, 28], which determines whether a transaction only accesses pages modified by transactions previously executed on the same worker; if so, the transaction can commit as soon as its logs are hardened, as Figure 7 (right) depicts.

This approach has many advantages over the PCT – reduced memory footprint, minimal log metadata, and lower complexity – making GSNs compelling for dependency tracking in decentralized logging systems, as summarized in the table below:

	overheads[1]	robustness[2]	impl.
DistDGCC [60]	high	reliable	hard
Taurus [57]	high	reliable	hard
GSN/RFA [28, 54]	low	unstable	easy
GSN/RFA+Barrier (Our)	low	reliable	easy

[1] Storage and/or memory [2] In realistic scenarios

Weak commit condition in total ordering. Total ordering comes with a trade-off for its efficiency: it does not explicitly track transaction dependencies. In other words, it is impossible for the DBMS

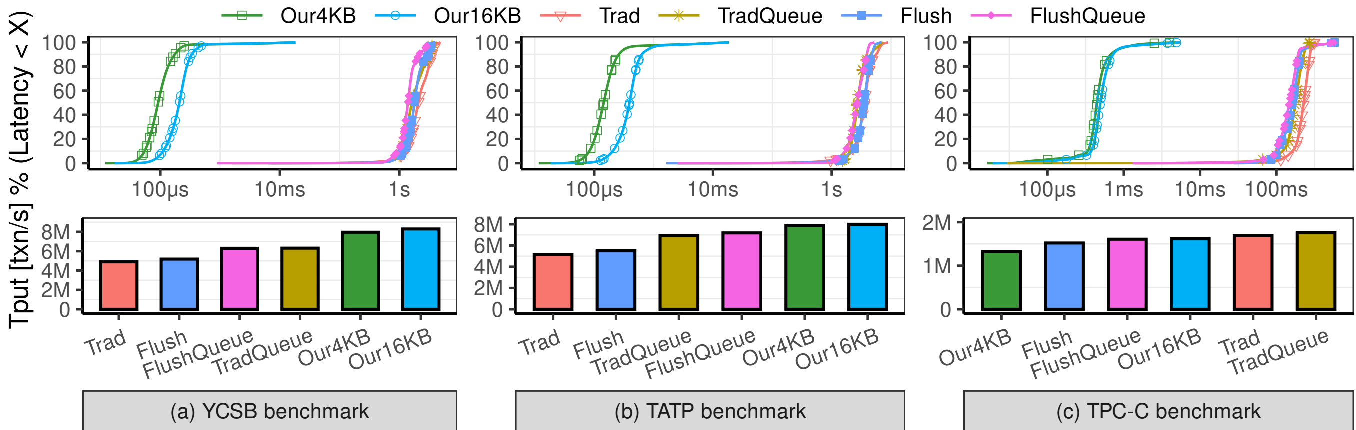


Figure 9: Close-looping benchmarks: Stressing both throughput and latency.

to precisely determine whether any particular transaction's dependencies are committed. Instead, a *weaker* commit condition is used: a transaction can commit if its GSN is less than or equal to the smallest durable GSN across workers [54].

Why is this commit condition weaker than that of PCT? We demonstrate this with an example in Figure 8, which involves three transactions: `TX50`, `TX20`, and `TX30`, all of which have been hardened, and `TX50` depends on `TX30`. All three transactions can be committed if we use any of the PCT approaches. That is, `TX20` and `TX30` have no dependency and can commit. For `TX50`, its only dependency is `TX30` which has already been committed, so the system can commit `TX50` as well. However, under the weaker commit condition of total ordering, both `TX50` and `TX30` cannot be committed because their GSNs are higher than the minimum durable GSN, which is 20 (from `TX20`). We call this the *straggler* problem.

5.2 Barrier transaction

Advancing minimum GSN. The straggler problem arises because slow or idle workers do not advance their GSNs during inactivity, as no transactions are being processed. To resolve this issue, we introduce *barrier transactions*. A barrier transaction is a lightweight transaction generated by a worker – typically when it wants to trigger a *force commit* (cf., Section 3.5). This transaction does not modify any data or generate log entries. Its sole purpose is to advance the worker's GSN, thereby increasing the global minimum GSN and allowing other transactions to be committed. An advantage of barrier transactions is that they integrate seamlessly with the existing commit logic, making the implementation straightforward.

Working example. Figure 8 provides a working example for barrier transaction. (1) Because the last transaction in worker 1 is `TX20`, thus other workers are trapped at `minGSN=20`. As a result, these workers can not commit `TX30` and `TX50` even if those transactions satisfy all commit conditions. Assuming that the current global GSN is 52, (2) we can push a *barrier* transaction with `GSN=52` to worker one, `B52`, and then trigger the commit operation on it. By doing so, (3) the minimum GSN becomes 52, so the system can then commit `TX30` and `TX50` with a minimal waiting time.

By strategically using barrier transactions, the system prevents idle workers from hindering overall progress, ensuring that transactions across all workers can be committed promptly.

