Boosting OLTP Performance with Per-Page Logging on NVDIMM

BOHYUN LEE[*†], Technical University of Munich, Germany
SEONGJAE MOON[*], Sungkyunkwan University, Korea
JONGHYEOK PARK[*], Korea University, Korea
SANG-WON LEE[†], Seoul National University, Korea

When running OLTP workloads on flash SSDs, relational DBMSs still face the write durability overhead, severely limiting their performance. To address this challenge, we propose *NV-PPL*, a novel database architecture that leverages NVDIMM as a *durable log cache*. *NV-PPL* captures per-page redo logs and retains them on NVDIMM to absorb writes from DRAM to SSD. Our *NV-PPL* prototype, deployed on an actual NVDIMM device, demonstrates superior transaction throughput, surpassing the *same-priced* Vanilla MySQL by at least 6.9× and NV-SQL, a page-grained NVDIMM caching scheme, by up to 1.5×. Beyond write reduction, the page-wise logs in NVDIMM enable novel approaches such as redo-less recovery and redo-based multi-versioning. Compared to Vanilla MySQL, redo-less recovery reduces recovery time by one-third, while redo-based multi-versioning enhances the latency of long-lived transactions in HTAP workloads by 3× to 18×.

CCS Concepts: • **Information systems → DBMS engine architectures**.

Additional Key Words and Phrases: Per-Page Logging, NVDIMM, MVCC, OLTP, Transaction

ACM Reference Format:
Bohyun Lee, Seongjae Moon, Jonghyeok Park, and Sang-Won Lee. 2025. Boosting OLTP Performance with Per-Page Logging on NVDIMM. *Proc. ACM Manag. Data* 3, 1 (SIGMOD), Article 17 (February 2025), 28 pages. https://doi.org/10.1145/3709667

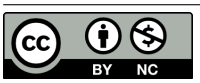
1 Introduction

As Solid State Drives (SSDs) continue to evolve with higher throughput and better cost-effectiveness, storage-centric database systems remain the mainstream and economical choice for most applications [9, 27]. Despite the advancement of modern SSDs, they still exhibit high latency at the microsecond level. For instance, the latency of the Samsung 960 Pro SSD exceeds 70 μs [72]. This latency is significantly higher than that of DRAM, which operates at the tens-of-nanoseconds level, typically ranging from 50 to 100 ns.

Write durability overhead This leads to non-trivial durability overhead when updated pages in the DRAM buffer must be persisted on the SSD upon checkpointing and buffer eviction. Despite the asynchronous nature of storage writes facilitated by write-ahead logging (WAL), they remain a major performance bottleneck for write-intensive Online Transaction Processing (OLTP) workloads on SSDs. Notably, excessive writes to flash storage devices with asymmetric read and write speeds

[*]They equally contributed for multi-versioning, basic *NV-PPL*, and recovery, respectively, and are listed in alphabetical order of the last name.
[†]Part of the work was done while in Sungkyunkwan University.

Authors' Contact Information: Bohyun Lee, bohyun.lee@tum.de, Technical University of Munich, Munich, Germany; Seongjae Moon, sungjae.moon@skku.edu, Sungkyunkwan University, Suwon, Korea; Jonghyeok Park, jonghyeok_park@korea.ac.kr, Korea University, Seoul, Korea; Sang-Won Lee, swlee69@snu.ac.kr, Seoul National University, Seoul, Korea.



ACM 2836-6573/2025/2-ART17
https://doi.org/10.1145/3709667

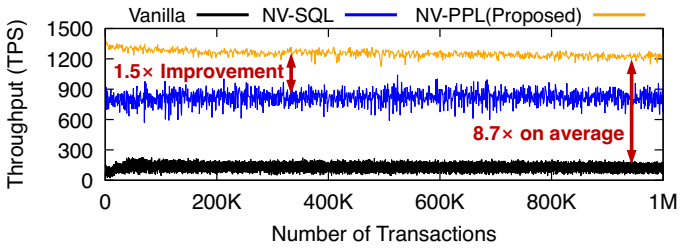


Fig. 1. TPC-C Throughput (Buffer Size = 10%) - *NV-PPL* outperforms the *equal-priced* Vanilla MySQL and NV-SQL (the state-of-the-art solution) by 8.7× and 1.5×, respectively.

impede overall performance. This is because faster reads are often blocked by preceding slower writes at the database buffer layer, worsening throughput and latency [5, 43]. Meanwhile, our analysis of redo logs in OLTP workloads indicates that per-page changes are quite small, often spanning only several tens of bytes. In many cases, however, pages with a small update footprint must be written to storage in their entirety, worsening write amplification.

NVDIMM DRAM-based non-volatile DIMM (NVDIMM) [24, 30, 54, 62] operates at DRAM speeds, offers byte-addressability on the memory bus, and ensures data persistence even during power loss by integrating traditional DRAM and flash memory on a single DIMM with capacitors [25]. This unique architecture allows for direct access to NVDIMM-resident data using `load` and `store` instructions, thereby bypassing the software storage stack. Consequently, various research efforts have explored the effective use of NVDIMM to bridge the latency gap between DRAM and SSDs [4, 14, 22, 75, 77]. Additionally, the recent emergence of Compute Express Link (CXL) represents a significant advancement for NVDIMM devices, with CXL-supported models now available in disaggregated data center environments [73].

NVDIMM as a durable log cache To address the durability overhead, we propose a novel database architecture, *NV-PPL*. This architecture integrates NVDIMM into the two-tier memory hierarchy of DRAM and SSD by capturing redo logs upon page updates and storing them page-wise in NVDIMM for immediate durability. Although NVDIMM is approximately 3× costlier than DRAM and typically limited in capacity to tens of gigabytes, our findings suggest that storing small-sized, per-page redo logs (i.e., Per-Page Log, or PPL) in NVDIMM is effective for bridging the latency gap between DRAM and SSD. Therefore, we advocate for utilizing NVDIMM as a *durable log cache* between DRAM and SSD to reduce writes to SSD (e.g., by one-fifth). This reduction in writes enables the underlying SSD to handle more reads [37], substantially boosting transaction throughput with *NV-PPL*. Meanwhile, *NV-PPL* incurs two runtime overheads: First, reading *PPL pages* (i.e., pages containing PPLs stored in NVDIMM) from the SSD takes slightly longer because the redo logs stored in NVDIMM must be applied before the pages can be read into the buffer cache. Second, under the assumption of equivalent cost, trading DRAM capacity for NVDIMM usage results in a reduced hit ratio. However, these moderate overheads are greatly offset by the substantial advantages gained from reducing writes.

The concept of employing NVDIMM as a durable log cache is straightforward; however, its implementation within existing relational DBMS engines poses several challenges. These challenges include selective *PPLization* (i.e., converting a normal page to a *PPL page*), cost-effective utilization of NVDIMM space, and the process of *de-PPLization* (i.e., restoring *a PPL page* to a normal one). To address these challenges, we have extended key DBMS modules, including the buffer manager and I/O interface, to seamlessly integrate our design. Additionally, we propose enhanced recovery

and multi-versioning schemes by leveraging the per-page logs stored in NVDIMM. While the use of NVDIMM, per-page logging, and redo-based multi-versioning are not novel in themselves, the primary contribution of *NV-PPL* lies in its targeted design for per-page redo logging on NVDIMM. Our contributions can be summarized as follows:

- We observe that pages in OLTP workloads typically exhibit a small write working set, where only a small portion of most pages (e.g., 256 bytes of a 4KB page) is updated between reads and writes to storage. This insight forms the basis for proposing *NV-PPL*, which guarantees the durability of page updates by leveraging a minimal amount of NVDIMM as a durable, page-wise log cache.
- To implement *NV-PPL* in an actual system, we propose ways to integrate *NV-PPL* into existing buffer managers and I/O modules. Firstly, we introduce a selective *PPLization* tailored to OLTP workloads. Secondly, to address situations where NVDIMM space allocated to *PPL pages* becomes scarce, we introduce the foreground reclamation and background PPL cleaner to reclaim NVDIMM space and derive a rationale for which *PPL page* to *de-PPLize* (Section 3).
- *NV-PPL* facilitates a lightweight recovery mechanism for *PPL pages* by ensuring immediate persistence of updates, thus eliminating the need for redo recovery and enabling asynchronous undo (Section 4). Additionally, *NV-PPL* supports redo-based multi-versioning for *PPL pages* by using PPLs in NVDIMM, which allows for constant-time version construction, unlike traditional undo schemes. This improves both the latency of long-lived transactions and OLTP throughput in HTAP (Section 5).
- We prototyped *NV-PPL* with moderate modifications to the MySQL codebase and ran the TPC-C and LinkBench benchmarks on an actual NVDIMM device and multiple commercial flash SSDs. Compared to the equivalently priced NV-SQL [4], *NV-PPL* reduces SSD writes by half through its more fine-grained log caching strategy, resulting in up to a 1.5× increase in throughput when executing the TPC-C benchmark. Furthermore, when compared to flash-tuned Vanilla MySQL at the same price point, *NV-PPL* increases TPC-C throughput by up to 8.7× (see Figure 1). Additionally, *NV-PPL* shortens the recovery process by half compared to NV-SQL, and by one-fourth compared to Vanilla MySQL. *NV-PPL* also substantially improves the latency of long-lived transactions in HTAP environments by 3× to 18× over Vanilla MySQL, all while maintaining OLTP throughput.

2 Background and Motivation

2.1 Flash Memory and NVDIMM

Durability cost in DBMSs In relational DBMSs, transactional atomicity and durability are ensured through the log-force-at-commit and write-ahead logging (WAL) protocols [56]. These protocols permit dirty pages to be asynchronously written back to storage. However, writes constitute a major performance bottleneck for write-intensive OLTP workloads, even on high-performance SSDs. Excessive writes to flash storage can hinder overall performance due to the asymmetric read and write speeds of SSDs. Read operations are frequently stalled by preceding write operations within both the database buffer and SSD buffer layers, resulting in diminished throughput and latency [3, 5, 43]. To address this issue, one viable strategy is to divert excessive writes to a faster, durable cache. By diminishing the write traffic to the SSD, the flash storage can accommodate more read requests, thus enhancing overall performance [4].

Non-volatile DIMMs Recently, various non-volatile memory (NVM) technologies have been developed that utilize the same form factor as DRAM, including Intel's Optane DCPMM [33] and NVDIMM [24]. Installed on the memory bus, these NVMs provide performance comparable to DRAM while offering persistence and byte-addressability. NVDIMM-N [62] uniquely combines DRAM and flash, positioning it as an optimal choice for durability and latency. During standard

Table 1. Storage Media: Speed and Price Characteristics

Storage Media	Random IOPS (Unit)	Random IOPS Read	Random IOPS Write	Unit Capacity	Unit Price
DRAM [71]	-	-	-	32GB	$180
NVDIMM [62]	(256B)	25,523K	27,827K	32GB	$540
DCPMM [81]	(256B)	6,164K	1,905K	128GB	$695
SSD [72]	(4KB)	555K	28K	1TB	$310

operation, DRAM is actively memory-mapped, while flash memory serves as a backup to ensure durability during power outages. Consequently, NVDIMM-N operates at DRAM speed and is not subject to the wear issues common to other NVM devices. However, NVDIMM-N necessitates capacitors as a backup power source in case of power failures, which restricts its capacity to several tens of gigabytes. Additionally, the inclusion of extra flash chips and capacitors causes its price to be approximately three times higher than that of DRAM, as detailed in Table 1.

Table 1 displays the read and write bandwidth of NVDIMM-N and DCPMM as measured on our testbed machine, which is equipped with a single 16GB HPE NVDIMM-N and a single 128GB Optane DCPMM. To assess the random I/O per Second (IOPS), we use the FIO benchmark [13] for SSDs and the Intel Memory Latency Checker [34] for both NVDIMM-N and DCPMM devices. Using a unit size of 256 bytes, NVDIMM-N significantly outperforms DCPMM, achieving 14.6× and 4.1× higher performance in random write and read operations, respectively. We select this unit size because the XPLine length in DCPMM is 256 bytes, whereas NVDIMM supports read and write operations with a finer granularity of 64 bytes. Furthermore, the read-write asymmetry is significantly more pronounced in DCPMM, showing a 3.2× difference between random read and write operations, whereas NVDIMM demonstrates less asymmetry between the two. Therefore, when using DCPMM, frequent random updates can rapidly deplete the write bandwidth, resulting in a substantial decline in overall performance. Given that OLTP workloads typically involve frequent, small, and random updates, along with cache-line flush commands [81], the extended write latency makes DCPMMs less favorable for storing small data compared to NVDIMM-Ns [4]. Therefore, in this paper, we focus exclusively on NVDIMM-N as the storage medium for a durable cache and will refer to it simply as NVDIMM henceforth.

