NVM: Is it Not Very Meaningful for Databases?

Dimitrios Koutsoukos
ETH Zurich, Switzerland
dkoutsou@inf.ethz.ch

Raghav Bhartia
ETH Zurich, Switzerland
rbhartia@ethz.ch

Michal Friedman
ETH Zurich, Switzerland
mkorenberg@inf.ethz.ch

Ana Klimovic
ETH Zurich, Switzerland
aklimovic@inf.ethz.ch

Gustavo Alonso
ETH Zurich, Switzerland
alonso@inf.ethz.ch

ABSTRACT

Persistent or Non Volatile Memory (PMEM) offers expanded memory capacity and faster access to persistent storage. However, there is no comprehensive empirical analysis of existing database engines under different PMEM modes, to understand how databases can benefit from the various hardware configurations. To this end, we analyze multiple different engines under common benchmarks with PMEM in AppDirect mode and Memory mode. Our results show that PMEM in Memory mode does not offer any clear performance advantage despite the larger volatile memory capacity. Also, using PMEM as persistent storage usually speeds up query execution, but with some caveats as the I/O path is not fully optimized and therefore does not always justify the additional cost. We show this to be the case through a comprehensive evaluation of different engines and database configurations under different workloads.

PVLDB Reference Format:

Dimitrios Koutsoukos, Raghav Bhartia, Michal Friedman, Ana Klimovic, and Gustavo Alonso. NVM: Is it Not Very Meaningful for Databases?. PVLDB, 16(10): XXX-XXX, 2023. doi:XX.XX/XXX.XX

PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at https://github.com/dkoutsou/database-benchmarking-optane.

1 INTRODUCTION

To alleviate the memory pressure [43] and to accelerate I/O, Persistent or Non Volatile Memory (PMEM) has been proposed as a solution. PMEM is both cheaper and has a higher capacity than DRAM, it is byte-addressable, and it persists data. This comes at the price of higher latency and lower bandwidth.

Database researchers have extensively studied how to integrate PMEM into a DBMS [22–25, 53]. Nevertheless, most of these studies are based on simulating PMEM since they were done before it was commercially available. The picture changed with the release of Intel®Optane©DC [2]. Intel®Optane©DC can be configured in Memory, AppDirect or Mixed mode. In Memory mode it operates as a volatile memory extension, in AppDirect mode it serves as byte-addressable persistent memory, and finally in Mixed mode part of it runs in AppDirect mode and the rest in Memory mode.

Proceedings of the VLDB Endowment, Vol. 16, No. 10 ISSN 2150-8097.
doi:XX.XX/XXX.XX

Optane sparked numerous studies benchmarking and explaining the behaviour of the hardware [42, 48, 50, 57]. These studies evaluate the hardware's characteristics and compare them to DRAM and SSDs. For example, they show the large latency difference between sequential and random accesses and the asymmetric behaviour of PMEM between read and write bandwidth.

However, DBMSs are complex systems with many knobs that behave very differently as workloads and input size vary. Existing studies [29, 34, 59] give only a partial picture as they provide only high-level conclusions and are mostly based on micro- or small-scale benchmarks. Furthermore, they do not explore how database knobs and different operations (e.g. join types) or query plans affect PMEM's behaviour. To this end, in this paper we provide the first extensive analysis of database engines using Intel®Optane©DC Persistent Memory. We run OLAP, OLTP, and key-value store workloads on various engines (PostgreSQL, MySQL, SQLServer, DuckDB, VoltDB, and RocksDB) under various PMEM configurations altering a variety of database/workload knobs.

Our results provide a complex picture showing that using PMEM does not always lead to better performance, despite its nominal advantages. Although PMEM is faster than SSDs, the I/O path is not fully optimized since there is no OS prefetching and the CPU is involved in I/O when using PMEM as persistent memory, wasting valuable and often scarce processing capacity. Furthermore, in systems with resource contention due to mixed or write-heavy workloads, PMEM experiences a large performance drop. This makes SSDs competitive in scenarios such as CPU-heavy queries with high selectivity. Similarly, although PMEM offers extra volatile memory capacity in Memory mode, this does not translate to performance benefits. In conclusion, our findings indicate that in its current form, PMEM does not provide large advantages to database engines, particularly when considering its additional cost.

2 BACKGROUND

2.1 Persistent Memory

Persistent memory, also known as non-volatile or storage class memory, combines the byte-addressability and low latency of DRAM with the persistence and high capacity of disks. The technology is available in the Intel®Optane©DC Persistent Memory Module [2] (referred to as PMEM in the rest of the paper). It is faster and more expensive than NAND Flash but slower and cheaper than DRAM. PMEM comes in a DIMM form factor in 128, 256, or 512 GB sizes and it is attached to a memory channel.

Each memory channel connects to the CPU via an integrated Memory Controller (iMC) (Figure 1), which maintains read/write

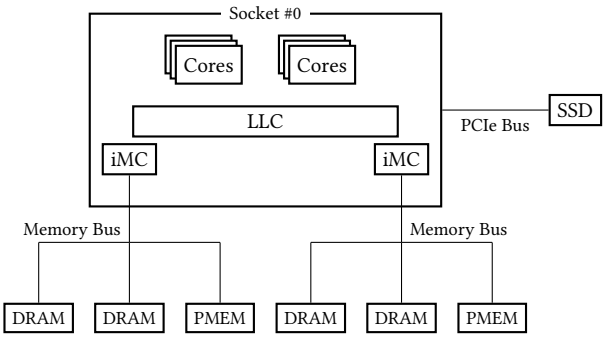

Figure 1: Socket topology

queues for PMEM DIMMs and ensures persistence on power failure. With a maximum memory capacity of 6TB for a 2-socket server, PMEM's internal access granularity is 256 bytes, which can cause read/write amplification [42]. Sequential access patterns have highest bandwidth, just 2-3x lower than DRAM [42].

PMEM operates in three modes: Memory Mode, AppDirect Mode, and Mixed Mode. In Memory Mode, PMEM acts as volatile memory while DRAM serves as an L4 direct-mapped cache. The iMC manages data traffic between DRAM and PMEM, with applications experiencing DRAM latency for cache hits and both PMEM and DRAM latency for cache misses. Applications do not need any source code changes to use this mode.

In AppDirect Mode, PMEM serves as persistent storage, offering striped read/write operations in interleaved or standalone regions. Configurable namespaces modes include `fsdax`, `devdax`, `sector`, and `raw` mode [19], with Intel recommending `fsdax` for DAX-aware file systems. [19] `Fsdax` eliminates the OS page cache in the I/O path, allowing `mmap(2)` to directly map to persistent memory media.

Finally, Mixed Mode combines Memory and AppDirect modes, using DRAM as an L4 cache. Users can configure the PMEM percentage for volatile memory or disk usage, but Intel suggests a minimum 1:4 DRAM to PMEM ratio for cost efficiency.

In the rest of the section, we briefly compare memory management in traditional systems and AppDirect Mode, covering the OS page cache, prefetching, and Direct Memory Access (DMA) operations, as these are crucial for the results of the paper.

The OS page cache caches recently accessed data from disk. This improves the performance of I/O operations. User-level programs access the page cache through system calls like `read(2)` and `write(2)`. The page cache uses techniques such as prefetching to further improve I/O. Prefetching anticipates the data that will be accessed and proactively caches them to ensure that the data is readily available in memory when needed.

Direct memory access (DMA) is a feature that allows data to be transferred between peripheral devices (such as network or storage devices) and memory without going through the CPU. DMA operations transfer data directly between the peripheral device and memory, bypassing the CPU and the page cache altogether.

In a traditional system, the OS page cache and prefetching are used to improve the performance of I/O operations. However, in the case of AppDirect mode, the OS page cache is not available and applications use directly `mmap(2)` to establish mappings to persistent memory. Also since Intel Optane Persistent Memory is not a peripheral device, DMA is not available and the CPU is necessarily involved in every data transfer between PMEM and the CPU. Finally, although PMEM has its own prefetcher, it cannot use the OS page cache for extra memory capacity and it has to rely directly in the memory allocated by the application.

2.2 Related work

Even before the first PMEM commercial implementation, researchers adapted DBMS components to PMEM characteristics. Chatzistergiou et al. [30] developed a library for transactional updates in data structures designed for PMEM and Wang et al. [56] came up with a distributed logging protocol for PMEM. Similarly, Arulraj and Pavlo adapted different parts of a database engine to PMEM, such as the buffer manager [64], and the storage and recovery [24, 26] protocol. With their findings, they developed a DBMS, Peloton [1, 49], built to leverage PMEM. Similarly, SAP HANA adapted their engine to integrate PMEM [21], Alibaba developed their own durable PMEM database [39] and Liu et al. [45] built a log-free OLTP engine for PMEM. Van Renen et al. [53] evaluated how DBMSs perform when they use either PMEM exclusively or a hybrid approach with a page-based DRAM cache in front of PMEM. There has also been considerable work on algorithms such as index joins [51], or data structures such as B+-Trees [31, 32, 41, 55, 62], hash tables [35, 37, 38, 46, 47, 65], and other indexes [40, 44, 63] on PMEM.

Most of the above efforts simulate PMEM using software tools [36]. With the introduction of Intel Optane, it is possible to experiment on the actual hardware. This has led to several performance studies. Outside of the database context, Izraelevitz et al. [42] provided the first comprehensive performance measurements of Intel Optane including read/write latency with sequential and random access patterns as well as the effect of factors such as the number of threads on different PMEM modes (Memory-mode and AppDirect-mode). This study was followed by others [50, 61] providing more in-depth analysis and elaborating on the insights of Izraelevitz et al. Xiang et al. [60] study the read/write buffering performance of PMEM and they provide new insights on how read/write buffers are managed. In HPC, a number of studies [48, 57] have shown promising results in hybrid DRAM-NVM configurations.