6 Evaluation

Experimental setup. We integrate our proposal, denoted as *Our*, and all existing commit processing protocols into LeanStore, an open-source storage engine that integrates decentralized logging [6]. We use a combination of total ordering mechanism [33, 54] and Remote Flush Avoidance [9, 28] to manage transactions' ordering information. The size of the buffer pool is 32 GB.

Hardware & OS. All experiments were run on a single-socket machine with an AMD EPYC 9654P Processor with 96 cores / 192 hardware threads and 384 GB memory. The storage device used in the experiments is an enterprise-grade KIOXIA CM7-R 3.8TB NVMe PCIe5 SSD. We use Ubuntu with Linux kernel version 6.8.

Variants. We use two autonomous commit variants for evaluation:

- *Our4KB*: Every worker flushes its log buffer whenever the size of dirty log entries (i.e., log flush unit) is at least 4 KB.
- *Our16KB*: Similar to *Our4KB*, but the log flush unit is 16 KB. This variant offers both high throughput and low latency.

We use a stealing group size of eight, which follows the CPU topology as explained in Section 3.4. The acknowledgment group size is two (Section 6.6 will explain the reasoning). The queue size of all variants is 10 MB per worker.

Competitors. We compare our design with group commit, denoted as *Trad*. Another competitor is flush pipelining [9, 34], which runs group commit in a background thread, denoted as *Flush*. We implement all strategies in the same test system to isolate the conceptual differences from incidental differences. We also combine these two protocols with the queue design in Section 4, denoted as *TradQueue* for group commit and *FlushQueue* for flush pipelining. As we will show later, transactions in these competitors are queued for an extensive period, necessitating a larger queue size compared to our design. Hence, we use a queue size of 100 MB per worker.

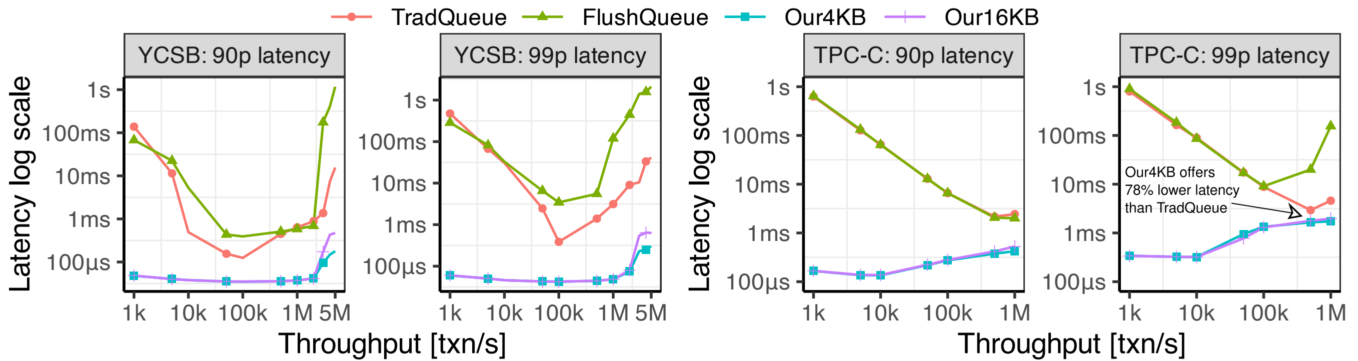


Figure 10: Open-looping benchmarks. The open-looping scheduler significantly undermines the background commit thread of flush pipelining, leading to its poor commit latency.

6.1 Stress Benchmark

In this benchmark, we use YCSB, TATP, and TPC-C workloads to evaluate all commit processing protocols. We run all experiments with 96 threads. For YCSB, we use 50% update ratio and 100 million tuples. The number of warehouses in the TPC-C workload is 96. And for TATP, the number of subscribers is 1 million. The experimental results are shown in Figure 9.

YCSB: Competitor latency. As Figure 9(a) (top left) illustrates, all competitors experience high commit latency. In this highly concurrent environment, most transactions take more than 1s to commit in our competitors. This is primarily because of the expensive *commit acknowledgment* and *queuing* overheads, as explained in Section 2.3. Moreover, because a single thread can not commit fast enough to keep up with the workers, the number of uncommitted dependencies is very high, exacerbating the *queuing* overheads.

YCSB: Latency of autonomous commit. *Our4KB* provides the lowest latency with its 90p is only 175μs, 12431× lower than that of *FlushQueue*, which provides the best 90p latency amongst all competitors. However, since autonomous commit fully saturates the I/O capacity of modern SSDs, the SSD occasionally struggles to serve write operations swiftly. This leads to rare delays in log flushing, causing high tail latencies for autonomous commit.

YCSB: Why competitors provide poor throughput? The main reason is that the log buffers are frequently full because the group commit processes them not fast enough. As a result, when a worker wants to append a new log record, it usually has to wait for the group commit to finish the *log flush*, negatively affecting system throughput. If we increase the size of the log buffers, the competitors' throughput improves at the trade-off of higher latency.