2.2 Per-Page Update Size in OLTP Workloads

Conventional databases do not immediately write an entire page to storage after each update; instead, they buffer updates in memory. Once a page is cached in the buffer pool, it may undergo multiple updates before being written to persistent storage. To ensure atomicity and durability, database systems record these updates in the redo log [57]. For simplicity, we refer to such update logs as *redo logs* throughout the paper.

To persist updates to non-volatile storage, data is typically written at the granularity of a page (e.g., 4KB or 16KB). Consequently, most workload analyses center around page-level I/O patterns [31, 51, 70], leaving detailed insights into the size of updates for each cached page largely unexplored. To explore this in OLTP workloads, we track redo log sizes for each 4KB page until it is flushed to storage during six-hour runs of the TPC-C (500 warehouses) and Linkbench (50M nodes) benchmarks, using a 10% buffer cache size. We modified MySQL/InnoDB to collect this data. The findings, shown in Figure 2, illustrate redo log sizes per page write plotted against the cumulative fractions of written pages on the x-axis. Note that the same page ID may appear multiple times if it is written repeatedly. We also ran the same experiment with a larger buffer size (50%)

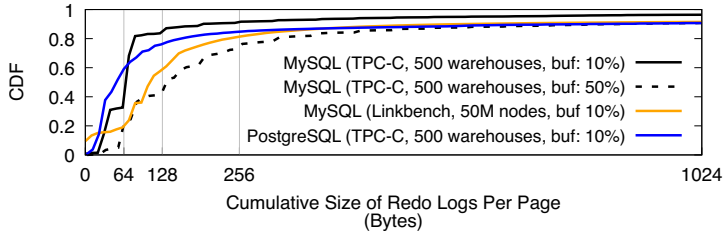


Fig. 2. Redo Log Size Distribution per Page Write

to assess distribution changes. This study offers valuable insights into determining the optimal amount of NVDIMM space to allocate per page for caching its redo logs, aiming to achieve both cost-effectiveness and write reduction efficiency.

Figure 2 reveals that with a 10% buffer cache size, 80% of TPC-C and 60% of LinkBench page writes have log sizes under 128 bytes. Furthermore, 30% of TPC-C and 20% of LinkBench page writes have log sizes under 64 bytes, while only 10% and 20%, respectively, exceed 256 bytes. Caching the redo logs of pages with log sizes exceeding 256 bytes is cost-inefficient, as they consume a relatively larger portion of NVDIMM space for a single disk write. Therefore, we exclude these pages from NVDIMM caching consideration. Even when the buffer size is increased to 50% in TPC-C, up to 40% of pages still have log sizes below 128 bytes during the period they are cached in the buffer cache, despite the larger buffer size allowing for more updates per page. We also conducted the same experiment with a 10% buffer cache size on PostgreSQL [68]. As shown in the final line of Figure 2, PostgreSQL exhibits a similar trend in redo log size distribution, indicating that our findings are not limited to MySQL/InnoDB but also apply to PostgreSQL and potentially other DBMSs. Overall, most OLTP workload page writes involve updates of less than 256 bytes per page, regardless of the system. It is clear that the write volume required to ensure durability (i.e., the page size) is much larger than the actual modifications made during OLTP workloads.

2.3 Durability Cost: SSD vs. NVDIMM

Consider three approaches for ensuring the durability of a page modification: writing the entire 4KB page to an SSD, caching the entire page in NVDIMM (i.e., NV-SQL [4]), and caching only the per-page redo Logs in NVDIMM (i.e., *NV-PPL*). For the discussion, we assume the price and performance of SSD and NVDIMM devices as provided in Table 1.

Suppose a 4KB page in the DRAM buffer cache undergoes cumulative updates totaling 256 bytes. Utilizing the five-minute rule [26], we estimate the durability cost for each of these alternatives. In the first approach, writing an entire page to the flash SSD costs about \$0.011 (≈ \$310 / 28K writes). The second approach of caching the entire page of 4KB in NVDIMM costs \$0.000065 (≈ \$540 / 32GB / 4KB). In the last approach, the cost of caching the 256-byte long update redo log in NVDIMM is \$0.000004 (≈ \$540 / 32GB / 256B). As such, in terms of durability cost, *NV-PPL* is 2,750× and 16× cheaper than writing to SSD and caching the entire page in NVDIMM, respectively. Furthermore, as per the specifications in Table 1, both *NV-PPL* and NV-SQL achieve approximately 1,000 times faster durability for a 256-byte page update compared to SSD writes. This confirms that caching fine-grained redo logs on NVDIMM represents the most cost-effective and high-performance solution for ensuring write durability.

Comparing NVDIMM space efficiency: NV-SQL vs. *NV-PPL* As mentioned earlier, the NV-SQL page caching scheme [4], which caches entire pages in NVDIMM to reduce writes, leads to inefficient use of NVDIMM space compared to *NV-PPL*, which only stores redo logs for each page.

According to TPC-C benchmarking results shown in Figure 2, 80% of evicted pages in NV-SQL cached in NVDIMM had redo log sizes under 256 bytes. This implies that NV-SQL allocates 4KB of NVDIMM space to manage only 256 bytes of updates across four out of every five pages. This inefficiency highlights NV-SQL's poor use of costly NVDIMM space due to its page-level caching approach, which limits its ability to reduce writes more effectively on other pages.

In contrast, the fine-grained log caching scheme of *NV-PPL* utilizes NVDIMM space more effectively by absorbing more writes across more pages, especially when the accumulated updates per page are small. For instance, as shown in the lower-left corner of Figure 2, where cumulative redo logs are less than 64 bytes, the log caching scheme can absorb four writes for a single page using just 256 bytes of NVDIMM space. As a result, with 4KB of NVDIMM, the log caching scheme can absorb four writes for each of 16 pages, whereas the page caching scheme can only absorb four writes for only one page.

Small write working set The benefits of *NV-PPL* can be influenced by the size of the write working set, which naturally varies depending on the workload. For example, applications that update many pages in batch may see limited performance improvements with *NV-PPL*. In such scenarios, even a log caching scheme may be infeasible for buffering all updates due to the constrained capacity of NVDIMM relative to the number of modified pages. However, OLTP workloads generally exhibit a small write working set (e.g., 2.6~6.7% of the total database size), due to temporal locality and skewed writes [10, 31, 49, 70]. This compact write working set size allows NVDIMM to effectively absorb a significant portion of writes in general OLTP workloads [4].

3 *NV-PPL*: Per-Page Logging on NVDIMM

NV-PPL aims to minimize writes to flash storage by leveraging NVDIMM as a durable log cache, requiring minimal modifications to conventional relational database systems. Given that OLTP workloads often involve small updates to each page and maintain a small write working set, *NV-PPL* can drastically reduce writes to SSDs by caching redo logs on a modestly sized NVDIMM. The primary advantage of using NVDIMM as a durable log cache, alongside the conventional DRAM page buffer, lies in its ability to persist page updates at NVDIMM speeds while seamlessly integrating with the existing DRAM-SSD two-tier architecture. With these factors in mind, *NV-PPL* is designed to ensure cost-effective durability for per-page updates within a minimal NVDIMM footprint. Figure 3 illustrates the architecture of *NV-PPL* and the interactions between its components.

3.1 Basic Framework

PPL page and per-page log Firstly, a page referred to as *a PPL page* indicates it has allocated NVDIMM space for sequentially storing its redo logs, known as *Per-Page Logs* (PPLs). This architecture is illustrated in the upper-right section of Figure 3. Conversely, pages that do not have such dedicated NVDIMM space are referred to as *normal* pages within the *NV-PPL* framework. The PPLization process allows a normal page to acquire dedicated NVDIMM space for its redo logs, and vice versa. Currently, *NV-PPL* selectively PPLizes only user data pages while excluding other types of pages, such as secondary index pages, undo pages, and system tablespace pages. Our experiments show that these user data page writes account for over 95% of all page write operations during the TPC-C benchmark.

Page update When a database page is updated in the buffer cache, *NV-PPL* generates a corresponding redo log and stores it in the WAL log buffer, just like conventional DBMSs [59, 63, 68]. Additionally, if the page is PPLized, *NV-PPL* appends the redo log to the page's PPL space in NVDIMM (Step 1 in Figure 3). This means that PPL pages maintain two copies of the redo logs: one in NVDIMM and another in the WAL files on disk. For each normal page that is not PPLized,

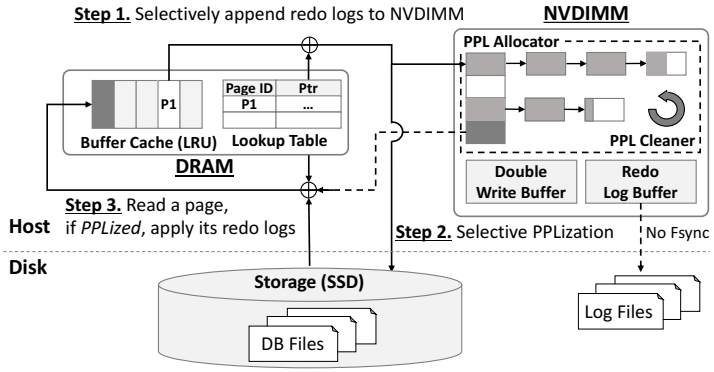


Fig. 3. NV-PPL Overview

NV-PPL retains its redo logs in its dedicated DRAM log space. The allocation of the DRAM log space occurs when the normal page is first updated after being loaded into the buffer cache. Later, during page eviction, *NV-PPL* checks this DRAM space to determine whether the page should be PPLized.

Page write and selective PPLization When a dirty page is written to storage, *NV-PPL* handles it selectively based on its status. For a PPL page, *NV-PPL* skips the disk write because its updates are already preserved in NVDIMM. Instead, *NV-PPL* marks the page as clean, evicts it from the buffer cache, and retains its PPL in NVDIMM, avoiding unnecessary writes to storage. For a normal page, *NV-PPL* determines whether to PPLize it based on the size of its accumulated redo logs in the DRAM log space. If the total log size is less than or equal to the PPL_{max} threshold (e.g., 256 bytes), *NV-PPL* allocates new NVDIMM space for the page and transfers the *PPLs* from DRAM to NVDIMM. This process, known as `PPLization`, takes place when sufficient NVDIMM space is available for allocation. Otherwise, the page must be written to storage. Once PPLized, updates to the page are durably preserved, eliminating the need for *NV-PPL* to store the page for durability. However, if logs in the DRAM space exceed the PPL_{max} parameter, *NV-PPL* will not PPLize the page but will write it to storage instead.

The rationale for selective PPLization based on the PPL_{max} value is to prevent the inefficient use of limited NVDIMM space by avoiding PPLization of pages with large redo logs. For instance, using NVDIMM space to prevent only one write for a page with a 256-byte redo log is less economical than using the same space to absorb four writes from pages with 64-byte logs each. Therefore, pages with large redo logs are written directly to storage (Step 2 in Figure 3), ensuring a more economical use of the available NVDIMM capacity.

We chose the default value of PPL_{max} as 256 bytes based on the findings from Figure 2, taking cost-effectiveness into account. The results indicate that over 70% of page writes in both TPC-C and Linkbench have accumulated redo logs smaller than 256 bytes before being written to disk. By allocating 256 bytes of NVDIMM space per page, we can effectively eliminate over 70% of disk writes for these workloads. While we believe this value will remain effective when running other OLTP workloads across different systems, it may need to be adjusted to achieve the maximum throughput.

Page read When a requested page is not present in the buffer cache, it is first fetched from storage. *NV-PPL* then checks whether the page is PPLized. If it is, all associated PPLs stored in the NVDIMM are sequentially applied to the retrieved page in chronological order, updating it before it

is inserted into the buffer cache (Step 3 in Figure 3). If the page is not PPLized, it is directly loaded into the buffer cache without any additional processing.

