

Autonomous Commit and Commit Causality Clocks: Building Blocks for Logging and Recovery on Modern Hardware

LAM-DUY NGUYEN, Technische Universität München, Germany

ADNAN ALHOMSSI*, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany

TOBIAS ZIEGLER†, Technische Universität München, Germany

VIKTOR LEIS, Technische Universität München, Germany

Traditional ARIES has long been the standard for logging and recovery in database systems. Yet, its overhead limits system efficiency on modern hardware, leading to a shift towards decentralized logging for improved performance and scalability. Decentralized logging, however, poses two major challenges: (1) its integration with existing commit processing protocols fails to provide low-latency commits, (2) current variants introduce many critical trade-offs that hinder widespread adoption.

This paper aims to address these gaps. To tackle the first issue, we introduce *Autonomous Commit*, the first commit protocol that fully utilizes modern NVMe SSDs to achieve low-latency durable commits with decentralized logging. Second, we propose *Commit Causality Clocks*, a new dependency tracking mechanism optimized for decentralized logging that retains the strengths of existing approaches while eliminating their limitations. We also revisit the recovery subsystem, enabling DBMS logging to work directly on block devices for optimal performance during normal operation. Our experimental evaluation demonstrates that these techniques together deliver very good scalability and low-latency durable commits across diverse workloads, enabling a high-performance logging and recovery design.

CCS Concepts: • **Information systems** → **Database transaction processing**.

Additional Key Words and Phrases: Database Management Systems, Transaction Processing, Logging, Recovery, Group Commit, Commit Processing, Latency, Partial Order, Vector Clocks

ACM Reference Format:

Lam-Duy Nguyen, Adnan Alhomssi, Tobias Ziegler, and Viktor Leis. 2026. Autonomous Commit and Commit Causality Clocks: Building Blocks for Logging and Recovery on Modern Hardware. *ACM Trans. Datab. Syst.* 1, 1 (January 2026), 32 pages. <https://doi.org/10.1145/3859374>

1 Introduction

Data durability with ARIES. For decades, ARIES-style write-ahead logging (WAL) [61] has been the standard approach for ensuring data durability in disk-based DBMSs. Its widespread adoption stems from its numerous advantages, including efficient handling of out-of-memory datasets, fast and reliable crash recovery, support for fuzzy checkpoints, and many more.

Scalability issue of ARIES and how decentralized logging addresses that. While ARIES offers strong qualitative guarantees, its scalability is limited, especially on modern hardware. The

*Now unaffiliated.

†Also with TigerBeetle.

Authors' Contact Information: Lam-Duy Nguyen, lamduy.nguyen@tum.de, Technische Universität München, Germany; Adnan Alhomssi, adnan.alhomssi@gmail.com, Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany; Tobias Ziegler, tobias@tigerbeetle.com, Technische Universität München, Germany; Viktor Leis, leis@in.tum.de, Technische Universität München, Germany.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).

ACM 1557-4644/2026/1-ART

<https://doi.org/10.1145/3859374>

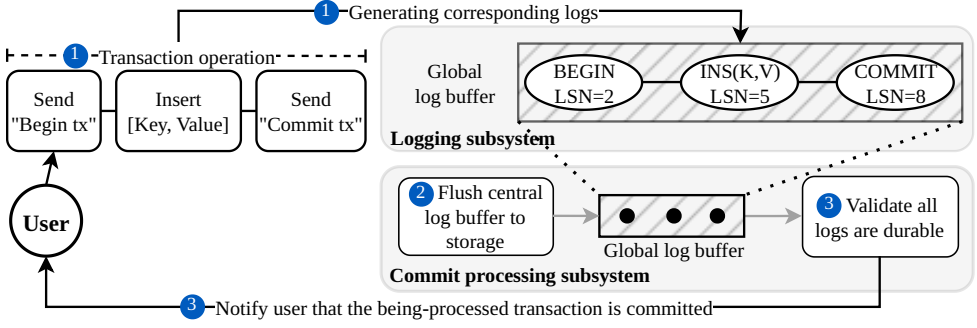


Fig. 1. The logging architecture of an ARIES-based DBMS can be conceptually divided into two key components: (1) Logging subsystem, which generates ARIES-style log entries [61] and temporarily buffers these records; and (2) the commit processing subsystem, which handles concurrency control (such as optimistic validation [65, 74, 76, 85]) and ensures the logs produced by the logging subsystem are safely persisted.

primary bottleneck lies in the logging subsystem (top-right of Figure 1), specifically the *global log buffer*, where every thread must acquire a latch protecting the shared buffer before generating log entries. To address this contention, many prior studies have proposed *decentralized logging*, which partitions the global log buffer into multiple buffers, each exclusively owned by a single thread [32, 74, 76, 80, 86], eliminating contention and enabling higher scalability.

Decentralized logging causes latency problems. While decentralized logging enhances scalability by removing contention on the centralized log buffer, it also introduces two new performance challenges that both significantly worsen commit latency. These challenges emerge from both the *commit processing subsystem* – responsible for concurrency control validations [65, 74, 76, 85] and log persistence (bottom-right of Figure 1) – and the decentralized logging mechanism itself, each introducing latency overheads in different scenarios. To achieve robust low-latency commits under decentralized logging, it is therefore crucial to understand and mitigate these sources of overhead.

Problem #1: Trade-off in the logging subsystem. The first problem originates in the logging subsystem and reflects a fundamental trade-off between commit latency and operational overheads. That is, while multiple log buffers help reduce contention, they complicate durability conditions for transactions, which can either (1) negatively affects commit latency during system inactivity or (2) introduce superfluous storage overhead for log metadata and additional runtime synchronization across threads (cf. Section 5). Achieving every desideratum – minimal log metadata, lightweight synchronization, and consistently low-latency commits – requires a rethinking of decentralized logging design.

Problem #2: Traditional commit processing protocol is the bottleneck. The second source of latency is from the commit processing subsystem, where group commit – the most widely used commit protocol – cannot keep pace with the higher throughput enabled by decentralized logging, leading to excessive commit delays under high system load. To demonstrate this limitation, we benchmark several existing DBMSs – focusing on their commit processing protocols – with a YCSB benchmark (details in Section 7.1) on modern NVMe SSDs. This benchmark contrasts two widely adopted approaches: (1) Group commit with centralized ARIES logging [61] as used by WiredTiger, SQLite, and RocksDB, and (2) Group commit with a variant of decentralized logging [32, 76, 80] (cf., Section 5) as used by LeanStore [32]. As the figure shows, the ARIES approach can achieve low 90th-percentile latency but offers limited throughput due to contention on the global log buffer. In contrast, the decentralized design scales well but incurs significantly higher commit latency, with

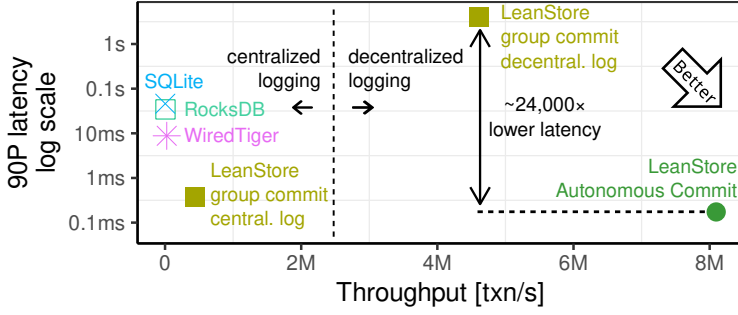


Fig. 2. No existing DBMS or commit protocol can achieve both low-latency commits and high throughput. Our proposed solution, **autonomous commit**, accomplishes both.

the 90th-percentile latency exceeding four seconds – 99.999% of which is attributed to the group commit runtime.

The missing complete solution. Both the logging and commit processing subsystems are fundamental to DBMS architecture. Beyond their direct impact on system performance, they also shape the correctness and efficiency of the *recovery* process. This paper proposes a unified approach to logging, commit processing, and recovery, addressing all the performance challenges outlined above. The result is a storage engine that achieves high throughput, low latency, and fast recovery. Our key contributions are as follows:

Contribution #1: Autonomous commit. We propose *autonomous commit*, a novel protocol that addresses the problems of the commit processing subsystem. When paired with any decentralized logging variant, the DBMS can achieve both high throughput and low-latency commits. This is done by leveraging modern NVMe SSDs, which support small, random, parallel, and durable writes with double-digit microsecond latency. Instead of batching large writes as in traditional group commit, autonomous commit parallelizes small log flushes across workers. Additionally, we decouple and parallelize the dependency checking from the flushing phase. Together with a series of optimizations, autonomous commit is able to achieve a 90th-percentile commit latency of under a millisecond, as shown in Figure 2.

Contribution #2: Commit causality clocks. As discussed earlier, existing decentralized logging variants face a fundamental trade-off between achieving low-latency commits during low system activity and maintaining lightweight metadata and synchronization. This trade-off arises from how they track the ordering of committed states (i.e., up to which transactions have been persisted) across multiple log buffers: lightweight clocks introduce unnecessary latency under inactivity, whereas precise causality tracking incurs substantial metadata and synchronization overhead. To address this limitation, we propose a new dependency tracking mechanism for decentralized logging, termed *commit causality clocks* (C3), which combines the strengths of both approaches. The key idea is that precise causality tracking is only necessary at transaction commit; during normal operation, the cheaper total ordering mechanism is sufficient. By deferring exact dependency tracking until commit time, C3 supports low-latency commits during inactivity while maintaining minimal metadata overhead and lightweight synchronization.