YCSB: Why our design provides higher throughput? On the other hand, autonomous commit writes logs frequently upon meeting the threshold. As a result, there are no *blocking* periods in the hot path. Therefore, as shown in Figure 9, the two autonomous commit variants have higher throughput than the two competitors. Specifically, the variant that provides the lowest throughput, *Our4KB*, outperforms the best competitor, *TradQueue*, by 26.1%.

TATP summary. As shown in Figure 9(c), all autonomous commit variants show superior commit latency to group commit and flush pipelining, similar to previous benchmarks. TATP and YCSB share similar throughput and latency patterns because the transactions in these two workloads are very lightweight, pressuring the commit processing subsystem considerably.

TPC-C latency: Autonomous commit vs. competitors. From Figure 9(b), autonomous commit is also better than all competitors. Specifically, *Our4KB* and *Our16KB* provide 283× and 265× lower 90p latency, respectively, compared to the best competitor, *FlushQueue*. The differences between autonomous commit and the competitors are less pronounced than in the previous benchmarks because the *queuing* and *commit acknowledgment* overheads are less severe in TPC-C compared to YCSB. TPC-C transactions are more complex and produce larger log records, leading to lower throughput.

TPC-C throughput. Only *Our16KB* variant provides comparable throughput to the group commit techniques. In contrast, *Our4KB* invokes *log flush* and *commit acknowledgment* much more frequently than other variants, thus generating fewer transactions and is not able to amortize the overhead. We argue that the latency improvements outweigh minor reductions in throughput.

6.2 Open-Loop Benchmark

To better simulate real-world scenarios with frequent user think times, we integrate the *open-system model* [8, 18, 20, 25, 28, 45, 51, 64] with this benchmark. We use both YCSB and TPC-C workloads in this benchmark, with similar configurations as in previous experiments. The results are shown in Figure 10.

Autonomous commit. Autonomous commit, *Our4KB* and *Our16KB*, always has the lowest commit latency in both YCSB and TPC-C, with its 99p latency usually less than 100μs for YCSB and 1ms for TPC-C. This shows that barrier transactions effectively resolve the *stragger* problem of GSN-based decentralized logging. Specifically, when the throughput is small, workers in autonomous commit will likely trigger force commit several times during system idle. With barrier transactions, all workers know that everyone has flushed all dirty logs and can acknowledge transactions in time. Without *barrier transactions*, autonomous commit will have high commit latency for low-throughput scenarios, even though still lower than that of the two competitors (not shown in the figure). This explains the conceptual issue of GSN-based decentralized logging and the important role of barrier transactions in this problem.

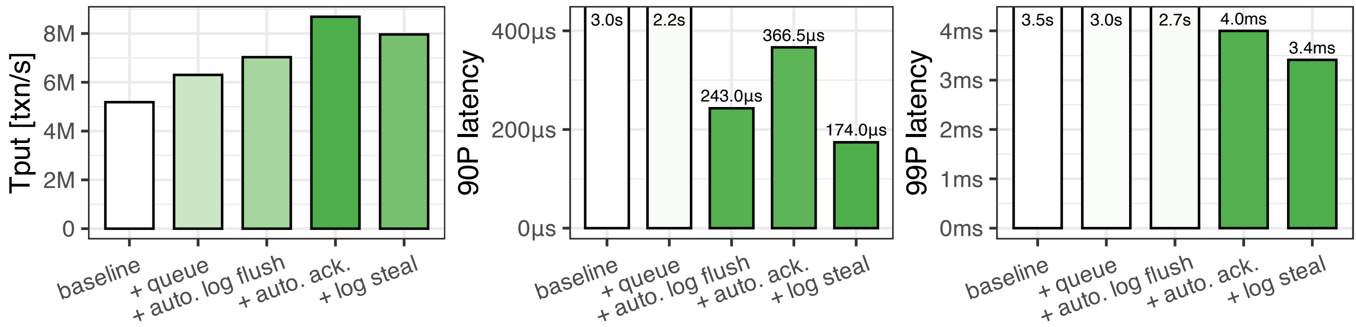


Figure 11: Impact of different optimizations on throughput and latency.

Competitors vs. autonomous commit. In contrast, *TradQueue* and *FlushQueue* do not implement *barrier transaction* and *force commit*, explaining why they suffer from the *straggler* problem of GSN-based logging in open-loop experiments. These two competitors only achieve their best commit latency when the throughput is sufficiently high – 100K and 500K transactions per second for YCSB and TPC-C, respectively. This occurs when the benchmark framework generates transactions fast enough to advance the minimum GSN in sync with their commit rate. However, even in the most favorable scenarios for the competitors, i.e., large TPC-C transactions with high throughput, the lowest 99P latency they can achieve is still 78% higher than that of *Our4KB*.