Page split and merge In B-tree-based storage engines, *PPL* pages undergoing splits or merges present complex recoverability issues [74]. Currently, we choose to `de-PPLize` *PPL* pages upon page splits or merges, and revert them back to their normal state. Once these pages are written to storage, *NV-PPL* treats them as normal pages eligible for PPLization. Our decision is based on observations from Figure 2, which indicate that the large logs generated during page splits or merges typically trigger de-PPLization due to their size. Specifically, the average redo log size for these operations exceeds 1 KB. Given the constraints of limited NVDIMM space, PPLizing such pages is not an efficient use of resources, so we exclude them from PPL targets.

Benefit and overhead *NV-PPL* reduces writes to storage by utilizing NVDIMM as a fast, byte-addressable, and durable log cache. When redo logs per page write are minimal, as shown in Figure 2, *NV-PPL* efficiently buffers multiple writes within its limited NVDIMM space. This approach not only alleviates the read stall problem caused by slow write operations on flash storage [5, 43], but also significantly boosts throughput. As demonstrated in Section 6, *NV-PPL* with a moderately-sized NVDIMM reduces average writes per transaction by one-fifth compared to Vanilla MySQL and by half compared to NV-SQL. In addition to enhancing transaction throughput, this reduction in user-level writes mitigates internal SSD garbage collection overhead [44], extending device lifespan and improving read operation speeds [36].

However, achieving write reduction in *NV-PPL* comes with two associated costs. First, compared to a DRAM-only system of equivalent cost, *NV-PPL* experiences a reduced buffer hit ratio. This trade-off arises because *NV-PPL* allocates part of its DRAM resources to NVDIMM for storing PPLs and other relevant small objects. Second, *NV-PPL* introduces additional read latency and CPU overhead when applying redo logs while reading PPL pages. Specifically, our measurements from the TPC-C benchmark indicate that the average time required to apply redo logs to a PPL page is 13 μs, contributing an additional 7.8% to the average page read latency of 165 μs. The associated CPU overhead is estimated to be only 2.7%. Despite this overhead, as demonstrated in Section 6, *NV-PPL* far outperforms the equally priced Vanilla MySQL and NV-SQL, primarily due to its advantage in write reduction.

3.2 Design of Per-Page Logging

With the overall framework in mind, we now explore how *NV-PPL* manages per-page logging and resolves associated technical challenges.

3.2.1 PPL Layout.

Per-page log Upon a page update, *NV-PPL* captures a redo log of the update and stores it in both the NVDIMM space and the original WAL buffer. A key difference between the two is that PPLs in NVDIMM are durably appended in chronological order, whereas the WAL buffer interleaves logs from various pages based on their generation order. Each PPL (as shown in the upper section of Figure 4) contains the same payload as the redo log in the WAL buffer, but also includes additional metadata, known as the PPL header. The header consists of a 1-byte type, a 2-byte log length, and a 6-byte transaction ID. The transaction ID aids in faster recovery, as will be discussed in Section 4.

PPL block The traditional centralized logging system, where all threads and transactions share a single log buffer, often becomes a bottleneck in multi-threaded systems, even in the absence of conflicts among threads and transactions [14, 18, 28, 29, 52, 80]. To address this limitation and enhance transaction scalability, *NV-PPL* employs a distributed logging approach [80]. It partitions the mmapped NVDIMM space into dynamically allocable units called PPL blocks. Once pages

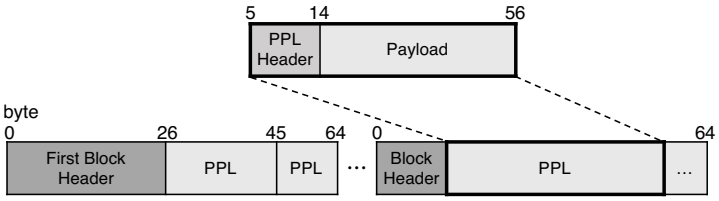


Fig. 4. PPL Block and PPL Format

are PPLized, they can write PPLs concurrently to their designated blocks, enabling simultaneous logging for multiple PPL pages.

As illustrated in Figure 4, each PPL block consists of a block header and multiple PPL entries. The first block allocated to each PPL page contains a specialized metadata structure known as the *first block header*, which includes an additional 26 bytes of data compared to the standard block header. The *first block header* comprises six distinct elements, described below:

- *is_first_block* (1 byte) indicates the first PPL block of the corresponding PPL page. This is used to construct a mapping table when probing PPLs during database recovery.
- *depplized* (1 byte) indicates that the page has once been PPLized and now de-PPLized. Thus, the page is de-PPLized but still contains PPLs.
- *next_block* (4 bytes) points to the address of the next PPL block.
- *ppl_len* (4 bytes) represents the total length of the PPLs stored within all PPL blocks.
- *ppl_lsn* (8 bytes) stores the log sequence number (LSN) of the latest PPL that has been appended to the current PPL block.
- *page_id* (8 bytes) contains the page ID of the corresponding PPL page.

Whereas, subsequent blocks have a standard structure called a *block header*, which only contains two elements: *is_first_block* and *next_block*.

The size of each PPL block is set to 64 bytes. As more PPLs are appended, additional blocks are allocated to accommodate the new logs. The most recently allocated block for a PPL page is referred to as the current PPL block. If the size of accumulated PPLs for a page exceed PPL_{max}, or the number of PPL blocks reaches the maximum limit, *NV-PPL* begins a de-PPLization process, which is detailed later.

We have configured the PPL block size to 64 bytes and the maximum number of PPL blocks per page to 4. As confirmed by our preliminary experiments described in Section 2, these settings optimally support OLTP workloads. Choosing an excessively small total PPL size can lead to frequent de-PPLization, thereby reducing the effectiveness of *NV-PPL*'s write reduction strategy. Conversely, an excessively large PPL size risks underutilizing the costly NVDIMM space, compromising the cost efficiency of our design. While the current configuration is well-suited for typical OLTP workloads, these parameters remain adjustable to accommodate other workload characteristics as needed.

The metadata space overhead, including the first block header, block header, and PPL header, could represent a disproportionately large share of the total space if the payload size per PPL is very small (e.g., just a few bytes). However, our analysis reveals that the average payload size per PPL is 42 bytes in TPC-C and 71 bytes in Linkbench. For TPC-C, the additional metadata accounts for approximately 30% of the total space, with the PPL header alone contributing about 47% of the total metadata space. While this overhead is not negligible, it is important to note that its proportion depends on the average payload size of the redo logs generated during the workload. More importantly, this metadata plays a critical role in reducing parsing overhead, skipping undo

recovery, and supporting PPL-based multi-versioning, which are essential features for enhancing performance and scalability.

Lookup table To quickly locate PPLs in NVDIMM, *NV-PPL* maintains an in-memory hash table that stores page IDs and the first PPL block pointers as key-value pairs. When a page is PPLized, its page ID and the pointer to the first PPL block are stored in this table. This pointer indicates the start of the chain of PPL blocks in NVDIMM. Upon reading a page into the buffer cache, *NV-PPL* searches the table using the page ID. If the key is found, it confirms that the page has been PPLized and provides the starting point for the PPL blocks. To ensure consistency amid concurrent lookups, insertions, and deletions, *NV-PPL* employs MySQL's `rw_lock` [64].

3.2.2 PPL Operations.

PPL write To secure durability, *NV-PPL* employs a *header* logging scheme [77], which is similar to *libpmemlog* in PMDK [67]. *NV-PPL* appends the *PPL header* and the payload to the end of the current PPL block, then updates and persists the cumulative log size in the *first block header*. Since the valid length (i.e., *ppl_len*) is directly stored in the *first block header*, this simplifies the inspection of valid PPLs during recovery. If the remaining space in a PPL block is less than the size of a newly generated PPL, *NV-PPL* spans the PPL across multiple blocks. The *ppl_len* field in the *first block header* is updated only after the PPL has been completely written to the NVDIMM blocks, ensuring that PPLs are not partially stored.

To guarantee the order and persistence of memory writes, NV-PPL ensures atomicity and durability by invoking the `mfence` instruction before and after each `clflush` call. Since *NV-PPL* accesses data stored on NVDIMM via the CPU cache using `load` and `store` instructions, similar to PMem, it ensures that data is properly flushed from the CPU cache using cache-line flush instructions. This allows *NV-PPL* to handle metadata for both PPL blocks and PPLs in NVDIMM, guaranteeing correct recovery upon failure.

PPL apply When reading a PPL page from the storage to the buffer cache, *NV-PPL* retrieves the address of the first PPL block associated with the page and applies the PPLs from this block to the page version fetched from disk. It then sequentially applies each subsequent PPL by following the chain indicated in the *next_block* field.

De-PPLization *De-PPLization* is the process where a PPL page releases its allocated PPL blocks, reverting from a PPLized to a normal state. This occurs under four conditions: firstly, if a PPL page needs a new PPL block but none are available, *NV-PPL* de-PPLizes the page. Secondly, as noted in Section 3.1, de-PPLization is triggered by structural changes such as B-tree node splits or merges, which simplifies recovery processes. Thirdly, de-PPLization happens if the cumulative size of PPLs surpasses the PPL_{max} threshold. Lastly, when only a few PPL blocks remain available, *NV-PPL* de-PPLizes a victim page to reclaim NVDIMM space, as will be demonstrated in Section 3.2.3. While adding more PPL blocks is also feasible, it can lead to excessive use of NVDIMM space, diminishing efficiency. Thus, to better distribute limited NVDIMM space and enhance write reduction, *NV-PPL* de-PPLizes update-intensive pages.

Once a PPL page reverts to its de-PPLized state, *NV-PPL* updates the LSN in the page header to reflect the most recent update. The redo logs of subsequent updates are no longer stored in the PPL block but are managed in the global WAL buffer. To ensure durability, *NV-PPL* retains the PPLs of a de-PPLized page until it is safely flushed to storage. Then, *NV-PPL* removes the corresponding entry from the lookup table and returns PPL blocks as free, making it possible for the page to be PPLized in the future if needed.

3.2.3 NVDIMM Space Reclamation.

An intrinsic and critical challenge with *NV-PPL* is that as the utilization of PPL blocks increases, the number of available PPL blocks decreases. When there is a shortage of available PPL blocks,

pages can no longer be PPLized, and existing PPL pages undergo de-PPLization. This reduces the benefits of write reduction. Therefore, when *NV-PPL* exhausts NVDIMM space and only a small number of PPL blocks remain available, it initiates de-PPLization on victim pages to reclaim NVDIMM space.

Foreground reclamation To prevent *NV-PPL* from running out of NVDIMM space, it begins de-PPLizing PPL pages with the maximum number of PPL blocks (i.e., four PPL blocks) in the foreground once the number of available PPL blocks falls below the threshold. By preemptively de-PPLizing these pages, which are likely to be de-PPLized soon, *NV-PPL* can reclaim the PPL blocks and use the reclaimed space to PPLize other pages. This foreground reclamation continues until the number of available PPL blocks reaches the threshold.

Background PPL cleaner and victim selection The background PPL cleaner de-PPLizes pages that have been PPLized but have remained unused for a certain period of time. Since these pages are likely not present in the buffer cache, their PPL blocks cannot be reclaimed through foreground reclamation. As a result, the background PPL cleaner must read these pages from storage, de-PPLize them, and reclaim the unused PPL blocks. This process involves additional read, apply, and write operations.

To minimize the additional overhead caused by this process, efficient victim selection is crucial when the PPL cleaner determines which PPL page to de-PPLize. The victims should be the pages least likely to be updated. Conventional victim selection policies, such as Least Recently Used (LRU), may be effective, as they predict which pages are less likely to be accessed. However, these policies require introducing a new linked list for PPL pages, leading to additional CPU cycles and increased mutex contention for their management.

Hence, we opt for an alternative, lightweight victim selection scheme that requires no additional data structures or runtime overhead. A practical and cost-effective rule of thumb for selecting victim pages for de-PPLization involves applying the formula used to derive the five-minute rule [26] with the costs of SSD and NVDIMM from Table 1. According to this analysis, the break-even write interval for PPL pages is 45 minutes. In other words, if a PPL page has not been updated in the last 45 minutes or longer, it would be more economical to de-PPLize it and reallocate its PPL blocks to other pages. Note that implementing the five-minute rule is straightforward since *NV-PPL* already tracks recent update timestamps (i.e., *ppl_lsn*) for PPL pages. Although better victim selection methods could be explored, we defer a comprehensive analysis to future work.