Database researchers have primarily focused on OLAP workloads. Benson et al. [29, 34] employ microbenchmarks and OLAP benchmarks (Star Schema, TPC-H) to identify best practices for using Intel Optane and assess its viability as an NVMe SSD replacement, exclusively utilizing the AppDirect mode (with and without `fsdax`). Shanbhag et al. [52] evaluate the AppDirect mode in an OLAP context using the SSB benchmark. Wu et al. [59] conduct a brief study running microbenchmarks, TPC-H, and TPC-C on SQLServer 2019 using PMEM in Memory and AppDirect mode (with and without `fsdax`), presenting high-level conclusions on DBMSs leveraging PMEM, primarily related to SQLServer, without experimenting with database knobs or examining how query plans affect PMEM performance. Renen et al. [54] measure Intel Optane Persistent Memory's bandwidth and latency, while tuning log writing and block flushing. Lastly, Benson et al. [28] develop a benchmark framework to evaluate various customizable database-related PMEM access characteristics [28].

Table 1 summarizes the landscape of available benchmarking papers, denoting our contributions with the symbol ‡ and prior

Table 1: Landscape of Intel Optane benchmark papers in the DB community (our contributions are denoted with ‡)

	PostgreSQL	MySQL	SQLServer	DuckDB	VoltDB	Bench
SSD	‡	‡	‡, [59], [58]	‡	-	TPCH
SSD	‡	‡	‡, [59], [58]	-	‡	TPCC
NVMe	[29]	-	[58]	-	-	TPCH
NVMe	-	-	[58]	-	-	TPCC
AppDir	‡, [29]	‡	‡, [59]	‡	-	TPCH
AppDir	‡	‡	‡, [59]	-	‡	TPCC
Mem	‡	‡	‡, [59]	‡	-	TPCH
Mem	‡	‡	‡, [59]	-	‡	TPCC

work with references. We use NVMe for Intel Optane SSD disks and AppDir and Mem rows to represent the corresponding Intel Optane Persistent Memory modes. As shown in the table, we evaluate five DBMSs in both AppDirect and Memory modes using complete benchmarks (TPC-H and TPC-C) rather than microbenchmarks. In TPC-H, we demonstrate how query plans, operations, and system statistics are affected by using PMEM. We also extensively evaluate Memory mode, providing insights on its usefulness with working sets larger than DRAM. We use both row and column stores to show how memory accesses and compression affect Intel Optane. In TPC-C, we explore the impact of database knobs. Furthermore, we are the only ones benchmarking a key-value store (RocksDB) under varying read-write workloads.

3 EXPERIMENTAL SETUP

3.1 System specification

All the experiments in this paper were conducted on a dual-socket server (Table 2) and the socket topology shown in Figure 1. Each CPU socket has two memory controllers and three channels per controller. The DRAM and Intel Optane DIMMs are installed in a 1-1-1 topology. We disable the turbo mode and we set the CPU power governor to performance mode with 2GHz as the maximum frequency, so that we can have reproducible experiments with minimum variation. We refer to *Default Mode* as the configuration that does not use PMEM at all and, unless otherwise mentioned, to *AppDirect mode* as the configuration that mounts PMEM in AppDirect mode with `fsdax` enabled, since this is the one recommended by Intel. When we use PMEM in the AppDirect mode, we do not use the SSD at all. In *Memory mode*, we configure all the available PMEM capacity (512GB) as volatile memory. We collect system statistics with the Intel Vtune Platform Profiler [3]. To get more accurate statistics, we modify the source code of the profiler to increase the sampling hardware counter frequency. We use PostgreSQL 13.2, MySQL 8.0.23, SQLServer 2019-15.0.4178, DuckDB v0.6.1 919cad22e8, VoltDB 11.4.0.beta1 Community Edition, YCSB 0.17.0 and RocksDB 5.11.3. Finally, we are aware that our SSD is not the fastest, but having a cheaper SSD gives a better perspective on the average case and shows even more clearly how small can the performance gap be besides the hardware difference. We expect that with a high-end SSD disk, the performance advantage of PMEM would become even smaller.

Table 2: Server specifications

Component	Specs
Sockets	2
CPU	Intel(R) Xeon(R) Gold 6248R
Microarchitecture	Cascade Lake
Cores	48 physical (96 logical)
L1 cache	64KB
L2 cache	1MB
L3 cache	36MB
RAM	128GB (16GB DDR4 @ 2666 MHz × 8)
PMEM	512GB (128GB × 4)
SSD	KIOXIA KPM6XRUG1T92 (2TB)

For all our benchmarks, we pin database processes to socket 0 to avoid remote NUMA access to better interpret the results, as done by other studies [29, 59]. That also ensures the working set is larger than the DRAM capacity for most queries. Before executing each query or workload, we clear the OS page and the database cache.

For TPC-H, we use SF-100 and a buffer cache of 16GB across databases. We also add foreign indexes, when the DBMS does not do that automatically. We use PMEM in interleaved mode, since the data together with the indexes do not fit in a single PMEM DIMM.

We use 1000 warehouses for all TPC-C experiments and set the warmup and running time to 3 minutes to sufficiently load the buffer pool and stabilize database activity. To reduce the proportion of reads to writes, we employ a large buffer pool (48GB). Additionally, we found that using only one of the two available PMEM DIMMs in the socket was sufficient to reach maximum tpmC for most experiments. Thus, unless otherwise noted, we use non-interleaved mode (e.g., one PMEM DIMM) for TPC-C in AppDirect mode, ensuring a better comparison with Default mode, which uses only one SSD. For MySQL, PostgreSQL, and SQLServer, we use HammerDB [16] as a TPC-C implementation, while for VoltDB, we use the official implementation provided in the VoltDB repository [17].

We adjust DBMS checkpointing strategies and parameters for frequent checkpoints, offering a clearer picture of hardware impact on performance. Specifically, for PostgreSQL (using PGTune [6]), we set `checkpoint_timeout` to 30 seconds and `max_wal_size` to 5GB. For SQLServer, we do automatic checkpoints every minute. For MySQL, we disable binary logs, keeping default log buffer and log file sizes, causing checkpointing every second. For VoltDB, we set the flush interval to 1 second and use one snapshot copy that's continuously overwritten, exploring PMEM's behaviour to concurrent writes. Frequent checkpoints, while atypical, expose PMEM's weaknesses under stress, providing valuable insights.

Furthermore, SQLServer requires `fsdax` to store data and `sector` to store logs [12]. We thus have to use both memory sockets, as it is not possible to create a mixed namespace in one socket that has the capacity for 1000 warehouses. Finally, although we run the Memory mode for TPC-C for many configurations across DBMSs, in most cases its performance is almost identical to the Default mode as requests are served by the L4 DRAM cache and not by PMEM. We therefore do not report any Memory mode results for TPC-C. Finally, we do not compare against Intel Optane SSDs, because Optane Persistent Memory is significantly faster that Optane SSDs [29].

For RocksDB, we use the Yahoo Cloud Serving Benchmark [33], specifically Workload A (update-heavy), Workload B (read-heavy), and Workload C (read-only). We generate 110GB of data per workload and set the operation count to 10 million to enable a more comprehensive analysis of the hardware behavior. We leave all other parameters of the benchmark to their default values.

Finally, we do not run experiments for Mixed mode. This mode requires a minimum volatile memory ratio of 1:4, which Intel suggests for PMEM to be cost and performance-effective over a DRAM only solution. Furthermore, different sizes and ratios are needed

Table 3: Latency measurements for different memory types

Memory type	Read latency [us]	Write latency [us]
DRAM	0.147	0.250
PMEM	0.333	0.262
SSD (random)	4.97	108.9
SSD (sequential)	4.85	94.9

to understand this mode but each one requires its own hardware configuration and individual insights might not generalize.

3.2 Basic microbenchmarks

As a baseline, we measure the latency and bandwidth of our system for different memory types (Tables 3 and 4), following a similar methodology to Wang et al. [61]. We use one socket for all microbenchmarks to avoid NUMA-node effects, employing AVX-512 instructions and PMEM in interleaved mode. We don't perform measurements for PMEM in Memory Mode due to Intel's lack of public information on its operation [18, 20].

For PMEM and DRAM, we measure load and store latencies by issuing a 64-byte instruction for sequential and random memory accesses and not using any caching. For loads, we issue a 64-byte load instruction with a cold cache. For stores, we load the memory address into the cache, and afterwards measure the 64-byte store, followed by a flush (clwb) and a fence (sfence) instruction. For the SSD we use ioping [11]. The SSD has 10× larger read latency than PMEM and more than 30× larger read latency than DRAM. The gap is much larger for write latencies, where the SSD has up to 435× larger latency than both DRAM and PMEM.

We measure bandwidth on sequential and random accesses and we vary the number of threads from 1 to 8. We pick the access granularity that maximizes bandwidth for every hardware type (64B for DRAM, 256B for PMEM, 4KB for SSD). For DRAM and PMEM, we execute a flush and a fence instruction after each store. For loads, we only execute a fence instruction. For SSDs we use fio [10]. We notice that for 1 thread, DRAM has 2.9× more bandwidth for sequential reads and 12% more bandwidth for sequential writes than PMEM. The gap increases as we increase the number of threads, where DRAM has more than 5× larger bandwidth for reads and more than 3× more bandwidth for writes. For random accesses and 1 thread, DRAM has 2.4× more read bandwidth and the same write bandwidth. Increasing the thread count to 8, shows that DRAM has 4.7× larger read bandwith and 3× larger write bandwidth. If we compare PMEM with the SSD, there is a very large gap for reads, but a smaller gap for writes. For 8 threads and sequential accesses, PMEM has 25× higher read bandwidth and 5× larger write bandwidth. However, for random accesses and 8 threads, PMEM has 9× higher read bandwidth and 2.8× larger write bandwidth. The difference is analogous for smaller number of threads with random accesses. For sequential accesses and 1 thread, PMEM has 20× more read bandwidth and 5× more write bandwidth than SSD.