Summary. This article is an extended version of a conference paper introducing autonomous commit, a novel commit processing protocol that achieves high scalability while delivering microsecond-level commit latency [63]. The new contributions presented in this extended version are as follows:

- *Commit causality clocks* (C3), a new dependency tracking mechanism optimized for decentralized logging. By exploiting transaction semantics, C3 combines the strengths of prior

approaches – the simplicity of Lamport clocks and the precise causality information of vector clocks – while eliminating their drawbacks.

- The first complete design and implementation of a high-performance logging architecture that supports parallel logging and recovery without relying on an intermediate file system – unlike prior systems that rely on file systems for parallel recovery [32, 71, 86].

Outline. The structure of the paper is as follows. Section 2 provides the necessary background on the commit processing subsystem. Section 3 introduces *autonomous commit*, detailing its design and key techniques. A fundamental aspect of autonomous commit, which also benefits other commit protocols like group commit, is the *scalable transaction queue*, which we will discuss in Section 4. Next, in Section 5, we discuss the technical design considerations of decentralized logging, explaining in detail how it influences commit conditions and commit latency, and why C3 outperforms all previous variants. Then, in Section 6, we describe how recovery is implemented to meet the block device requirements of autonomous commit. Section 7 empirically evaluates the performance of autonomous commit and C3. Finally, we discuss related work in Section 8 and conclude in Section 9.

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 3(a), the life cycle of a transaction in a DBMS using ARIES-style logging follows several stages [61]. 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 state¹, 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 3(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 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

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

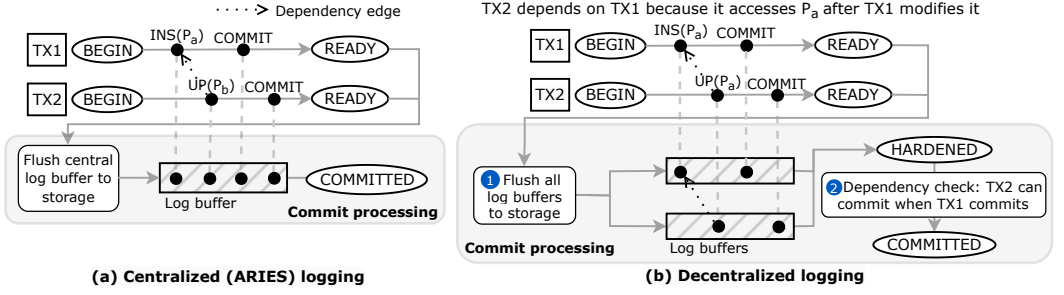


Fig. 3. State machine of a transaction, in centralized (ARIES) logging (a) vs. decentralized logging (b).

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 [30, 39, 76, 80].

Decentralized logging: Scalability with a trade-off. To address the scalability issues of centralized logging, many works propose *decentralized logging* as a viable solution [30, 32, 43, 76, 80]. Under a decentralized logging scheme, each worker thread manages a dedicated log buffer and flushes its associated log entries independently. When a transaction enters the DBMS, it is assigned to a specific worker thread, which executes *all* its constituent operations; consequently, all log records generated by a given transaction are appended exclusively to that single worker's log buffer. As a result, the main contention point on a shared global log buffer is eliminated, improving system scalability. However, such performance improvements come with a trade-off: commit processing becomes significantly more complicated due to dependency tracking, as we will discuss next.

Decentralized logging: Complex commit processing. In decentralized logging, commit processing becomes more complex due to transaction dependencies: As illustrated in Figure 3(b), log records from dependent transactions may end up in separate buffers. Consider the same example as before: transaction TX1, which inserts a new tuple to page P_a but has not yet completed its execution. After that, transaction TX2 updates another tuple in page P_a , 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. In practice, dependency chains are far more complex and span multiple log buffers; thus, ensuring transaction durability becomes considerably more complicated, as the system must verify that both the transaction's own log entries and all preceding transactions it depends on – which are managed by other workers – are fully persisted before allowing it to commit.

Decentralized logging: Commit conditions. Because of the complexity, it is necessary to formally define the commit conditions of transactions in a decentralized logging-based DBMS. As discussed above, these conditions must ensure both log durability and dependency durability – meaning a transaction cannot safely commit until its own log entries and all its prerequisite transactions are fully persisted. We can map these dual requirements directly to the transaction lifecycle – illustrated in Figure 3(b) – where the system enforces them during two main state transitions:

- 1 **Log durability** (READY $\Rightarrow$ HARDENED): A READY transaction advances to the HARDENED state only after the system flushes all its log entries to the storage.

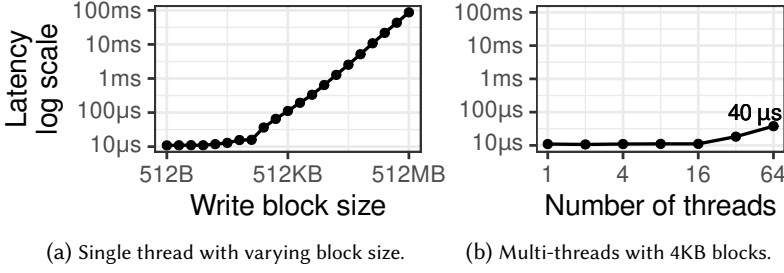


Fig. 4. Average random-write latency of an enterprise NVMe SSD, measured using microbenchmarks.

- ② **Dependency checking** (HARDENED $\Rightarrow$ COMMITTED): A HARDENED transaction reaches the final COMMITTED state only after the system verifies that all transactions it depends on have also committed.

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 [11, 28, 38] – then moving the state of those transactions to HARDENED. Finally, the system calls *commit acknowledgment* to determine the new set of safe transactions (satisfying the *commit conditions*) and acknowledges them as COMMITTED.

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 throughput, as shown in Figure 2, 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 to amortize the high costs of disk I/O. While this design effectively hides the ms-level latency of magnetic disks, it fails to exploit the modern enterprise-grade NVMe SSDs that offers μ s-level write performance, leading to high commit latencies in modern systems. To bridge this gap, we must first establish the behavioral characteristics of enterprise NVMe SSDs in comparison with both magnetic disks and consumer-grade flash storage.

Enterprise-grade SSD characterization. The enterprise SSD class is defined by strict performance and durability guarantees, primarily achieved through certain power-loss protection (PLP) mechanisms. This distinguishes enterprise-grade drives from M.2 NVMe SSDs (used in consumer devices like laptops) that lack equivalent performance guarantees. These enterprise-grade SSDs are common in private data centers, usually in the U.2 form factor, with examples including KIOXIA CM6-R, Samsung PM1733, Intel D7-P5520, and many more. In the cloud, empirical evidence from [27, 48] confirms that storage-optimized cloud instances (e.g., AWS I3 family) provide nearly identical performance profiles, validating these SSDs as a standard primitive for modern data systems.

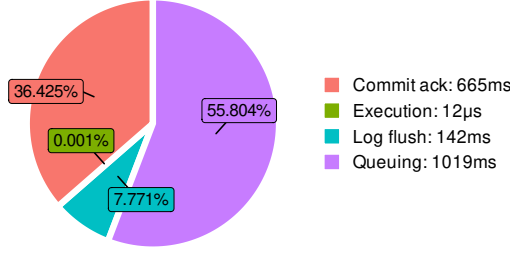


Fig. 5. Latency breakdown of the group commit with decentralized logging in a multi-threaded YCSB benchmark.

Smaller writes (4-16KB). Enterprise NVMe SSDs offer low-latency writes [26, 28, 33, 41]. For example, as Figure 4a shows, our Kioxia SSD achieves random write latencies as low as $11\mu\text{s}$ for a 4KB write unit. Latency stays below $11\mu\text{s}$ 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 μs -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 4b, 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()` [26]. This is because the physical flash 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 [8, 14, 25, 41]. 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 [26, 28, 47, 62, 66, 68, 81]. 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 [11, 32]. In terms of throughput, this implementation is capable of executing millions of transactions per second.

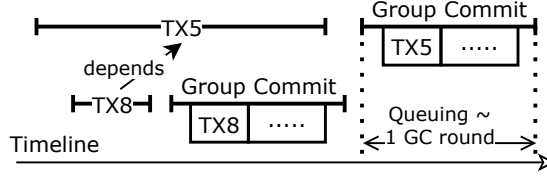


Fig. 6. Queuing time is significant in group commit.
TX8 can only be committed when its dependency, TX5, is committed.

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 μ s 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 4 shows, writing 300MB to SSD takes 45ms in a microbenchmark, significantly increasing the latency for all transactions in that round.

Problem 2: Unithreaded 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 6 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

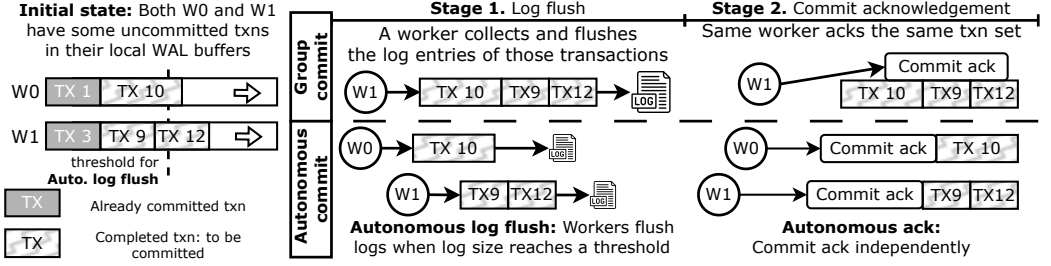


Fig. 7. Traditional group commit vs. our proposed approach, autonomous commit.

second technique, *autonomous commit acknowledgement*, 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 and improve the robustness of 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` [12, 28, 32, 54, 67, 79], 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 acknowledgement* stage. Figure 7 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.