Group commit vs. flush pipelining. The only difference between group commit and flush pipelining is whether to run the group commit task in a background thread. As Figure 10 shows, the differences between these two are significant, unlike the experiments in Figure 9. This difference is primarily because the CPU cores are oversubscribed with the background commit thread. Particularly, because the transaction scheduler (which is in the hot path of the workers) is compute-intensive [4, 18], hence workers will consume CPU resources considerably and thus likely preempt the background commit thread, a finding aligns with numerous previous studies [19, 25, 33, 35, 53].

6.3 Ablation Study

To better understand the impact of our conceptual building blocks, we now dissect how each optimization affects the performance. We start with the flush pipelining as the baseline and end with *Our4KB* variant. Step by step, we add additional features: the optimized queue in Section 4, autonomous log flush in Section 3.1, autonomous commit acknowledgment in Section 3.2, and log stealing optimization in Section 3.4. We also use the YCSB workload with the same parameters as the previous benchmarks. Figure 11 shows the ablation study for throughput (left), 90p latency (middle), and 99.9p latency (right).

Lock-free queue. As shown in Figure 11, replacing a latch-based data structure with a lock-free queue improves throughput and commit latency. Specifically, the optimized queue improves the throughput of the flush pipelining by 21.5% and reduces 90p latency by 41%. These improvements show that commit operations can block workers from adding new transaction objects to the pre-committed queue for a significant amount of time.

Autonomous log flush: Eradicating I/O spikes. By moving the log flush to the workers, the background thread is responsible only for commit acknowledgment. Autonomous log flush can significantly reduce the 90th-percentile latency by $8,867\times$. There are two main reasons for this improvement. First, autonomous log flush effectively mitigates the overheads from I/O spikes. Second, it also reduces queuing overhead. Specifically, because the workload of the background thread is lighter, it can initiate many more commit rounds, thereby considerably decreasing queuing time.

Autonomous log flush: Tail latencies. However, autonomous log flush exhibits a very high 99th-percentile latency, similar to flush pipelining. This primary reason, unsurprisingly, is the CPU oversubscription issue on the background commit thread. That is, the number of queued transactions is occasionally considerable because the background thread does not have enough computing resources, leading to considerable *queuing* and the *commit acknowledgment* overheads, thus resulting in the phenomenon.

Autonomous acknowledgment: Eliminating high 99p latency. Autonomous commit acknowledgment is free of CPU oversubscription by eliminating the background thread, resulting in lower 99p latency. One surprising finding is that autonomous commit acknowledgment also improves system throughput. This improvement occurs because, at times, the pre-committed queues become full while the background commit thread cannot progress due to resource contention.

Stealing logs to optimize latency. Figure 11 shows that the 90p latency of autonomous commit without log stealing optimization is even worse than that of autonomous log flush. With log stealing optimization, autonomous commit can cut the 90th-percentile latency in half, and the 99th-percentile latency also reduces by 17.6%. The only issue with the stealing optimization is that it reduces throughput slightly by 9.1% because of extra synchronization. However, we argue that the latency gain is more significant than the reduction in throughput, which is still higher than all other variants.

6.4 Scalability

In this benchmark, we use the YCSB workload with the same configurations as previous benchmarks. We vary the number of worker threads to evaluate our design's scalability and compare it with group commit and flush pipelining. The experimental results are depicted in Figure 12.

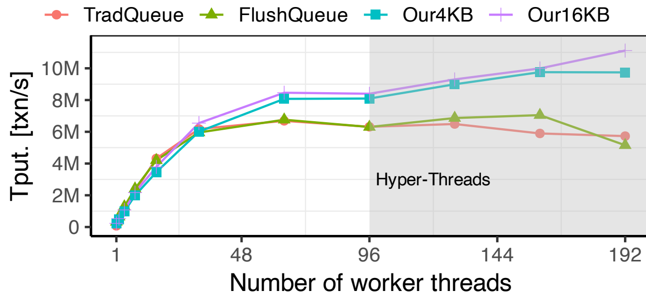


Figure 12: Scalability benchmark.

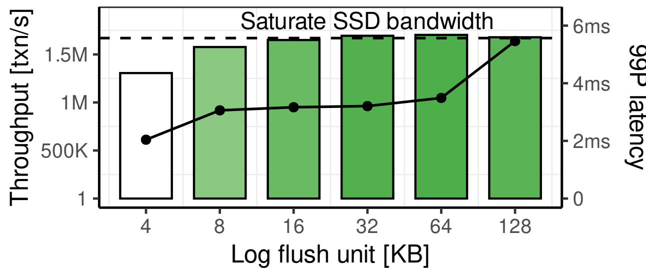


Figure 13: Log flush unit benchmark.

Result. As shown, both autonomous commit and the two competitors provide similar performance until 32 threads. When the number of threads exceeds 32, *TradQueue* and *FlushQueue* cannot scale more. This is because the single-threaded nature is insufficient for that massive volume of transactions. Consequently, workers often wait for the group commit to finish the log flush, leading to considerable delays and poor scalability. Larger log buffers might improve their scalability but at a higher memory usage. In contrast, autonomous commit writes log regularly, hence does not require large log buffers while still exhibiting good scalability, with *Our16KB* reaching 11 million transactions per second with 192 threads.