4 OLAP WORKLOADS (TPC-H)

4.1 General observations

We present the running times of the TPC-H benchmark for all four systems in Figures 2. DuckDB cannot execute queries 17 and 21

Table 4: Bandwidth for different memory types

Memory type	Read bw [MB/s]	Write bw [MB/s]
DRAM (random, 8 threads)	20480	2529
DRAM (sequential, 8 threads)	67550	7615
DRAM (random, 1 thread)	3185	324
DRAM (sequential, 1 thread)	11460	969
PMEM-storage (random, 8 threads)	4341	830
PMEM-storage (sequential, 8 threads)	12628	2425
PMEM-storage (random, 1 thread)	1306	323
PMEM-storage (sequential, 1 thread)	3977	861
SSD (random, 8 threads)	470	293
SSD (sequential, 8 threads)	504	477
SSD (random, 1 thread)	160	157
SSD (sequential, 1 thread)	191	166

because of insufficient memory. In general, the AppDirect mode is consistently faster than the Default mode. This happens because PMEM has lower latency and higher througput than SSDs and, thus, all I/O is faster. MySQL is the slowest database, since it does not support multiple threads per query, except for particular type of queries, e.g., `SELECT COUNT(*)` [5]. When doing I/O with only one thread, the bandwidth in both PMEM in AppDirect mode and the SSD are far from their maximum. PostgreSQL is in the middle, as it uses 8 threads for processing and I/O. SQLServer and DuckDB use in general all the available logical cores in socket 0 (48 in total) and that is why the running time is much lower than the other two databases. The PMEM read bandwidth is up to 8 GB/s for some queries of SQLServer. Besides using all the available logical cores in socket 0, DuckDB uses a native compressed column format to which it pushes selections and projections and therefore it reduces I/O costs vastly. For MySQL, PostgreSQL, and SQLServer, the time difference is more obvious for queries involving larger tables (e.g. the `lineitem` table, which is around 100GB of storage together with indexes). For DuckDB, although the AppDirect mode is faster for the majority of the queries, the time difference is smaller, since I/O is not in the critical path. Additionally, as mentioned in Section 2.1, DMA is not available in AppDirect mode. As a result, CPU resources are used for I/O. In scenarios with CPU-intensive queries or a low number of threads, the CPU spends computation time on I/O, leading to suboptimal PMEM bandwidth and making the Default mode's performance competitive. We can observe this for DuckDB when the optimizer chooses CPU-heavy operations (e.g. index joins, hash group-bys) where the AppDirect mode is very close or has worse performance than the Default mode (e.g. queries 2, 15, 18).

The Memory mode has similar or slightly worse running times than the Default mode for the vast majority of queries. To understand where the overhead comes from, in Figure 3 we show the DRAM read and write bandwidth consumed by each query for the Default and Memory modes, respectively, only for PostgreSQL. We observe similar results for MySQL and SQLServer. DuckDB has a different behaviour, since it minimizes I/O because of its columnar compressed format. For all the queries, we read and write more data from DRAM in Memory Mode compared to the Default mode across databases. For reads, that happens because a DRAM (L4 cache) read miss in Memory mode results in a read from PMEM, which consequently leads to a DRAM write. Another reason for the

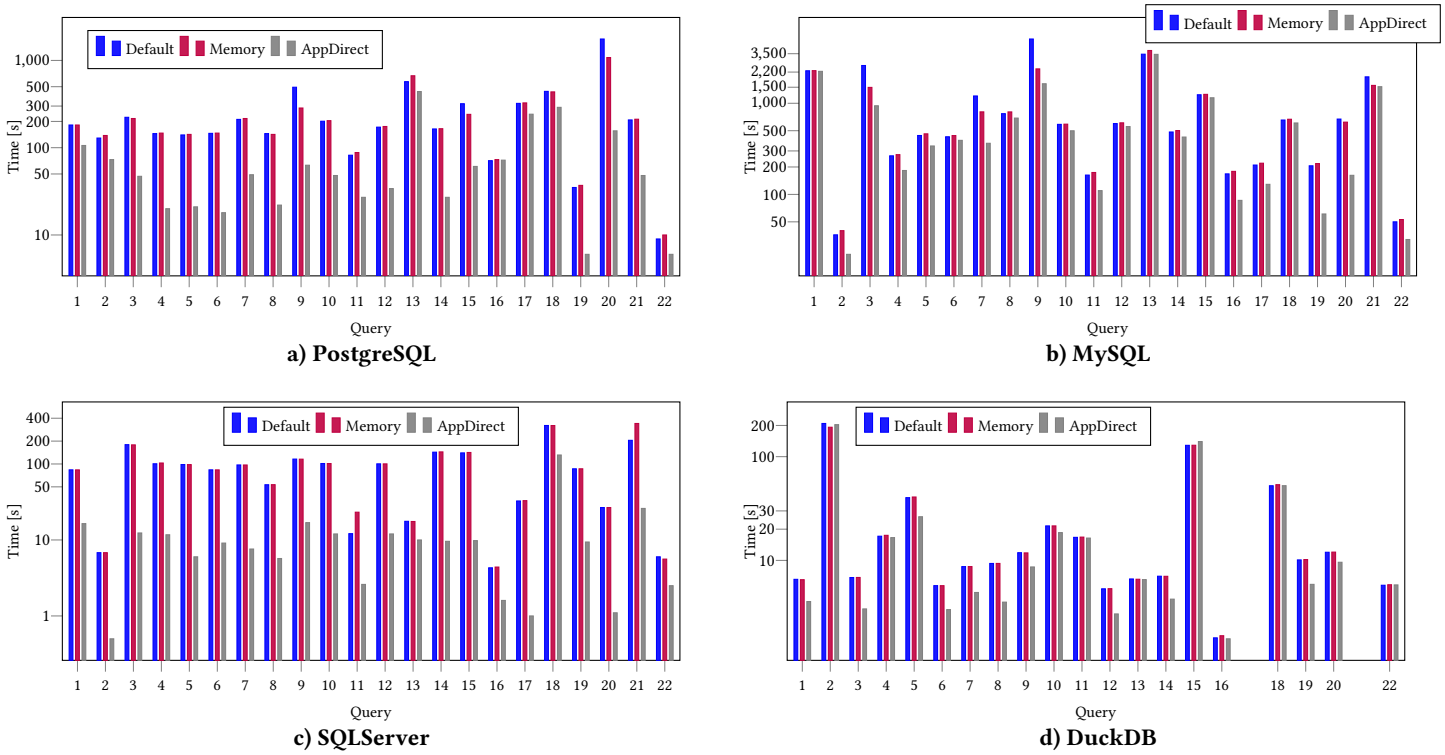

Figure 2: Running time (log-scale) for TPC-H SF-100 on various DBMSs for different system configurations

increased DRAM reads is that during a DRAM write, the system has to read the dirty lines and flush them to PMEM. Due to prefetching by the iMC, the DRAM writes are more than the L4 cache misses. Also, the increased DRAM bandwidth for both reads and writes is because there are collisions in the L4 direct-mapped cache, since multiple PMEM memory lines map to the same DRAM memory line. If we allocate memory space such that two PMEM memory lines map to different DRAM memory lines (e.g. by having the exact same capacity in PMEM and DRAM), that will decrease the cache conflicts and makes the two modes comparable.

Insights. For general OLAP workloads, Memory mode does not offer a significant performance advantage, as it slightly hinders performance for the majority of the queries. AppDirect mode can significantly speed up workload execution, if the queries are I/O intensive and there are enough system resources dedicated to the database. If system resources are limited, the design restricts the number of cores/threads involved or the queries are CPU-intense, SSDs offer a competitive and cheaper alternative.

4.2 Scans

Sequential scans in AppDirect mode have a much lower latency in PMEM [34, 42]. We observe this advantage for the AppDirect mode compared to the Default mode for all the systems. However, this advantage becomes more pronounced for queries that involve larger sequential scans (e.g. queries 3, 4, 5, 6, 7, 8, 11, 12, 19, and 20), especially the more threads are used. Although the lower latency does provide a significant benefit, it is not as substantial as we initially anticipated based on our microbenchmarks. We expected to see a 9-25× difference across all queries, but the actual advantage is somewhat lower. This is due to the fact that, as explained in Section 2.1, the AppDirect mode with `fsdax` does not utilize DMA or the OS page cache. This is especially true for queries that have large sequential scans and are also compute-heavy (such as query 1), where valuable CPU resources are used by PMEM for memory transfers in the absence of DMA, and the Default mode gains an advantage. In contrast, simpler queries that mostly involve scans (such as query 6) benefit greatly from PMEM. To validate that, we run a variant of TPC-H SF-100 query 6 using PostgreSQL, where we increase the selectivity of the predicates (i.e. we select more data) and we present the results in Figure 4. As we observe, by increasing the selectivity, both PMEM and the SSD have a sublinear time increase but their relative difference stays constantly between 9-10x and it is due to the sequential scan of the lineitem table. Furthermore, SQLServer writes data directly to physical hardware, bypassing the OS page cache [13], and it is also more efficient at pipelining I/O and processing. As a result, there is a larger time difference between queries, since SQLServer can take full advantage of the higher bandwidth. Finally, MySQL prefers nested loop joins over hash or merge joins. Nested loop joins have many random accesses when there are no indexes available compared to hash or sort/merge joins. Especially in the case of MySQL, the secondary indexes are non-clustered and in case of matches, the DBMS has to go into the base table to materialize the result. However, as these accesses have large block sizes (i.e. 16KB), PMEM has similar

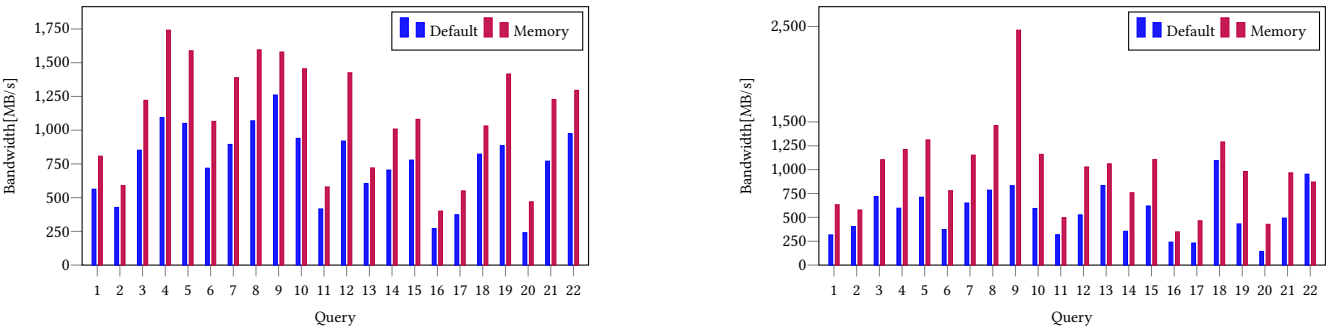

Figure 3: DRAM average write (left) and read (right) bandwidth on PostgreSQL for different system configurations

bandwidth to sequential accesses. In contrast, prefetching and the OS page cache cannot help in Default mode for random accesses.

