NV-SQL: Boosting OLTP Performance with Non-Volatile DIMMs

Mijin An[†]
Sungkyunkwan University
meeeejin@skku.edu

Jonghyeok Park[†*]
Hankuk University of Foreign Studies
jonghyeok.park@hufs.ac.kr

Tianzheng Wang
Simon Fraser University
tzwang@sfu.ca

Beomseok Nam
Sungkyunkwan University
bnam@skku.edu

Sang-Won Lee
Sungkyunkwan University
swlee@skku.edu

ABSTRACT

When running OLTP workloads, relational DBMSs with flash SSDs still suffer from the *durability overhead*. Heavy writes to SSD not only limit the performance but also shorten the storage lifespan. To mitigate the durability overhead, this paper proposes a new database architecture, NV-SQL. NV-SQL aims at absorbing a large fraction of writes written from DRAM to SSD by introducing NVDIMM into the memory hierarchy as a durable write cache. On the new architecture, NV-SQL makes two technical contributions. First, it proposes the *re-update interval-based admission policy* that determines which write-hot pages qualify for being cached in NVDIMM. It is novel in that the page hotness is based solely on pages' LSN. Second, this study finds that NVDIMM-resident pages can violate the *page action consistency* upon crash and proposes how to detect inconsistent pages using per-page `in-update` flag and how to rectify them using the redo log. NV-SQL demonstrates how the ARIES-like logging and recovery techniques can be elegantly extended to support the caching and recovery for NVDIMM data. Additionally, by placing write-intensive redo buffer and DWB in NVDIMM, NV-SQL eliminates the *log-force-at-commit* and *WAL* protocols and further halves the writes to the storage. Our NV-SQL prototype running with a real NVDIMM device outperforms the *same-priced* vanilla MySQL with larger DRAM by several folds in terms of transaction throughput for write-intensive OLTP benchmarks. This confirms that NV-SQL is a cost-performance efficient solution to the durability problem.

PVLDB Reference Format:
Mijin An, Jonghyeok Park, Tianzheng Wang, Beomseok Nam and Sang-Won Lee. NV-SQL: Boosting OLTP Performance with Non-Volatile DIMMs. PVLDB, 16(6): 1453 - 1465, 2023.
doi:10.14778/3583140.3583159

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/meeeejin/mysql-57-nvdimm-caching.

Proceedings of the VLDB Endowment, Vol. 16, No. 6 ISSN 2150-8097.
doi:10.14778/3583140.3583159

† Both authors contributed equally.
* Work done while in Sungkyunkwan University.

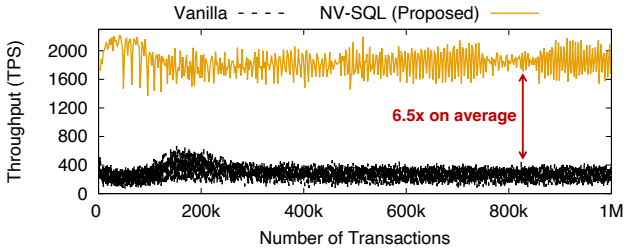

Figure 1: Throughput of two systems under the same budget. NV-SQL achieves 6.5× higher throughput with NVDIMM-N.

1 INTRODUCTION

As SSDs continue to evolve with higher performance and better cost-effectiveness, storage-centric database systems remain the mainstream and economical choice for most applications [4]. Despite the high bandwidth provided by modern SSDs, however, they still exhibit high latency at the microsecond level (e.g., Samsung 970 PRO's latency can be as high as over 300μs [1]), which is significantly higher than DRAM latency at the tens of nanosecond level (e.g., 50-100ns). This leads to non-trivial durability overheads when data has to be persisted from the DRAM-resident buffer to SSD due to checkpointing, eviction, and log flushing upon commit. Such storage accesses happen frequently in relational DBMSs and especially limit the performance of write-intensive OLTP workloads [2, 3].

DRAM-based non-volatile DIMMs (NVDIMMs) [14, 27] offer ample opportunity to bridge the latency gap. In particular, NVDIMM-N bundles normal DRAM and flash memory on a single DIMM with super-capacitors. It operates as normal DRAM. Upon power failure, the supercapacitor provides enough power to flush the DRAM contents to the on-board flash memory. The data is loaded back from flash to DRAM when the system is powered up again. Like recent scalable persistent memory products such as Optane PMem [18], NVDIMMs offer persistence and byte-addressability on the memory bus. This allows them to be placed side-by-side with DRAM and accessed using `load` and `store` instructions without going through the software storage stack. However, unlike scalable PMem, which is typically 3–5× slower than DRAM, NVDIMMs exhibit the same performance characteristics as typical DRAM.

Notably, NVDIMMs are based on DRAM, so their capacity is limited by DRAM capacity in the order of tens of GBs. However, we observe that, in fact, the key to bridging the latency gap is allowing frequently checkpointed/flushed data to be quickly persisted. More

importantly, OLTP workloads tend to focus on a small hot data set. Typically, the amount of hot data is small enough to fit well within NVDIMM capacity. Based on this observation, we advocate using an NVDIMM-resident durable cache for staging hot pages. To address the write durability overhead in the memory hierarchy of DRAM and flash SSD, we propose a new database architecture, NV-SQL, which leverages NVDIMMs as a cost-effective way to reduce write traffic to flash SSD and thus boosts transaction throughput.

While the idea of using NVDIMM as a durable cache for hot data is straightforward, realizing it in existing relational DBMS engines is non-trivial due to several challenges. First, the system must be able to efficiently identify and admit hot data to maximize the use of NVDIMM. Second, since the DBMS already uses a general-purpose buffer pool in DRAM, adding the NVDIMM cache, in essence, introduces a second buffer pool to the system. Hence, it must be managed with a replacement policy that guarantees data consistency across the two caching locations. To meet those challenges, NV-SQL demonstrates how the ARIES-like logging and recovery techniques [30] can be successfully and elegantly extended to support NVDIMM caching. Specifically, we make the following contributions by tackling these challenges:

- We observe *write working set* in OLTP workloads: only a tiny fraction of the entire database is actively updated at a time window. In addition, we show that NVDIMM is a very economical device to keep such write working set.
- We propose a *re-update interval-based admission policy* to determine which pages qualify for being cached in NVDIMM based on their LSN and a *replacement policy* for NVDIMM buffer whose goal is to evict false or less hot pages quickly.
- Since the effect of logical update actions (*e.g.*, record insertion) against NVDIMM pages becomes immediately durable, some pages can violate the *page action consistency* upon crash. NV-SQL suggests detecting inconsistent pages using the per-page `in-update` flag and rectifying them using the redo log.
- By placing write-intensive objects of redo buffer and DWB in NVDIMM, NV-SQL eliminates the *log-force-at-commit* and *WAL* protocols and further halves the writes to the storage.
- We prototype NV-SQL with moderate non-intrusive modification to the codebase of MySQL. With the NVDIMM smaller than 2% of the database size, NV-SQL improves throughput by up to 6.5x over the *same-priced* vanilla MySQL with larger DRAM. In addition, by skipping the redo recovery for durable write-hot pages in NVDIMM, NV-SQL can also reduce the recovery time by one-fourth.

2 BACKGROUND

2.1 Durability Tax in DBMSs

In relational DBMS, transactional atomicity and durability are guaranteed by the *log-force-at-commit* and *write-ahead-logging (WAL)* protocols. With the logging techniques, dirty pages can be written back to storage asynchronously. Though asynchronous in nature, however, writes are one of the major performance bottlenecks when running write-intensive OLTP workloads, even on high-performance SSDs. In particular, excessive writes to flash storage with asymmetric read and write performance hinders overall performance, and the read operations are often blocked by its preceding write operations in both database buffer and SSD buffer layers, which degrades the throughput and latency [2, 3]. To improve the read performance of flash storage and boost OLTP performance, excessive writes to flash storage need to be steered to a faster durable cache so that the write traffic going to the flash storage can be throttled; the fewer write requests, the more read operations the flash storage can process.

Table 1: Write Bandwidth: NVDIMM-N vs. DCPMM

Bandwidth	Sequential Write			Random Write		
(GB/s)	64B	128B	256B	64B	128B	256B
NVDIMM-N	5.8	10.6	15.8	4.9	6.2	8.3
DCPMM	1.6	1.9	2.0	0.5	0.6	0.7

2.2 Non-Volatile DIMMs

Although the performance of storage devices has been getting faster, it is still bounded by the PCIe bus speed as they are connected as peripheral devices [5]. To make the performance of storage surpasses that of PCIe bus, various non-volatile memory (NVM) technologies, such as Intel's Optane DCPMM [18] and NVDIMM [14], have been developed to use the same form factor as DRAM. As installed on the memory bus, they exhibit performance similar to DRAM while providing persistence and byte-addressability.

NVDIMM-N uses both DRAM and Flash, and it is the best in terms of latency. DRAM is memory mapped and used during normal operation, and Flash is used as backup media for durability in the event of a power outage. Therefore, NVDIMM-N provides similar performance to DRAM and is practically free from wearing issues, unlike other NVMs. However, most supercapacitors in NVDIMM-N are used as a backup power source when a power failure occurs, so its capacity is limited and smaller than several tens of GB.

Table 1 shows the write bandwidth of DCPMM and NVDIMM-N that we measure in our testbed machine equipped with a single 128 GB Optane DCPMM and a single 16 GB HPE NVDIMM-N. With 256B random writes, the bandwidth of NVDIMM-N is up to 12× higher than that of DCPMM. Since OLTP workloads frequently perform small and random updates followed by cache-line flush commands [55], DCPMM's tardy write latency makes DCPMMs unsuitable for durable write buffer cache compared to NVDIMMs. Hence, only NVDIMM-N is considered a storage medium for durable write buffer, and NVDIMM-N will be referred to as NVDIMM hereinafter.