6.5 Log Flush Unit

We use the TPC-C workload in this experiment because it more effectively illustrates the throughput differences between various log flush units compared to YCSB and TATP. The log flush unit varies from 4 KB to 128 KB.

Result. As Figure 13 depicts, 4 KB results in the lowest commit latency. The main reason is that smaller log flush units trigger *log flushes* more frequently, which advances the minimum durable GSN more quickly. Additionally, smaller log flush units increase the number of independent transactions that meet the RFA conditions, thereby avoiding remote log flushes (cf., Section 5). Together, these factors contribute to lower commit latency with smaller log flush units. On the other hand, Figure 13 also shows that even with 16 KB, the autonomous commit can saturate the SSD bandwidth, i.e., the throughput can not increase much with a higher log flush unit. In short, the autonomous commit with a log flush unit of 4 KB provides the lowest latency, while a log flush unit of 16 KB is an all-around solution.

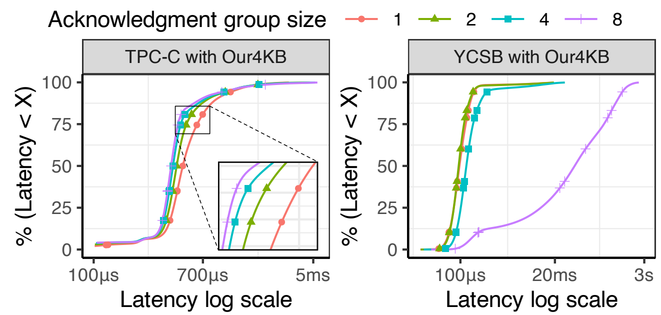


Figure 14: Acknowledgment group size experiment.

6.6 Acknowledgment Group Size

We will evaluate the acknowledgment group size in this benchmark using both TPC-C and YCSB with the same configurations as in previous experiments. The group size varies from one to eight – with a group size of one, the lock-free queue closely resembles the private transaction queue proposed by Huang et al. [33]. The autonomous commit variant we use is *Our4KB*.

Result. As Figure 14 shows, in TPC-C experiments, the group size of eight results in the lowest commit latency. The primary reason is that, given that the number of transactions in TPC-C is low, a bigger group size means every acknowledgment round can commit more transactions in time. However, in YCSB, that group size has the highest commit latency. This is because the number of completed transactions in YCSB is much higher; thus, a bigger group size leads to more expensive commit acknowledgment, and ultimately, the *queuing* overhead is very high.

Furthermore, bigger group sizes also boost throughput, with a group size of eight delivering 10% higher throughput compared to a group size of one (not shown in the figure). We also want to note that the dependency ratio (i.e., how many transactions require remote log flush or not) only depends on how fast autonomous log flush is triggered, and thus is orthogonal to acknowledgment group size. In short, a group size of two or four allows low-latency commit in both YCSB and TPC-C, which covers a wide range of workloads.

6.7 Mixed Workloads

This experiment runs TPC-C and YCSB workloads on different workers to simulate a scenario where workers generate log entries at varying rates. Specifically, to challenge the autonomous commit mechanism, we configure each stealing group so that half of the workers (i.e., four workers) run TPC-C while the other four workers execute YCSB. This makes it hard for workers to steal log entries from their peers and introduces additional synchronization overhead. In contrast, the two competitors, *TradQueue* and *FlushQueue*, remain unaffected by this configuration. The experimental results are presented in Figure 15.

Result. Surprisingly, both *TradQueue* and *FlushQueue* provide reasonable commit latency in this experiment, unlike previous experiments. This is because the data size of this experiment exceeds the buffer pool capacity, i.e., out-of-memory. As a result, the average

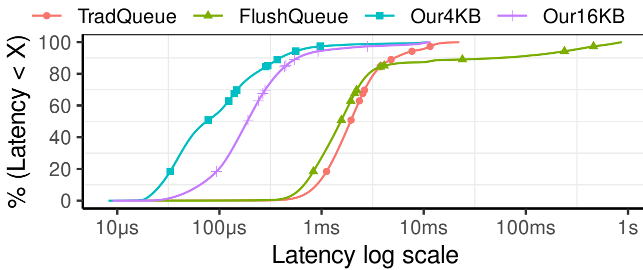


Figure 15: Mixed-workloads benchmark.

Table 1: Comparative analysis of existing commit protocols.

	throughput	latency	support OOM[1]	require PM[2]
Group commit [2, 3]	medium	high	yes	no
Flush pipelining [34]	medium	high	yes	no
Epoch-based [52]	high	high	no	no
PM[2]-based [28, 54]	high	low	yes	yes
Autonomous commit	high	low	yes	no

[1] Out-of-memory workloads [2] Persistent memory

execution time of transactions is several times slower than in previous experiments, leading to significantly lower log volume and lower transaction objects. Despite that, autonomous commit still outperforms both traditional group commit and flush pipelining, especially *Our4KB* whose 90th-percentile latency is 13.2× and 136.8× lower than that of *TradQueue* and *FlushQueue*, respectively.