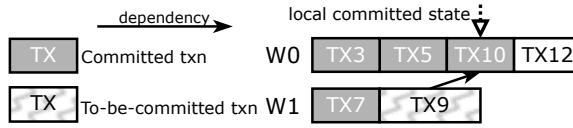
3.2 Autonomous Commit Acknowledgment

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. 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 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 7 (Stage 2) shows 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 [32, 76, 80]:



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 to the next worker, leaving the transaction to wait until the next round. This explains why 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 6, 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.

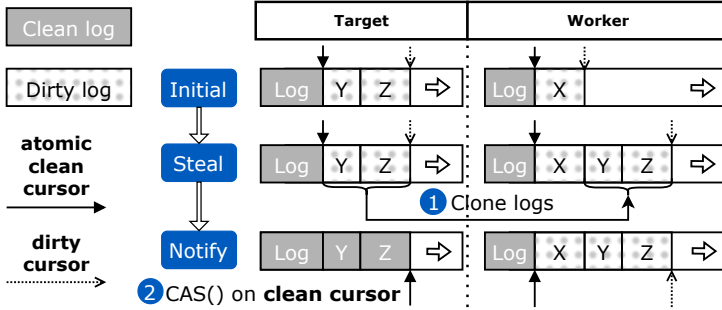
Tackling *queuing*: Frequent acknowledgment. Transactions with remote dependencies require workers to trigger commit acknowledgment *frequently* such that the worker owning those dependencies will perform *log flush* to harden them. However, performing commit acknowledgment too often (e.g., after every 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 $6\mu\text{s}$ to execute, the average delay before being hardened is $300\mu\text{s}$, 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.

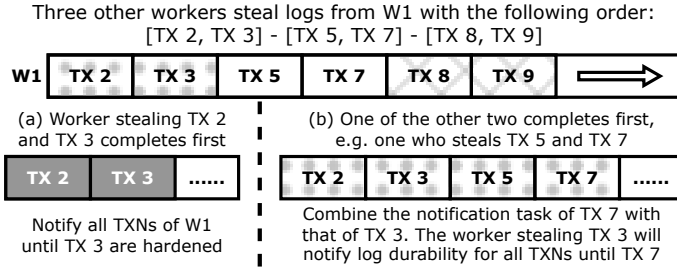
Solution: Log stealing. Instead, we introduce log stealing as a solution to mitigate excessive delays caused by workers accumulating log entries. We trigger stealing operations only at the end of a transaction's execution. Upon the end of a transaction's execution, the worker loops through all target 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 persists 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 CPUs². 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 [15, 53], as is illustrated in the following figure:



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 Durability Delay Under Low Load

Problem. Combining all techniques described so far ensures fast *log durability*³ 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 [10, 18, 72]. 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.

²Modern server CPUs usually comprise multiple dies [1, 6], in which inter-die communications are slower than intra-die communications.

³While log durability is essential for achieving low commit latency, it alone does not guarantee that a transaction is committed in decentralized logging, cf., Section 5.

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 $50\mu s$ 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 protocol successfully eliminates the many bottlenecks analyzed in Section 2.3, it exposes a significant new bottleneck: The contention and synchronization overhead of managing transaction states. In this section, we explain why this overhead was previously hidden, formalize the requirements of transaction state management, and show that a simple design is sufficient for the proposed requirements.

Requirements for transaction state management. Transaction queue management is a previously hidden overhead relative to the dominant overheads found in traditional ARIES-based and/or decentralized logging systems (cf. Section 2.3). Once the autonomous commit protocol eliminates those primary performance issues, the commit processing engine becomes highly parallel, and the management of shared transaction states across threads emerges immediately as a primary bottleneck. This bottleneck is directly caused by the high-frequency, cross-thread interactions required in such environments, with scenarios including:

- **Asynchronous BLOB logging:** Committing threads must batch-persist BLOBs managed by concurrent worker threads before acknowledging those transactions as committed [66].
- **Autonomous commit acknowledgment:** As described in Section 3, threads must frequently poll the transaction states of other workers to evaluate transaction dependencies.
- **Certifier-based concurrency control:** These protocols require transactions to validate their eligibility to commit by evaluating the dependency metadata of concurrent transactions (e.g., "dangerous structures" in SSI [16, 69] or watermark in SSN [77, 78]).

Two key requirements arise from these scenarios: Transaction queues must *provide cross-thread visibility* and *maintain performance under high-frequency access*. Existing designs fail to meet these requirements: latched shared queues [2, 11] become a contention bottleneck, and private per-worker queues [38] cannot provide the cross-thread visibility at all.

Solution: SPSC queue. At first glance, the transaction queue appears to be a multi-consumer interface – where any thread may access its enqueued transactions. One interesting observation: For any specific queue, at most *one* thread acts as the consumer at any given moment – which effectively maps the workload to a Single-Producer, Single-Consumer (SPSC) model. Another important observation is that the transaction state is typically small (e.g., ≈ 128 bytes in PostgreSQL [4]). Consequently, even a modest 10MB queue can accommodate more than 80K transactions, making a simple bounded queue sufficient in practice. Furthermore, the number of queued transactions is typically much smaller, as the autonomous commit protocol aggressively commits transactions to maintain low commit latency. We, therefore, suggest that transaction queues can be efficiently implemented using a bounded circular buffer.

```

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

```

Listing 1. Lock-free queue of serialized transactions.

Implementation. Listing 1 shows our queue implementation. For simplicity, we do not include the code for the queue operations. First, ❶ we ensure that all serialized states are aligned to the CPU cache line to prevent false sharing. Next, ❷ the system waits until there is enough space to serialize a new transaction state into the queue. After that, ❸ we serialize the state 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, ❹ the system can iterate through the completed transaction set efficiently to verify their commit state. This batching dequeue mechanism also reduces cache coherence traffic [82], thus improving performance.

5 High-Performance And Robust Decentralized Logging

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 in system performance and robustness: Different tracking mechanisms (1) require varying amounts of information to be tracked [76, 80] and (2) dictate how workers assess the durability of a transactions, i.e., by evaluating whether remote dependencies are committed. Consequently, not only affects the efficiency of the commit processing subsystem differently, but each variant also influences the correctness and performance of the recovery subsystem.

In this section, we will begin by reviewing existing dependency tracking strategies, highlighting their advantages and drawbacks. We then introduce *Commit Causality Clocks* (C3), which integrates the strengths of all other approaches without their limitations, laying the foundation for robust recovery which will be detailed in Section 6.

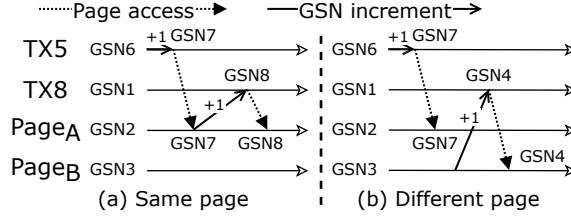


Fig. 8. Dependency tracking with GSNs.

5.1 Existing Decentralized Logging 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 [83] and Taurus [80]. DistDGCC [83] includes the whole dependency graph in log records, incurring considerable runtime and storage overheads. Taurus [80] achieves the same effect by using *Log Sequence Number vectors* (LVs), which is based on *vector clocks* [22, 60], 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. This excessive metadata also affects synchronization: retrieving the committed state of all workers incurs approximately 100 cache misses, which is very expensive.

Advantages of total ordering. Wang et al. proposed using *Global Sequence Numbers* (GSNs) based on Lamport clocks [49], in decentralized logging to achieve a global total order of transactions [76]. 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 8 (left) illustrates, whenever a transaction accesses a page (for both reads and writes), it synchronizes the local GSN to $\max(\text{localGSN}, \text{pageGSN})$. When a new log 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) [11, 32], 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 8 (right) depicts.

Straggler problems of 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 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 [76].

Explanation. Why is this commit condition weaker than that of PCT? We demonstrate this with an example in Figure 9, 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.

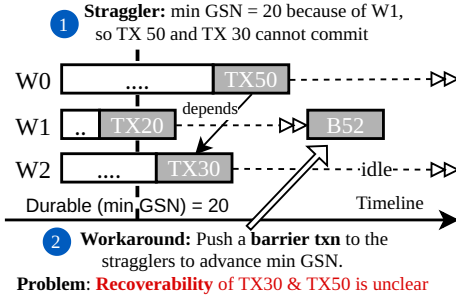


Fig. 9. Barrier transactions mitigate the straggler problem at the cost of potential transaction loss.

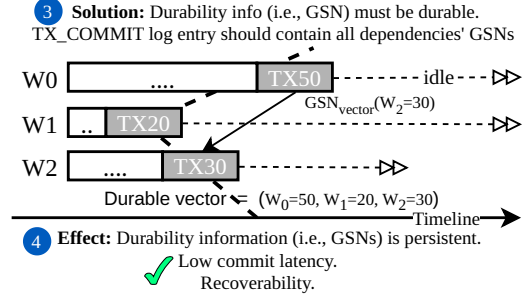


Fig. 10. In contrast, C3 addresses the straggler issue while ensuring recoverability for all transactions.

Barrier transaction to address the stragglers. The straggler problem arises because slow or idle workers do not advance their GSNs during inactivity, as no transactions are being processed. One way to address the issue is to use *barrier transactions* [63]. 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.