Insights. The AppDirect mode is largely beneficial for non-CPU intensive queries with sequential scans. When the majority of the data has to be heavily processed, then the lack of the page cache and prefetching in AppDirect mode reduces most of PMEM's advantages. Users should tune the block size to a large size (≥ 8KB), when the DBMS performs random scans, because then PMEM has similar bandwidth to sequential accesses.

4.3 Index lookups

Index lookups cause random accesses. Thus, we would expect a performance drop in AppDirect mode. However, because all tested databases use large block sizes (≥ 8KB) the lookup accesses are about 10× faster for the AppDirect mode (queries 2, 4, 5) or have the same latency as a sequential scan for PMEM (queries 8, 9, 10, 19, 20, 21). We also observe that the type of index can affect the lookup time. More specifically, MySQL stores primary clustered indexes together with the data and secondary indexes separately. That puts PMEM at a disadvantage, because an access to a secondary index needs to locate the data in the tables and in the Default mode part of these accesses are served by the OS page cache. This behaviour is not present in the other engines, because PostgreSQL does not have the notion of a clustered index and SQLServer bypasses the page cache. DuckDB uses indexes only to perform index joins once the data are already in DRAM, and therefore the storage medium is irrelevant at this point in the query plan. Compared to the other systems it has much less I/O due its compressed columnar format, making the effects of the page cache inconspicuous.

Insights. When creating indexes or running query plans that operate on indexes (e.g. joins), the block size should be tuned to be large (≥ 8KB) to maximize the read bandwidth in AppDirect mode for PMEM. Additionally, whenever possible, clustered indexes should be used. The additional lookup in a non-clustered index has a performance penalty for PMEM due to the lack of an OS page cache.

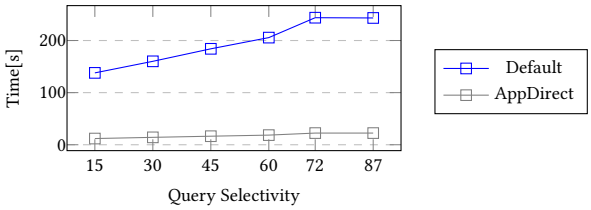

Figure 4: Increasing selectivity microbehmark for PostgreSQL

4.4 Working set size in Memory mode

As databases try to minimize I/O as much as possible, we would expect that adding a larger volatile memory capacity with PMEM in Memory Mode would improve the running time of queries with a large working set. However, across all three engines, we observe that the majority of the queries perform the same or slightly (less than 5%) worse in Memory Mode because of the increased traffic between the L4 DRAM cache and PMEM as mentioned in Section 4.1. We can split the queries in 2 categories based on the working set size. Queries 1, 4, 5, 6, 12, 14 and 17 have a working set of less than 64GB and therefore most of the accesses are served from the L4 DRAM cache. The rest of the queries have a working set larger than the L4 and/or the OS page cache, but the majority of them does not benefit from the larger volatile memory capacity and any improvements of the Memory mode over the Default mode are due to the memory structures that the systems use to handle memory and how the DBMS utilizes the OS page cache. For PostgreSQL, these are queries 9, 15, and 20. For MySQL, these are queries 3, 7, 9, and 21. For SQLServer, no query is significantly faster in Memory mode as the DBMS does not utilize the OS page cache. Finally, DuckDB has very small working sets for all the queries. Thus, in Memory mode the data are directly transferred to the L4 DRAM cache and PMEM is not used.

Insights. A very large working size can run slightly (up to 10%) faster in some cases in Memory mode, due to the larger OS page cache available in main memory. On the other hand, the running times of queries with small working sets is not affected at all by the larger volatile memory capacity.

4.5 Query plans

PostgreSQL, MySQL and DuckDB as open-source databases offer valuable statistics about their query plans. We isolate a subset for all, providing valuable observations on how the DBMSs use the underlying hardware. In this section we only compare the Default with the AppDirect mode.

4.5.1 PostgreSQL. Query 1 has a parallel sequential scan and a parallel aggregation of `lineitem`. The selection predicate is satisfied by 98% of the rows. In AppDirect mode, 85 out of the 106 seconds

are spent in the aggregations, and only 17 seconds are spent in scanning the data. PMEM has a latency of 10us for 4KB application reads [4]. Therefore, for the block size of Postgres (8KB), the latency should be less than 20us. We can confirm that because we have 10 parallel workers with an overall bandwidth of 4-5 GB/s which explains the 17 seconds needed to scan the 86GB table. In contrast, in Default mode, SSDs have a 10x latency, but only 5x of the time is spent in scanning because of the OS page cache and prefetching.

In query 3 the Default mode spends the majority of the time scanning the `lineitem` table. The time is larger than query 1 because the prefetcher does not work so effectively. That happens because the inter-arrival time of read calls is lower as fewer operations are performed in higher-level operators. For the AppDirect mode, the scan time increases to 19 seconds compared to query 1. This is due to the fact that parallel workers are also involved in writing data to PMEM during the join and sorting of the query. It is well known that writing in PMEM requires more CPU and memory resources and provides lower bandwidth [34, 42]

Query 13 has an index-only sequential scan on `customer`, which takes the same time in both modes due to prefetching in the Default mode. There is also an index scan in `orders`, which takes approximately the same time in the AppDirect and Default mode. That happens because the index is not correlated, and therefore the database can scan more than one block at the same time. This leads to a large working set that is larger than the buffer cache, which subsequently leads to many reads from PMEM in AppDirect mode. The amount of data read in AppDirect mode is 12× more than that in the Default mode. However, AppDirect mode still performs better, because PMEM bandwidth is 6× larger and it does not involve the synchronous read from DRAM as the Default mode.

In query 15, we observe the asymmetric behaviour of PMEM as we have to read `lineitem` twice. The maximum read bandwidth is 4 GB/s and the maximum write bandwidth is 1 GB/s. To maintain the price/performance ratio, we need to increase the working memory or use a separate storage drive for temporary writes.

Lastly, in query 17 there is a hash join between `lineitem` and `part` with `lineitem` on the probe side without any filtering. For the sequential scan of lineitem, prefetching is effective and the main performance difference is caused by index lookups on lineitem. In such CPU intensive queries, when CPU operations overshadow I/O requests, prefetching is very effective in sequential accesses. On the other hand, for random accesses asynchronous I/O is more beneficial. Therefore, if Postgres adopts asynchronous I/O, the performance difference between the two modes would be negligible.

4.5.2 MySQL. In query 1, most of the time is spent on aggregation and filtering. The scan time of `lineitem` in the Default mode is 436s and in AppDirect mode it is 392s. The small performance difference has two reasons. First, prefetching and the OS page cache is very effective for sequential scans of large tables. Second, the average block size is around 100KB for both modes. The increase in response time for larger block sizes is sublinear in SSDs due to the inherent parallelism. On the other hand, in AppDirect mode the CPU is involved in reading, and the response time is exactly linear.

In query 3, there is a table scan on `customer` and index lookups on `orders` that use a secondary index. The cost of a lookup is 140us for the AppDirect mode and 250us for the Default mode. The difference is not as large as we would expect based on the hardware, but the Default mode has the advantage of prefetching and the OS page cache. Additionally, PMEM has a larger latency when secondary indexes are used because some accesses are served by the OS page cache in the Default mode.

In queries 4 and 5 there are index lookups on `lineitem` using a primary clustered index, and the cost of a lookup is 15us for the AppDirect and 30us Default mode. The lookup times are close, because the queries have a smaller working set that mainly fits in the buffer cache. In general, for smaller working sets secondary and primary index lookups have comparable lookup times since they fit in the buffer cache and the OS page cache is not useful. We also observe this behaviour in query 7.

In query 13, both modes take the same amount of time, because the working set is larger than the buffer cache but smaller than the page cache. Thus, a memory copy from DRAM to DRAM has similar latency to a memory copy from PMEM to DRAM. Queries 19 and 20 involve a join with `lineitem` using a secondary index lookup. Every lookup returns 30 and 7 tuples, respectively. PMEM is faster, because of large block random accesses, even when a small number of tuples is returned within a lookup.

4.5.3 DuckDB. Query 1 has a sequential scan on lineitem that takes 2.6s for PMEM and 5s for the Default mode which is a 2x performance increase, much less than our microbenchmarks suggest. DuckDB streams data in chunks using a vectorized push-based model, meaning that CPU resources have to be split between I/O and processing. That gives an advantage to the Default mode, which is visible in the subsequent operations that take 1.4s for the Default mode and 1.8s for the AppDirect mode.