2.3 Write Characteristics of OLTP Workloads

Understanding the I/O characteristics of a given workload is a prerequisite for improving I/O performance. In OLTP workloads, there are two main write characteristics. First, write queries have *temporal locality* [6, 26, 28]. OLTP workloads write a small number of new pages repeatedly over a short period of time. Once they are updated heavily, they tend not to be accessed for a while until the access pattern changes, *i.e.*, write operations alternate between hot and cold phases. Second, *write skews* are common [6, 26, 28]. Certain tables have a small number of tuples updated frequently during

Table 2: Write Fraction by Types (TPC-C)

Buffer Size (%)	10	20	30	40	50
Replacement Write (%)	97.3	92.9	81.0	66.3	46.8
Checkpoint Write (%)	2.7	7.1	19.0	33.7	53.2

short periods that repeat periodically, whereas most other tuples are rarely updated [6, 26]. As detailed later, NV-SQL considers these characteristics when designing the page admission policy.

2.4 Page-Action consistency

A transaction invokes page modifications, which are translated into a sequence of *logical page actions*, each applied to an individual physical page. In that the conventional DBMSs keep track of every such *logical action* for every *physical page* for recovery, they are said to take the *physio-logical logging* approach. An important assumption made by page-oriented recovery schemes such as ARIES is that all database pages are consistent with respect to each logical page-updating action [20]. To guarantee the action consistency for pages, a process has to strictly follow the *fix-rule* [21] when updating a page: the process has to fix the page first, then make all modifications and generate log records, and finally unfix the page. However, note that a physical page in DRAM can be inconsistent because a logical action typically updates multiple parts of a physical page. Thus, if the system crashes after only some subtask completes, the page will be in an inconsistent state.

In the conventional memory hierarchy with volatile DRAM and non-volatile secondary storage, such a transient inconsistency due to a page action does not cause any recovery issue. This is because only *unfixed* and thus action-consistent pages are written to the storage, and DRAM-resident pages are lost despite the page might be in an inconsistent state at the time of failure. The recovery module can restore the database from a failure by transforming the action-consistent database into the transaction-consistent one using logical redo and undo logs. On the other hand, in the memory hierarchy with NVDIMM, since transient inconsistency for NVDIMM-resident pages is durable, the database engine needs a new mechanism to guarantee the page-action consistency for NVDIMM pages upon failures (which is our focus).

3 MOTIVATION

3.1 Write Working Set and Write Types

Given the cost per gigabyte of commercial NVDIMM devices, it is not cost-effective to use a large NVDIMM as the sole buffer cache to keep all pages in the working set, regardless of their update hotness. However, OLTP workloads are known to have a small set of pages that are frequently updated during a specific time window due to temporal locality and skewed writes [6, 26]. We call such pages as *write-hot* and refer to the set as *write working set*. The size of write working set is typically small (*e.g.*, 2.6~6.7% of total database size) [11], but such a write working set allows a large portion of the writes to be absorbed in NVDIMM cost-effectively.

Meanwhile, there are only two types of writes that occur in DBMSs: (i) *Replacement writes* caused by the buffer replacement policy for cold dirty pages and (ii) *checkpoint writes* for hot dirty pages that the checkpointing module flushes to disk to reduce recovery time. To get insight into which pages need to be kept in small NVDIMM, we examine the type of pages in the write working set. Table 2 shows the proportion of each write type. As the buffer size increases, replacement writes decrease. This is because a larger buffer cache can hold more dirty pages, which results in fewer dirty pages being replaced. On the other hand, a larger buffer cache increases transaction throughput, and thus redo logs accumulate faster. As a result, the DBMS triggers checkpointing more frequently; thus, the proportion of checkpoint writes increases.

As detailed later, our proposed NV-SQL exploits these characteristics in designing its admission policy to effectively identify write-hot pages worth keeping in NVDIMM regardless of buffer cache sizes and the dominant write types.

Table 3: Price and Performance Characteristics

Storage	Random IOPS (4KB)		Unit	Unit
Media	Read	Write	Capacity	Price
DRAM [43]	-	-	32GB	$180
NVDIMM [33]	8,960K	5,152K	32GB	$540
SSD [44]	530K	51K	512GB	$350

3.2 Five-Minute Rule for Durability

When should data be kept in DRAM, and when should it be kept on disk? A simple guide to data placement is to keep infrequently accessed data on disk and bring it into DRAM when needed while keeping frequently accessed data in DRAM. As a practical rule of thumb to determine the access interval which justifies data to be cached in memory, Jim Gray et al. put forth the five-minute rule for trading memory to reduce disk accesses [16]. The rule explores the trade-off between the cost of memory and the cost of disk accesses and computes the break-even access interval using the formula:

$$\frac{\text{Pages per MB of Memory}}{\text{IOPS of Disk}} \times \frac{\text{Disk Price}}{\text{Memory Price per MB}}$$

When NVDIMM is used as a durable cache (*i.e.*, *Memory* in the formula) for SSD (*i.e.*, *Disk* in the formula), the above formula can be applied to calculate the break-even write interval for data to be cached in NVDIMM. That is, this break-even interval can be used as an economic formula for durability against the aforementioned write working set. Calculating the cost of writing a single 4KB page in SSD and the cost of caching the same page in NVDIMM based on Table 3, the break-even point is 106 seconds. That is, if a page has to be repeatedly written to SSD every or less than 106 seconds, it is more cost-effective to cache the page in NVDIMM durably.

Considering that the capacity of NVDIMM is much smaller than SSD, if too many pages are written frequently in the break-even interval, the NVDIMM will not be able to absorb all of the write-hot pages, *i.e.*, it will drain write-hot pages to SSD, and the performance may be bounded by SSD performance. To determine whether NVDIMM can be used as a cost-effective cache for write working sets, we ran the TPC-C workload for 6 hours and measured the percentage of pages written at least once every 106 seconds. We set

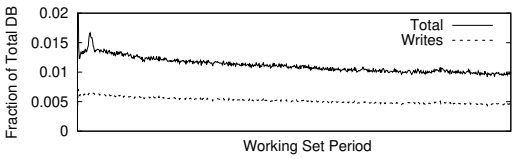

Figure 2: Working Set Size over Time (TPC-C)

the database size to 54 GB (*i.e.*, 500 warehouses) and the buffer size to 11 GB (*i.e.*, 20% of the database size). Figure 2 shows the fractions of pages written more than once in the time window (denoted as `writes`) out of the total number of pages accessed in the same time window (denoted as `total`). Overall, the write working set size accounts for a very small portion (*i.e.*, 0.54% on average, 300 MB). This result is consistent with results reported in [11].

3.3 Threats to Page-Action Consistency

If a system crashes while logical actions are updating NVDIMM-resident pages, those pages may remain action-inconsistent upon restart. If a page is durable in NVDIMM and action-consistent despite a system failure, the page does not require redo recovery but only undo recovery. However, if the logical update action for an NVDIMM page is not completed at the time of the crash, the page is action-inconsistent upon restart and cannot be recovered using the logical undo/redo logs. Furthermore, if an NVDIMM page becomes page-action inconsistent, the durability of all its previous committed updates, while it resides in NVDIMM, is also lost. For this reason, the guarantee of page-action consistency for logical actions on the NVDIMM-resident page must be ensured. While providing write buffering and durability, the NVDIMM caching brings the problem of page-action consistency. Thus, as we described in Section 5.1, we need to devise a mechanism to guarantee the action consistency for NVDIMM-resident pages upon failures: *to detect action-inconsistent pages* and *to rectify them to be consistent.*

4 NV-SQL

NV-SQL leverages NVDIMM to absorb write-hot pages with minimal modifications to a mature DBMS implementation, MySQL. The key benefit of using NVDIMM as a durable buffer cache alongside the conventional DRAM buffer is that it can fully utilize the existing mature DRAM-SSD architecture while persisting dirty pages at the speed of NVDIMM. Due to the small write working set size, NV-SQL can reduce the number of writes to SSD by placing frequently updated pages on a small NVDIMM. With this goal in mind, NV-SQL has to identify the write working set (*i.e.*, write-hot pages in OLTP workloads) to cache in the small NVDIMM buffer cache.

4.1 Design Choice for Tiering

NV-SQL is a non-hierarchical DBMS architecture in that the NVDIMM buffer cache is located at the same tier as the DRAM buffer cache, unlike the three-tier architectures proposed in the previous literature [52, 56]. That is, the pages cached in the DRAM buffer cache and those in the NVDIMM buffer cache are disjoint in NV-SQL, *i.e.*, a page can reside in only one of the two caches at any moment. Considering that NVDIMM works at the same speed

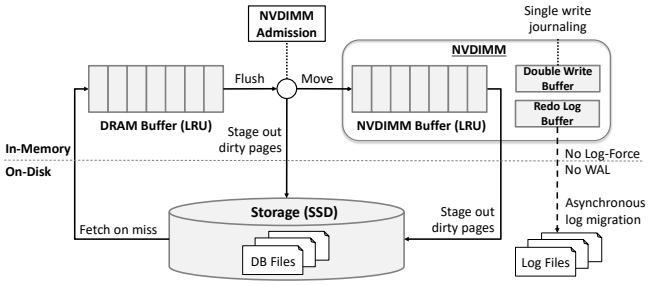

Figure 3: NV-SQL: Overview

as DRAM, it is neither beneficial nor economical to use NVDIMM as a cache between DRAM and SSD. In such a three-tier architecture, foreground database processes can not directly access pages in NVDIMM despite DRAM and NVDIMM being in the same address space. This results in a lower hit ratio than our two-tier architecture, allowing CPUs to access NVDIMM directly. Furthermore, upon a miss for a page in NVDIMM, the page should be copied back to DRAM. In addition, if a page is updated in DRAM, it should eventually be copied into the middle NVDIMM tier when it is evicted. This memory copy overhead for write will stand out for frequently checkpointed pages. As a result, the frequent memory copy between DRAM and NVDIMM makes it impossible to achieve better performance than our two-tier architecture. More importantly, considering that NVDIMMs are costlier than DRAM and all dirty pages evicted from DRAM, regardless of hot or cold, move to the NVDIMM cache with limited capacity, the effect of absorbing writes in NVDIMM is marginal. For this reason, we decided to choose the two-tier architecture over the three-tier one.