Example for barrier transaction. Figure 9 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, the minimum GSN becomes 52, so the system can then commit TX30 and TX50 with a minimal waiting time.

Lost transaction. A critical issue of barrier transactions is the risk of lost transactions. Specifically, transactions that have already been committed and acknowledged may be lost and become unrecoverable upon system crashes. As shown in Figure 9, the minimum durable GSN is 20, yet both TX30 and TX50 are acknowledged due to barrier transactions. Suppose the DBMS crashes after 2 and then restarts. In this case, the system has no way of determining whether these two transactions were truly committed. This is because (1) total ordering would not recognize those two transactions as committed, and (2) barrier transactions do not generate logs, hence the recovery subsystem cannot verify whether any barrier transactions were issued to acknowledge them. As a result, these transactions will be lost if the DBMS crashes, highlighting a fundamental flaw of the barrier transactions.

5.2 Commit Causality Clocks

As discussed above, precise and efficient causality tracking is essential for ensuring durability and low-latency commits. However, existing precise tracking techniques often incur high storage and runtime overheads – an issue that total ordering approaches avoid. In this section, we introduce *Commit Causality Clocks*, a novel mechanism that preserves the simplicity and efficiency of total ordering while enabling precise causality tracking, making it a compelling design for decentralized logging.

Intuition. A key but often overlooked characteristic of the GSN is that each worker’s local GSN precisely reflects its local durability knowledge. Specifically, all modifications by a worker satisfy

the following condition:

$$\text{modifications' GSN} \leq \text{owning worker's local GSN}$$

This observation leads to a question: What if we leverage a vector of *local GSNs* similarly to vector clocks? Such a design would have the advantages of total ordering (i.e., minimal metadata overhead and straightforward implementation) with those of precise causality tracking (i.e., explicit dependency tracking).

High-level idea. We propose *Commit Causality Clocks (C3)*, a novel decentralized logging scheme that accurately captures remote dependencies across workers. Similar to vector clocks, C3 maintains a vector of logical clocks, each representing the knowledge of a corresponding worker. However, unlike traditional vector clocks with logical timestamps that operate independently, the local clocks in C3 correspond to each worker's *local GSN* – which is periodically synchronized with the GSN. Consequently, C3 combines the best of both worlds – offering precise causality tracking like Taurus while maintaining the lightweight nature and simplicity of total ordering approaches.

Design. Capturing transaction causality across worker threads relies on the following data:

- (1) For each page, we maintain two metadata fields:

- L_{last} : log that contains information about the most recent modification of this page.
- W_{last} : worker ID generating that log entry.

These values do not need to be persisted and can be kept in the in-memory buffer frame [88]. Notably, they are already required for database checkpointing and thus do not introduce additional metadata overhead.

- (2) Each transaction maintains a *GSNvector* data structure, which is initially empty.

When a transaction accesses a page, it checks if the committing transaction belongs to W_{last} . If it is, then the DBMS is aware that the committing transaction only depends on prior operations from the same worker and, hence, proceeds without additional bookkeeping. Otherwise, it records the dependency information – W_{last} and L_{last} – in the transaction's *GSNvector*, which helps determine *precisely* whether all dependencies of that transaction have been committed.

Example. For instance, with the scenario illustrated in Figure 10, the *GSNvector* of TX50 is:

$$\text{Dependencies} = [(L_{last} = TX_COMMIT(TX30) = GSN30, W_{last} = W2)]$$

As such, during commit acknowledgment (cf., Section 3.2), W0 is aware that TX50 depends on GSN30 (i.e., TX30) of W2. Therefore, because the committed state of W2 is GSN30, W0 can confirm that TX50 has been committed, avoiding the *straggler* problem similar to other *precise causality-tracking* mechanisms.

Log metadata overheads. One interesting observation is that causality tracking is only required for the commit operations to ensure transactional durability. Consequently, only the commit log entry needs to store the *GSNvector* to ensure recoverability, as shown in Figure 10 ③. All other log entries only require a single GSN, similar to the original GSN protocol [76]. As a result, C3 introduces minimal metadata overhead, making it nearly as lightweight as the GSN protocol while still capturing complete dependency information similar to Taurus [80] and thus avoiding the *straggler* problem.

Optimization. To further reduce metadata overhead, C3 can leverage the concept of $GSN_{flushed}$, which tracks the highest GSN for which all log entries have been flushed [11, 32]. During a page access, the transaction compares if *page's* $L_{last} \leq GSN_{flushed}$; if it is, the DBMS recognizes that the last modification is already durable and excludes it from the *GSNvector*, further minimizing metadata overhead. The key conceptual differences between C3 and existing decentralized logging approaches are shown in Table 1.

Table 1. Conceptual comparison of decentralized logging designs.

	metadata overheads	latency in realistic wl.	lost txns in recovery	impl.
DistDGCC [83]	high	low	no	hard
Taurus [80]	high	low	no	hard
GSN/RFA [32, 76]	low	high	no	easy
GSN/RFA+Barrier txn [63]	low	low	yes	easy
<i>Commit causality clocks</i>	low	low	no	easy

6 Design and Implementation of Correct and Efficient Recovery

In this section, we discuss the low-level constraints that autonomous commit and decentralized logging impose on the recovery process, and then detail our design and implementation of a parallel recovery scheme that effectively addresses these constraints.

6.1 Background: Recovery Basis

To understand the challenges of recovery when integrating with autonomous commit and decentralized logging, please let us first revisit how the conventional recovery works. Physiological logging-based recovery (e.g., ARIES) consists of three phases: log analysis, redo, and undo. Each phase must be completed before the next one begins.

- (1) *Log analysis*: The DBMS analyzes the log entries to identify dirty pages in the buffer pool, as well as to classify *winner* transactions (i.e., completed transactions) and *loser* transactions (i.e., transactions that were active at the time of the crash). This phase begins with the *log materialization* step, which involves loading logs from durable storage into main memory, followed by the analysis phase, which identifies loser and winner transactions.
- (2) *Redo*: The DBMS replays all actions in order of their *Log Sequence Numbers* (i.e., GSN in case of C3-based decentralized logging), including those of aborted transactions.
- (3) *Undo*: The DBMS reverses the effects of transactions that were not committed before the crash.

From the recovery perspective, an aborted transaction is a special case of a normal (committed) transaction. When a transaction is aborted – whether by the user or the concurrency control protocol – it generates compensation log records (CLRs) [61]; afterward, the DBMS can proceed with the next transaction without flushing the log entries of the aborted transaction to storage because:

- If the log entries of the aborted transaction (including its CLRs) are not made durable, the recovery subsystem treats it like any other incomplete transaction and fully *undoes* its effects.
- If the aborted transaction is persisted in storage, the recovery subsystem *redos* all its log entries, including CLRs, just as it would for a committed transaction. Afterward, the recovery subsystem acknowledges that the previous database session has already rolled back the transaction; hence, it continues undoing normally with other transactions.

6.2 Parallel Log Materialization in Autonomous Commit Systems

Parallel recovery in existing architectures. Previously, systems typically store log entries across multiple files [4, 32, 36, 39, 61, 71, 86]. This design naturally enables parallel log materialization, as each worker thread can independently process a different log file. Additionally, file systems

maintain rich metadata, allowing userspace applications (i.e., DBMSs) to efficiently detect log tails, thereby simplifying recovery.

Recovery without file systems: New challenges. The main problem with using the file system as the middleware between the database system and the NVMe SSDs is the high cost of persisting logs [26, 28]; this overhead is why autonomous commit relies on the `O_DIRECT` flag to bypass file systems, achieving microsecond-scale commit latency. However, completely removing the file system from the logging path introduces two new challenges for recovery. First, the recovery subsystem can only operate directly with (typically one) block devices, making it unclear how to effectively implement parallel log materialization, i.e., each thread works with one log file as in previous designs. Second, the DBMSs also do not know when to stop log materialization because they cannot detect the end of the log region: previous approaches do not have this issue because they can check the size of each log file before loading it into memory. Retrieving the entire SSD into memory is impractical: it is slow, consumes excessive SSD bandwidth, and has many other drawbacks.

Addressing 1st problem: Organize logs in large blocks. To allow parallel log materialization, the logs must be structured in a way that enables efficient parallel processing. One approach is to organize logs through indexing [71], but this incurs substantial bookkeeping, including continuous log reorganization through merging and compaction, reliance on group commit, and many more. Instead, our solution is to logically divide the storage medium into a list of log I/O blocks (illustrated in Figure 11 ①), where each block is logically defined in the form `[start_offset : end_offset]`. For example, a 1TB NVMe SSD can be divided into 1024 blocks of 1MB each: `[0 : 1MB]`, `[1MB : 2MB]`, ..., `[1023MB : 1024MB]`. During autonomous log flushes, every worker writes to the assigned I/O block until it becomes nearly full. If the remaining space is insufficient for the next log flush, the worker requests a new I/O block from the DBMS, leaving the unused space as logical padding. This padding naturally enables random read access, facilitating *parallel log materialization*. An additional benefit is that beyond supporting autonomous log flush – where each worker manages its own I/O blocks, this design is also compatible with other commit processing techniques, such as group commit.

Addressing 2nd problem: Log tail validation with signature. The other key challenge is determining when to stop reading logs. Without a file system, log file metadata is unavailable, making log tail detection an open question. Our solution, inspired by SQLite’s magic header [9], is to include additional metadata, a *signature*, before flushing each batch of dirty logs⁴. Another way to look at the signature is that it serves as a lightweight, early-filtering mechanism designed to quickly identify the log tail without computing expensive block-level hashes.