Query 2 is very CPU intensive and does not perform a lot of I/O, because it does not involve `lineitem`. Most of the time is spent on index joins between the tables, and therefore since I/O is a negligible portion of the query execution time, the total runtime for the Default and the AppDirect mode is almost the same.

Query 15 has two sequential scans on `lineitem`, that for PMEM take 3.04s while for the Default mode they take 5.75s combined. However, since there is a hash group by and an index join afterwards that are very CPU intensive, the absence of DMA gives an advantage to the Default mode. These two operations take 20s for the Default mode and 23s for the AppDirect mode, essentially covering PMEM's bandwidth advantage completely.

Query 19 has a lot of projections and selections on the lineitem and part tables. As DuckDB pushes most of these directly to the columnar storage, the I/O operations contain effectively most of the running time for this query. The Default mode takes 7.5s to scan the lineitem table, whereas the AppDirect mode does 3.5s.

Insights. A few of the characteristics presented in the last sections are now revealed in more detail through the query plans. We see that the absence of DMA and the page cache closes the gap from 25 (the value observed in our microbenchmarks) to 2 times between PMEM and the SSD in some queries. We also notice the advantage of prefetching in CPU intensive queries (e.g. queries 17, 18) that involve large tables on the probe side of a join.

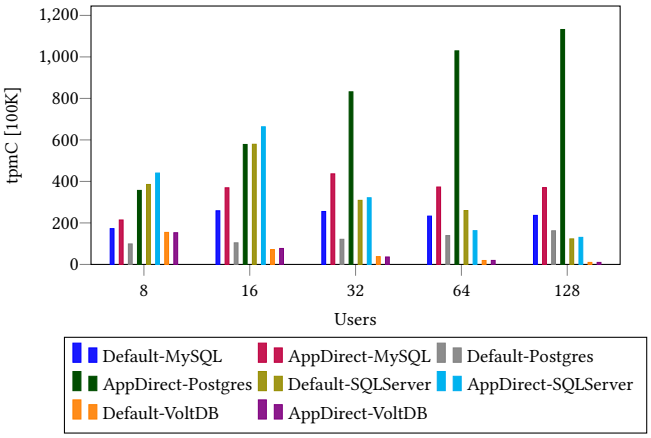

Figure 5: tpmC for TPC-C with 1000 warehouses on all three systems for different system configurations

5 OLTP WORKLOADS (TPC-C)

5.1 General observations

We present the TPC-C results for MySQL, PostgreSQL, SQLServer, and VoltDB in the Default and AppDirect modes in Figure 5. In general, and for different number of concurrent users, the AppDirect mode performs better than the Default mode but depending on the engine the differences vary.

For PostgreSQL, the AppDirect mode outperforms the Default mode by 3-10× for 8-128 threads respectively. That happens because the checkpoint, background, and WAL writer processes use only one thread and this write combining minimizes resource contention for PMEM. We also conduct experiments with the `no-dax` mode, where the page cache is enabled, but the tpmC drops drastically. This happens because the latency of PMEM is low compared to that of SSDs/HDDs and since PostgreSQL relies on the OS for prefetching, keeping/updating the page cache is an additional overhead.

SQLServer, as mentioned, uses all the available cores in socket 0. As we notice, for small number of concurrent users, the higher bandwidth PMEM offers gives a small performance advantage of around 10% compared to the Default mode, due to the lower latency PMEM has compared to SSDs. However, as the number of concurrent users increases, the Default mode is very competitive and outperforms the AppDirect mode for 64 concurrent users by 37%. For such high number of users, there is a lot of interference between I/O and processing in the system and the tpmC drops drastically because of thrashing. Additionally, there is a large number of concurrent reads and writes, which causes a significant performance drop for PMEM [34]. Finally, we observe that CPU utilization is very low for the AppDirect mode, indicating that I/O is in the critical path, despite the much lower latency that PMEM offers.

In MySQL, the AppDirect mode consistently outperforms the Default mode by up to 20%. This happens because of the smaller latency that PMEM offers and also due to the fact that MySQL does not use a large number of threads for writing. The latter helps in avoiding write contention in the iMC, which is one of the main reasons that affect significantly PMEM's performance.

VoltDB's performance is worse than other databases in all but a few cases, with the AppDirect mode showing only marginal improvement over the Default mode. The lower transaction count is due to several factors. Firstly, VoltDB is designed for distributed, high-throughput environments, limiting its performance in single-node setups. Furthermore, VoltDB is the only tested system written in Java, while the others use C/C++. Additionally, upon examining VoltDB's TPC-C implementation, we find that it relies heavily on writing to the storage medium whole snapshots instead of just the changes in the database/log like the other databases. High concurrency high volume writing does not scale well, particularly for PMEM, which struggles with numerous concurrent users. This is also the main reason that contributes to the minimal performance difference between AppDirect and Default modes.

Insights. As TPC-C is a write-heavy workload, we confirm what other studies have found about the effect of writes on the performance of PMEM [42]. Having low number of write threads and/or combining writing operations in one thread provides a much larger transaction rate than the Default mode and it also avoids thrashing, as we observe in the case of MySQL and PostgreSQL. Conversely, heavy contention and mixed read/write workloads significantly impact PMEM performance, bringing it close to or even worse than the Default mode, as seen with SQLServer and VoltDB.

5.2 Log and data placement

We assess PMEM's effectiveness as a data or WAL store for MySQL, PostgreSQL, and SQLServer (Table 5). We could not separate the log and data checkpointing process for VoltDB. We tested configurations with data and WAL on SSD (`SSD-both`), data on SSD and WAL on PMEM (`SSD-data-PMEM-wal`), data on PMEM and WAL on SSD (`PMEM-data-SSD-wal`), and data and WAL on PMEM (`PMEM-both`).

For PostgreSQL, for a small number of clients, when we place the WAL in PMEM instead of the SSD, we see only a slight increase in tpmC. That means that buffered I/O does not significantly affect performance in log writing. However, the gap increases when we increase the clients due to the bandwidth limitations of the SSD. On the other hand, when we place the data in PMEM instead of the SSD, we see a large increase in tpmC, even for a small number of clients. Thus, PMEM should be used as a data rather than just as a log store, because databases can utilize PMEM's read bandwidth more effectively than its write bandwidth.

In SQLServer, placing the log in PMEM offers only a small performance advantage, compared to placing the data in PMEM, indicating that I/O is more on the critical path than the redo logs. As the number of clients increases, log placement does not affect performance significantly. Lastly, placing the data on PMEM and the log on SSD has a higher throughput than placing both on PMEM, but this is due to the remote accesses to socket 1 for the log (that is necessary due to the configuration enforced by SQLServer).

For MySQL that log placement in PMEM does not offer a significant performance advantage since the tpmC increases marginally (up to 5%). Thus, redo logs are not the bottleneck and buffered I/O hides the latency for the Default mode.

Insights. Placing only the logs in PMEM does not offer a significant performance advantage irrespective of the DBMS. We therefore conclude that PMEM should be used as a data or a data and log store

Table 5: tpmC (in 100K) for different database configurations and concurrent users

DB/Users / Configuration	PostgreSQL					SQLServer					MySQL				
	8	16	32	64	128	8	16	32	64	128	8	16	32	64	128
SSD-both	170.8	239.8	332.7	387	474.4	385.2	578.9	308.7	259.4	123	180.6	258.7	262.6	274.1	282.2
SSD-data-PMEM-wal	210.3	313.3	373.5	502.6	495.9	414.3	613.5	321.2	162.5	130.3	189.3	271.5	270	279.7	288.6
PMEM-data-SSD-wal	324.4	520.1	541.6	598.7	645.7	406	662.9	328	172.6	133.2	-	-	-	-	-
PMEM-both	350.6	577.4	854.2	1001.7	1088.4	440.2	663.2	321.2	162.5	130.3	-	-	-	-	-

rather than just a WAL store, because DBMSs can utilize PMEM's read bandwidth more effectively than its write bandwidth.

5.3 Varying database configurations

As open source DBMSs, PostgreSQL and MySQL give us the freedom to experiment with a number of parameters that affect transactions. We analyze various options in this section.

5.3.1 WAL compression. WAL compression is widely used for PostgreSQL to close the CPU-storage gap. However, in our case even though it is slightly useful for the Default mode, it causes a drop in tpmC for the AppDirect mode. The number of transactions is almost 50% less for 16 clients. Due to the lower latency of PMEM, the compression step involving an additional read and write from/to memory becomes unnecessary. Lastly, because PostgreSQL depends on the OS page cache for read-ahead and buffering of writes, we enable the page cache in AppDirect Mode with the no-DAX option. However, as we have mentioned, in AppDirect mode the CPU is involved in reading/writing and this causes an increase in CPU usage and a drop in tpmC as we can see in Figure 8. Therefore, although PostgreSQL is very effective at hiding the storage latency of HDDs and SSDs, this is not the case with PMEM and the techniques it uses for HDDs and SSDs hurt performance in PMEM.

5.3.2 Double write buffer. In MySQL, the double-write buffer is a storage area where dirty pages are flushed to, before writing to their respective positions in the data files [8]. This buffer does not incur additional costs, because InnoDB writes data in a large sequential chunk with a single `fsync` call at the end. We disable the double-write buffer and present the results in Figure 6 for the two modes. We can observe that the tpmC for the Default mode stays stable, since double writes are a part of the background flushing. Therefore, the storage can follow along and transactions are not affected. Conversely, the tpmC increases significantly by up to 20% for the AppDirect mode when double-writes are disabled, confirming again that concurrent writes are a pain point for PMEM.