4.2 Basic Framework

Figure 3 illustrates the architecture of NV-SQL and the interactions between its components.

- When a page is requested, DRAM and NVDIMM buffer caches are searched for that page. If the page is found in either one, it is fetched and used. Otherwise, it is read from storage and cached in the DRAM buffer. Note that pages in storage cannot be fetched to NVDIMM without going through the DRAM buffer.
- When a page is flushed from the DRAM buffer, it is either staged to the NVDIMM buffer or written back to storage, depending on whether it is hot enough to be cached in the NVDIMM. That is, the admission criteria described below identify whether it is worth caching in NVDIMM. If it is a write-hot page, it is moved to the NVDIMM buffer. Otherwise, it is written back to storage.
- When a page is evicted from the NVDIMM buffer according to the replacement policy, it is written to storage as in the conventional buffer cache replacement mechanism.

4.3 Page Admission Policy

Pages in database tables with skewed access patterns often have high temporal locality. Since it is not difficult to identify skewed write access at the table level, it can be beneficial to steer such

write requests to the NVDIMM buffer if a table is known to have skewed access. However, if such a table is large and there exist a large number of pages that are within the break-even interval for durability, it is not possible to steer all those pages, but they need to be selectively cached in NVDIMM. To select the most write-hot pages, we use two database semantics: (i) *checkpoint* and (ii) *log sequence number (LSN)*. Let us detail how these two classic database semantics are used to identify the write-hotness.

Checkpoint-based Admission To expedite the recovery process, most DBMSs, including MySQL, Oracle, PostgreSQL, and IBM DB2, track dirty pages using a list (called *dirty page list*) sorted by LSN, an ever-increasing value representing offset recorded in the log file, checkpoint them to the durable storage and truncate log files periodically [17, 20, 22, 29, 34, 42, 45]. This fuzzy checkpointing asynchronously flushes dirty pages from the head of the dirty page list in small batches without blocking database operations, including the LRU replacement [34, 45]. That is, the fuzzy checkpointing spreads disk writes across the checkpoint interval.

If a dirty page is not accessed for a long time, it is evicted by the LRU replacement mechanism even before being selected for the checkpoint target and removed from the list. Therefore, NV-SQL regards the pages in the head of the dirty page list as write-hot and, upon checkpoints, unconditionally migrates them to the NVDIMM buffer, even if some of them are no longer write-hot and thus *falsely* admitted. By leveraging the existing checkpoint mechanism, this admission policy does not require any prior analysis of the workload. As such, it is a general policy that can be easily implemented in any DBMS supporting the fuzzy checkpointing [17, 22, 29, 34, 42, 45].

Re-update Interval-based Admission As shown in Table 2, most writes occur due to the buffer cache replacement, especially when the buffer cache size is small. Therefore, it is necessary to distinguish write-hot pages from uncheckpointed pages and migrate them to NVDIMM to reduce write traffic. For this (*i.e.*, LRU writes), we utilize the concept of LSN maintained per page to estimate the write-hotness of the page. DBMSs supporting ARIES-like recovery and fuzzy checkpointing mechanism manage two types of LSN for each page [30]: (i) `Last-Flush-LSN` and (ii) `First-Update-LSN`. `Last-Flush-LSN` is LSN when a page is last flushed to disk, and `First-Update-LSN` is LSN when a page is first updated after being fetched into the buffer. Leveraging these two LSNs, we can obtain the following equation:

$$\text{Re-update Interval} = \text{First-Update-LSN} - \text{Last-Flush-LSN}$$

Re-update Interval is an indicator of how quickly an evicted page is brought back into the cache. If the difference between these two LSNs of a page is small, it means that the page is, as soon as flushed to disk, read back to the DRAM buffer and updated in a short time. Conversely, a large difference indicates that the page is read back and updated long after the last write. These pages are write-cold, so they are not worth caching in NVDIMM. That is, pages with a short Re-update Interval can be considered as write-hot, so they need to be cached in the NVDIMM buffer.

Due to the limited NVDIMM capacity, given a Re-update Interval, we need to convert it into a relative write-hotness rank, *i.e.*, the rank of the page among all the working set pages. If the rank of the page is within a predefined threshold percentage (*e.g.*, 10%), the page is admitted to the NVDIMM buffer cache. In order to convert the Re-update Interval score to the write-hotness rank, the distribution of the Re-update Interval scores of all the write working set pages must be known. The distribution of Re-update Interval scores can be sampled and predicted on-the-fly. Alternatively, the workload can be profiled in advance, and a threshold can be manually selected. We currently assume offline profiling to set the threshold. We defer dynamically changing the threshold online to future work.

4.4 NVDIMM Buffer Replacement Policy

Given the limited NVDIMM capacity, the data pages in the NVDIMM buffer need to be managed judiciously for better utilization. Which buffer replacement algorithm does work best for the NVDIMM cache? Recall that the main role of NVDIMM is not to maximize the hit ratio but to minimize writes to SSD, while the primary role of DRAM is to keep frequently accessed pages cached for fast processing and to increase the hit ratio. Given its simplicity and adaptability, the obvious first choice is the Least Recently Used (LRU) replacement policy. It manages pages in the buffer cache according to the page access recency and selects a page not used for the longest time as a victim since it is least likely to be accessed soon. Because pages with high temporal locality are migrated to the NVDIMM buffer, LRU can be a good alternative.

However, we need to prevent *false-positive* write-hot pages as much as possible from being cached into the NVDIMM buffer. Some pages might be checkpointed or have a short Re-update Interval, even though they are no longer write-hot. Our admission policy may unfortunately admit such pages that do not deserve to be cached in the NVDIMM buffer. Hence, for better utilization of limited NVDIMM, it is desirable to evict those pages at the earliest time.

For this purpose, we inherit the midpoint insertion algorithm from MySQL [36]. The LRU list is divided into a new sublist and an old one. The new sublist holds the younger data, while the other has the older data. When a new page is read into the buffer cache, it is initially inserted at the head of the old list, i.e., in the middle of the entire LRU list. If a page hit occurs in the old sublist, it is made young and moved to the head of the new sublist. Otherwise, if a page hit occurs in the new sublist, the page is transferred to the head of the new sublist only if it is within a predefined distance from the head. If not, the page stays in place. As the database operates, pages that are not accessed age by moving toward the tail of the list. Eventually, they reach the tail of the old sublist and are evicted.

Despite these efforts to prevent false-positive write-hot pages from polluting the new sublist, those pages can still enter the new sublist and evict warm pages. To mitigate this problem, we introduce *an update count-based midpoint insertion*, the variant of frequency-based replacement [41]. To be specific, we maintain an update count for each page, which increases when it is updated in the NVDIMM cache. The update count indicates how many times a page has been updated after entering the old list in the NVDIMM buffer. Hence, if a page hit occurs in the old sublist, but its update count is less than one (*i.e.*, never been updated since it entered the buffer), it remains in the old sublist. Otherwise, it moves to the new sublist. As a result, the NVDIMM buffer can quickly age out pages that are only accessed once while fully utilizing the new sublist for true-positive write-hot pages.

4.5 Prototype Implementation

The NV-SQL scheme has been implemented as an extended buffer manager module of MySQL 5.7.

Buffer Allocator This module is modified to allocate the NVDIMM buffer and extend buffer management to the NVDIMM area, for which we leverage the existing buffer allocation mechanism in MySQL. Since pages are managed across DRAM and NVDIMM buffers, data structures needed for buffer management, such as hash tables for page lookup and locks/latches for page consistency, are managed for each buffer. NV-SQL stores buffer frames in NVDIMM to ensure persistence, and other data structures for buffer management are stored on DRAM. The key to such placement is that those metadata, even for NVDIMM resident pages, can be reconstructed during recovery without requiring persistence. An update count is also added to each page in the NVDIMM buffer to decide whether to move a page to the new sublist. The NVDIMM buffer size is 1GB by default, which is relatively small compared to the DRAM buffer size (*e.g.*, 8GB). NV-SQL organizes the NVDIMM buffer into 8 instances (*i.e.*, as Vanilla does) and dedicates each instance to its DRAM counterpart, thereby reducing contention when migrating pages from DRAM to NVDIMM.

Buffer Replacement This module manages the page admission policy for the NVDIMM buffer. When a page is evicted from the DRAM buffer, the aforementioned criteria (*i.e.*, checkpoint write or short Re-Update Interval) are tested, and a boolean flag is set to indicate whether the page will be moved to the NVDIMM buffer. Then, to migrate the target page to the NVDIMM buffer, one free buffer frame is removed from the NVDIMM buffer, and the content of the DRAM buffer frame is copied to the NVDIMM buffer frame using `memcpy`. To ensure the order and durability of memory writes, `mfence` is called before and after each `clflush` call. If a failure occurs during `clflush`, we follow the conventional redo semantics of DBMS for recovery. We then insert the NVDIMM buffer frame into the midpoint of the NVDIMM buffer. Finally, we discard the content of the original DRAM buffer frame, and the emptied page is added to the free list of the DRAM buffer.