7 Related Work

In Section 2, we already introduced the prior work that our approach is based on. In section, we discuss other commit processing approaches and contrast their properties with our proposal.

Epoch-based group commit. Epoch-based group commit [52, 62] divides time into a sequence of *epochs*, during which a *batch* of transactions is persisted to storage. While this provides high throughput, it leads to high commit latency, often taking milliseconds to commit. This is because the commit state is driven by the epoch period, which is typically tens of milliseconds [52]. Additionally, this technique is unsuitable for storage-based DBMSs because it relies on *value logging*, which requires the whole dataset to reside in memory.

Persistent memory-based commit. To simultaneously achieve instant commits and high throughput, previous research has used PM to store transaction logs [28, 31, 38, 39, 54, 61]. However, the discontinuation of the Intel Optane project [16, 26], along with the limited availability of NVDIMMs compared to NVMe SSDs [11, 32], has made PM-based commit protocols less attractive. Furthermore, adapting PM-based commit protocols for NVMe SSDs is challenging because (1) the SSD programming model differs from that of persistent memory; (2) modern NVMe SSDs still exhibit significantly higher I/O latency compared to persistent memory devices; and (3) the CPU overhead required to persist a log entry is higher on SSDs. We summarize the conceptual differences between existing commit protocols and autonomous commit, when combined with decentralized logging, in Table 1.

Disk-based commit protocol. Throughout this paper, we have highlighted that the autonomous commit protocol is optimized explicitly for modern NVMe SSDs, which support low-latency, parallel writes. However, if the DBMS is deployed on mechanical disks with millisecond-scale write latencies, the performance of autonomous commit is likely to degrade significantly. In such scenarios, it may underperform compared to traditional approaches like group commit or flush pipelining. Addressing this limitation is an area we intend to explore in future research.

Delays in distributed DBMSs. As discussed in Section 5, the GSN-based dependency tracking mechanism suffers from the *straggler* problem, where idle workers fail to process transactions, hence preventing the advancement of the minimum GSN and delaying transaction commits. This issue is also prevalent in distributed DBMSs and can be even more pronounced due to challenges caused by the networking infrastructure, such as network partitioning and high communication overheads. One approach to address this problem is *write-TID buffering* [41, 42], which optimizes transaction ID assignment to reduce delays from cross-node communication. Both techniques, barrier transaction and write-TID buffering, aim to resolve global progress dependencies, effectively mitigating the impact of delays and stragglers on system performance.

8 Summary

In this paper, we have demonstrated that achieving both high scalability and low commit latency in transactional DBMSs is possible. By designing and integrating the autonomous commit protocol with decentralized logging, we have created a system that scales efficiently while maintaining low latency. Our performance study shows that our approach can commit transactions swiftly, achieving 90th-percentile latencies in the microsecond range across diverse workloads. We believe that autonomous commit provides a practical solution for high-performance DBMSs. Our implementation is open source and available at https://github.com/leanstore/leanstore/tree/latency.

Acknowledgments

We are grateful to the reviewers for their thoughtful feedback. Our sincere thanks go to Goetz Graefe for his invaluable suggestions that helped shape this research. Additionally, we also thank Tianzheng Wang and Kaisong Huang for an insightful discussion about decentralized logging and dependency tracking mechanisms.

References

[1] 2023. 4th Gen AMD EPYC™ Processor Architecture. https://www.amd.com/system/files/documents/4th-gen-epyc-processor-architecture-white-paper.pdf.
[2] 2023. MySQL source code. https://github.com/mysql/mysql-server/tree/mysql-cluster-8.0.33.
[3] 2023. PostgreSQL source code. https://github.com/postgres/postgres/tree/REL_15_3.
[4] 2024. BenchBase source code. https://github.com/cmu-db/benchbase.
[5] 2024. Intel® Xeon® 6 Processors. https://www.intel.com/content/www/us/en/products/details/processors/xeon.html.
[6] 2024. LeanStore. https://github.com/leanstore/leanstore.
[7] 2024. NVMe Specifications. https://nvmexpress.org/specifications/.
[8] Dana Van Aken, Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2015. BenchPress: Dynamic Workload Control in the OLTP-Bench Testbed. In *SIGMOD Conference*. ACM, 1069–1073.
[9] Adnan Alhomssi, Michael Haubenschild, and Viktor Leis. 2023. The Evolution of LeanStore. In *BTW (LNI, Vol. P-331)*. Gesellschaft für Informatik e.V., 259–281.

[10] Mijin An, Soojun Im, Dawoon Jung, and Sang Won Lee. 2022. Your Read is Our Priority in Flash Storage. *Proc. VLDB Endow.* 15, 9 (2022), 1911–1923.