Failure model and torn writes. A potential concern with this scheme is the scenario where a torn write coincidentally produces a valid signature. Should such a false positive occur, the recovery subsystem safely resolves the resulting ambiguity through a two-step verification:

- **Step 1 – Early Signature Filtering:** During sequential scans of the log I/O blocks and the log batches they contain, the recovery engine validates each batch’s signature. If the signature is missing or invalid, the engine immediately stops processing the current log batch and terminates the I/O process, avoiding unnecessary storage reads.
- **Step 2 – Per-Entry Checksum Validation:** If a torn write accidentally mimics a valid signature (a false positive), the batch passes the initial filter. However, as recovery threads parse the batch, they strictly validate the independent checksum embedded within every

⁴Another alternative approach is to include the signature at the beginning of every I/O block. The main reason we couple signature with log flushes instead is to re-use code and simplify overall implementation.

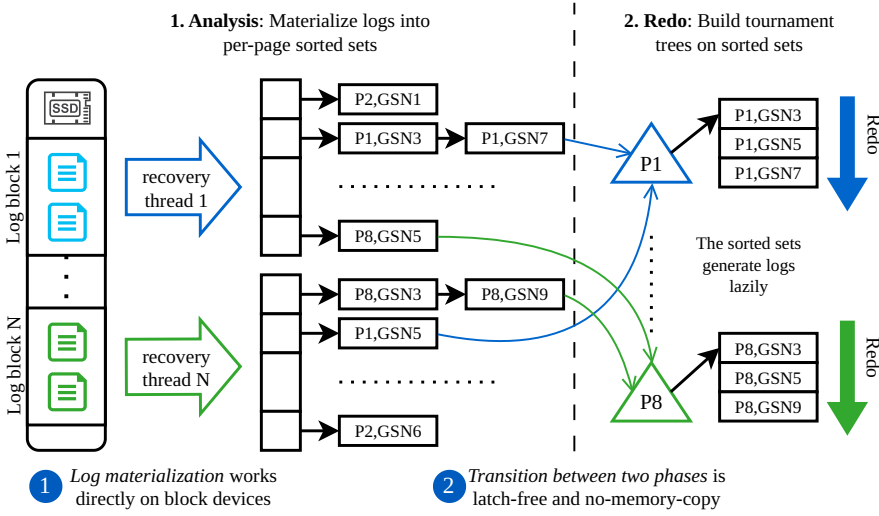


Fig. 11. Parallel log analysis and redo recovery (Undo phase is not shown). Our design ① organizes logs into multiple logical blocks, enabling logging and recovery to bypass the file system and operate directly on block devices, as described in Section 6.2. In addition, as in Section 6.3, ② we exploit the characteristics of log entries to eliminate the synchronization step between the *analysis phase* and the *redo phase* – which previously requires latching and log cloning – further optimizing the performance of parallel recovery.

individual log entry. This check immediately exposes the malformed logs, stopping the recovery process for this block immediately.

Consequently, any partial or corrupted data that sneaks past the initial signature filter is caught at log-entry granularity and automatically discarded. And because the recovery engine never replays these invalid records, system maintains its correctness. Ultimately, the signature works as a performance heuristic for efficient log-tail detection, leaving data integrity verification to the per-entry checksums.

6.3 Low-Level Optimization for Parallel Recovery

Importance of parallel recovery. Parallel recovery is crucial for reducing recovery time, as it splits the recovery work across multiple worker threads. Consequently, this technique has been adopted in many systems, including Silo [74, 86], LeanStore [32, 52], and a recent proposal for PostgreSQL [3].

Performance problem of prior designs. A subtle but important limitation in these designs arises from the synchronization step between the *log analysis phase* and the *redo phase*. In prior implementations, this step requires moving log entries between threads, i.e., from the log reader thread’s buffer to the redo thread’s buffer [3, 32, 52, 86]. This process involves latching and memory copying, which constrains parallelism and can reduce overall recovery efficiency.

Observation: Logs form multiple sorted lists. A key insight often neglected in previous works is that every log flush (i.e., either from group commit or autonomous commit) produces multiple sorted sequences of log entries – ordered by $[PID, GSN]^5$. As shown in Figure 11 (left), together with the parallel log materialization design in Section 6.2, every worker now maintains multiple

⁵Log stealing technique in Section 3.4 makes this ordering constraint more complicated, additionally requiring the TransactionID that made the modification.

sorted sets of logs. Consequently, the problem of redo recovery reduces to a familiar one: how to efficiently merge multiple sorted sets without relocating data, thereby avoiding log buffer latching overhead.

Solution: Lazy tournament tree. Our design, illustrated in Figure 11 (2), integrates the concepts of tournament trees [19, 23, 46] to answer this question. Specifically, during log analysis and materialization, each worker selects random I/O blocks to materialize their log entries into multiple thread-local per-page sorted sets, as discussed earlier. Subsequently, in the redo recovery step, a per-pageID tournament tree is constructed from these sorted sets. Crucially, the tournament trees *store only references* to log records rather than copies. This approach allows redo recovery to avoid cloning log entries from the buffers populated during the *log analysis* phase, eliminating unnecessary data movement and log buffer latching.

7 Evaluation

Experimental setup. We integrate autonomous commit, denoted as *Our*, and all existing commit processing protocols into LeanStore, an open-source storage engine optimized for multi-core CPUs and NVMe SSDs [7]. The dependency tracking mechanism used for the evaluation is C3 – Section 7.2 will experimentally show the performance differences between C3 and the state-of-the-art GSN/RFA tracking variant [63]. By default, the buffer pool is dynamically adjusted to ensure that working datasets always fit in memory, unless stated otherwise. The isolation level used in all experiments is *read uncommitted*. The primary evaluation goal is to stress-test the commit processing subsystem, particularly its dependency tracking mechanisms. We therefore run LeanStore in the *read uncommitted* isolation level, which eliminates execution-phase overheads (e.g., lock contention), thereby maximizing transaction throughput, and making it the ideal experimental setup for our goal. We note that, the proposed protocols – autonomous commit and commit causality clocks – work independently of the transaction execution phase, hence perfectly support stronger isolation levels such as *serializable*.

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. This is the latency-optimized variant.
- *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 7.6 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 *TradQueue*. Another competitor is flush pipelining [11, 39], which runs group commit in a background thread, denoted as *FlushQueue*. We implement all strategies in the same test system to isolate the conceptual differences from incidental differences.

7.1 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

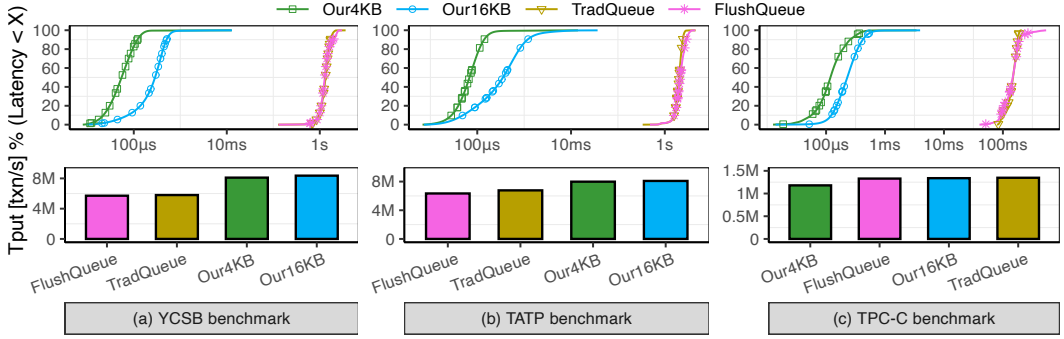


Fig. 12. Close-looping benchmarks: Stressing both throughput and latency.

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 12.

YCSB: Competitor latency. As Figure 12a (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 119μs, 14285× lower than that of *TradQueue*, which provides the best 90p latency amongst two 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? As we can see from Figure 12a, two competitors provide limited throughput because of the logging subsystem. 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. In short, workers have to wait regularly to have enough free space in the log buffer, 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, 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 39.7%.

TATP summary. As shown in Figure 12b, 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 12c, autonomous commit is also better than all competitors. Specifically, *Our4KB* and *Our16KB* provide 704× and 427× lower 90p latency, respectively, compared to the best competitor, *TradQueue*. The differences between autonomous commit and the competitors are less pronounced than in the previous benchmarks

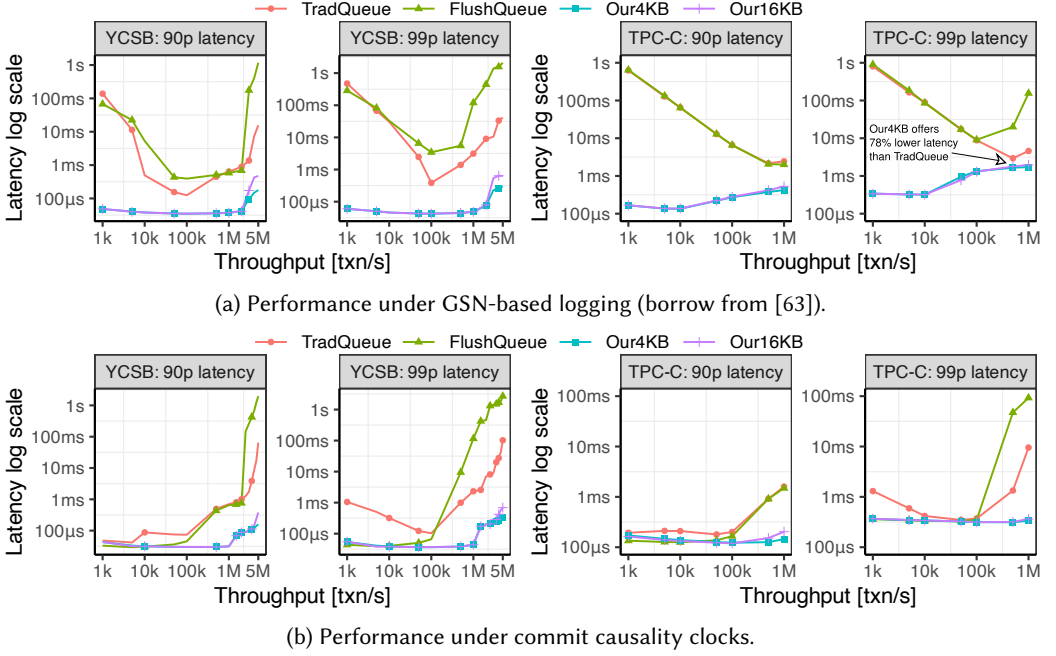


Fig. 13. Open-looping benchmarks.