3.3 Placing Other NVDIMM-worth Objects

Besides database pages, DBMS engines also manage critical, write-intensive objects like the redo log buffer and double write buffer (DWB). Allocating the redo log buffer to NVDIMM mitigates the transactional durability and atomicity overhead imposed by the *log-force-at-commit* and *WAL* protocols in conventional DBMSs thas assume volatile DRAM-based buffers [4]. Similarly, hosting the DWB in NVDIMM facilitates the atomic writing of dirty pages through *single-write journaling*, instead of redundant writes to DWB and its original location [37].

3.4 Prototype Implementation

We have implemented *NV-PPL* on MySQL/InnoDB with moderate code changes. Considering that MySQL adheres to standard buffer management [59], we expect that integrating *NV-PPL* into other relational DBMSs will also be straightforward.

PPL allocator *NV-PPL* extends each of MySQL's default buffer instances (eight by default) with a dedicated PPL-Module that facilitates PPL operations discussed in Section 3.2.2. This module manages the NVDIMM space by storing the address of each PPL block in a queue. It uses a dequeue operation to allocate a PPL block when a page needs more NVDIMM space for PPLs, and an enqueue

operation to reclaim the PPL block during page de-PPLization. A mutex ensures the consistency of these operations against concurrent accesses.

Lookup table We implement the lookup table using a hash map. Each entry in the lookup table has a key for `page_id` and a value for `first_PPL_ptr`, each occupying 8 bytes. Therefore, *NV-PPL* requires additional DRAM space for the lookup table. On average, the memory size used by the lookup table is approximately 90MB. Even in the worst-case scenario—such as a batch-update transaction that updates each page with a single PPL block—the memory overhead remains low, as the available NVDIMM limits the number of PPL blocks. For example, in a system with 2GB of DRAM and 1GB of NVDIMM (a relatively high NVDIMM ratio), the maximum memory usage for the lookup table would be 256MB ($1GB/64B * 16B$) or 12.5% of the DRAM buffer size. As the ratio of NVDIMM to DRAM decreases, the additional memory required for the lookup table also decreases.

Extending buffer objects We add two 1-bit flags to each buffer frame object: `PPLized` and `de-PPLized`. The `PPLized` flag indicates whether the page is PPLized, while the `de-PPLized` flag signals whether the page has been de-PPLized, indicating if its PPL should be relinquished. Additionally, to track the first PPL block and the location for appending subsequent logs, the buffer frame object is augmented with two pointers, each occupying 8 bytes: `first_ppl_block_ptr` and `wr_ptr`.

PPL cleaner The PPL cleaner module includes a coordinator thread and several background threads that manage the reuse of NVDIMM space, as detailed in Section 3.2.3. The coordinator thread uses LSNs to set and record a 45-minute break-even update interval every minute. Subsequently, the cleaner threads identify victim pages that have not been updated for 45 minutes or longer.

4 Recovery

System failure is inevitable in large-scale database servers, particularly in cloud environments where hardware failures are prevalent [7, 32]. Upon system restarts, databases typically undergo a multi-phase recovery process to ensure the atomicity and durability of transactions. However, *NV-PPL* simplifies the restart procedure compared to conventional systems. Updates are made durable in NVDIMM, clustered by page, and ordered chronologically. This structure aligns effectively with existing recovery schemes such as ARIES [57], where pages serve as the recovery unit.

4.1 Prototype Implementation

NV-PPL leverages the existing Vanilla MySQL design, making it compatible with other DBMSs that use ARIES-style recovery. By using durable per-page redo logs in NVDIMM, NV-PPL enables redo-less and undo-less recovery for most PPL pages. The ARIES-like recovery module in MySQL involves three steps: Analysis, Redo, and Undo [58]. Here, we outline how *NV-PPL* expedites each phase through the use of persistent PPLs in NVDIMM.

Analysis *NV-PPL* constructs a mapping table by scanning the PPL region in NVDIMM and parsing the redo log file. This process identifies the PPLs in NVDIMM, enabling instant access to them during page read operations. Additionally, *NV-PPL* records the transaction IDs of active transactions at the time of failure, ensuring proper handling of PPL pages modified by these transactions.

Redo phase *NV-PPL* handles redo recovery differently for two types of pages: normal and `PPL` pages. For all normal pages, recovery follows the same procedure as in Vanilla MySQL. However, for most `PPL` pages, *NV-PPL* can achieve redo-less recovery. Since all updates for each `PPL` page are already durable as PPLs, *NV-PPL* can skip applying redo logs for those pages. Thus, any changes made to `PPL` pages by committed transactions are guaranteed to persist even in the event of crashes.

When a page is de-PPLized, it ensures that the page has been safely flushed to disk before removing the PPLs of the page. If a failure occurs during the flush process, it is crucial to retain

updates made before the de-PPLization to ensure their persistence. To do this, *NV-PPL* orchestrates the interaction between PPLs and redo logs in the WAL file by introducing a `ppl_lsn` that records the LSN of the latest PPL. For a de-PPLized page where the `ppl_lsn` is greater than the lsn in the page header, it indicates that the page has not been safely flushed to disk. In this case, *NV-PPL* first applies PPL to the old disk page and then replays the redo logs from the WAL file to rectify the situation. Conversely, if the `ppl_lsn` is smaller than the lsn in the page header, *NV-PPL* skips applying redo logs and returns the PPL blocks of the page, since the de-PPLized page has already been persisted to disk.

Undo phase *NV-PPL* restores uncommitted updates for normal pages using undo logs while it skips replaying PPLs from incomplete transactions for PPL pages. This distinction arises because *NV-PPL* keeps the original pages on disk unchanged, while PPLs are persisted in NVDIMM, enabling the identification of uncommitted transaction IDs within the PPL. The undo process in *NV-PPL* shares similarities with the logical revert mechanism used in [7], which restores the committed version to the data page. Unlike this undo-less scheme[7], *NV-PPL* allows the system to simply skip PPLs associated with incomplete transactions.

The recovery process in *NV-PPL* is *idempotent*. *NV-PPL* determines the need for redo application by comparing the page's LSN with that of the log records. If the page's LSN exceeds that of the log records, it indicates that the page has not only been flushed but also that any applicable PPL records have been applied after the application of this record, ensuring idempotency within *NV-PPL*.

5 Redo-based Multi-Version Support

Many disk-based database systems use multi-versioning for concurrency control, maintaining the latest tuple versions for write transactions and creating snapshots for read transactions at lower isolation levels [2, 50, 59]. Snapshots are built using undo logs to revert updates made by write transactions to the state when the read transaction began. While version construction typically has minimal impact in OLTP or OLAP workloads, it poses significant overhead in HTAP applications due to the increasing version gap between concurrent read and write transactions [1, 38]. To mitigate this, we propose a redo-based multi-version store scheme, inspired by *NV-PPL*, which groups redo logs for PPL pages on fast NVDIMM. This approach constructs tuple versions for long-lived transactions (LLTs) by applying PPLs from NVDIMM to the old disk page, significantly reducing query latency for LLTs and maintaining OLTP throughput in HTAP environments.

5.1 Undo-based Multi-Version Store

In this paper, we use the undo-based multi-version store as a representative comparative design, given its widespread adoption in prominent systems like Oracle, Azure, and InnoDB [8, 35, 61]. However, it is important to note that our scheme can be seamlessly integrated into any system that implements multi-versioning by managing historical versions and redo logs based on timestamps.

Undo-based version construction When a write transaction modifies a tuple, the system preserves the previous version (*i.e.*, delta), alongside other preceding versions. The `TRX_ID`, which works as the global atomic timestamp, is updated to the value of `TRX_ID` at the start of the write transaction. The `ROLL_PTR` field is also updated to point to the most recent old version, indicating the start of the undo chain [60]. During a read transaction, the system creates a `readview` to validate the visibility of the constructed tuple version. A record becomes visible if it meets two conditions: (1) `TRX_ID` falls within the minimum and maximum `TRX_ID` among all active transactions, and (2) its `TRX_ID` is not in the `active_trx_id_list`. If the current version is not visible, the system retrieves the most recent undo log by using the `ROLL_PTR`, rolls back the modification, and rechecks visibility. This process repeats until the current version becomes visible, as shown in Figure 5 (Step 3a).

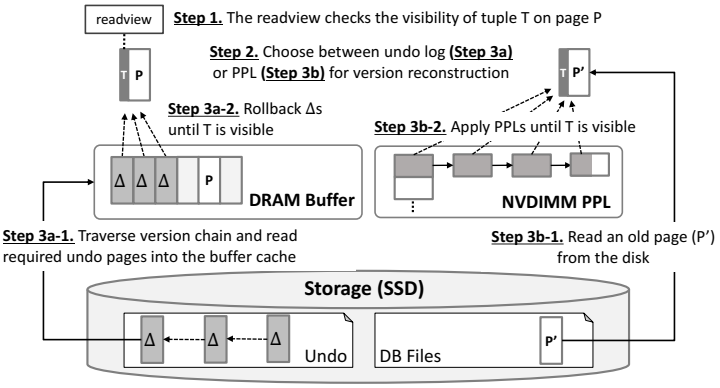


Fig. 5. Undo-based vs. PPL-based Multi-Versioning

Problems with HTAP workloads While performing well on standard OLTP workloads with short write-intensive transactions, undo-based MVCC schemes are known not to be suitable for HTAP applications [1, 38, 39, 48, 82]. The write transactions rapidly accumulate new versions in the version chain, but stale versions, though irrelevant to LLT queries, cannot be garbage-collected until the LLT commits. As such, the conventional undo-based multi-versioning has a detrimental impact on HTAP workloads, mainly because of two reasons. First, multiple undo pages have to be read from the undo tablespace to the buffer cache when building tuple versions for LLTs, which prolongs the latency of the query [1]. Second, concurrent OLTP transactions experience more buffer misses since the buffer cache becomes polluted by undo pages fetched during the version construction process, worsening throughput.

5.2 PPL-based Multi-version Support for HTAP

To address the LLT-induced performance degradation, we propose a hybrid version store that utilizes per-page logs in NVDIMM.

Supporting consistent view with PPL The severe latency of LLTs in conventional undo-based multi-version stores arises because undo logs are stored by transaction order, not per page or tuple. This disperses the logs necessary for version reconstruction across multiple undo pages, requiring numerous disk reads. Conversely, *NV-PPL* stores the recent update history of each PPL page in chronological order in a dedicated NVDIMM space. This unique design makes it feasible to use PPLs when constructing consistent tuple versions to achieve lower latency of LLTs.

To serve a read transaction, the system generates a visible version of a specific tuple by following the process outlined in Figure 5. After a visibility check through the `readview` (Step 1), the system decides to construct an older version. If the previously read tuple and the current tuple are on the same page, a previously constructed page is reused. If not, the system retrieves an old page from the disk without adding it to the buffer cache (Step 3b-1), then applies logs to make the record visible (Step 3b-2). This approach reduces the volume of logical data read from the disk by utilizing redo logs from NVDIMM, rather than fetching multiple undo pages.

Version pre-construction of de-PPLized pages One issue is that *NV-PPL* discards the redo logs of a PPL page upon de-PPLization, and if an LLT started before this process, it would be impossible to use PPLs for version reconstruction. For this reason, we prebuild the version of the page before flushing a de-PPLized page to the disk. Before de-PPLization, *NV-PPL* examines the active transaction list and identifies potential LLTs. If multiple LLT candidates are running, *NV-PPL*

selects the one with the latest start timestamp. This prebuilt version is discarded once the LLT commits.

Hybrid version reconstruction When an LLT requires version reconstruction, *NV-PPL* selects the appropriate reconstruction strategy based on the availability of a prebuilt version and whether the page remains PPLized. If a prebuilt version exists with a matching timestamp, it is utilized directly without further processing. If the start timestamp of the LLT is earlier than that of the prebuilt version, *NV-PPL* employs undo-based version reconstruction, starting from the prebuilt version. This approach shortens the version chain by minimizing undo traversal, as rollback operations begin from the prebuilt version rather than the newest version inside the buffer pool. If no prebuilt version is available for PPL pages, *NV-PPL* uses redo-based version reconstruction for PPL pages. For normal or de-PPLized pages, *NV-PPL* resorts to undo-based reconstruction.

Advantages Our approach offers two benefits. First, *NV-PPL* can significantly reduce LLT query latency due to constant version construction time for a page that is either PPLized or has a prebuilt version. Unlike undo-based versioning, which may incur hundreds or even thousands of random disk reads for undo pages in the worst case, our approach reads one old page from the disk and retrieves logs from NVDIMM at DRAM-like speeds when using PPLs for version construction. Second, the buffer hit ratio improves because *NV-PPL* does not load undo pages into the buffer cache. Consequently, the number of disk reads by other concurrent transactions is reduced, making them less impeded by LLTs.