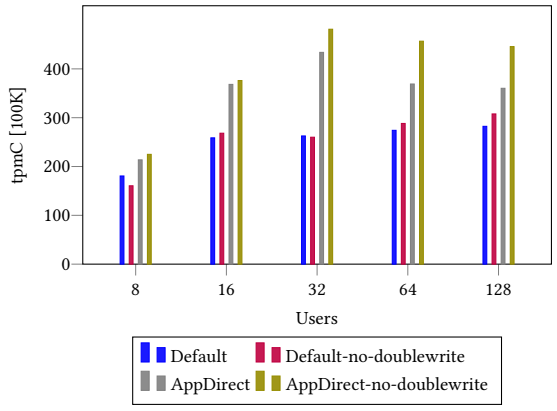

Figure 6: tpmC for TPC-C on MySQL for different number of clients with and without the double-write buffer

5.3.3 Number of write threads. By default, MySQL uses async I/O to perform reads and writes. This removes control over how many concurrent requests are pending, which might be detrimental to PMEM's performance. By using sync I/O, we can control the number of background write threads. Read threads are mainly used for prefetching and with a large buffer pool they do not play a significant role in the TPC-C workload. We vary the number of write threads from 1 to 16 for different numbers of concurrent users and we present the results in Figure 7. The throughput reaches a maximum for 8 threads across all numbers of concurrent users and decreases after that, confirming again that PMEM is not effective at performing many concurrent accesses.

5.3.4 Flushing methods. In MySQL, we have to opportunity to experiment with different flushing methods of InnoDB, e.g. enabling the `O_DIRECT` flag, which uses direct I/O (bypassing the OS page cache) for data files and buffered I/O for logs. We present the results in Figure 9. In general, the AppDirect mode is faster than the Default mode, both with and without direct I/O, due to the smaller latency that PMEM offers. However, direct I/O does not offer a significant performance advantage in the AppDirect mode, because it does not provide any additional benefit on top of DAX. Furthermore, using the AppDirect mode with the `no-dax` option and the `O_DIRECT` flag, provides the same performance as the default AppDirect mode (with `dax` enabled). This is expected, since both configurations skip the OS page cache. For the Default mode, when we have a small number of clients this happens because MySQL manages its own buffer pool and does not depend upon the OS page cache for

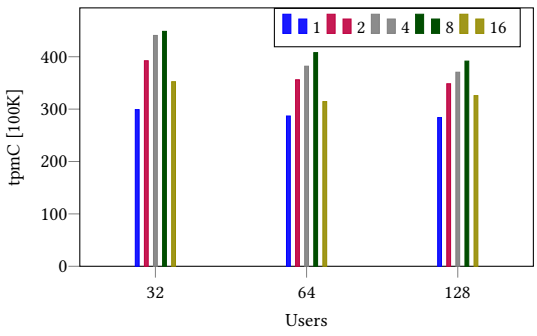

Figure 7: tpmC for TPC-C on MySQL for different number of write threads

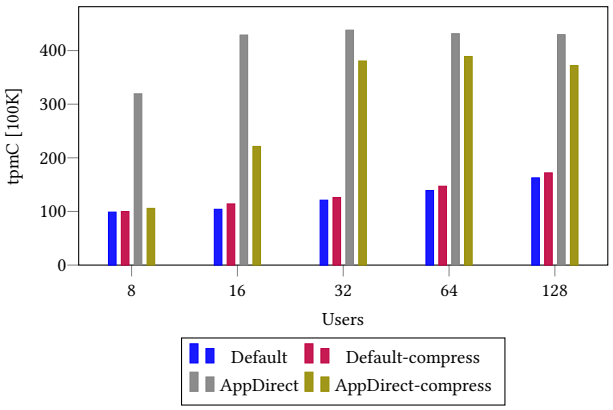

Figure 8: tpmC for TPC-C with 1000 warehouses on PostgreSQL with and without log compression for different configurations

prefetching and caching. As the number of clients increases, the Default mode with direct I/O performs much better, since we have restricted the execution to one socket. Thus, the database is not doing unnecessary memory copy from/to the OS page cache, which in turn reduces contention and leaves more resources available.

We also investigate how disabling `fsync` affects performance. This function is used to ensure that writes are flushed to disk and not to the device cache. Because these calls may degrade performance, MySQL provides a flushing method called `O_DIRECT_NO_FSYNC` that takes them away. As we see in Figure 9, when using this flag, the performance increase is negligible, especially in AppDirect mode. This happens because of two reasons. First, `fsync` calls are not totally eliminated and they are still used for synchronizing file system metadata changes, e.g. during appends [9]. Second, writing dirty pages to the tablespace is done in batches that require a single `fsync` call that happens in the background. Therefore, this does not affect transactions directly [7]. Contrary to the Default mode, when we disable `fsync` we see a large improvement in the AppDirect mode. This happens because overwrites in AppDirect mode with DAX use non-temporal stores. Thus, no cache lines are flushed and only a `sfence` instruction is issued to make sure writes complete.

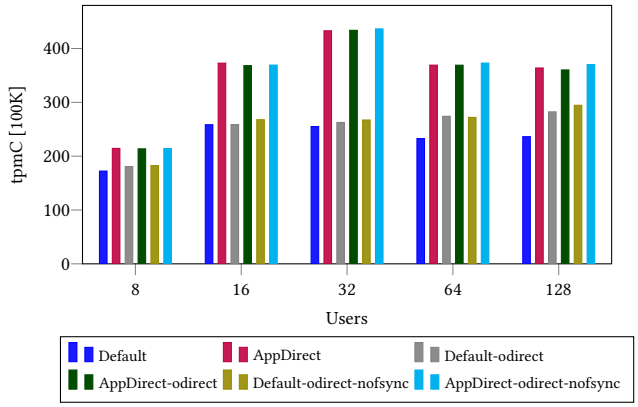

Figure 9: tpmC for TPC-C on MySQL with different flushing methods

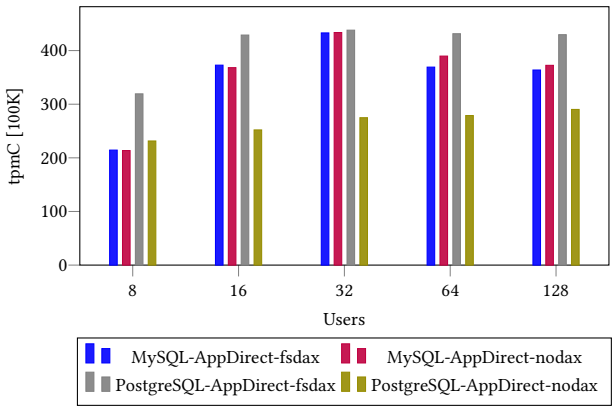

Figure 10: tpmC for TPC-C on MySQL and PostgreSQL with and without fsdax

5.3.5 ext4 block size. In MySQL, we evaluate how the ext4 block size affects PMEM's performance. (Table 6). We set the ext4 block size to 1KB and 4KB in AppDirect mode with `nodax` and `O_DIRECT` enabled, because direct access is only supported when the block size is equal to the system page size. As we observe in the figure, for a small number of clients 1KB block size performs worse than 4KB, because the amount of I/O is increased and I/O is more in the critical path for a small number of clients. Contrary to that, tpmC is larger for 1KB block size for larger number of clients, because access size must be lower for higher thread counts [34].

5.3.6 `no-dax` *mode.* Regarding different configurations in the AppDirect mode, we experiment with the `no-dax` mode in both PostgreSQL and MySQL. In MySQL, the `no-dax` options is on par with `fsdax`. However, on PostgreSQL, the `no-dax` options performs significantly worse. This is because PostgreSQL depends heavily on the OS page cache for read-ahead and buffering of writes and as we have mentioned, in AppDirect mode, the CPU is involved in reading/writing from DRAM to PMEM as there is no DMA available. This causes an increase in CPU usage and a drop in tpmC.

5.3.7 Interleaved vs. non-interlaved. We compare the AppDirect mode in PostgreSQL using only one PMEM DIMM with the AppDirect Mode using both PMEM DIMMs (denoted with AppDirect-interleaved) in Table 7. The interleaving access block size is 4KB and any read/write for PostgreSQL happens in multiples of 8KB. Therefore, when using PMEM in interleaved mode, we would expect a large increase in tpmC, since the access latency is halved and the max bandwidth is doubled. However, as we see in the figure, the increase in tpmC is minimal. For a small number of clients, that happens because the workload is already CPU-bound even when using a single PMEM DIMM. For a larger number of clients, the bottleneck is the checkpointing process. Thus, we increase the

Table 6: tpmC [in 100K] for TPC-C on MySQL for different block sizes on AppDirect mode

Configuration \ Users	8	16	32	64	128
AppDirect-nodax-bs-1kb	210.2	358.9	450.4	404.2	415.8
AppDirect-nodax-bs-4kb	213.6	368.2	433.7	389.7	372.5

Table 7: tpmC for TPC-C on PostgreSQL using different number of DIMMs

Configuration \ Users	8	16	32	64	128
AppDirect	319.3	428.9	438	431.4	429.6
AppDirect-interleaved	346.8	524.2	524.6	533.9	477.4
AppDirect-interleaved-increased-wal	356.6	578	831.7	1029.1	1132

`max_wal_size` to reduce the number of checkpoints and the tpmC for 128 clients is more than 2x higher and is bounded by the max bandwidth of the interleaved setup.

Insights. As PMEM is sensitive to write workloads and its performance can deteriorate heavily, we should carefully tune the underlying parameters to avoid additional writes (e.g. the double-write buffer), use small access sizes for higher thread counts and limit the number of write threads to at most 8 threads. Additionally, since the bandwidth of PMEM in AppDirect mode is much larger than the one that SSDs offer, we do not have to resolve to I/O optimizations, especially if these involve CPU operations.

6 KEY-VALUE STORES (YCSB)

We evaluate RocksDB, an open-source key-value store, using YCSB. We choose YCSB due to its use in PMEM-native key-value store studies [27]. Figure 11 presents throughput and read latency results, while Figure 12 shows write latency for two workloads as the thread count increases. Workload A is update-heavy (50% reads - 50% writes), Workload B is read-heavy (95% reads - 5% writes), and Workload C is read-only (100% read).