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. However, as we will show later in Section 7.4, if we increase the number of threads, all autonomous commit variants will be more performant than all competitors in terms of throughput.

Autonomous commit: Effects of log flush unit. Theoretically, a smaller log flush unit should result in better commit latency. Figure 12 proves this statement by showing that *Our4KB* results in lower latency compared to *Our16KB*. However, the latter provides higher throughput in complicated workloads such as TPC-C. We will further delve into the log flush unit in Section 7.5.

7.2 Open-Loop Benchmark

To better simulate real-world scenarios with frequent user think times, we integrate the *open-system model* [10, 72] with this benchmark. We use both YCSB and TPC-C workloads in this benchmark, with similar configurations as in previous experiments.

Latency advantages of C3. Figure 14 highlights this advantage by presenting the commit latency of the flush pipelining protocol under TPC-C workloads. As shown, flush pipelining experiences high commit latencies with GSN/RFA logging due to the straggler problem [63], which is most extreme when the configured throughput is as low as 100 transactions per second. In contrast, C3 logging mitigates this issue, enabling flush pipelining to maintain low commit latencies at low throughput.

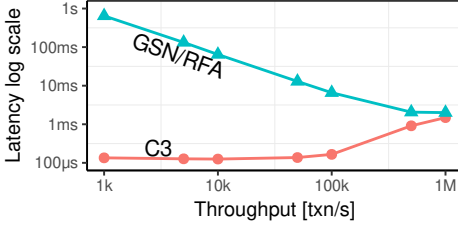


Fig. 14. C3 avoids the straggler problem, improving commit latency of flush pipelining protocol in TPC-C workloads.

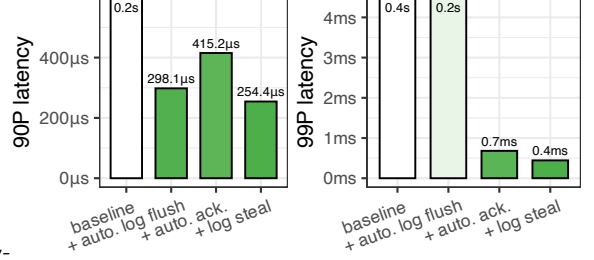


Fig. 15. Impact of different optimizations on TPC-C latency.

Autonomous commit. As Figure 13 depicts, *Our4KB* and *Our16KB* consistently achieves the lowest commit latencies in both YCSB and TPC-C, with its 99p latency remains below 100μs for YCSB and under 1ms for TPC-C. This demonstrates that C3 effectively eliminates the straggler problem. Specifically, when throughput is low, workers in autonomous commit frequently trigger force commits during system idle periods. With C3, each worker is aware when all dirty logs have been flushed, allowing transactions to be acknowledged promptly. We note that C3 also outperforms the barrier transaction technique in terms of commit latency. In TPC-C experiments, *autonomous commit* & C3 consistently achieves 99p latencies well below 1ms. In contrast, *autonomous commit* & *barrier transaction* exceeds 1ms when throughput surpasses 10,000 transactions per second.

Competitors vs. autonomous commit. Thanks to C3, *TradQueue* and *FlushQueue* also avoid the straggler problem [63]. However, as depicted in Figure 13b, their commit latencies rise sharply at higher throughput, with 99p latencies exceeding 100ms. On the other hand, autonomous commit consistently maintains low commit latencies across all workload configurations.

Group commit vs. flush pipelining. The key difference between group commit and flush pipelining is whether to run the group commit task in a background thread. As Figure 13b shows, the differences between these two are significant, unlike the experiments in Figure 12. Specifically, at low throughput, group commit exhibits higher latencies because commit tasks are only triggered upon transaction completion, which does not occur during system inactivity. Flush pipelining, in contrast, continuously executes commit operations in the background, ensuring timely transaction acknowledgments. However, at high throughput, flush pipelining suffers from excessive latencies due to CPU oversubscription by the background commit thread. Because the transaction scheduler (which is in the hot path of the workers) is compute-intensive [5, 18], hence workers will consume CPU resources considerably and thus likely preempt the background commit thread, a finding aligns with numerous previous studies [21, 28, 38, 40, 75].

7.3 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 (with the lock-free transaction queue in Section 4) as the baseline and end with *Our4KB* variant. Step by step, we add additional features: autonomous log flush in Section 3.1, autonomous commit acknowledgment in Section 3.2, and log stealing optimization in Section 3.4. For a realistic benchmark, we use TPC-C workload with the same parameters as the previous benchmarks. Because all protocols give similar throughput in TPC-C (as in earlier experiment), we only show the latency patterns here. Figure 15 shows the ablation study for 90p latency (left) and 99.9p latency (right).

Autonomous log flush: Eradicating I/O spikes. By moving the log flush to the workers, the background thread is responsible only for commit acknowledgment, thereby significantly reducing

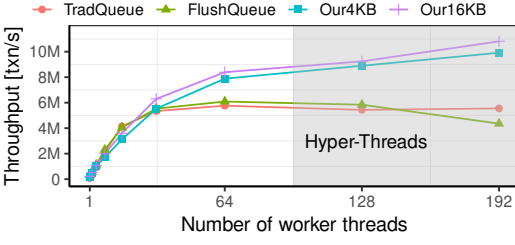


Fig. 16. Scalability benchmark.

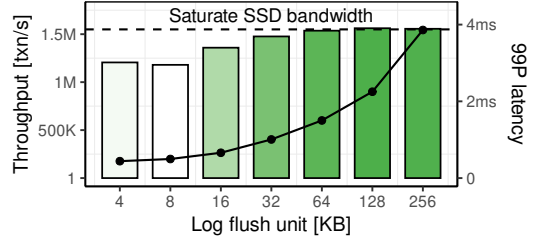


Fig. 17. Log flush unit benchmark.

90p latency by 631 $\times$. 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 more lightweight, it can initiate many more commit rounds, thereby considerably decreasing queuing time.

Autonomous log flush: Tail latencies. However, autonomous log flush exhibits a high 99p latency, approximately half that of flush pipelining. The primary cause is CPU oversubscription on the background commit thread: At times, a significant number of transactions accumulate in the queue because the background commit thread is preempted out and can not acknowledge transactions. This phenomenon leads to substantial *queuing* delays and *commit acknowledgment* overheads, ultimately increasing latency.

Autonomous acknowledgment: Eliminating high 99p latency. By incorporating autonomous commit acknowledgment, we effectively eliminate high tail latency. This improvement stems from two key advantages:

- **Avoiding CPU oversubscription:** Removing the background thread prevents CPU oversubscription, ensuring that acknowledgment tasks are triggered without OS scheduling overheads.
- **Parallelizing commit acknowledgment:** Autonomous acknowledgment distributes the expensive acknowledgment process across multiple workers, allowing transactions to be acknowledged promptly rather than idling in transaction queues.

Stealing logs to optimize latency. As shown in Figure 15, the 90p latency of autonomous commit without log stealing optimization is slightly higher than that of autonomous log flush. This overhead arises from the additional synchronization from gathering the committed state of workers required for commit acknowledgment. With log stealing optimization, autonomous commit reduces the 90p latency by 38.7% and the 99p latency by 34.9%, ensuring low-latency durable commits.

7.4 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 16.

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 unithreaded 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 10.8 million transactions per second with 192 threads.

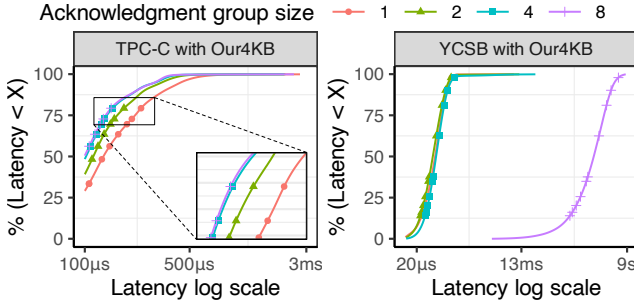


Fig. 18. Acknowledgment group size experiment.

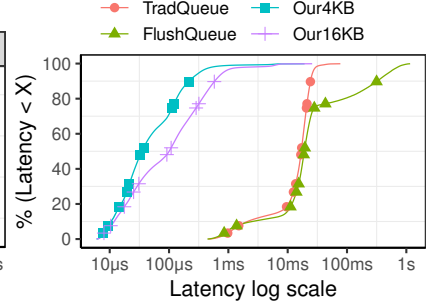


Fig. 19. Mixed-workloads benchmark.