Read Buffer In NV-SQL, data pages are cached across DRAM and NVDIMM buffers, so both buffers need to be searched for a read request, but note that hash table lookups incur negligible overhead.

Flush Buffer A background writer dedicated to the NVDIMM buffer is introduced to make free space timely. The background writer flushes dirty pages when it receives a signal that the buffer manager runs out of free pages.

5 LOGGING AND RECOVERY IN NV-SQL

One of the challenges that NVDIMM caching needs to address is the page-action consistency for each individual page. In this section, we describe the failure-atomic page update method. We also discuss the benefit of placing a redo buffer and double write buffer at NVDIMM.

5.1 Alternatives for Page-Action Consistency

There are at least three alternatives to achieve page-action consistency for NVDIMM pages upon crashes - Copy-on-Write (CoW), undo-based, and redo-based approaches. This subsection explains the first two approaches and discusses why they are discarded.

CoW-based Page-Action Consistency The CoW approach maintains action-consistent pages by occupying additional space in NVDIMM and ensures an action-consistent state by means of invoking memory copy and cache flush operations. With the copy-on-write (CoW) approach, page action consistency is no more problematic since action-consistent pages are preserved as shadow pages. Despite the benefit in terms of consistency, unfortunately, the CoW approach has two drawbacks: (i) it halves the capacity of NVDIMM to accommodate shadow pages, lowering the write-absorbing effect. and (ii) it suffers from the overhead of memory copy and cache flush on every page modification. In fact, we found from a separate experiment that the CoW version underperforms our scheme by 20% in terms of transaction throughput.

Undo-based Page-Action Consistency Intel's PMDK [39] provides transactional APIs that perform undo logging to ensure failure-atomicity when updating data in non-volatile memory. However, PMDK transactions are known to have a significant overhead [32, 46], and database systems must be changed to use its 16-byte fat pointers. In addition, such undo-based page action consistency techniques entail the following problems. First, storing fine-grained and complex logs for microoperations within pages incurs excessive `clflush` instruction overhead. If we enlarge the atomic write granularity to a page, the log size will increase and waste scarce NVDIMM space. Second and more importantly, NV-SQL should support recovery not only for durable pages in the NVDIMM buffer cache but also for volatile pages in DRAM buffer cache. That is, volatile pages are recovered by redoing the logs in a durable storage and then undoing the incomplete actions in most DBMSs, including MySQL. If the NVDIMM buffer cache employs a recovery method that is independent of the recovery method for the volatile buffer cache, restored pages in DRAM may conflict with their corresponding restored pages in the NVDIMM buffer cache.

5.2 Redo-based Page Action Consistency

A simple but effective recovery method for the NVDIMM buffer cache while compatible with the existing recovery method is to recover inconsistent NVDIMM pages by replaying redo logs against their old versions in SSDs. Therefore, NV-SQL creates a redo log entry for each update to NVDIMM-resident pages as well as each update to its undo page, respectively. Specifically, NV-SQL performs the following three steps for logging as in vanilla MySQL.

- Step 1: Before modifying a page, NV-SQL creates an undo page in the rollback segment.
- Step 2: NV-SQL creates a redo log entry for the undo page and persists it in NVDIMM.
- Step 3: After completing page modification, redo logs are recorded for the micro operations performed on the page.

Detecting Action-Inconsistent Pages On the other hand, a potential disadvantage of the redo-based recovery method is that the number of the redo logs can be very large, and replay times can be long, especially for write-hot pages. To alleviate this problem, NV-SQL uses a bit flag to keep track of which pages and which redo logs need to be recovered and replayed, as follows.

For NVDIMM-resident pages, the following steps are added to check the consistency. After writing an undo log page before page

modification, NV-SQL sets and persists a bit flag (`in-update` flag) in the page to indicate the page is being modified. After persisting a redo log entry, the flag is cleared. Thereby, NV-SQL ensures page action consistency for logical operations that occur on NVDIMM-resident pages. Since NV-SQL can, according to the fix-rule, modify a page only after acquiring the exclusive semaphore for the page, toggling a bit flag will not incur further concurrency overhead.

In addition, NV-SQL has to invoke `clflush` instructions for a logical action on a page, *i.e.*, modifications to cache lines on a page, prior to resetting the page's `in-update` flag. Unless those modified cache lines are flushed to NVDIMM, the page-action consistency of the page may be violated. For instance, let us assume that a page's in-update flag has been durably cleared at the time point of the crash, but its modified cache lines have not yet reached NVDIMM. Then, the page is incorrectly regarded as consistent. `clflush` and its variants `clflushopt` and `clwb` instructions prevent the CPU and the memory controller from optimizing memory accesses and incur latency of over 100 nsec [10]. However, NV-SQL offsets this overhead by saving SSD writes (*i.e.*, with latency of over 100 μs).

By introducing the `in-update` flag, NV-SQL can detect the action-inconsistent pages easily upon reboot. In addition, the number of inconsistent pages is very limited. Recall that, at the time point of a crash, only a small number of NVDIMM pages are being modified by logical update actions while most other pages are in a consistent state. Thus, the number of action-inconsistent pages is proportional to the number of CPU cores executing logical update actions. Only for those pages, NV-SQL will replay the corresponding redo logs to make them action-consistent.

5.3 Prototype Implementation

NV-SQL extends the ARIES-style recovery module of MySQL to ensure page-action consistency for NVDIMM-resident pages. The recovery module of MySQL consists of three steps: Analysis, Redo, and Undo. Below we describe how NV-SQL modifies each step for page-action consistency.

Analysis The analysis phase of NV-SQL works in two steps. First, it has to handle the problem of *torn* pages, as the Vanilla MySQL [35]. For every DWB page in NVDIMM, NV-SQL checks whether its corresponding page from the user tablespace is partially written (*i.e.*, torn), and, when torn, overwrites the page using its corresponding copy from DWB. Given that DWB has at most 128 pages, the overhead of this additional step will be small. Second, it reads `checkpoint_lsn` and the checkpoint offset of the most recent checkpoint from the redo log file as in vanilla MySQL. `checkpoint_lsn` determines the scope to "redo", *i.e.*. log entries with an LSN greater than `checkpoint_lsn` must be replayed, and the checkpoint offset points to the starting position of the redo logs to be scanned. Then, NV-SQL examines the `in-update` flag of all NVDIMM pages, looking for pages that were being updated by incomplete transactions, *i.e.*, the pages with the `in-update flag` set. The LSNs of the partially updated pages are used to determine where in the redo log file to scan from. While scanning and parsing redo logs, the redo log entries are copied to a volatile buffer and indexed into a hash table with the page ID as the key for fast lookup.

Redo Once redo logs are scanned and parsed, the recovery process replays the redo logs from the earliest point in the redo log file identified in the analysis phase. Volatile pages in the DRAM buffer cache are recovered no different than vanilla MySQL's recovery process. For NVDIMM-resident pages with the `in-update` flag unset, NV-SQL does nothing because the logical operations in the redo log have already been reflected in the pages. Otherwise, NV-SQL reads the pages from the durable storage and applies the redo logs to convert them into consistent pages. Upon completion of the redo phase, NV-SQL restores the state of the system to the state it was in at the time of failure. In addition, NV-SQL also restores on-disk organization structures (*i.e.*, rollback segments) for undo as in vanilla MySQL.

Undo After the redo phase, NV-SQL must roll back any effects of uncommitted transactions to ensure failure-atomicity. For NVDIMM pages with updates made by uncommitted transactions, NV-SQL restores them using undo logs as in vanilla MySQL [31].

5.4 Placing Redo Buffer and DWB on NVDIMM

In addition to write-hot data pages, DBMS engines have a few more objects that are small and write-intensive and are worth storing in NVDIMM. In the case of MySQL, such objects are in redo buffer and double write buffer (DWB). By placing such write-intensive and performance-critical objects in NVDIMM, we expect better transactional latency and throughput. In addition, fewer writes to the storage will prolong the lifespan of SSD.

Redo Log Buffer Placing redo log buffer in NVDIMM can avoid the overhead of the *log-force-at-commit* and *write-ahead-log (WAL)* protocols in the conventional DBMSs, which are inevitable with volatile DRAM-resident redo buffer [40]. Such protocols for transactional durability and atomicity are known to incur the process communication overhead and the synchronous I/O overhead [20]. Meanwhile, as depicted in Figure 3, with a redo buffer placed on NVDIMM, transaction durability, and atomicity can be achieved without following those protocols. Being durable once written to NVDIMM, redo logs need not be synchronously flushed to the storage for durability at commit. The *no-log-force-at-commit* protocol removes the log flushing latency from the critical path of transaction execution, improving the transaction latency. The accumulated redo logs in NVDIMM will be asynchronously flushed to the log device. In addition, since undo logs are also durable in NVDIMM, they need not be written ahead prior to evicting pages dirtied by non-committed transactions (*i.e.*, *no-WAL* protocol).

DWB Since neither file systems nor storage devices support atomic page writes to the storage, MySQL engine resorts to redundant journaling using DWB so as to guarantee the atomic propagation of dirty pages from DRAM to the storage despite failures: for each dirty page, write it first to DWB and then to the original location. Unfortunately, the redundant writes for atomic page propagation sacrifice the performance a lot (*e.g.*, enabling DWB can drop throughput by one-third [24].)