6 Performance Evaluation

In this section, we conduct a performance evaluation to highlight the cost-performance effectiveness of *NV-PPL* compared to the *same-priced* Vanilla MySQL and the state-of-the-art NV-SQL solution [4]. Throughout this paper, Vanilla MySQL refers to flash-tuned MySQL. We tune flash-aware knobs in MySQL to establish a baseline [5].

6.1 Experimental Setup

We conduct all experiments on a dual-socket Linux machine equipped with two Intel Xeon E5-2460 CPUs (32 cores at 2.5GHz), 64GB DRAM, and 16GB NVDIMM-N [62]. Storage includes a Samsung 960 PRO 1TB NVMe SSD for data and a Samsung 850 PRO 256GB SSD for logs, with the `ext4` file system in direct I/O mode and NVDIMM mounted with the `DAX` option. We configure the redo log buffer size to 32MB and the DWB size to 2MB. We set the buffer cache to 10% of the database size, the page size to 4KB, and the number of concurrent client threads to 32. All systems are cost-equivalent, with NVDIMM priced three times higher than DRAM, as per Table 1. *NV-PPL* utilizes an additional 162MB of DRAM for the PPL allocator, the lookup table, and the DRAM log space for normal pages, while Vanilla MySQL and NV-SQL receive more DRAM for the buffer cache. As suggested in [4], we use a 1GB NVDIMM configuration for NV-SQL. Below are descriptions of the three workloads used in the experiments:

TPC-C We use tpcc-mysql [66] to run the TPC-C workload. TPC-C is a standard transaction processing system benchmark, exhibiting heavy random reads and writes. For all experiments, we set the initial database size to 54GB (*i.e.*, 500 warehouses).

Linkbench Linkbench [10] is a graph-serving benchmark that simulates a read-intensive transactional workload based on production traces from TAO [19] at Meta. The generated dataset consists of 50 million nodes, which is approximately 64GB.

HTAP To simulate the HTAP workload, we add a thread to the original TPC-C workload [66] that executes only LLT queries. Section 6.5 includes more details regarding the experimental method.

Table 2. Performance Metrics (TPC-C, Buffer Size = 10%)

DRAM:NVDIMM	Vanilla	NV-SQL	NV-PPL		
(GB)	(5:0)	(2:1)	(2:1)	(3:0.66)	(1:1.33)
Average TPS	140	811	1,228	907	1,059
Avg Latency (ms)	7.8	1.2	0.8	1.1	0.9
99th Latency (ms)	399.5	135.4	24.7	113.2	26.8
Write/tx (KB)	158.4	78.5	29.7	37.1	41.1
Read/tx (KB)	98.2	139.1	187.2	144.8	285.3
Write/Sec. (MB)	21.7	62.2	35.6	32.8	42.5
Read/Sec. (MB)	13.5	110.1	224.6	128.3	295.1
Hit Ratio (%)	97.0	95.2	94.6	95.7	92.0
User CPU (%)	5.9	33.6	54.4	34.8	51.4

6.2 Basic Performance Analysis with TPC-C

To compare the cost-effectiveness of three systems – DRAM-only `Vanilla` (5GB DRAM), `NV-SQL(2:1)`, and `NV-PPL(2:1)` (2GB DRAM and 1GB NVDIMM) – we measure TPS and relevant metrics while running the TPC-C benchmark on each system. The results are presented in Table 2.

Transaction throughput and latency The first row of Table 2 shows TPS for 30 minutes after a 5-minute warm-up time. `NV-PPL(2:1)` outperforms Vanilla and `NV-SQL(2:1)` by 8.7× and 1.5×, respectively. This result indicates that the equal-cost *NV-PPL* can reduce writes to flash storage more effectively than NV-SQL. Even if the buffer hit ratio decreases in *NV-PPL* compared to Vanilla or NV-SQL, as shown in the eighth row of Table 2, the write reduction far offsets the reduced hit ratio in terms of throughput. In addition to throughput, the write reduction in *NV-PPL* improves transaction latency as well. As shown in the second and third rows of Table 2, *NV-PPL* reduces the average and 99th-percentile latencies by 32% and 82%, respectively, compared to NV-SQL.

Disk reads and writes per transaction The fourth row shows the amount of data written to the SSD per transaction. Compared to Vanilla and `NV-SQL(2:1)`, `NV-PPL(2:1)` reduces per-transaction write traffic to the SSD by 81% and 62%, respectively. This highlights that the fine-grained log caching in *NV-PPL* can absorb more writes using the same amount of NVDIMM than the page-grained caching in NV-SQL.

The fifth row represents the amount of data read from SSD per transaction. As expected, *NV-PPL* results in higher per-transaction read traffic than other versions. `NV-PPL(2:1)` yields 1.9× more read/tx than Vanilla. Considering that *NV-PPL* trades DRAM space for NVDIMM space to achieve write reduction, the reduced buffer cache size degrades the hit ratio and thus incurs more per-transaction disk reads. Specifically, the miss ratio of `NV-PPL(2:1)` is 1.8× higher than that in Vanilla. However, the improved transaction throughput from offsetting the hit ratio with write reduction consistently demonstrates the effectiveness of this trade-off.

Reads and writes per second The sixth and seventh rows in Table 2 display the amount of writes and reads per second, respectively. They reveal that the ratio of per-second read and write amounts in Vanilla, NV-SQL, and *NV-PPL* are approximately 1:1.6, 1:0.6, and 1:0.2. This underscores that *NV-PPL* transforms the write-heavy I/O pattern in Vanilla into a more read-intensive, and thus more SSD-friendly, pattern. The higher I/O traffic in *NV-PPL* is not due to its failure to absorb writes to storage but because it can process transactions much faster: NV-PPL achieves 8.7 times higher TPS than Vanilla and 50% higher TPS than NV-SQL (1st row). For this reason, in terms of

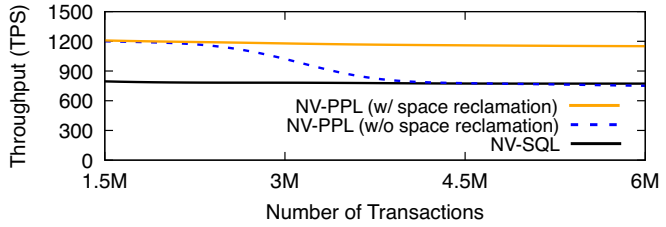


Fig. 6. Effect of NVDIMM Space Reclamation

per-second reads (7th row), *NV-PPL* is 16.5 times (= 8.7 × 1.9) higher than Vanilla. Meanwhile, user CPU utilization is proportional to transaction throughput (9th row).

Varying NVDIMM size To explore *NV-PPL*'s performance with a different DRAM:NVDIMM ratio but the same cost, we evaluate `NV-PPL(3:0.66)` and `NV-PPL(1:1.33)` and present the result in the fourth and fifth columns in Table 2. The experiment results confirm that `NV-PPL(2:1)` achieves the best trade-off between write reduction and hit ratio, resulting in the highest TPS. The write reduction in `NV-PPL(1:1.33)` is not enough to offset the reduced hit ratio. The hit ratio improvement in `NV-PPL(3:0.66)` is also insufficient to overcome the lowered write reduction. We leave the investigation of determining the optimal NVDIMM size as a future work.

In addition, we measure the throughput of NV-PPL by incrementally increasing the NVDIMM size from 0.6GB to 1.6GB in 0.2GB steps while keeping the DRAM buffer size fixed at 2GB. The experimental results indicate that throughput saturates at an NVDIMM size of 1GB. This saturation occurs because this size is sufficient to buffer the write working set, meaning that additional NVDIMM capacity does not lead to further improvements in throughput.

CPU overhead of *NV-PPL* The runtime CPU overhead of *NV-PPL* arises from two main sources: (i) managing PPLs in NVDIMM (i.e., PPL insertion, deletion, and search) and (ii) applying the PPLs upon reading PPL pages from SSD. Since measuring the exact CPU time for these sources is challenging, we estimate the additional CPU overhead of *NV-PPL* based on the metrics in Table 2. Given the average TPS of *NV-PPL* (i.e., 1,228) and Vanilla (i.e., 140), and the CPU utilization of Vanilla (i.e., 5.9%), the CPU utilization required for pure transaction processing in *NV-PPL* is estimated to be 51.6%. The 2.7% gap between this estimated utilization and the user CPU utilization of *NV-PPL* (i.e., 54.4%) can be regarded as the additional CPU overhead of *NV-PPL*. Though not negligible, this overhead can be justified by the 8.7× throughput enhancement.

Effect of NVDIMM space reclamation To explore the impact of NVDIMM shortages and demonstrate the effectiveness of the NVDIMM space reclamation, we run the TPC-C benchmark until it executes 6 million transactions across three setups: *NV-PPL* with and without the NVDIMM space reclamation and NV-SQL. We set the PPL cleaner to select victim pages for de-PPLization and de-PPLize them every 10 seconds. The dashed line in Figure 6 indicates that *NV-PPL* without the NVDIMM space reclamation sees a drop in throughput after 2.4 million transactions due to full NVDIMM utilization and increased competition for PPL blocks, eventually matching NV-SQL's performance. In contrast, *NV-PPL* with the NVDIMM space reclamation sustains throughout consistently, while adding 1.8% read and 15% write overhead. This underscores the benefits of NVDIMM space reclamation.

6.3 Effect of *NV-PPL* on Other Parameters

Different SSDs To verify the effect of *NV-PPL* on different SSDs, we measure the transaction throughput while running the TPC-C benchmark using two MySQL versions on three other

Table 3. Effect of *NV-PPL* on Different SSDs (TPC-C)

	SSD-A†	SSD-B♯	SSD-C*
Read IOPS(4KB)	370K	300K	95K
Write IOPS(4KB)	500K	100K	90K
Vanilla (TPS)	174	22	17
NV-PPL (TPS)	1205	896	776
TPS Improvement	6.9×	40.7×	45.6×
Write Reduction Ratio	77%	86%	89%

† Samsung 970 Pro NVMe SSD 512GB, ♯ Samsung SM951 256GB, * Micron Crucial MX500 250GB

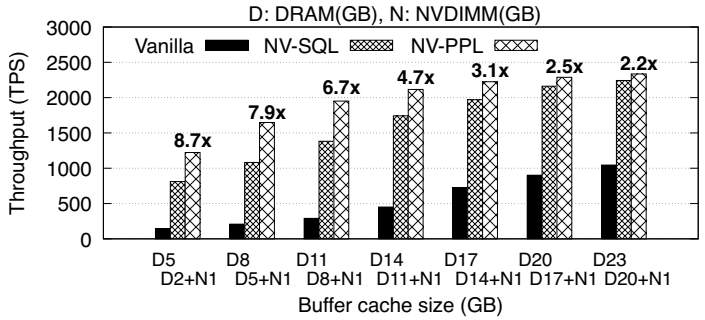


Fig. 7. TPC-C Throughput: Varying Buffer Sizes

commercial SSDs. The results are summarized in Table 3. The first two rows of the table illustrate varying random read and write IOPS capabilities across the three SSDs tested. Across all SSDs, *NV-PPL* consistently outperforms Vanilla, though the degree of improvement varies depending on the SSD. Specifically, for `SSD-A`, which shares a vendor and similar internal architecture with the SSD used in Figure 1, *NV-PPL* achieves a performance improvement over Vanilla of 6.9×.

Meanwhile, on `SSD-B` and `SSD-C`, *NV-PPL* outperforms Vanilla by more than 40 times. Both SSDs exhibit lower random write IOPS than `SSD-A`, resulting in Vanilla achieving only 22 TPS or fewer (approximately one-tenth of `SSD-A`'s TPS). Under the *NV-PPL* scheme, which transforms the write-heavy I/O pattern into a read-heavy one, the random read performance of SSDs becomes more critical than the random write performance, provided that a certain level of random write performance is maintained.