7.5 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 256 KB.

Result. As shown in Figure 17, a 4 KB log flush unit results in the lowest commit latency. This is because smaller log flush units trigger *log flushes* more frequently, allowing committed GSNs to advance more quickly. At the same time, Figure 17 also demonstrates that autonomous commit can saturate SSD bandwidth with a 64 KB log flush unit, i.e., the throughput can not increase much with a higher log flush unit. Overall, a 4 KB log flush unit offers the best balance – delivering high throughput while maintaining the lowest commit latency, ensuring a smooth user experience.

7.6 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. [38]. The autonomous commit variant we use is *Our4KB*.

Result. As Figure 18 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.

7.7 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

Fig. 20. Per-transaction logging overhead of C3 compared to other TO designs. (measured in bytes per transaction)

Variant	GSN	GSN/RFA	C3
YCSB	339	339	355
TATP	227	187	203
TPC-C	5798	5718	5767

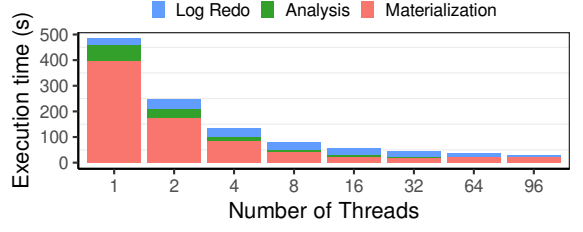


Fig. 21. Recovery benchmark.

competitors, *TradQueue* and *FlushQueue*, remain unaffected by this configuration. The experimental results are presented in Figure 19.

Result. Both *TradQueue* and *FlushQueue* provide acceptable 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 execution time of transactions is several times slower than in previous experiments, leading to significantly lower log volume and lower number of transaction objects. Despite that, autonomous commit still significantly outperforms both traditional group commit and flush pipelining, especially *Our4KB* whose 90p latency is 111.4× and 1489.2× lower than that of *TradQueue* and *FlushQueue*, respectively.

7.8 Logging Overheads

In this section, we compare the logging overhead of C3 with other decentralized logging designs across three workloads: TPC-C, YCSB, and TATP. The workload configurations remain the same with those in Section 7.1. For this experiment, we use the autonomous commit variant *Our4KB*.

C3 vs. total ordering variants. As illustrated in Figure 20, the logging overhead of C3 is comparable to that of total ordering approaches, thanks to its efficient design. That is, most log entries contain only a single clock (similar to GSN-based approaches), while only the TX_COMMIT log maintains a vector of clocks. Even then, this vector remains minimal, as it stores only the GSNs of the workers on which the being-committed transaction depends. This additional metadata incurs negligible overhead compared to *operational* log entries, which store undo and redo information.

C3 vs. PCT variants. In contrast, two other PCT logging approaches, DistDGCC [83] and Taurus [80], introduce considerable storage overhead. Even the more lightweight design, Taurus, incurs substantial logging costs, necessitating the use of compression techniques to alleviate the issue [80]. Despite these optimizations, Taurus is still several folds more expensive than C3: while Taurus logs a complete clock vector for each entry, C3 only stores a single clock representing the local GSN.

7.9 Recovery

The objective of this experiment is to evaluate whether our design can (1) correctly identify the tail of the log when log records are stored directly on block devices, and (2) effectively parallelize all phases of the recovery procedure. In this experiment, we employ a lightweight YCSB workload that generates roughly 28 GB of logs during normal operation. Afterward, we restart the database and measure the recovery time. The number of recovery threads is varied to demonstrate that our proposed design, described in Section 6, effectively parallelizes the recovery process. Note that because we run everything under the *read uncommitted* isolation level, the recovery process does not include an *undo* phase. All experiments are conducted under the *read uncommitted* isolation level as discussed earlier. This setup remains representative of performance under stronger isolation levels because the overhead of the UNDO phase – incurred by transaction aborts in higher isolation

levels – is negligible compared to the dominant costs of recovery, such as log I/O and REDO processing, as discussed in [24, 61].

As shown in Figure 21, the log materialization phase is consistently the most expensive step. This is because DBMSs load all log records into memory (because log size is bounded) before the actual recovery begins, making the process I/O-bound. Nevertheless, by leveraging the I/O logical block mechanism in Section 6, we not only detect the end of the logs efficiently but also parallelize the expensive log materialization step. Consequently, increasing the number of recovery threads to 96 reduces the total recovery time by a factor of 16.4×, from 484.6s to 29.5s.

8 Related Work

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

Epoch-based group commit. The epoch-based paradigm has been used by many systems for different purposes because of its simple and robust design, from durability as in prefix recovery [57, 70], memory reclamation [15, 31], to synchronization [58]. For transactional commit processing, specifically group commit, this means dividing time into a sequence of epochs, where a batch of transactions is persisted to storage together [74, 86]. While this epoch-based group commit provides high throughput, it results in high commit latency, often in the range of milliseconds. This is because the commit state is driven by the epoch period, which is typically tens of milliseconds [74]. 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 [32, 36, 44, 45, 76, 84]. However, the discontinuation of the Intel Optane project [17, 29], along with the limited availability of NVDIMMs compared to NVMe SSDs [13, 37], 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 the table below:

	throughput	latency	support OOM ¹	require PM ²
Group commit [2, 4]	medium	high	yes	no
Flush pipelining [39]	medium	high	yes	no
Epoch-based [74]	high	high	no	no
PM ² -based [32, 76]	high	low	yes	yes
<i>Autonomous commit</i>	high	low	yes	no

¹ Out-of-memory workloads

² Persistent memory

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. One possible starting point is to integrate with eventual durability [42] to hide the latency of mechanical disks.

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* [50, 51], 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.

Causal orderings in distributed systems. At its core, dependency tracking in decentralized logging resembles causal ordering techniques in distributed environments: GSN originates from Lamport clocks [49], and Taurus adopts vector clocks [22, 60]. More recently, Mathur et al. [59] introduced tree clocks – a vector clock variant that can compute the happens-before partial order optimally. Similarly, distributed prefix recovery [57] and speculative execution frameworks [56] leverage these same causality tracking principles to enable fault-tolerance in modern cloud applications. Tailored for decentralized logging, our commit causality clocks described in Section 5 capture the strong causality of vector clocks while maintaining the constant-size footprint of Lamport clocks, as discussed in Section 7. This efficiency positions C3 as a promising solution for the causality-tracking problem space in the distributed environments mentioned above, an exploration we leave to future work.

Cloud-native data management. The DBMS market has undergone a significant transformation from on-premise to the cloud in the last few years. Following this trend, a key question arises: Is it possible to achieve both high scalability and low-latency durable commit for cloud-native DBMSs, especially with write-intensive OLTP workloads? We believe this is achievable by integrating our solution with a dedicated low-latency cloud-native log service. One particular work following the same design is Amazon MemoryDB [73], an in-memory storage engine backed by a low-latency log service. However, Amazon MemoryDB is proprietary and only exposes Redis APIs, limiting its applicability. Therefore, we believe there are many exciting research opportunities as the building blocks for a full-fledged DBMS optimized for OLTP workloads, a view supported by many previous studies [20, 34, 35, 55, 64, 87].

9 Summary

In this paper, we outlined two key techniques – Autonomous Commit and Commit Causality Clocks – for achieving high scalability and low-latency commits. We also examined the recovery subsystem, detailing the adaptations required to support parallel recovery in DBMSs operating directly on block devices. Our performance evaluation demonstrates that our approach enables fast transaction commits, achieving 90p latencies in the microsecond range across diverse workloads. We believe these techniques offer a compelling solution for high-performance DBMSs. Our implementation is open source and available at <https://github.com/leanstore/leanstore/tree/latency-j>.

Acknowledgments

We are grateful to the reviewers for their thoughtful feedback. Additionally, we also thank Goetz Graefe, Tianzheng Wang, Kaisong Huang, and Michael Haubenschild for many insightful discussions that helped shape this research.

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. Parallel Recovery in PostgreSQL. https://wiki.postgresql.org/wiki/Parallel_Recovery.
- [4] 2023. PostgreSQL source code. https://github.com/postgres/postgres/tree/REL_15_3.
- [5] 2024. BenchBase source code. <https://github.com/cmu-db/benchbase>.
- [6] 2024. Intel® Xeon® 6 Processors. <https://www.intel.com/content/www/us/en/products/details/processors/xeon.html>.
- [7] 2024. LeanStore. <https://github.com/leanstore/leanstore>.
- [8] 2024. NVMe Specifications. <https://nvmexpress.org/specifications/>.
- [9] 2024. SQLite: Database File Format . <https://www.sqlite.org/fileformat.html>.
- [10] 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.
- [11] 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.
- [12] 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.
- [13] 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.
- [14] Mijin An, In-Young Song, Yong Ho Song, and Sang-Won Lee. 2022. Avoiding Read Stalls on Flash Storage. In *SIGMOD Conference*. ACM, 1404–1417.
- [15] 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.
- [16] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable isolation for snapshot databases. In *SIGMOD Conference*. ACM, 729–738.
- [17] 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@OSP*. ACM, 23–30.
- [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] Thanh Do, Goetz Graefe, and Jeffrey F. Naughton. 2022. Efficient Sorting, Duplicate Removal, Grouping, and Aggregation. *ACM Trans. Database Syst.* 47, 4 (2022), 16:1–16:35.
- [20] Haowen Dong, Chao Zhang, Guoliang Li, and Huanchen Zhang. 2024. Cloud-Native Databases: A Survey. *IEEE Trans. Knowl. Data Eng.* (2024), 1–20.
- [21] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proc. VLDB Endow.* 16, 11 (2023), 2769–2782.
- [22] Colin J. Fidge. 1987. *Timestamps in message-passing systems that preserve the partial ordering*. Australian National University. Department of Computer Science.
- [23] Goetz Graefe. 2023. Priority queues for database query processing. In *BTW (LNI, Vol. P-331)*. Gesellschaft für Informatik e.V., 27–46.
- [24] Jim Gray and Andreas Reuter. 1993. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann.
- [25] 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.
- [26] Gabriel Haas, Michael Haubenschild, and Viktor Leis. 2020. Exploiting Directly-Attached NVMe Arrays in DBMS. In *CIDR*. www.cidrdb.org.
- [27] Gabriel Haas, Bohyun Lee, Philippe Bonnet, and Viktor Leis. 2025. SSD-iq: Uncovering the Hidden Side of SSD Performance. *Proc. VLDB Endow.* 18, 11 (2025), 4295–4308.
- [28] 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.
- [29] Jim Handy and Tom Coughlin. 2023. Optane’s Dead: Now What? *Computer* 56, 3 (2023), 125–130.
- [30] 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.
- [31] Michael Haubenschild and Viktor Leis. 2023. Lock-Free Buffer Managers Do Not Require Delayed Memory Reclamation. In *SiMoD@SIGMOD*. ACM, 1:1–1:3.
- [32] 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.
- [33] 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.