Simply by placing the 2MB-sized DWB buffer in NVDIMM, we can atomically propagate dirty pages to the storage in single-write journaling. First, when evicting a DRAM-resident dirty page to SSD, we can avoid redundant writes to SSD just by copying the page to DWB at NVDIMM and then writing the page to its original location in SSD. That is, the storage operation of flushing the page to DWB in the disk is replaced by the memory writes to NVDIMM-resident

DWB. In this way, even when the system fails while overwriting the old copy in SSD, the new copy of DWB in NVDIMM can be used to guarantee the atomic write. Second, when evicting an NVDIMM page, NV-SQL does not write the page redundantly to DWB in the disk, either. Since the page is already durable and thus can be atomically propagated to the SSD in *single-write* despite failures. To summarize, placing DWB in NVDIMM enables the elimination of redundant writes to SSD, further halving the physical writes in NV-SQL. This is another positive side-effect of NV-SQL.

6 PERFORMANCE EVALUATION

This section evaluates the performance of NV-SQL on the *same-priced* DRAM-SSD and DRAM-NVDIMM-SSD configurations to highlight the cost-performance effectiveness of our solution.

6.1 Experimental Setup

We use a Linux platform equipped with Intel Xeon E5-2640 CPU with 32 total cores, 32GB main memory, and 16GB NVDIMM-N [33]. We use a Samsung 970 PRO 512GB NVMe SSD as a database storage device and a Samsung 850 PRO 256GB SSD as a database log device. All benchmarks use the `ext4` file system in direct I/O mode. For NVDIMM-N, we mount the device with the `DAX` option. Throughout all the experiments, unless otherwise stated, we set the buffer cache size to 20% of the database size, the database page size to 4KB, and the number of concurrent client threads to 32. As shown in Table 3, the price per unit capacity of NVDIMM is three times higher than that of DRAM. Thus, we evaluate NV-SQL using various DRAM and NVDIMM configurations to demonstrate which one is more cost-effective in performance, given the same cost (*i.e.*, a 1GB NVDIMM or a 3GB DRAM). Below are described the two OLTP workloads used in the experiments:

TPC-C tpcc-mysql from Percona [38] is used for TPC-C workload on MySQL. TPC-C is an industry-standard OLTP benchmark for transactional database systems, consisting of heavy random reads and writes. For all TPC-C experiments, the initial database is set to 54GB (*i.e.*, 500 warehouses).

LinkBench LinkBench [6] is an open-source database benchmark for a large-scale social graph, reflecting the read-intensive feature with approximately 30% writes. The dataset consists of 50 million nodes, amounting to 64GB.

6.2 Run-Time Performance

In this section, we analyze the performance impact of NV-SQL with respect to transaction throughput and reduction of writes to SSD.

6.2.1 Effect of Update Count-based Midpoint Insertion. Before discussing the performance of NV-SQL, let us first clarify the performance of the update count-based NVDIMM buffer management policy. The update count-based midpoint insertion policy is introduced to differentiate truly write-hot pages for the new sublist. Table 4 shows the detailed performance metrics of two midpoint insertion schemes: the default and the update count-based one.

With the update count-based policy, the average number of updates made to the pages in the new sublist (the `Update # / Page` column) increases from 31 to 77, and the proportion of the false-positive write-hot pages in the new sublist being written to the SSD

Table 4: Performance of NVDIMM Management Schemes

	Write Amount (%)		Update # / Page		TPS
	Old	New	Old	New	
Default	83	17	9	31	1477
Update Count	95	5	13	77	1831

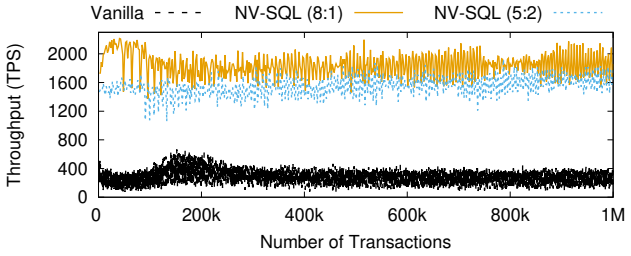

Figure 4: TPC-C Performance (Buffer Size = 20%)

(the `Write Amount(%)` column) decreases from 17% to 5%. This is because the policy admits a small amount of truly write-hot pages to the new list and absorbs writes more effectively while preventing false-positive write-hot pages from entering and polluting the new sublist. As a result, the update count-based policy improves transaction throughput by 24% than the default.

6.2.2 Basic Performance Analysis using TPC-C. We run the TPC-C benchmark for 1.5 hours and measure the throughput per second to compare the economic cost-effectiveness of the three same-priced systems, *i.e.*, the throughput of DRAM-only `Vanilla` (11 GB DRAM), 8GB DRAM–1GB NVDIMM, (`NV-SQL(8:1)`) and 5GB DRAM–2GB NVDIMM (`NV-SQL(5:2)`).

Transaction Throughput Figure 4 shows TPS for 1 million transactions; that is, the x-axis represents the number of completed transactions, and the y-axis indicates TPS. For `Vanilla` used as the baseline for comparison, we tuned flash-aware knobs known to be critical for I/O performance in MySQL [3]. The performance of the DRAM-only system is significantly lower than the other two equi-cost systems that benefit from the NVDIMM buffer. On average, `NV-SQL(5:2)` shows 5.8× better throughput than Vanilla and `NV-SQL(8:1)` does 6.5×. Furthermore, due to the active transaction processing, NV-SQL improves the I/O utilization from 60% in Vanilla to 82% and CPU utilization from 20% to 68%. These results clearly show that the equi-cost NV-SQL with *small but durable* cache can reduce writes to flash, and the benefit of lowering writes to SSD is more performance-critical than sacrificing the hit ratio.

To analyze the benefit of NV-SQL in detail, we measure several metrics for each mode as presented in Table 5. The first row is the amount of page writes per transaction issued by the MySQL buffer layer. Compared to Vanilla, NV-SQL reduces the per-transaction write traffic to the storage in half as its NVDIMM buffer absorbs writes. To further quantify the write reduction by NV-SQL, we measure the number of writes to SSD per page while conducting the same experiment in each mode. Figure 5 shows that NV-SQL cuts the number of writes to SSD approximately in half. Specifically,

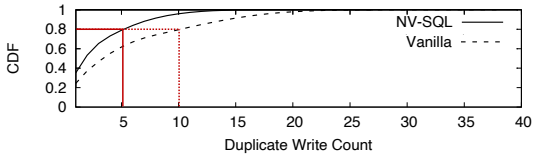

Figure 5: Cumulative Distribution of Skewed Writes

Table 5: I/O Metrics (Buffer Size = 20%)

(DRAM:NVDIMM)	Vanilla (11:0)	NV-SQL (8:1)	NV-SQL (5:2)
Write/TX (KB)	94	42	52
Write/Second (MB)	23.1	75.9	85.7
Read/Second (MB)	13.7	106.7	93.0
Hit Ratio (%)	98.6	97.9	97.5
Average TPS	279	1,831	1,631

as depicted by two red lines, pages ranked in the top 20% are written 5 times in NV-SQL while they are written 10 times in Vanilla. This result clearly indicates that NV-SQL can, as intended, absorb write-hot pages using a small amount of NVDIMM buffer.

The second and third rows in Table 5 are the amount of each I/O per second measured by `iostat`. They show that the ratio of per-second read and write amount in Vanilla is about 1:1.7, whereas the ratio in NV-SQL becomes roughly 1.4:1. That is, NV-SQL transforms the write-heavy I/O pattern in Vanilla into the read-intensive (and thus more SSD-friendly) pattern. Meanwhile, note that the data transfer rates of NV-SQL are much higher than those of Vanilla. This is not because NVDIMM fails to absorb the disk I/Os but because NV-SQL yields higher TPS. The hit ratio of NV-SQL with smaller memory is lower than that of Vanilla. Recall that this lowered hit ratio leads to more reads. In addition, note that SSD can obviously serve more reads because reads are less interfered with by writes inside SSD [9]. As a result, NV-SQL achieves 6.5× better throughput.

Another observation made from Figure 4 and Table 5 is the trade-off of *more DRAM vs. more NVDIMM*. From the throughput trends between `NV-SQL(8:1)` and `NV-SQL(5:2)`, compared to `NV-SQL(8:1)`, the benefit of write reduction in `NV-SQL(5:2)` is not large enough to offset the reduced hit ratio by far, and thus its TPS gain over Vanilla is less, though still significant. Finding the optimal NVDIMM size is beyond the scope of this paper and is left for future work.

Transaction Latency The high tail latency poses serious challenges for online service providers, as even a small increase in latency can reduce traffic and revenue. NV-SQL can reduce transaction latency mainly for two reasons. First, since fewer writes are in the critical path of transactions and thus the page-missing foreground processes will experience fewer or no read stalls [2, 3], NV-SQL will obviously reduce transaction latency. That is, SSD with fewer write requests can serve more flash reads faster; thus, the host buffer layer can read missing pages more quickly from the SSD. Second, each committing transaction no longer has to wait for the synchronous log flush (that is, no force flushing) as the redo buffer resides in durable NVDIMM. Once the redo log records become

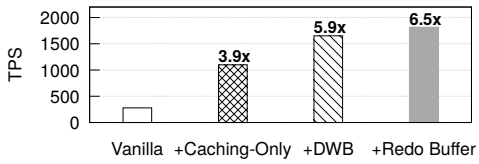

Figure 6: In-depth Analysis of Performance Gain

persistent, NV-SQL can eagerly release the exclusive semaphore, and thus this can reduce additional transaction latency.

To verify this effect, we measure transaction latency while running the TPC-C benchmark. The table is omitted due to the space limitation, NV-SQL considerably reduces the overall transaction latency and narrows the latency distribution. To be specific, the average, 95th, and 99th percentile latency of NV-SQL are 85%, 90%, and 90% lower than Vanilla, respectively.