Buffer cache size Increasing buffer cache size boosts performance by raising the hit ratio. To assess the impact on *NV-PPL*, we compare the TPC-C throughput across three MySQL versions with varying NVDIMM to DRAM ratios, maintaining cost parity. The results in Figure 7 show that all versions achieve higher hit ratios and improved throughput with larger buffers. *NV-PPL* consistently outperforms NV-SQL and Vanilla at all buffer sizes, with its advantage growing as buffer size decreases. For instance, `NV-PPL(2:1)` exceeds `Vanilla` and `NV-SQL(2:1)` by 8.7× and 50%, respectively. Notably, *NV-PPL* experiences a smaller decline in TPS than NV-SQL and Vanilla as buffer size reduces. In contrast, as the buffer size increases, the relative performance gain of *NV-PPL* diminishes due to CPU-bound (i.e., more than 85% CPU utilization). However, even with memory configurations larger than 40% of the database size, as shown by the rightmost bar in Figure 7, the impact of *NV-PPL* remains significant.

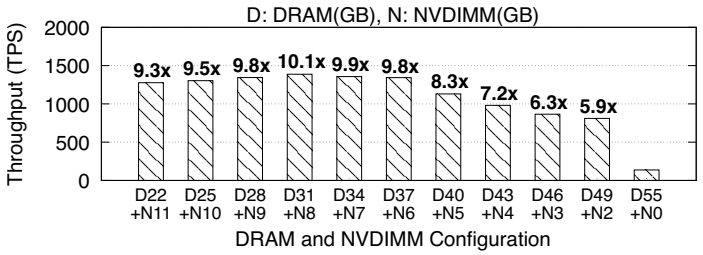


Fig. 8. TPC-C Throughput on Larger Database (554GB, 5,000 Warehouses)

Another important observation from Figure 7 is the cost-performance efficiency. For example, `NV-PPL(D2+N1)`, which costs only one-fifth in terms of memory, outperforms `Vanilla(D23)` by 1.17× in terms of TPS. Given that memory comprises about half of the cloud OLTP service cost [53], *NV-PPL* offers an economical solution for cloud service providers.

Database size To verify the scalability of *NV-PPL* in large databases, we measure transaction throughput for both *NV-PPL* and Vanilla while running the TPC-C experiment for three hours. The initial database size is set to 554GB (i.e., 5,000 warehouses), with a buffer size of 55GB (10% of the database size). The results are presented in Figure 8. The rightmost bar in the figure, labeled `D55+N0`, represents the transaction throughput of Vanilla. For *NV-PPL*, we vary the NVDIMM size from 2GB to 11GB while decreasing the DRAM size. As shown in Figure 8, the best-performing configuration, `NV-PPL(31:8)`, outperforms `Vanilla(55:0)` by a factor of 10.1. This result confirms that *NV-PPL* is scalable to large database sizes.

One interesting point from the experiment is that the best TPS improvement ratio of *NV-PPL* (i.e., 10.1×) is higher than that observed in the smaller TPC-C database with 500 warehouses (i.e., 8.7× in Table 2). This difference arises because the relative size of the write working set becomes smaller in the larger database. According to our separate analysis using Vanilla, while the database size increases tenfold, the number of distinct pages updated increases only five times. This indicates that the relative write working set is not proportional to the database size. Consequently, *NV-PPL* can achieve the same write reduction ratio with a relatively smaller NVDIMM capacity while also achieving a higher hit ratio with a larger DRAM capacity. Specifically, the hit ratio of `NV-PPL(31:8)` for 5,000 warehouses is 96.3%, while those for `NV-PPL(2:1)` with 500 warehouses is 95.2%.

Furthermore, we conduct the same experiment while varying NVDIMM sizes, with the results presented in Figure 8. For *NV-PPL* with NVDIMM sizes of 5GB or smaller, which is insufficient to buffer the write working set, the write reduction ratio significantly decreases compared to `NV-PPL(31:8)`, and a higher hit ratio does not compensate for this decline. Meanwhile, as the NVDIMM size increases from 9GB to 11GB, the write reduction ratio of *NV-PPL* remains relatively constant compared to `NV-PPL(31:8)`, while the hit ratio gradually declines, resulting in a corresponding decrease in TPS. Consequently, `NV-PPL(31:8)` achieves the optimal balance between write reduction and hit ratio.

Other workloads To demonstrate the effectiveness of *NV-PPL* on other OLTP workloads, we measured Operations Per Second (OPS) using the LinkBench benchmark across three MySQL versions for 25 million operations. While we were not able to display the OPS graph due to space limitations, *NV-PPL* outperforms DRAM-only Vanilla and NV-SQL by 14× and 1.36×, respectively. The performance gain of *NV-PPL* over NV-SQL is slightly smaller in LinkBench than in TPC-C. This is mainly because the average payload size in LinkBench is larger than that in TPC-C, thus causing more frequent de-PPLizations.

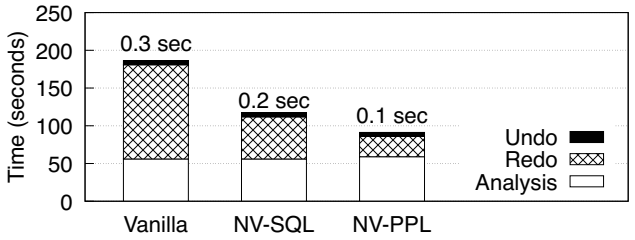


Fig. 9. Recovery Performance

In addition, to evaluate how *NV-PPL* performs under read-intensive workloads, we compare the performance of Vanilla and *NV-PPL* using the TPC-E benchmark, with an initial database size of 64GB and a buffer size of 6GB for 30 minutes. Unlike TPC-C and Linkbench, which have approximately a 2:1 read-to-write I/O ratio, the TPC-E benchmark features a 10:1 ratio [21]. In such read-intensive OLTP workloads, Vanilla does not experience significant write durability overhead, thereby limiting the performance improvement achievable with *NV-PPL*. Nevertheless, *NV-PPL* enhances transaction throughput by approximately 22%.

Finally, to evaluate the impact of bulk insert/append workloads, we experiment with a bulk-insert transaction continuously appending tuples to a dedicated table while running the TPC-C benchmark. The initial database size is set to 54GB, and the buffer size is 5GB, using 31 TPC-C client threads and a single bulk-insert thread via sysbench [42]. When bulk tuples are inserted into a page, the accumulated redo logs in its DRAM log space exceed the PPL_{max}, preventing PPLization (as described in Section 3.1). Consequently, both *NV-PPL* and Vanilla experience increased page writes. Our results show that while the TPS of Vanilla drops by 16%, *NV-PPL* decreases only by 6%. As a result, during the concurrent bulk-insert transaction, *NV-PPL* outperforms Vanilla by 9.7× in TPC-C throughput and by 6.1× in bulk-insert thread throughput. This throughput improvement arises from *NV-PPL*'s ability to reduce page writes from TPC-C, allowing the SSD to handle more writes from the bulk-insert thread, while excessive page writes from TPC-C hinder those from the bulk-insert thread in Vanilla. This demonstrates that *NV-PPL* maintains a performance advantage over Vanilla, even under various concurrent workloads, thanks to selective PPLization and reduced page writes.

6.4 Recovery Performance

To evaluate recovery performance, we simulate a power-off event to force the system to crash after 10 minutes while running the TPC-C benchmark on three MySQL versions (Vanilla, NV-SQL, and *NV-PPL*). For each system, we measure the time to recover the system and present the result in Figure 9.

Analysis phase All three MySQL versions in Figure 9 have the same process to scan and parse the redo log files. In the case of NV-SQL and *NV-PPL*, they have to scan the NVDIMM region additionally but differ in terms of contents and size for inspection. Specifically, *NV-PPL* reads PPL regions in NVDIMM and constructs a mapping table that contains page_id, PPL offset in chunks of tens to hundreds of megabytes, while NV-SQL detects pages in action-inconsistent state by accessing NVDIMM in the unit of page. As such, *NV-PPL* spends more time parsing PPL in NVDIMM and constructing the mapping table. As shown in Figure 9, *NV-PPL* and NV-SQL spend 59 and 56 seconds, respectively, scanning the redo log file and NVDIMM region, whereas Vanilla MySQL takes 56 seconds to scan only the redo log file.

Table 4. Related Performance Metrics (10 warehouses)

		Vanilla	NV-PPL	PPL-MV
Q1	Avg LLT Latency (*s*)	164	87	9
	Version Build (μs)	10,089	3,781	362
	Hit Ratio (%)	97.1	87.8	89.6
Q2	Avg LLT Latency (*s*)	44	23	6
	Version Build (μs)	9,851	1,872	197
	Hit Ratio (%)	97.2	89.1	90.9

Redo phase Since, in the case of *PPL* pages, all their updates are already durable in the presence of crashes, the redo phase in *NV-PPL* can safely skip the redo for *PPL* pages. However, for normal pages, it replays the redo logs in the same manner as Vanilla. Considering that a large fraction of all pages are updated once or more since the last checkpoint and thus they are presumably *PPLized*, *NV-PPL* can skip their redo and thus reduce the read time significantly. Meanwhile, NV-SQL can also reduce the redo phase time by skipping the redo for action-consistent NVDIMM pages. However, NV-SQL experiences inevitable replaying redo logs against the corresponding old action-inconsistent pages, prolonging the recovery time. In our experiment, the redo phase of Vanilla, NV-SQL, and *NV-PPL* took 124, 56, and 27 seconds, respectively. This corroborates our other observation that *NV-PPL* applied about 22% of the entire redo log, thereby reducing disk IOs up to 50%.

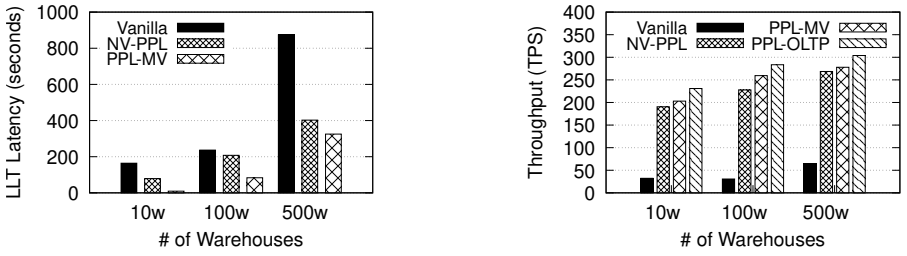
Undo phase Three MySQL versions follow the same undo process, except that *NV-PPL* skips undo operations for *PPL* pages. This is achieved by omitting the application of PPL when background rollback threads initially trigger a page read. However, the rollback thread executes asynchronously to revert any effect of uncommitted transactions using undo logs. Furthermore, as most undo operations are conducted in memory, the undo phase takes less than one second for all MySQL versions.

6.5 Effect of *NV-PPL* on HTAP Workloads

To assess the impact of PPL-based version construction on HTAP, we extend the TPC-C benchmark [66] with an additional thread executing LLT queries `Q1` and `Q2` on `Stock` and `Customer` relations, respectively. We test these queries across various database sizes using three MySQL versions: Vanilla MySQL, *NV-PPL* (*i.e.*, *NV-PPL* supporting undo-based multi-versioning), and *PPL-MV* (*i.e.*, *NV-PPL* supporting PPL-based multi-versioning).

Latency of LLT queries The first row of Table 4 displays the average latency for queries `Q1` and `Q2` during an HTAP workload involving 16 TPC-C threads and 1 OLAP query thread executed against a 10-warehouse TPC-C database. Over an hour, the `PPL-MV` configuration executed 29 `Q1` queries, significantly outperforming `NV-PPL` and `Vanilla`, which completed only 5 and 4 queries, respectively. This performance advantage is primarily attributed to `PPL-MV`'s ability to construct tuple versions in constant time. Specifically, the average version construction time for `PPL-MV` is only 10.4% and 3.5% of the time for `NV-PPL` and `Vanilla`, respectively. The slower performance of `Vanilla`, compared to `PPL-MV` can be explained by its reduced overall system speed.

Access skewness influences the efficacy of `PPL-MV`. While both the `Stock` and `Customer` tables exhibit page-level update skewness, the `Stock` table demonstrates a higher degree of skewness [49]. Pages that are frequently updated and accumulate a high number of PPLs are excluded from PPLization due to the selective PPLization strategy, requiring these pages to rely on undo-based multi-versioning. This effect is more pronounced in the `Stock` table due to its higher skewness. To



(a) Average LLT Latency (b) Average OLTP Throughput

Fig. 10. Varying Number of Warehouses on HTAP (Q1)

quantify this, we observe that 7% of pages in the `Stock` table are not PPLized, compared to only 1.8% in the `Customer` table. Consequently, the average version build time for `Q1` is higher than that for `Q2`.