- [34] Pat Helland. 2022. Decoupled Transactions: Low Tail Latency Online Transactions Atop Jittery Servers. In *CIDR*. www.cidrdb.org.
- [35] Pat Helland. 2024. Scalable OLTP in the Cloud: What's the BIG DEAL?. In *CIDR*. www.cidrdb.org.
- [36] Jian Huang, Karsten Schwan, and Moinuddin K. Qureshi. 2014. NVRAM-aware Logging in Transaction Systems. *Proc. VLDB Endow.* 8, 4 (2014), 389–400.
- [37] Kaisong Huang and Tianzheng Wang. 2024. *Lessons Learned and Outlook*. Springer Nature Switzerland, 77–87.
- [38] 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.
- [39] 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.
- [40] Ryan Johnson, Ippokratis Pandis, Radu Stoica, Manos Athanassoulis, and Anastasia Ailamaki. 2012. Scalability of write-ahead logging on multicore and multiset hardware. *VLDB J.* 21, 2 (2012), 239–263.
- [41] 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.
- [42] Tejasvi Kashi, Kenneth Salem, Jaemyung Kim, and Khuzaima Daudjee. 2024. Eventual Durability. *Proc. VLDB Endow.* 17, 13 (2024), 4733–4745.
- [43] Jong-Bin Kim, Hyeonwon 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.
- [44] 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.
- [45] Wonbae Kim, Chanyeol Park, Dongui Kim, Hyeonjun 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.
- [46] Donald E. Knuth. 1973. *The Art of Computer Programming, Volume III: Sorting and Searching*. Addison-Wesley.
- [47] 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.
- [48] Maximilian Kuschewski, Lam-Duy Nguyen, Matthias Jasny, Tobias Ziegler, Viktor Leis, and Muhammad El-Hindi. 2026. BtrLog: Low-Latency Logging for Cloud Database Systems. *Proc. VLDB Endow.* 19 (2026).
- [49] Leslie Lamport. 2019. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*. ACM, 179–196.
- [50] 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.
- [51] 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.
- [52] Viktor Leis. 2024. LeanStore: A High-Performance Storage Engine for NVMe SSDs. *Proc. VLDB Endow.* 17 (2024), 4536–4545.
- [53] 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.
- [54] 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.
- [55] Alberto Lerner, Carsten Binnig, Philippe Cudré-Mauroux, Rana Hussein, Matthias Jasny, Theo Jepsen, Dan R. K. Ports, Lasse Thostrup, and Tobias Ziegler. 2023. Databases on Modern Networks: A Decade of Research that now comes into Practice. *Proc. VLDB Endow.* 16, 12 (2023), 3894–3897.
- [56] Tianyu Li, Badrish Chandramouli, Philip A. Bernstein, and Samuel Madden. 2024. Distributed Speculative Execution for Resilient Cloud Applications. *CoRR* abs/2412.13314 (2024).
- [57] Tianyu Li, Badrish Chandramouli, Jose M. Faleiro, Samuel Madden, and Donald Kossmann. 2021. Asynchronous Prefix Recoverability for Fast Distributed Stores. In *SIGMOD Conference*. ACM, 1090–1102.
- [58] Tianyu Li, Badrish Chandramouli, and Samuel Madden. 2024. Performant almost-latch-free data structures using epoch protection in more depth. *VLDB J.* 33, 6 (2024), 1793–1812.
- [59] Umang Mathur, Andreas Pavlogiannis, Hünkar Can Tunç, and Mahesh Viswanathan. 2022. A tree clock data structure for causal orderings in concurrent executions. In *ASPLOS*. ACM, 710–725.
- [60] Friedemann Mattern. 1989. Virtual time and global states of distributed systems. In *Proc. International Workshop on Parallel and Distributed Algorithms*. Holland, 215–226.
- [61] 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.
- [62] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *CIDR*. www.cidrdb.org.
 - [63] 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. *Proc. ACM Manag. Data* 3, 3 (2025), 191:1–191:24.
 - [64] Lam-Duy Nguyen, Gabriel Haas, Muhammad El-Hindi, Viktor Leis, and Philip A. Bernstein. 2027. The Case for Single-Version Concurrency Control in the Cloud. In *CIDR*. www.cidrdb.org.
 - [65] Lam-Duy Nguyen, Sang Won Lee, and Beomseok Nam. 2022. In-Page Shadowing and Two-Version Timestamp Ordering for Mobile DBMSs. *Proc. VLDB Endow.* 15, 11 (2022), 2402–2414.
 - [66] Lam-Duy Nguyen and Viktor Leis. 2024. Why Files If You Have a DBMS?. In *ICDE*. IEEE, 3878–3892.
 - [67] Hamish Nicholson, Periklis Chrysogelos, and Anastasia Ailamaki. 2022. HPCache: Memory-Efficient OLAP Through Proportional Caching. In *DaMoN*. ACM, 7:1–7:9.
 - [68] Jonghyeok Park, Soyeon 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.
 - [69] Dan R. K. Ports and Kevin Grittnner. 2012. Serializable Snapshot Isolation in PostgreSQL. *Proc. VLDB Endow.* 5, 12 (2012), 1850–1861.
 - [70] Guna Prasaad, Badrish Chandramouli, and Donald Kossmann. 2019. Concurrent Prefix Recovery: Performing CPR on a Database. In *SIGMOD Conference*. ACM, 687–704.
 - [71] Caetano Sauer, Goetz Graefe, and Theo Härder. 2018. FineLine: log-structured transactional storage and recovery. *Proc. VLDB Endow.* 11, 13 (2018), 2249–2262.
 - [72] Bianca Schroeder, Adam Wierman, and Mor Harchol-Balter. 2006. Open Versus Closed: A Cautionary Tale. In *NSDI*. USENIX.
 - [73] Yacine Taleb, Kevin McGehee, Nan Yan, Shawn Wang, Stefan C. Müller, and Allen Samuels. 2024. Amazon MemoryDB: A Fast and Durable Memory-First Cloud Database. In *SIGMOD Conference Companion*. ACM, 309–320.
 - [74] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy transactions in multicore in-memory databases. In *SOSP*. ACM, 18–32.
 - [75] Benjamin Wagner, André Kohn, and Thomas Neumann. 2021. Self-Tuning Query Scheduling for Analytical Workloads. In *SIGMOD Conference*. ACM, 1879–1891.
 - [76] Tianzheng Wang and Ryan Johnson. 2014. Scalable Logging through Emerging Non-Volatile Memory. *Proc. VLDB Endow.* 7, 10 (2014), 865–876.
 - [77] Tianzheng Wang, Ryan Johnson, Alan D. Fekete, and Ippokratis Pandis. 2015. The Serial Safety Net: Efficient Concurrency Control on Modern Hardware. In *DaMoN*. ACM, 8:1–8:8.
 - [78] Tianzheng Wang, Ryan Johnson, Alan D. Fekete, and Ippokratis Pandis. 2017. Efficiently making (almost) any concurrency control mechanism serializable. *VLDB J.* 26, 4 (2017), 537–562.
 - [79] 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.
 - [80] 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.
 - [81] Hiroyuki Yamada, Toshihiro Suzuki, Yuji Ito, and Jun Nemoto. 2023. ScalarDB: Universal Transaction Manager for Polystores. *Proc. VLDB Endow.* 16, 12 (2023), 3768–3780.
 - [82] Stephen Yang, Seo Jin Park, and John K. Ousterhout. 2018. NanoLog: A Nanosecond Scale Logging System. In *USENIX ATC*. USENIX Association, 335–350.
 - [83] 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.
 - [84] 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.
 - [85] Jianqiu Zhang, Kaisong Huang, Tianzheng Wang, and King Lv. 2022. Skeena: Efficient and Consistent Cross-Engine Transactions. In *SIGMOD Conference*. ACM, 34–48.
 - [86] 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.
 - [87] 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.
 - [88] Michael Zinsmeister, Lam-Duy Nguyen, Viktor Leis, and Thomas Neumann. 2026. Predictive Translation: High-Performance Buffer Management Without the Trade-Offs. In *SIGMOD Conference*. ACM.