CPU overhead of NVDIMM Cache Management The overhead of maintaining the NVDIMM cache consists of two parts: (i) the costs of persisting data pages on NVDIMM (*i.e.*, data movement and persistence overhead) and (ii) the cost of guaranteeing page-action consistency (*i.e.*, `in-update` flag manipulation upon NVDIMM-resident pages are modified). For caching overhead, we measured CPU cycles while calling `memcpy` and `clflush` instructions on flushed pages from the DRAM buffer. We accumulated CPU cycles on every manipulation of the `in-update` flag of NVDIMM-resident pages to identify the page-action consistency overhead. We confirmed that about 2% of CPU cycles are spent on Caching and 7% on guaranteeing page action consistency. Although the CPU overhead of frequent calling of `clflush` is non-marginal, we believe eADR [19] can mitigate this to enhance NV-SQL further. That is, once CPU caches are also included in the power-failure protection domain (*e.g.*, *extended ADR* [19]), NV-SQL doesn't need to call `clflush` anymore to ensure the durability.

6.2.3 In-Depth Analysis of Performance Gain. In order to drill down the total gain and thus understand the performance contribution of each technique, we measure TPS while running the TPC-C benchmark used in Figure 4 using three different NV-SQL configurations: Caching-Only, DWB, and Redo Buffer.

Caching-Only Caching-Only NV-SQL outperforms Vanilla by up to 3.9×. This result corroborates our design goal that caching hot data pages in a durable cache yields considerable write reduction, resulting in performance improvement.

DWB Placing DWB on NVDIMM enables single-write journaling for dirty pages. This boosts the throughput of NV-SQL by reducing writes to SSD and enabling SSDs to serve more reads faster. The performance gain by placing DWB on NVDIMM (denoted as `+DWB` in Figure 6) corresponds to the result of `DWB off` [24].

Redo Buffer In addition to DWB, placing the redo buffer at NVDIMM will allow us to take the no-log-force-at-commit and no-WAL protocols. However, this does not lead to significant performance improvement because MySQL flushes logs to disk every second for higher throughput, even while taking the risk of losing the durability guarantee. We expect the benefit of placing a redo

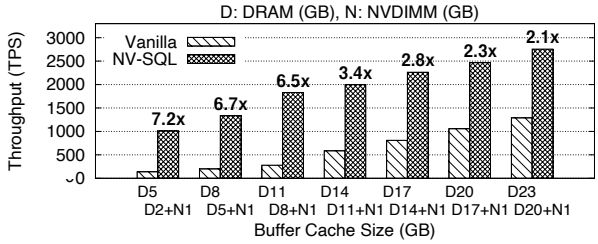

Figure 7: TPC-C Throughput: Varying Buffer Sizes

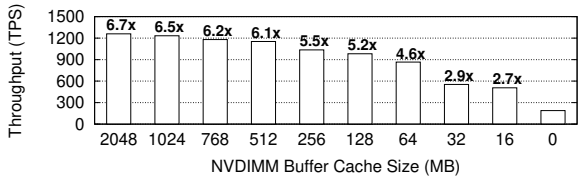

Figure 8: Scalability for Different NVDIMM Cache Sizes

buffer at NVDIMM to be more outstanding under full ACID compliance, which is a strong requirement in transaction systems [47, 51].

6.3 Effect of NV-SQL on Other Parameters

Buffer Cache Size In general, investing more memory resources in the buffer cache helps to improve performance because of the higher hit ratio. To delve into the effect of different buffer sizes, we run the TPC-C benchmark by varying the ratio of NVDIMM to DRAM while ensuring the same cost for each system.

Across different buffer sizes, as presented in Figure 7, NV-SQL consistently outperforms Vanilla in throughput and write reduction. On the one hand, as the buffer size increases, the throughput improves, and the average write per transaction reduces in Vanilla and NV-SQL. This result is obvious, considering a higher hit ratio leads to fewer I/Os. Though the relative gap between the two shrinks as the buffer size increases, the effect of NV-SQL remains considerable under a memory configuration of larger than 40% of the database size, as is illustrated by the rightmost bar in Figure 7. On the other hand, as the buffer size is reduced, the gap between the two is widened. NV-SQL with 2GB DRAM and 1GB NVDIMM (denoted as `D2+N1`)) wins over Vanilla even by 7.2× in terms of throughput. In summary, when investing the same cost in the buffer cache, NV-SQL always outperforms the DRAM-only Vanilla.

Another intriguing point in Figure 7 is the cost-performance efficiency of NV-SQL. For example, while the cost of NV-SQL with (`D2+N1`) is 70% lower than that of Vanilla with 17GB DRAM (denoted as `D17`), NV-SQL even outperforms Vanilla by 24%. This result highlights that NV-SQL is a cost-performance effective solution.

Impact of A Larger Write Working Set The results presented so far only have about 300MB of write working set, as mentioned in Section 3.2, which is smaller than the NVDIMM buffer size (*i.e.*, 1GB) and thus fits into it. Readers might question whether the design of NV-SQL can scale to a larger write working set. In this situation, NV-SQL will encounter a type of *sequential flooding* phenomenon [40] in terms of durability, which is the worst case to NV-SQL.

Table 6: Three TPC-C Transaction Mixes

	Write/TX (KB)		Improvement by NV-SQL	
	Vanilla	NV-SQL	WR Reduction (%)	TPS
`Default`	94	42	55	6.5
`NO+PM`	66	36	44	5.0
`NO-Only`	45	27	41	4.3

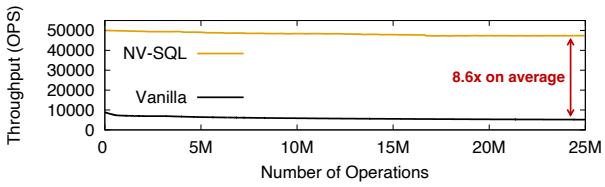

Figure 9: LinkBench Throughput (Buffer Size = 20%)

To evaluate the impact of a larger write working set, we run the TPC-C benchmark with a larger TPC-C database of 200GB. We set the total buffer cache size to 20GB (*i.e.*, 10% of DB size) and size the NVDIMM and DRAM buffers considering the 3× more expensive NVDIMM cost. We present the results in Figure 8. The last bar with 0MB NVDIMM buffer indicates the throughput of Vanilla.

The result clearly shows that the performance gain of NV-SQL reduces as the NVDIMM buffer size is reduced from 64MB to 32MB. This is because the working set size becomes larger than the NVDIMM buffer size. However, NV-SQL still outperforms Vanilla by caching fewer but write-hotter pages to NVDIMM and eliminating the write overhead due to DWB. Another thing to note is that, according to Section 3.2, the write working set size for a 200GB database would be around 1GB. However, this indicates the theoretical amount of pages worth caching in NVDIMM, and the number of actual write-hot pages cached in NVDIMM is very small, as mentioned in Section 4.4. Therefore, the sharp performance degradation occurs at a much smaller NVDIMM buffer size than expected.

Different I/O patterns To investigate how NV-SQL behaves across OLTP workloads with different I/O patterns, we conduct two additional experiments. First, to verify the effect of NV-SQL with different write/read ratios, we run the TPC-C using three mixes of five transaction types in the benchmark, including the standard one (denoted as `default`), a mix of New-Order and Payment (`NO+PM`), and New-Order only (`NO-Only`). While running those mixes using Vanilla and NV-SQL, we measure the write amount per transaction as well as TPS and present the result in Table 6. The result clearly indicates that *NV-SQL is more effective with a more write-intensive workload.* For instance, for the `default` with the highest write amount per transaction under Vanilla, the write reduction ratio of NV-SQL over Vanilla is the largest. Accordingly, the TPS improvement ratio is also the biggest.

Secondly, to demonstrate that NV-SQL works well for other OLTP workloads, we measure OPS while running LinkBench [6] using two configurations while executing 25 million operations. As shown in Figure 9, NV-SQL outperforms Vanilla (*i.e.*, DRAM-only) by 8.6× on average, with an average of 52% reduction in writes per

Table 7: Effect of NV-SQL on Various SSDs (TPC-C)

	SSD-A [49]	SSD-B [50]	SSD-C [48]
Vanilla	215	413	30
NV-SQL	1,725	1,979	326
WR Reduction (%)	48	46	68
Perf. Improvement	8.0×	4.8×	10.9×

operation. Since the I/O pattern in LinkBench, as in TPC-C, has temporal locality and write skew [28], the relative performance gain of NV-SQL over Vanilla is similar to that of default TPC-C.

Different SSDs To verify the effect of NV-SQL on different SSDs, we conduct the same experiment as Figure 4 using three commercial SSDs. As shown in Table 7, NV-SQL consistently outperforms Vanilla on all SSDs, although the improvement ratio varies by SSD type. In the case of SSD-A, which is the latest version of the SSD used in Figure 4 and thus has almost the same internal architecture, the improvement ratio (*i.e.*, 8 times) is similar to the result in Figure 4. On the other hand, on SSD-B with higher IOPS, NV-SQL outperforms Vanilla only by 4.8×. This is because NV-SQL becomes CPU-bound (*i.e.*, more than 85% CPU utilization) in effect, not because its write reduction ratio is less. Lastly, SSD-C shows the best performance improvement we have observed so far. The reason for this is the large write reduction ratio, which is 68%.

The Number of Concurrent Threads We also vary the number of concurrent client threads to 16, 32, 64, 128, 256, 512, and 1024 while running the TPC-C benchmark. The graph is omitted due to the space limitation, but as the number of threads increases, the TPS of both Vanilla and NV-SQL decrease. As illustrated in Table 7, the performance gain of NV-SQL decreases in CPU-bound. However, increasing the number of concurrent threads does not result in high CPU usage. Rather, it shifts the performance bottleneck to concurrency control (*i.e.*, locking), dropping the throughput.