To evaluate the scalability of `PPL-MV` with respect to database size, we repeat the experiments while varying the number of TPC-C warehouses from 10 to 100 and 500, as illustrated in Figure 10. The results, shown in Figure 10a, reveal that the average latency of LLTs increases with database size. This trend is primarily driven by the greater number of tuples requiring version traversal during a full table scan. However, as the database size grows, the version chain length per tuple decreases, reducing the relative performance advantage of PPL-based version construction.

OLTP throughput In addition, we measure the transaction throughput (TPS) of the TPC-C benchmark while running the same experiment described in Table 4, with the results presented in Figure 10b. For context, the comparison includes throughput from a TPC-C-only run without LLTs, labeled as `PPL-OLTP`. The results indicate that LLT queries significantly impair OLTP performance, with the interference being more pronounced in *NV-PPL* compared to `Vanilla`. This outcome is expected, as *NV-PPL* achieves approximately 8× higher throughput than `Vanilla` under standard TPC-C conditions, leading to a faster accumulation of versions in the undo table. However, `PPL-MV` results in a milder drop in transaction throughput. This can be attributed to the 1.8% higher hit ratio of the `PPL-MV` scheme, as indicated in Table 4, which is a result of our hybrid version construction scheme that mitigates buffer pollution caused by undo pages. Meanwhile, while Vanilla shows a better hit ratio due to a larger buffer pool size, its performance still falls short when compared to any *NV-PPL* version.

Another notable insight from Figure 10b is that the relative decline in OLTP throughput becomes less pronounced as the database size increases. This can be attributed to the shorter average version chain length per tuple in larger databases, which reduces the frequency of stale version purging and its disruptive impact on OLTP transactions [17]. As a result, the potential for improvement offered by the `PPL-MV` scheme diminishes for larger workloads.

7 Related Work

In that *NV-PPL* improves transaction throughput, recovery time, and the latency of LLT queries by leveraging per-page redo logs in NVDIMM, we categorize related works into five distinct types.

Write optimizations using redo logs The IPL (in-page logging) [47] aims to enhance write performance in flash-based DBMSs by storing redo logs within a dedicated DRAM area similar to *NV-PPL*. When a page is evicted from the buffer cache, only the logs, rather than the entire page, are written to a dedicated log sector in flash storage. Upon reading the page from SSD, these redo

logs are applied to the old copy to update it. However, unlike *NV-PPL*, IPL requires additional I/O operations for page reads and logging, a drawback exacerbated by the increased size of flash write units [40].

In a similar vein to IPL, cloud-native database systems (*e.g.*, Aurora) have adopted the strategy of shipping redo logs instead of sending the entire updated pages to the disaggregated storage to reduce the write traffic over the network [6, 55, 78, 79]. Upon receiving the redo log stream, background threads at the storage server continuously apply these logs to their corresponding pages [78]. Although leveraging redo logs to reduce write operations is akin to the approach used by *NV-PPL*, cloud databases could further benefit by adopting an *NV-PPL*-like methodology in their storage servers.

NV-SQL NV-SQL [4] and *NV-PPL* both aim to minimize writes to flash storage by leveraging NVDIMM as a cache between DRAM and SSD. However, *NV-PPL* outperforms NV-SQL in several key aspects. Firstly, *NV-PPL* achieves greater write reduction efficiency. While NV-SQL caches entire pages in NVDIMM, *NV-PPL* stores only per-page redo logs. This approach enables *NV-PPL* to absorb writes for more pages using the same amount of NVDIMM, thereby optimizing the utilization of the more expensive NVDIMM device. Secondly, *NV-PPL* offers faster recovery compared to NV-SQL, especially during the redo phase, due to more pages being eligible for redo-less recovery in *NV-PPL* than in NV-SQL. Thirdly, the NVDIMM-resident logs in *NV-PPL* enable a novel PPL-based multi-versioning, significantly improving the latency of LLT queries. In contrast, NV-SQL relies on existing undo-based multi-versioning, which is suboptimal for HTAP workloads. Finally, NV-SQL necessitates workload-specific manual tuning to determine the optimal LSN gap, whereas *NV-PPL* does not require such adjustments.

NVM-aware DBMS architectures Since the emergence of NVM, various studies have explored integrating NVM into traditional database memory hierarchies. For example, NVM Direct [11] uses NVM solely as storage, while MARS [23], SOFORT [65], and FOEDUS [41] use NVM as secondary storage, replacing SSDs. These are more costly per GB as they use high-capacity NVM products like Optane DCPMM [33] for the entire database. Three-tier architectures utilizing DRAM, NVM, and SSD position NVM as an intermediate cache. For example, 3 Tier BM [75] employs NVM as a warm cache for write-hot data, while HyMem [76] and Spitfire [83] migrate records from DRAM to NVM. However, the slower write latency of NVM like Optane DCPMM, compared to NVDIMM and DRAM, makes it less suitable for storing fine-grained logs [4]. For this reason, *NV-PPL* places NVDIMM at the same tier as DRAM, storing redo logs instead of entire pages, thus maximizing the utilization of NVDIMM's persistence and byte-addressability. Furthermore, persistent memory has been widely adopted for database logging due to its intrinsic durability. X-SSD [46], designed for SSD-based systems similar to *NV-PPL*, poses implementation challenges due to necessary SSD modifications. Write-behind logging [12] suggests reordering log writes and using a force and steal policy. However, *NV-PPL* adheres to the traditional steal and no-force policy, making it more compatible with systems utilizing ARIES.

Faster recovery using versions Several multiversion-based database recovery techniques were proposed in the early 1980s, including Reed's *atomic actions* [69] and its variants [16, 20]. A few multi-version-based snapshot isolation techniques [15, 35] can be traced back to these earlier works. Using multiple versions simplifies the implementation of multi-version concurrency control and ensures transaction atomicity. More recently, Microsoft Azure SQL Database introduced a novel recovery mechanism, Constant Time Recovery (CTR), combining ARIES recovery with multi-version concurrency control to achieve constant-time recovery [7]. Similar to *NV-PPL*, CTR leverages multiple versions to facilitate faster recovery. However, *NV-PPL* differs from CTR and other recovery schemes employing multiple versions in that it utilizes the redo logs in NVDIMM for dual purposes — write reduction and fast recovery.

I/O optimization for multi-version support The existing database engines supporting read consistent views construct versions by traversing long version chains in N2O (New to Old) or O2N (Old to New) order [1]. Either way, such traversal incurs multiple disk I/Os for reading undo records scattered over the system-wide tablespace, thus prolonging the query latency and hindering concurrent update transactions on HTAP applications. While existing research efforts [1, 38, 39, 45] address these issues by purging stale versions, they introduce sophisticated data structures with added management overhead. In stark contrast, simply by leveraging the per-page redo logs and thus without any extra data structure, *NV-PPL* can extend the existing disk-based DBMSs supporting the undo-based versions [50, 60, 68] to take the redo-based multi-versioning for PPL pages. Especially, *NV-PPL* can construct versions in constant time for a page that is either PPLized or has a prebuilt version and thus reduce the latency of LLT queries and also does not interfere with concurrent transactions on HTAP workloads.

8 Conclusion

In this paper, we proposed a novel database architecture, *NV-PPL*, which leverages NVDIMM as a *durable per-page redo log cache* so as to avoid the durability overhead in flash-based databases. It captures per-page redo logs and stores them on NVDIMM, thus avoiding a significant fraction of page writes to the storage and boosting the performance of OLTP workloads dramatically. We also showed how the per-page redo logs can be leveraged to boost the recovery phase and also to optimize the I/O overhead in the existing undo-based multi-versioning scheme.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (No.2022R1A2C2008225, 25%, No.RS-2023-00251665, 25%). This work was supported by ICT Creative Consilience Program through the Institute of Information & Communications Technology Planning & Evaluation(IITP) grant funded by the Korea government(MSIT)(IITP-2024-RS-2020-II201819, 25%). This research was supported in part by Samsung Electronics (25%).

References

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

[2] Ibrar Ahmed, Gregory Smith, and Enrico Pirozzi. 2018. *PostgreSQL 10 High Performance: Expert Techniques for Query Optimization, High Availability, and Efficient Database Maintenance.* Packt Publishing.

[3] Mijin An, Soojun Im, Dawoon Jung, and Sang-Won Lee. 2022. Your Read is Our Priority in Flash Storage. *Proceedings of the VLDB Endowment* 15, 9 (2022).

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

[5] Mijin An, In-Yeong Song, Yong-Ho Song, and Sang-Won Lee. 2022. Avoiding Read Stalls on Flash Storage. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22).*

[6] Panagiotis Antonopoulos, Alex Budovski, Cristian Diaconu, Alejandro Hernandez Saenz, Jack Hu, Hanuma Kodavalla, Donald Kossmann, Sandeep Lingam, Umar Farooq Minhas, Naveen Prakash, Vijendra Purohit, Hugh Qu, Chaitanya Sreenivas Ravella, Krystyna Reisteter, Sheetal Shrotri, Dixin Tang, and Vikram Wakade. 2019. Socrates: The New SQL Server in the Cloud. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD '19).* 1743–1756.

[7] Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mittur Venkataramanappa. 2019. Constant Time Recovery in Azure SQL Database. *Proceedings of the VLDB Endowment* 12, 12 (2019).

[8] Panagiotis Antonopoulos, Peter Byrne, Wayne Chen, Cristian Diaconu, Raghavendra Thallam Kodandaramaih, Hanuma Kodavalla, Prashanth Purnananda, Adrian-Leonard Radu, Chaitanya Sreenivas Ravella, and Girish Mittur Venkataramanappa. 2019. Constant Time Recovery in Azure SQL Database. *Proceedings of VLDB Endowment* 12, 12 (2019).

[9] Raja Appuswamy, Goetz Graefe, Renata Borovica-Gajic, and Anastasia Ailamaki. 2019. The Five-Minute Rule 30 Years Later and Its Impact on the Storage Hierarchy. *Commun. ACM* 62, 11 (oct 2019), 114–120.

[10] Timothy G. Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. LinkBench: A Database Benchmark Based on the Facebook Social Graph. In *Proceedings of the 2013 International Conference on Management of Data (SIGMOD '13)*.

[11] Joy Arulraj, Andrew Pavlo, and Subramanya R. Dulloor. 2015. Let's Talk About Storage Recovery Methods for Non-Volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. 707–722. https://doi.org/10.1145/2723372.2749441

[12] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind logging. *Proc. VLDB Endow.* 10, 4 (nov 2016), 337–348. https://doi.org/10.14778/3025111.3025116

[13] Jens Axboe. [n. d.]. FIO (Flexible IO Tester). https://github.com/axboe/fio.

[14] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: an efficient hybrid PMem-DRAM key-value store. *Proc. VLDB Endow.* 14, 9 (may 2021), 1544–1556. https://doi.org/10.14778/3461535.3461543

[15] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. 1995. A Critique of ANSI SQL Isolation Levels. In *Proceedings of ACM SIGMOD*. 1–10.

[16] Paul M. Bober and Michael J. Carey. 1992. On Mixing Queries and Transactions via Multiversion Locking. In *Proceedings of ICDE*. 535–545.

[17] Jan Böttcher, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Scalable Garbage Collection for In-Memory MVCC Systems. *Proceedings of the VLDB Endowment* 13, 2 (2019).

[18] Silas Boyd-Wickizer, M Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. 2012. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*. Citeseer, 119–130.

[19] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's Distributed Data Store for the Social Graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference (USENIX ATC'13)*. 49–60.

[20] Arvola Chan, Stephen Fox, Wen-Te K. Lin, Anil Nori, and Daniel R. Ries. 1982. The Implementation of an Integrated Concurrency Control and Recovery Scheme. In *Proceedings of ACM SIGMOD*. 184–191.

[21] Shimin Chen, Anastasia Ailamaki, Manos Athanassoulis, Phillip B. Gibbons, Ryan Johnson, Ippokratis Pandis, and Radu Stoica. 2011. TPC-E vs. TPC-C: Characterizing the New TPC-E Benchmark via an I/O Comparison Study. *SIGMOD Rec.* 39, 3 (Feb. 2011), 5–10.