In all the graphs, the AppDirect mode consistently achieves higher throughput and lower read latency across all workloads, with the gap widening as the number of threads increases. Meanwhile, Memory mode performs similarly to the Default mode. To understand why, we must examine the workload details. Each workload accesses all fields of a 1000-byte row. With RocksDB processing vast amounts of wide rows, the AppDirect mode experiences read and write amplification, particularly as thread count rises. This results in up to 6.5 GB/s read bandwidth for PMEM (compared to 0.9 GB/s for SSD) and up to 1.3 GB/s write bandwidth for PMEM (compared to 0.4 GB/s for SSD).

We can also make some interesting observations. For Workload A, as we increase the number of threads, reading and writing concurrently, causes the throughput to plateau and slightly drop for the Default and Memory modes, and to heavily drop for the AppDirect mode. In fact, the throughput for 48 threads for the AppDirect mode drops to the level of 16 threads for the same configuration, showing that mixed read-write workloads should keep a moderate number of threads to achieve maximum performance. The plateau effect for the AppDirect mode is also visible if we have a very small percentage of writes such as in Workload B, but it is not visible in Workload C where the throughput continues to increase as we increase the number of threads. For the Default and Memory modes the throughput plateaus consistently after 32 threads because we reach the bandwidth limits of the SSD.

While the read latency of the Default and Memory mode keep increasing for all the workloads, for PMEM in AppDirect mode we observe a plateau in Workload A for the read latency. For the rest of the workloads in AppDirect mode, the read latency keeps slighly increasing. Additionally, the read latency for the Memory mode is slighly lower for higher number of threads compared to the Default mode and the corresponding throughput is higher. Since the workload is 110GB in total the additional volatile memory capacity comes handy as the number of threads increases, because the DRAM capacity is not enough and the requests go to a faster storage medium (PMEM) instead of the SSD. Finally, the write latency of PMEM in AppDirect mode is consistently lower than the rest of the modes, with the exception of 48 threads for Workload B, where the latency is marginally higher. This confirms that even a very low number of writes with a high number of threads can be highly problematic for PMEM.

Insights. The AppDirect mode consistently achieves higher throughput and lower latency across all workloads. However, increasing the number of threads leads to a throughput plateau or drop for mixed workloads. Additionally, the results indicate that even a low number of writes combined with a high number of threads can be problematic for PMEM, highlighting the importance of proper tuning in achieving peak performance. Finally, the Memory mode performs almost identically to the Default mode. Even if the larger volatile memory capacity is handy, the advantage is very marginal.

7 DISCUSSION

7.1 Insights

We summarize the main insights and provide best practices that a DBMS should follow to maximize the PMEM utilization.

(i) Memory mode does not offer a significant performance advantage: As we observe for all engines and benchmarks, in the vast majority of cases, Memory mode with its larger volatile memory capacity does not offer any performance advantage. There are even cases, where performance is slightly worse. This happens due to the conflict cache misses in the DRAM L4 cache, which also causes additional memory traffic between PMEM and DRAM. The only case where Memory mode is useful is for queries with sequential accesses and low number of reads/writes to memory. That happens because the DBMS can take advantage of the larger OS page cache offered by the larger memory capacity and together with prefetching, these queries can have a small speedup compared to configurations that do not use PMEM at all. Nevertheless, it is worth noting that other studies in the HPC context [48] show that a working set considerably larger than typical DRAM capacities, makes the Memory mode useful.

(ii) Performance gains when using PMEM vs. SSDs can be due to application limitations rather than differences in hardware: Although PMEM has superior performance compared to SSDs for sequential read accesses with many threads, this is not the same for random accesses. If applications adopt asynchronous I/O for random accesses (e.g., index lookups) for workloads in which CPU requests overshadow I/O, the Default and the AppDirect mode will have a very small performance difference.

(iii) The lower latency of PMEM in AppDirect mode does

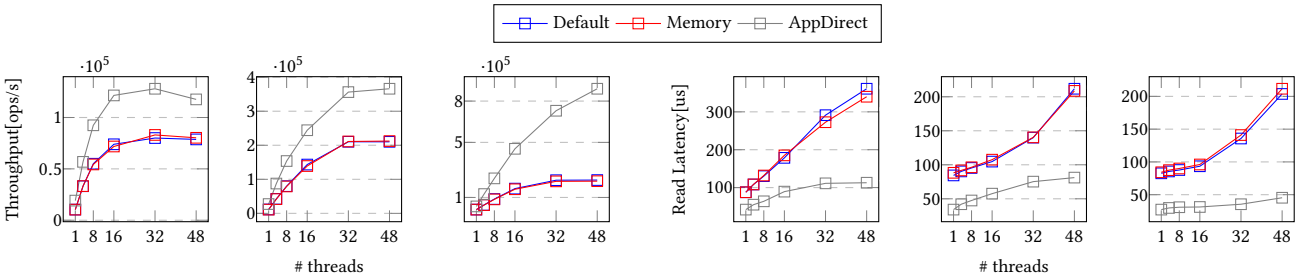

Figure 11: RocksDB throughput (left) and read latency (right) for Workload A, B, and C of the YCSB respectively

not translate to an advantage equal to the hardware characteristics: Our microbenchmarks reveal PMEM has 30x lower latency than SSDs and, in some cases, over 10x higher read and write bandwidth. However, this difference does not result in the expected runtime reduction, as the I/O path is not fully optimized. In AppDirect mode with `fsdax` (recommended by Intel), there is no OS page cache, and DMA is unavailable since PMEM is managed by the iMC, not as a peripheral device. Consequently, CPU resources are spent on I/O in AppDirect mode, while in Default mode, DMA and the OS page cache can cover part of the lower latency, especially in CPU-intensive, I/O-heavy queries or in cases involving secondary indexes with table data stored in the OS page cache.

(iv) In systems where resources are limited, PMEM in AppDirect mode is not as useful: In general PMEM in AppDirect mode involves the CPU as no DMA is available. When there is a lot of resource contention, the hardware advantage of PMEM is almost negligible. We notice this behaviour in various DBMSs (in MySQL for TPC-H, when we restricted PostgreSQL to one core, and in SQLServer and VoltDB for TPC-C).

(v) PMEM requires fine-tuning for write or mixed workloads: As noticed by previous studies [34, 42] heavy-write workloads and read/write interference decrease PMEM's performance dramatically. We notice the same behaviour when running TPC-C or when we use Workload A and B in the YCSB. Even a small number of writes with a high thread count in mixed workloads can hinder the performance of PMEM significantly. Thus, the number of (write) threads has to be carefully tuned to avoid interference. Especially in a DBMS context, several configurations that increase additional writes should be turned off to increase performance (e.g., the double-write buffer).

(vi) Optimizations made for SSDs/HDDs need to be re-designed when using PMEM: Many traditional optimizations avoid I/O as much as possible due to the latency gap between DRAM and storage (e.g. log compression). However, this gap is not as large with PMEM and these optimizations may not offer any performance advantage. In general, as the CPU is involved in I/O in AppDirect mode, it is preferable to avoid devoting CPU resources to optimize I/O.

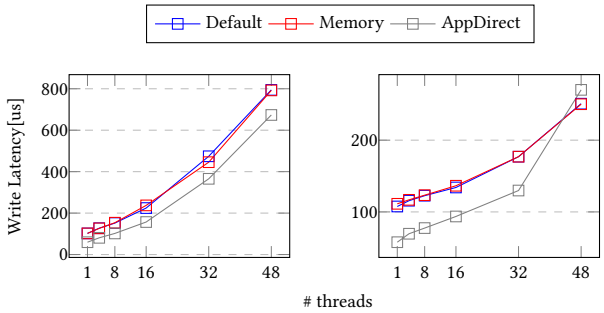

Figure 12: RocksDB write latency for Workload A and B of the YCSB respectively

(vii) Log placement in PMEM does not increase performance significantly: Across DBMSs, placing only the log in PMEM does not increase the transaction rate for TPC-C significantly. Placing data or both the log and data in PMEM increases throughout for TPC-C due to the lower latency and higher bandwidth of PMEM compared to SSDs/HDDs.

7.2 Future directions

As of July 2022, Intel decided to discontinue the Optane series [15]. Although DRAM may reach capacities up to 512GB with DDR5, those will come with a significant cost. Therefore, there is still a need for an intermediate storage tier between DRAM and SSDs. That gap can be filled with CXL [14], which is a cache-coherent interface for connecting CPUs, memory and accelerators. CXL-attached memory can increase the memory capacity of servers, offering a fast storage memory tier to place "hot" data that does not fit in DRAM. Several characteristics of PMEM are relevant to CXL-attached memory, such as byte-addressability and near-DRAM performance with higher capacity and opportunities for higher energy efficiency. Especially the last point will be highly valuable in the context of data centers, where CXL will be deployed first. Therefore our study provides insights on what system function (e.g., pre-fetching, low CPU utilization, query optimization hints, etc.) are valuable in the database context, to avoid performance regressions.

8 CONCLUSION

We evaluated PMEM on various engines and popular benchmarks (TPC-H, TPC-C, and YCSB). Our study sheds light on how to efficiently integrate PMEM into the memory hierarchy and how to tune relational and non-relational systems to get the best performance out of each configuration, given the higher cost both in terms of storage and CPU. In Memory mode, increasing volatile memory capacity using PMEM does not offer any performance advantage. In AppDirect mode, PMEM offers a lower latency than SSDs, which can increase performance significantly when I/O is in the critical path. However, when there is a lot of resource contention or mixed workloads and because the I/O path is not fully optimized in the case of PMEM, SSDs still remain a competitive solution.

REFERENCES

[1] Accessed 2021-10-18. Peloton. https://pelotondb.io/.

[2] Accessed 2021-10-19. Intel®Optane©DC Persistent Memory. https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html.