6.4 Comparison with Alternative Options

6.4.1 Alternative Baselines. We can consider two alternative options to NV-SQL: (i) *Using more DRAM for a low miss ratio* and (ii) *Using additional SSDs for high IOPS*. First, let us compare option (i) to NV-SQL using Figure 7. Even comparing Vanilla with 17GB DRAM to NV-SQL with 2GB DRAM and 1GB NVDIMM, NV-SQL still achieves better throughput, even though it costs less to configure than Vanilla. Regarding option (ii), when using multiple disks, it is more cost-effective to invest in NVDIMM to keep more write-hot pages than to invest in more disks to write back them faster. To confirm this, we run a TPC-C benchmark using two SSDs in Figure 7 in RAID-0. Vanilla with two SSDs improves TPS by 1.6x because of increased IOPS. Nonetheless, NV-SQL with one SSD still performs 4.1x better than the multiple disk option. The above results confirm that NV-SQL can address the page write-back bottleneck more cost-effectively than other options.

6.4.2 Other NVM-based Alternatives. To understand the end-to-end effect of NVM-based architecture, we compare the performance of NV-SQL with NVM-only DBMS design on NVDIMM-only and DCPMM-only systems. As described in [13], NVM-only architecture runs the entire database engine on pure NVM, which places a buffer pool in NVM while its data and log files are stored in NVM through the NVM-aware file system. To implement this approach, we allocate buffer frames in NVM, as NV-SQL does, and use the DAX-enabled filesystem. The TPC-C results are presented in Table 8.

Table 8: Performance of NVM-based Alternatives

	Vanilla	NV-SQL	NVDIMM-only	DCPMM-only
TPS	633	2265	2883	1029

Table 9: Performance over System Cost

Components	Vanilla	NV-SQL
CPU	$939	$939
DRAM	$61.88	$45.00
NVDIMM	*N/A*	$16.88
SSD	$39.61	$39.61
R (Performance/Cost)	0.96	2.47

NVDIMM-only shows superior performance, but this approach is difficult to scale out due to its limited capacity. DCPMM-only can scale much better with high capacity but has lower throughput due to its high access latency and asymmetric I/O speed. All data in both NVM-only approaches reside on NVM, so they do not incur any disk I/O. NV-SQL, on the other hand, despite having this I/O stack overhead, achieves approximately 3/4 of NVDIMM-only performance and 2.2x of DCPMM-only performance. This result illustrates that NV-SQL can cost-effectively improve performance using a small amount of NVM on a full-fledged database engine.

6.5 Performance/Cost Ratio

Now we consider the total cost of ownership of DRAM–SSD and DRAM–NVDIMM–SSD architectures. Assuming a multi-tenancy where multiple tenants share computing resources of the same server, the cost-effectiveness of NV-SQL can be different from the on-premise system. To confirm the effect of NV-SQL in the cloud environment following the pay-as-you-go pricing, we calculate the performance/cost ratio R based on the below formula proposed in the previous study [23]. In the equation, we divide the measured throughput P by the total storage system cost, including storage device cost \$$S$, DRAM cost \$$D$, and CPU computation cost \$$E$:

$$R = \frac{P}{\$S + \$D + \$E} = \frac{P}{\$S + \$D + (W * U) * (\$C * 1/T)}$$

\$$E$ is calculated from the number of worker threads W, the average CPU utilization U, the market price of CPU \$$C$ and the number of total hardware threads T. Therefore, combining them can yield the cost of active threads needed by the workload.

We present the cost of each component of Vanilla and NV-SQL in Table 9. Note that \$$S$ and \$$D$ are the same because we adjust the cost of memory devices used for each buffer cache to be equal. We calculate the performance/cost ratio based on the performance numbers obtained from the settings used for Vanilla(11:0) and NV-SQL(8:1) in Figure 4. The ratio R is 0.96 for Vanilla and 2.47 for

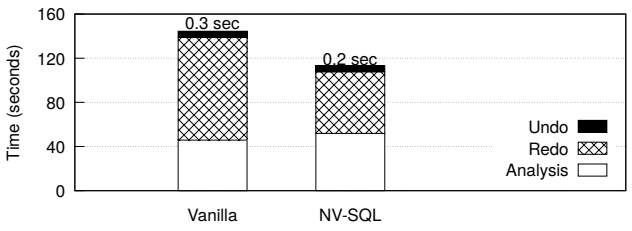

Figure 10: Recovery Performance

NV-SQL. That is, NV-SQL is 2.6× more cost-effective than Vanilla, although it uses more than 3× CPU and thus has high $E. This result clearly shows that NV-SQL can maximize the performance of existing DRAM–SSD architecture by using a small capacity of NVDIMM as a complementary device while providing high cost-effectiveness. In addition, given that R takes into account the storage, DRAM, and CPU space actually used, NV-SQL can provide cost benefits in cloud systems that follow the pay-as-you-go pricing.

6.6 Recovery Performance

To evaluate recovery performance, we use `SIGKILL` signal to force the system to crash after 10 minutes launching the same benchmark used in Figure 4. We measure the amount of time to restore the system, and the average recovery time is presented in Figure 10.

Analysis Phase Both MySQL versions have the same process to scan and parse the redo log files. However, NV-SQL needs to scan more redo log entries to recover action-inconsistent NVDIMM-resident pages from on-disk pages as explained in Section 5.3. Moreover, NV-SQL also scans NVDIMM pages to detect action-inconsistent pages using the `in-update` flag. In our experiment, this new phase for inspecting the 1GB NVDIMM buffer pool takes about 1 second to complete. Thus, NV-SQL takes slightly more time than Vanilla (51 and 43 seconds, respectively).

Redo Phase The reduction in the redo phase is because NV-SQL can skip the redo for action-consistent NVDIMM pages. Though the number of action-consistent NVDIMM pages is small, those pages account for about half of the amount of redo logs. In NV-SQL, however, for action-inconsistent pages, all their redo logs have to be replayed against the corresponding old pages in SSD. However, NV-SQL cancels out this overhead by avoiding replaying a bunch of redo log entries for action-consistent pages. In our experiment, NV-SQL reduces the redo phase time by ~ 40% compared to Vanilla.

Undo Phase Both Vanilla MySQL and NV-SQL have the same undo process. The most time-consuming task (*i.e.*reconstructing the on-disk layout for undo) is done in the redo phase. Here, the asynchronous rollback thread executes to revert any effect of uncommitted transactions using undo logs. Most operations are conducted in memory, so it takes less than 1 sec for both versions.

7 RELATED WORK

NVM-Aware DBMS Architectures. Recently proposed NVM-aware designs can be classified into three types: NVM direct [7], two-tiers with DRAM and NVM (MARS [12], SOFORT [37] and FOEDUS [25]), and three-tiers with DRAM, NVM and SSD (HyMem [52] and Spitfire [56]). The first two designs place entire databases in NVM, which is quite costlier than SSDs in terms of $/GB. Most of them target scalable NVM products such as Optane DCPMM [18], which are slower than DRAM but come with higher capacity. Therefore, the tiered designs use DRAM as a caching layer for NVM.

Unlike previous studies, NV-SQL places NVDIMM at the same tier as DRAM and leverages NVDIMM's high performance to mitigate durability overheads in traditional DRAM-SSD DBMS architectures. Compared to HyMem and Spitfire, which transfer fine-grained records or mini-pages between DRAM and NVM, NV-SQL admits entire pages to NVDIMM. Moreover, NV-SQL leverages existing checkpoint logic and per-page metadata to decide page admission while the focus of other systems is on cache replacement policies tailored for scalable but slower NVMs. NV-SQL also combines properties of NVM direct designs to allow direct updates in NVDIMM without having to go through the storage hierarchy managed by the buffer pool in HyMem and Spitfire.

Logging and Recovery Many NVM-aware logging and recovery protocols have been proposed recently. Some [15, 54] advocates placing the log buffer in NVM for fast commit, as NV-SQL does. MARS [12] redesigns the ARIES protocol by discarding design decisions for disk and opting for the *force* and *no-steal* policies. It also introduces new storage primitive to encapsulate multiple random writes in a transaction as a single atomic operation, making undo logs unnecessarily and writing only redo logs ahead. Write-behind logging [8] suggests adopting the *force* and *steal* policies to recover NVM-resident databases. While both MARS and WBL assume the NVM-resident databases, NV-SQL targets SSD-resident databases assuming the traditional *steal* and *no-force* policies.

8 CONCLUSION

In this paper, we explored a new design for NVM-based caching and suggested NV-SQL. Its key contribution is to show that the ARIES-like logging and recovery scheme and the related concepts such as checkpoint and LSN can be leveraged to cache data in the limited NVDIMM effectively. We also pointed out the problem of page-action consistency and suggested a redo-based solution.

Future research topics remain that can make NV-SQL more practical and performant. First, we need to devise an *online mechanism* that automatically adapts the re-update interval threshold to workloads. We simply took the offline profiling-based approach because this paper aims to explore a new caching architecture. Second, we would like to evaluate the effect of NV-SQL on disaggregated cloud storage. Like Amazon's Aurora [53], NV-SQL will boost the OLTP performance by reducing write traffic over the cloud network.

ACKNOWLEDGMENTS

This work was supported by Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No. 2015-0-00314, NVRam Based High Performance Open Source DBMS Development) and the National Research Foundation of Korea (NRF) grant funded by the Korea government(MSIT) (No. 2022R1A2C2008225). We are thankful to the anonymous reviewers for their insightful feedback.