[22] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) *(ASPLOS '20)*. Association for Computing Machinery, New York, NY, USA, 1077–1091. https://doi.org/10.1145/3373376.3378515

[23] Joel Coburn, Trevor Bunker, Meir Schwarz, Rajesh Gupta, and Steven Swanson. 2013. From ARIES to MARS: Transaction Support for next-Generation, Solid-State Drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles* (Farminton, Pennsylvania) *(SOSP '13)*. 197–212. https://doi.org/10.1145/2517349.2522724

[24] Dell. 2023. Dell EMC NVDIMM-N Persistent Memory. https://dl.dell.com/topicspdf/nvdimm_n_user_guide_en-us.pdf.

[25] Peter Desnoyers, Ian Adams, Tyler Estro, Anshul Gandhi, Geoff Kuenning, Mike Mesnier, Carl Waldspurger, Avani Wildani, and Erez Zadok. 2023. Persistent Memory Research in the Post-Optane Era *(DIMES '23)*. Association for Computing Machinery, New York, NY, USA, 23–30.

[26] Jim Gray and Franco Putzolu. 1987. The 5 Minute Rule for Trading Memory for Disc Accesses and the 10 Byte Rule for Trading Memory for CPU Time. In *Proceedings of the 1987 International Conference on Management of Data (SIGMOD '87)*. 395–398.

[27] 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. *Proceedings of VLDB Endowment* 16, 9 (2023), 2090–2102.

[28] Stavros Harizopoulos, Daniel J Abadi, Samuel Madden, and Michael Stonebraker. 2018. OLTP through the looking glass, and what we found there. In *Making Databases Work: the Pragmatic Wisdom of Michael Stonebraker*. 409–439.

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

[30] HPE. 2016. HPE 8GB NVDIMM User Guide for HPE ProLiant Gen9 Servers. https://www.hpe.com/psnow/doc/c05351006.

[31] W. Hsu and A. J. Smith. 2003. Characteristics of I/O traffic in personal computer and server workloads. *IBM System Journal*. 42, 2 (april 2003), 347–372.

[32] Theo Härder, Caetano Sauer, Goetz Graefe, and Wey Guy. 2015. Instant recovery with write-ahead logging. *Datenbank-Spektrum* 15, 3 (2015), 235–239.

[33] Intel Corp. 2016. Intel 3D XPoint. https://www.intel.com/content/www/us/en/architecture-and-technology/intel-micron-3d-xpoint-webcast.html.

[34] Intel Corporation. 2023. Intel Memory Latency Checker v3.9. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.
[35] Ken Jacobs. 1995. *Concurrency Control, Transaction Isolation and Serializability in SQL92 and Oracle7*. Oracle White Paper.
[36] Jeong-Uk Kang, Jeeseok Hyun, Hyunjoo Maeng, and Sangyeun Cho. 2014. The Multi-Streamed Solid-State Drive. In *Proceedings of the 6th USENIX Conference on Hot Topics in Storage and File Systems* (Philadelphia, PA) *(HotStorage'14)*. USENIX Association, USA, 13.
[37] 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 *Proceedings of the 2014 International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 529–540.
[38] Jongbin Kim, Hyunsoo Cho, Kihwang Kim, Jaeseon Yu, Sooyong Kang, and Hyungsoo Jung. 2020. Long-Lived Transactions Made Less Harmful. In *Proceedings of the 2020 International Conference on Management of Data (SIGMOD '20)*. 495–510.
[39] Jongbin Kim, Jaeseon Yu, Jaechan Ahn, Sooyong Kang, and Hyungsoo Jung. 2022. Diva: Making MVCC Systems HTAP-Friendly. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD '22)*. 49–64.
[40] Kangnyeon Kim, Sang-Won Lee, Bongki Moon, Chanik Park, and Joo-Young Hwang. 2011. IPL-P: In-Page Logging with PCRAM(demo paper). *Proceedings of the VLDB Endowment* 4, 12 (2011), 1363–1366.
[41] Hideaki Kimura. 2015. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, 691–706.
[42] Alexey Kopytov. 2018. SysBench. https://github.com/akopytov/sysbench.
[43] Bohyun Lee, Mijin An, and Sang-Won Lee. 2023. LRU-C: Parallelizing Database I/Os for Flash SSDs. *Proceedings of the VLDB Endowment* 16, 9 (2023), 2364–2376.
[44] Bo-Hyun Lee, Mijin An, and S W Lee. 2023. A Case for Space Compaction of B-Tree Nodes on Flash Storage. *IEEE Access* 11 (01 2023), 38149–38156.
[45] Juchang Lee, Hyungyu Shin, Chang Gyoo Park, Seongyun Ko, Jaeyun Noh, Yongjae Chuh, Wolfgang Stephan, and Wook-Shin Han. 2016. Hybrid Garbage Collection for Multi-Version Concurrency Control in SAP HANA. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 1307–1318.
[46] Sangjin Lee, Alberto Lerner, André Ryser, Kibin Park, Chanyoung Jeon, Jinsub Park, Yong Ho Song, and Philippe Cudré-Mauroux. 2022. X-SSD: A Storage System with Native Support for Database Logging and Replication. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA) *(SIGMOD '22)*. Association for Computing Machinery, New York, NY, USA, 988–1002. https://doi.org/10.1145/3514221.3526188
[47] Sang-Won Lee and Bongki Moon. 2007. Design of Flash-Based DBMS: An in-Page Logging Approach. In *Proceedings of the 2007 International Conference on Management of Data (SIGMOD '07)*. 55–66.
[48] Baptiste Lepers, Oana Balmau, Karan Gupta, and Willy Zwaenepoel. 2020. KVell+: Snapshot Isolation without Snapshots. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 425–441.
[49] Scott T. Leutenegger and Daniel Dias. 1993. A Modeling Study of the TPC-C Benchmark. In *Proceedings of the 1993 International Conference on Management of Data (SIGMOD '93)*. 22–31.
[50] Jonathan Lewis. 2011. *Oracle Core: Essential Internals for DBAs and Developers* (1st ed.). Apress, USA.
[51] J.L. Lo, L.A. Barroso, S.J. Eggers, K. Gharachorloo, H.M. Levy, and S.S. Parekh. 1998. An analysis of database workload performance on simultaneous multithreaded processors. In *Proceedings. 25th Annual International Symposium on Computer Architecture (Cat. No.98CB36235)*. 39–50. https://doi.org/10.1109/ISCA.1998.694761
[52] Yunus Ma, Siphrey Xie, Henry Zhong, Leon Lee, and King Lv. 2022. Hiengine: How to architect a cloud-native memory-optimized database engine. In *Proceedings of the 2022 International Conference on Management of Data*. 2177–2190.
[53] Bertrand Matthelie. 2023. MySQL cloud services cost comparison: who provides the best value? https://blogs.oracle.com/mysql/post/mysql-cloud-services-cost-comparison-who-provides-the-best-value.
[54] Micron. 2017. Micron Advances Persistent Memory with 32GB NVDIMM. https://investors.micron.com/news-releases/news-release-details/micron-advances-persistent-memory-32gb-nvdimm.
[55] Microsoft. 2023. About log shipping (SQL Server). https://learn.microsoft.com/en-us/sql/database-engine/log-shipping/about-log-shipping-sql-server?view=sql-server-ver16.
[56] C. Mohan. 1993. A Cost-Effective Method for Providing Improved Data Availability During DBMS Restart Recovery After a Failure. In *Proceedings of VLDB*. 368–379.
[57] C. Mohan, Don Haderle, Bruce Lindsay, Hamid Pirahesh, and Peter Schwarz. 1992. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on*

Database Systems 17, 1 (1992), 94–162.
[58] MySQL Community. 2023. InnoDB Recovery. https://dev.mysql.com/doc/refman/5.7/en/innodb-recovery.html.
[59] MySQL Team (Oracle Corp.). 2021. MySQL 5.7 Reference Manual. https://dev.mysql.com/doc/refman/5.7/en/.
[60] MySQL Team (Oracle Corp.). 2021. MySQL Server (GitHub repository). https://github.com/mysql/mysql-server.
[61] MySQL Team (Oracle Corp.). 2024. The InnoDB Multi-Versioning. https://dev.mysql.com/doc/refman/8.4/en/innodb-multi-versioning.html.
[62] Netlist. 2019. Netlist NVvault DDR4 NVDIMM-N. https://www.storagenewsletter.com/2019/11/13/netlist-nvme-ssds-and-nvvault-ddr4-nvdimm-n-validated-on-2nd-gen-amd-epyc-processors/.
[63] Oracle Corp. 2021. Oracle Redo Log buffer. https://docs.oracle.com/en/database/oracle/oracle-database/21/tgdba/database-memory-allocation.html.
[64] Oracle (MySQL Team). 2024. rw_lock_t Struct Reference. https://dev.mysql.com/doc/dev/mysql-server/latest/structrw__lock__t.html.
[65] Ismail Oukid, Daniel Booss, Wolfgang Lehner, Peter Bumbulis, and Thomas Willhalm. 2014. SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware* (Snowbird, Utah) *(DaMoN '14)*. Article 8, 7 pages. https://doi.org/10.1145/2619228.2619236
[66] Percona. 2018. tpcc-mysql. https://github.com/Percona-Lab/tpcc-mysql.
[67] pmem.io. 2022. Persistent Memory Development Kit. https://pmem.io/pmdk/.
[68] PostgreSQL Global Development Group. 2023. PostgreSQL 16.1 Documentation. https://www.postgresql.org/docs/16/index.html.
[69] David Reed. 1983. Implementing Atomic Actions on Decentralized Data. *ACM Transactions on Computer Systems* 1 (1983), 3–23.
[70] Chris Ruemmler and John Wilkes. 1993. *A trace-driven analysis of disk working set sizes.* Technical Report HPL–OSR–93–23. Hewlett-Packard Laboratories, Palo Alto, CA, USA.
[71] Samsung. 2016. Samsung DDR4 Memory. https://semiconductor.samsung.com/dram/module/rdimm/m393a2g40db0-cpb/.
[72] Samsung. 2017. Samsung 960 PRO 1TB. https://www.samsung.com/us/computing/memory-storage/solid-state-drives/ssd-960-pro-m-2-1tb-mz-v6p1t0bw/.
[73] Samsung Electronics Corp. 2023. Memory Semantic SSD (Persistent Mode). https://semiconductor.samsung.com/us/news-events/tech-blog/webinar-memory-semantic-ssd/.
[74] Sauer, Caetano and Graefe, Goetz and Härder, Theo. 2018. FineLine: Log-Structured Transactional Storage and Recovery. *Proceedings of the VLDB Endowment* 11, 13 (2018).
[75] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 1541–1555. https://doi.org/10.1145/3183713.3196897
[76] Alexander van Renen, Viktor Leis, Alfons Kemper, Thomas Neumann, Takushi Hashida, Kazuichi Oe, Yoshiyasu Doi, Lilian Harada, and Mitsuru Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*.
[77] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Building blocks for persistent memory: How to get the most out of your new memory? *The VLDB Journal* 29, 6 (sep 2020), 1223–1241. https://doi.org/10.1007/s00778-020-00622-9
[78] Alexandre Verbitski, Anurag Gupta, Debanjan Saha, Murali Brahmadesam, Kamal Gupta, Raman Mittal, Sailesh Krishnamurthy, Sandor Maurice, Tengiz Kharatishvili, and Xiaofeng Bao. 2017. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 International Conference on Management of Data (SIGMOD '17)*.
[79] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, Yujie Wang, Baokai Chen, Chang Cai, Yubin Ruan, Xiaoyi Weng, Shibin Chen, Liang Yin, Chengzhong Yang, Xin Cai, Hongyan Xing, Nanlong Yu, Xiaofei Chen, Dapeng Huang, and Jianling Sun. 2023. PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba. *Proc. ACM on Management of Data* 1, 2, Article 199 (jun 2023), 25 pages.
[80] Tianzheng Wang and Ryan Johnson. 2014. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment* 7, 10 (2014), 865–876.
[81] Jianhua Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of USENIX conference on File and Storage Technologies (FAST'20)*.
[82] Jiacheng Yang, Ian Rae, Jun Xu, Jeff Shute, Zhan Yuan, Kelvin Lau, Qiang Zeng, Xi Zhao, Jun Ma, Ziyang Chen, Yuan Gao, Qilin Dong, Junxiong Zhou, Jeremy Wood, Goetz Graefe, Jeff Naughton, and John Cieslewicz. 2020. F1 Lightning:

HTAP as a Service. *Proceedings of VLDB Endowment* 13, 12 (2020), 3313–3325.
[83] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. *Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory*. 2195–2207. https://doi.org/10.1145/3448016.3452819

Received July 2024; revised September 2024; accepted November 2024