[11] Mijin An, Jonghyeok Park, Tianzheng Wang, Beomseok Nam, and Sang-Won Lee. 2023. NV-SQL: Boosting OLTP Performance with Non-Volatile DIMMs. *Proc. VLDB Endow.* 16, 6 (2023), 1453–1465.

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

[13] Jan Böttcher, Viktor Leis, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2020. Scalable and robust latches for database systems. In *DaMoN*. ACM, 2:1–2:8.

[14] Alvin Cheung. 2014. Rethinking the application-database interface. In *PSFW@HPDC*. ACM, 1–2.

[15] Jeffrey Dean and Luiz André Barroso. 2013. The tail at scale. *Commun. ACM* 56, 2 (2013), 74–80.

[16] Peter Desnoyers, Ian F. Adams, Tyler Estro, Anshul Gandhi, Geoff Kuenning, Michael P. Mesnier, Carl A. Waldspurger, Avani Wildani, and Erez Zadok. 2023. Persistent Memory Research in the Post-Optane Era. In *DIMES@SOSP*. ACM, 23–30.

[17] David J. DeWitt, Randy H. Katz, Frank Olken, Leonard D. Shapiro, Michael Stonebraker, and David A. Wood. 1984. Implementation Techniques for Main Memory Database Systems. In *SIGMOD Conference*. ACM Press, 1–8.

[18] Djellel Eddine Difallah, Andrew Pavlo, Carlo Curino, and Philippe Cudré-Mauroux. 2013. OLTP-Bench: An Extensible Testbed for Benchmarking Relational Databases. *Proc. VLDB Endow.* 7, 4 (2013), 277–288.

[19] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proc. VLDB Endow.* 16, 11 (2023), 2769–2782.

[20] Steffen Friedrich, Wolfram Wingerath, and Norbert Ritter. 2017. Coordinated Omission in NoSQL Database Benchmarking. In *BTW (Workshops) (LNI, Vol. P-266)*. GI, 215–225.

[21] Dieter Gawlick and David Kinkade. 1985. Varieties of Concurrency Control in IMS/VS Fast Path. *IEEE Database Eng. Bull.* 8, 2 (1985), 3–10. http://sites.computer.org/debull/85JUN-CD.pdf

[22] Goetz Graefe. 2012. A survey of B-tree logging and recovery techniques. *ACM Trans. Database Syst.* 37, 1 (2012), 1:1–1:35.

[23] Gabriel Haas, Adnan Alhomssi, and Viktor Leis. [n. d.]. Managing Very Large Datasets on Directly Attached NVMe Arrays. *Scalable Data Management for Future Hardware* ([n. d.]), 223.

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

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

[26] Jim Handy and Tom Coughlin. 2023. Optane's Dead: Now What? *Computer* 56, 3 (2023), 125–130.

[27] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP through the looking glass, and what we found there. In *SIGMOD Conference*. ACM, 981–992.

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

[29] Haochen He, Erci Xu, Shanshan Li, Zhouyang Jia, Si Zheng, Yue Yu, Jun Ma, and Xiangke Liao. 2023. When Database Meets New Storage Devices: Understanding and Exposing Performance Mismatches via Configurations. *Proc. VLDB Endow.* 16, 7 (2023), 1712–1725.

[30] Yongjun He, Jiacheng Lu, and Tianzheng Wang. 2020. CoroBase: Coroutine-Oriented Main-Memory Database Engine. *Proc. VLDB Endow.* 14, 3 (2020), 431–444.

[31] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. *Proc. VLDB Endow.* 8, 4 (2014), 389–400.

[32] Kaisong Huang and Tianzheng Wang. 2024. *Lessons Learned and Outlook*. Springer Nature Switzerland, 77–87.

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

[34] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2010. Aether: A Scalable Approach to Logging. *Proc. VLDB Endow.* 3, 1 (2010), 681–692.

[35] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2012. Scalability of write-ahead logging on multicore and multi-socket hardware. *VLDB J.* 21, 2 (2012), 239–263.

[36] Woon-Hak Kang, Sang-Won Lee, Bongki Moon, Yang-Suk Kee, and Moonwook Oh. 2014. Durable write cache in flash memory SSD for relational and NoSQL databases. In *SIGMOD Conference*. ACM, 529–540.

[37] Jong-Bin Kim, Hyeongwon Jang, Seohui Son, Hyuck Han, Sooyong Kang, and Hyungsoo Jung. 2019. Border-Collie: A Wait-free, Read-optimal Algorithm for Database Logging on Multicore Hardware. In *SIGMOD Conference*. ACM, 723–740.

[38] Wook-Hee Kim, Jinwoong Kim, Woongki Baek, Beomseok Nam, and Youjip Won. 2016. NVWAL: Exploiting NVRAM in Write-Ahead Logging. In *ASPLOS*. ACM, 385–398.

[39] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeongjun Park, Young-ri Choi, Alan Sussman, and Beomseok Nam. 2022. ListDB: Union of Write-Ahead Logs and Persistent SkipLists for Incremental Checkpointing on Persistent Memory. In *OSDI*. USENIX Association, 161–177.