REFERENCES

[1] ADAM ARMSTRONG. Samsung SSD 970 PRO Review. https://www.storagereview.com/review/samsung-ssd-970-pro-review.

[2] M. An, S. Im, D. Jung, and S.-W. Lee. Your Read is Our Priority in Flash Storage. In *Proceedings of VLDB Endowment*. VLDB Endowment, 2022.

[3] M. An, I.-Y. Song, Y.-H. Song, and S.-W. Lee. Avoiding Read Stalls on Flash Storage. In *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, page 1404–1417, 2022.

[4] R. Appuswamy, G. Graefe, R. Borovica-Gajic, and A. Ailamaki. The Five-Minute Rule 30 Years Later and Its Impact on the Storage Hierarchy. *Commun. ACM*, 62(11):114–120, oct 2019.

[5] R. Appuswamy, G. Graefe, R. Borovica-Gajic, and A. Ailamaki. The five-minute rule 30 years later and its impact on the storage hierarchy. *Communications of the ACM*, 62:114–120, 10 2019.

[6] T. G. Armstrong, V. Ponnekanti, D. Borthakur, and M. Callaghan. LinkBench: a Database Benchmark based on the Facebook Social Graph. In *Proceedings of the 39th SIGMOD International Conference on Management of Data (SIGMOD '13)*, pages 1185–1196, 2013.

[7] J. Arulraj, A. Pavlo, and S. R. Dulloor. 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*, SIGMOD '15, page 707–722, 2015.

[8] J. Arulraj, M. Perron, and A. Pavlo. Write-behind Logging. *Proc. VLDB Endow.*, 10(4):337–348, nov 2016.

[9] F. Chen, B. Hou, and R. Lee. Internal Parallelism of Flash Memory-Based Solid-State Drives. *ACM Transactions on Storage*, 12(3), May 2016.

[10] S. Chen and Q. Jin. Persistent B+-Trees in Non-Volatile Main Memory. *Proc. VLDB Endow.*, 8(7):786–797, feb 2015.

[11] J. W. Chris Ruemmler. A trace-driven analysis of disk working set sizes. 1993.

[12] J. Coburn, T. Bunker, M. Schwarz, R. Gupta, and S. Swanson. From ARIES to MARS: Transaction Support for next-Generation, Solid-State Drives. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 197–212, 2013.

[13] J. DeBrabant, J. Arulraj, A. Pavlo, M. Stonebraker, S. Zdonik, and S. Dulloor. A prolegomenon on oltp database systems for non-volatile memory. *ADMS@ VLDB*, 2014.

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

[15] R. Fang, H.-I. Hsiao, B. He, C. Mohan, and Y. Wang. High Performance Database Logging using Storage Class Memory. In *2011 IEEE 27th International Conference on Data Engineering*, 2011.

[16] J. Gray. The Five-Minute Rule. https://jimgray.azurewebsites.net/talks/FiveMinuteRule.ppt(Unpublisedtechnicaldocument).

[17] IBM. Understanding Fuzzy Checkpoints. https://www.ibm.com/support/pages/understanding-fuzzy-checkpoints.

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

[19] Intel Corp. eADR: New Opportunities for Persistent Memory Applications. https://www.intel.com/content/www/us/en/developer/articles/technical/eadr-new-opportunities-for-persistent-memory-applications.html, 2021.

[20] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann publishers Inc., San Francisco, CA, USA, 1992.

[21] Jim Gray and Andreas Reuter. Transactional Resource Manager Concepts. In *Transaction Processing: Concepts and Techniques*, chapter 10.3, pages 548–558. Morgan Kaufmann publishers Inc., San Francisco, CA, USA, 1992.

[22] A. Joshi, W. Bridge, J. Loaiza, and T. Lahiri. Checkpointing in Oracle. In *In Proceedings of 24 th International Conference on Very Large Data Bases (VLDB)*, 1998.

[23] T. W. Kaisong Huang, Darien Imai and D. Xie. SSDs Striking Back: The Storage Jungle and Its Implications on Persistent Indexes. In *12th Annual Conference on Innovative Data Systems Research*, CIDR '22, New York, NY, USA, 2022.

[24] W.-H. Kang, S.-W. Lee, B. Moon, Y.-S. Kee, and M. Oh. Durable Write Cache in Flash Memory SSD for Relational and NoSQL Databases. In *Proceedings of the 40th SIGMOD International Conference on Management of Data (SIGMOD '14)*, pages 529–540, 2014.

[25] H. Kimura. FOEDUS: OLTP Engine for a Thousand Cores and NVRAM. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 691–706. Association for Computing Machinery, 2015.

[26] S. T. Leutenegger and D. Dias. A Modeling Study of the TPC-C Benchmark. In *Proceedings of ACM SIGMOD*, pages 22–31, 1993.

[27] Micron. Micron NVDIMM. https://www.micron.com/products/dram-modules/nvdimm/.

[28] Minji Kang, Soyee Choi, Gihwan Oh, and Sang-Won Lee. 2R: efficiently isolating cold pages in flash storages. In *Proc. VLDB Endow. 13, 12*, 2020.

[29] C. Mohan. IBM's Relational DBMS Products: Features and Technologies. *SIGMOD Records*, 22(2):445–448, jun 1993.

[30] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1), 1992.

[31] MySQL Community. InnoDB Recovery. https://dev.mysql.com/doc/refman/5.7/en/innodb-recovery.html.

[32] S. Nalli, S. Haria, M. D. Hill, M. M. Swift, H. Volos, and K. Keeton. An Analysis of Persistent Memory Use with WHISPER. In *Proceedings of the 22nd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, page 135–148, 2017.

[33] Netlist. Netlist NVvault DDR4 NVDIMM-N. https://netlist.com/products/specialty-dimms/nvvault-ddr4-nvdimm.

[34] Oracle (MySQL Team). InnoDB Checkpoints. https://dev.mysql.com/doc/refman/5.7/en/innodb-checkpoints.html.

[35] Oracle (MySQL Team). InnoDB Disk I/O and File Space Management. https://dev.mysql.com/doc/refman/5.7/en/innodb-disk-io.html.

[36] Oracle (MySQL Team). Midpoint Insertion Strategy. https://dev.mysql.com/doc/refman/8.0/en/midpoint-insertion.html.

[37] I. Oukid, D. Booss, W. Lehner, P. Bumbulis, and T. Willhalm. SOFORT: A Hybrid SCM-DRAM Storage Engine for Fast Data Recovery. In *Proceedings of the Tenth International Workshop on Data Management on New Hardware*, DaMoN '14, 2014.

[38] Percona Lab. tpcc-mysql. https://github.com/Percona-Lab/tpcc-mysql, 2018.

[39] pmem.io. Persistent Memory Development Kit. https://pmem.io/pmdk/, 2022.

[40] Raghu Ramakrishnan and Johannes Gehrke. *Database Management Systems (3rd edition)*. McGraw-Hill Company, Inc., NewYork, NY, USA, 2003.

[41] J. T. Robinson and M. V. Devarakonda. Data cache management using frequency-based replacement. In *Proceedings of the 1990 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '90, page 134–142, New York, NY, USA, 1990. Association for Computing Machinery.

[42] E. Rogov. WAL in PostgreSQL: 3. Checkpoint. https://postgrespro.com/blog/pgsql/5967965.

[43] Samsung. Samsung DDR4 Memory. https://semiconductor.samsung.com/dram/module/rdimm/m393a2g40db0-cpb/.

[44] Samsung. SSD 970 PRO. https://www.samsung.com/semiconductor/minisite/ssd/product/consumer/970pro/.

[45] B. Schwartz. How InnoDB Performs a Checkpoint. https://www.xaprb.com/blog/2011/01/29/how-innodb-performs-a-checkpoint/.

[46] S. Shin, S. K. Tirukkovalluri, J. Tuck, and Y. Solihin. Proteus: A Flexible and Fast Software Supported Hardware Logging approach for NVM. In *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 178–190, 2017.

[47] M. Stonebraker, S. Madden, D. J. Abadi, S. Harizopoulos, N. Hachem, and P. Helland. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, page 1150–1160. VLDB Endowment, 2007.

[48] TechPowerUp. Crucial MX500 250 GB (Micron B16A). https://www.techpowerup.com/ssd-specs/crucial-mx500-250-gb.d75.

[49] TechPowerUp. Samsung 980 PRO 250 GB. https://www.techpowerup.com/ssd-specs/samsung-980-pro-250-gb.d45.

[50] TechPowerUp. Samsung PM981a 512 GB. https://www.techpowerup.com/ssd-specs/samsung-pm981a-512-gb.d783.

[51] A. Thomson and D. J. Abadi. The case for determinism in database systems. *Proc. VLDB Endow.*, 3(1–2):70–80, sep 2010.

[52] A. van Renen, V. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 International Conference on Management of Data*, SIGMOD '18, 2018.

[53] Verbitski, Alexandre and Gupta, Anurag and Saha, Debanjan and Brahmadesam, Murali and Gupta, Kamal and Mittal, Raman and Krishnamurthy, Sailesh and Maurice, Sandor and Kharatishvili, Tengiz and Bao, Xiaofeng. Amazon Aurora: Design Considerations for High Throughput Cloud-Native Relational Databases. In *Proceedings of the 2017 ACM International Conference on Management of Data*, SIGMOD '17, 2017.

[54] T. Wang and R. Johnson. Scalable Logging through Emerging Non-Volatile Memory. *Proceedings of VLDB Endowment*, 7(10):865–876, jun 2014.

[55] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *Proceedings of USENIX conference on File and Storage Technologies*, FAST'20, 2020.

[56] X. Zhou, J. Arulraj, A. Pavlo, and D. Cohen. *Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory*, page 2195–2207. 2021.