[40] Dimitrios Koutsoukos, Raghav Bhartia, Michal Friedman, Ana Klimovic, and Gustavo Alonso. 2023. NVM: Is it Not Very Meaningful for Databases? *Proc. VLDB Endow.* 16, 10 (2023), 2444–2457.

[41] Juchang Lee, Yong Sik Kwon, Franz Färber, Michael Muehle, Chulwon Lee, Christian Bensberg, Joo-Yeon Lee, Arthur H. Lee, and Wolfgang Lehner. 2013. SAP HANA distributed in-memory database system: Transaction, session, and metadata management. In *ICDE*. IEEE Computer Society, 1165–1173.

[42] Juchang Lee, Michael Muehle, Norman May, Franz Faerber, Vishal Sikka, Hasso Plattner, Jens Krüger, and Martin Grund. 2013. High-Performance Transaction Processing in SAP HANA. *IEEE Data Eng. Bull.* 36, 2 (2013), 28–33.

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

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

[45] Tianyu Li, Matthew Butrovich, Amadou Ngom, Wan Shen Lim, Wes McKinney, and Andrew Pavlo. 2020. Mainlining Databases: Supporting Fast Transactional Workloads on Universal Columnar Data File Formats. *Proc. VLDB Endow.* 14, 4 (2020), 534–546.

[46] 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 Trans. Database Syst.* 17, 1 (1992), 94–162.

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

[48] Lam-Duy Nguyen and Viktor Leis. 2024. Why Files If You Have a DBMS?. In *ICDE*. IEEE, 3878–3892.

[49] Hamish Nicholson, Periklis Chrysogelos, and Anastasia Ailamaki. 2022. HPCache: Memory-Efficient OLAP Through Proportional Caching. In *DaMoN*. ACM, 7:1–7:9.

[50] Jonghyeok Park, Soyee Choi, Gihwan Oh, Soojun Im, Moonwook Oh, and Sang-Won Lee. 2023. FlashAlloc: Dedicating Flash Blocks By Objects. *Proc. VLDB Endow.* 16, 11 (2023), 3266–3278.

[51] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Open Versus Closed: A Cautionary Tale. In *NSDI*. USENIX.

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

[53] Benjamin Wagner, André Kohn, and Thomas Neumann. 2021. Self-Tuning Query Scheduling for Analytical Workloads. In *SIGMOD Conference*. ACM, 1879–1891.

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

[55] Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2017. Query Fresh: Log Shipping on Steroids. *Proc. VLDB Endow.* 11, 4 (2017), 406–419.

[56] Hobin Woo, Daegyu Han, Seungjoon Ha, Sam H. Noh, and Beomseok Nam. 2023. On Stacking a Persistent Memory File System on Legacy File Systems. In *FAST*. USENIX Association, 281–296.

[57] Yu Xia, Xiangyao Yu, Andrew Pavlo, and Srinivas Devadas. 2020. Taurus: Lightweight Parallel Logging for In-Memory Database Management Systems. *Proc. VLDB Endow.* 14, 2 (2020), 189–201.

[58] Hiroyuki Yamada, Toshihiro Suzuki, Yuji Ito, and Jun Nemoto. 2023. ScalarDB: Universal Transaction Manager for Polystores. *Proc. VLDB Endow.* 16, 12 (2023), 3768–3780.

[59] Stephen Yang, Seo Jin Park, and John K. Ousterhout. 2018. NanoLog: A Nanosecond Scale Logging System. In *USENIX ATC*. USENIX Association, 335–350.

[60] Chang Yao, Meihui Zhang, Qian Lin, Beng Chin Ooi, and Jiatao Xu. 2018. Scaling distributed transaction processing and recovery based on dependency logging. *VLDB J.* 27, 3 (2018), 347–368.

[61] Jongyeon Yoo, Hokeun Cha, Wonbae Kim, Wook-Hee Kim, Sung-Soon Park, and Beomseok Nam. 2022. Pivotal B+tree for Byte-Addressable Persistent Memory. *IEEE Access* 10 (2022), 46725–46737.

[62] Wenting Zheng, Stephen Tu, Eddie Kohler, and Barbara Liskov. 2014. Fast Databases with Fast Durability and Recovery Through Multicore Parallelism. In *OSDI*. USENIX Association, 465–477.

[63] Tobias Ziegler, Philip A. Bernstein, Viktor Leis, and Carsten Binnig. 2023. Is Scalable OLTP in the Cloud a Solved Problem?. In *CIDR*. www.cidrdb.org.

[64] Tobias Ziegler, Carsten Binnig, and Viktor Leis. 2022. ScaleStore: A Fast and Cost-Efficient Storage Engine using DRAM, NVMe, and RDMA. In *SIGMOD Conference*. ACM, 685–699.

Received 8 October 2024; revised 3 December 2024; accepted 3 February 2025