[3] Accessed 2021-10-25. Intel Vtune Platform Profiler. https://www.intel.com/content/dam/develop/external/us/en/documents/vtuneplatformprofilerwhitepaper.pdf.

[4] Accessed 2021-11-01. Faster access to more data. https://www.intel.ca/content/www/ca/en/architecture-and-technology/optane-technology/faster-access-to-more-data-article-brief.html.

[5] Accessed 2021-11-02. InnoDB: Parallel read of index. https://dev.mysql.com/worklog/task/?id=11720#tabs-11720-2.

[6] Accessed 2021-11-04. PGTune. https://pgtune.leopard.in.ua/.

[7] Accessed 2021-11-09. Fsync performance on storage devices. https://www.percona.com/blog/2018/02/08/fsync-performance-storage-devices/.

[8] Accessed 2021-11-09. Mysql 8.0 reference manual: doublewrite buffer. https://dev.mysql.com/doc/refman/8.0/en/innodb-doublewrite-buffer.html.

[9] Accessed 2021-11-09. Mysql 8.0 reference manual: innoDB startup options and system variables. https://dev.mysql.com/doc/refman/8.0/en/innodb-parameters.html#sysvar_innodb_flush_method.

[10] Accessed 2021-11-16. fio - Flexible I/O tester rev. 3.27. https://fio.readthedocs.io/en/latest/fio_doc.html.

[11] Accessed 2021-11-16. Ioping. https://github.com/koct9i/ioping.

[12] Accessed 2021-11-23. Configure persistent memory (PMEM) for SQL Server on Linux. https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-configure-pmem?view=sql-server-ver15.

[13] Accessed 2021-11-25. Performance best practices and configuration guidelines for SQL Server on Linux. https://docs.microsoft.com/en-us/sql/linux/sql-server-linux-performance-best-practices?view=sql-server-ver15.

[14] Accessed 2022-11-22. Compute Express Link. https://www.computeexpresslink.org/.

[15] Accessed 2022-11-22. Discontinuation of Intel Optane. https://www.intel.com/content/www/us/en/support/products/99743/memory-and-storage/intel-optane-memory/intel-optane-memory-series.html.

[16] Accessed 2023-03-22. HammerDB. https://www.hammerdb.com/.

[17] Accessed 2023-03-22. VoltDB-TPCC. https://github.com/VoltDB/voltdb/tree/master/tests/test_apps/tpcc.

[18] Accessed 24-03-2023. How Does the DRAM Caching Work in Memory Mode Using Intel® Optane™ Persistent Memory? https://www.intel.com/content/www/us/en/support/articles/000055901/memory-and-storage/intel-optane-persistent-memory.html.

[19] Accessed 24-03-2023. Intel® Optane™ Persistent Memory startup guide. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel_Optane_Persistent_Memory_Start_Up_Guide.pdf.

[20] Accessed 24-03-2023. Intel® Optane™ Persistent Memory startup guide. https://www.intel.com/content/dam/support/us/en/documents/memory-and-storage/data-center-persistent-mem/Intel_Optane_Persistent_Memory_Start_Up_Guide.pdf.

[21] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, et al. 2017. SAP HANA adoption of non-volatile memory. *Proceedings of the VLDB Endowment* (2017), 1754–1765.

[22] Joy Arulraj and Andrew Pavlo. 2017. How to build a non-volatile memory database management system. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 1753–1758.

[23] Joy Arulraj and Andrew Pavlo. 2019. Non-volatile memory database management systems. *Synthesis Lectures on Data Management* (2019), 1–191.

[24] 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*. 707–722.

[25] Joy Arulraj, Andy Pavlo, and Krishna Teja Malladi. 2019. Multi-tier buffer management and storage system design for non-volatile memory. *arXiv preprint arXiv:1901.10938* (2019).

[26] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind logging. *Proceedings of the VLDB Endowment* (2016), 337–348.

[27] Lawrence Benson, Hendrik Makait, and Tilmann Rabl. 2021. Viper: An efficient hybrid pmem-dram key-value store. *Proceedings of the VLDB Endowment* (2021), 1544–1556.

[28] Lawrence Benson, Leon Papke, and Tilmann Rabl. 2022. PerMA-bench: benchmarking persistent memory access. *Proceedings of the VLDB Endowment* (2022), 2463–2476.

[29] Maximilian Böther, Otto Kißig, Lawrence Benson, and Tilmann Rabl. 2021. Drop It In Like It's Hot: An Analysis of Persistent Memory as a Drop-in Replacement for NVMe SSDs. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DaMoN 2021)*. 1–8.

[30] Andreas Chatzistergiou, Marcelo Cintra, and Stratis D Viglas. 2015. Rewind: Recovery write-ahead system for in-memory non-volatile data-structures. *Proceedings of the VLDB Endowment* (2015), 497–508.

[31] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* (2015), 786–797.

[32] Youmin Chen, Youyou Lu, Kedong Fang, Qing Wang, and Jiwu Shu. 2020. UTree: A Persistent B+-Tree with Low Tail Latency. *Proceedings of the VLDB Endowment* (2020), 2634–2648.

[33] Brian F Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. 2010. Benchmarking cloud serving systems with YCSB. In *Proceedings of the 1st ACM symposium on Cloud computing*. 143–154.

[34] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing persistent memory bandwidth utilization for OLAP workloads. In *Proceedings of the 2021 International Conference on Management of Data*. 339–351.

[35] Tudor David, Aleksandar Dragojević, Rachid Guerraoui, and Igor Zablotchi. 2018. Log-Free Concurrent Data Structures. In *Proceedings of the 2018 USENIX Conference on Usenix Annual Technical Conference (ATC 2018)*. 373–385.

[36] Subramanya R Dulloor, Sanjay Kumar, Anil Keshavamurthy, Philip Lantz, Dheeraj Reddy, Rajesh Sankaran, and Jeff Jackson. 2014. System software for persistent memory. In *Proceedings of the Ninth European Conference on Computer Systems*. 1–15.

[37] Michal Friedman, Naama Ben-David, Yuanhao Wei, Guy E. Blelloch, and Erez Petrank. 2020. NVTraverse: In NVRAM Data Structures, the Destination is More Important than the Journey. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2020)*. 377–392.

[38] Michal Friedman, Erez Petrank, and Pedro Ramalhete. 2021. *Mirror: Making Lock-Free Data Structures Persistent*. 1218–1232.

[39] Caixin Gong, Chengjin Tian, Zhengheng Wang, Sheng Wang, Xiyu Wang, Qiulei Fu, Wu Qin, Long Qian, Rui Chen, Jiang Qi, et al. 2022. Tair-PMem: a fully durable non-volatile memory database. *Proceedings of the VLDB Endowment* (2022), 3346–3358.

[40] Yuliang He, Duo Lu, Kaisong Huang, and Tianzheng Wang. 2022. Evaluating Persistent Memory Range Indexes: Part Two. *arXiv preprint arXiv:2201.13047* (2022).

[41] Deukyeon Hwang, Wook-Hee Kim, Youjip Won, and Beomseok Nam. 2018. Endurable Transient Inconsistency in Byte-Addressable Persistent B+-Tree. In *16th USENIX Conference on File and Storage Technologies (FAST 18)*. 187–200.

[42] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R Dulloor, et al. 2019. Basic performance measurements of the intel optane DC persistent memory module. *arXiv preprint arXiv:1903.05714* (2019).

[43] Andres Lagar-Cavilla, Junwhan Ahn, Suleiman Souhlal, Neha Agarwal, Radoslaw Burny, Shakeel Butt, Jichuan Chang, Ashwin Chaugule, Nan Deng, Junaid Shahid, et al. 2019. Software-defined far memory in warehouse-scale computers. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. 317–330.

[44] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment* (2019), 574–587.

[45] Gang Liu, Leying Chen, and Shimin Chen. 2021. Zen: a high-throughput log-free OLTP engine for non-volatile main memory. *Proceedings of the VLDB Endowment* (2021), 835–848.

[46] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302* (2020).

[47] Faisal Nawab, Joseph Izraelevitz, Terence Kelly, Charles B. Morrey, Dhruva R. Chakrabarti, and Michael L. Scott. 2017. Dalí: A Periodically Persistent Hash Map. In *31st International Symposium on Distributed Computing (DISC 2017)*. 37:1–37:16.

[48] Onkar Patil, Latchesar Ionkov, Jason Lee, Frank Mueller, and Michael Lang. 2019. Performance characterization of a dram-nvm hybrid memory architecture for hpc applications using intel optane dc persistent memory modules. In *Proceedings of the International Symposium on Memory Systems*. 288–303.

[49] Andrew Pavlo, Gustavo Angulo, Joy Arulraj, Haibin Lin, Jiexi Lin, Lin Ma, Prashanth Menon, Todd C Mowry, Matthew Perron, Ian Quah, et al. 2017. Self-Driving Database Management Systems.. In *CIDR*, Vol. 4. 1.

[50] Ivy B Peng, Maya B Gokhale, and Eric W Green. 2019. System evaluation of the intel optane byte-addressable nvm. In *Proceedings of the International Symposium on Memory Systems*. 304–315.

[51] Georgios Psaropoulos, Ismail Oukid, Thomas Legler, Norman May, and Anastasia Ailamaki. 2019. Bridging the latency gap between NVM and DRAM for latency-bound operations. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. 1–8.

[52] Anil Shanbhag, Nesime Tatbul, David Cohen, and Samuel Madden. 2020. Large-scale in-memory analytics on Intel® Optane™ DC persistent memory. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*. 1–8.

[53] 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*. 1541–1555.

[54] Alexander Van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent memory I/O primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. 1–7.

[55] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies (FAST 2011)*. 5.

[56] Tianzheng Wang and Ryan Johnson. 2014. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment* (2014).

[57] Michèle Weiland, Holger Brunst, Tiago Quintino, Nick Johnson, Olivier Iffrig, Simon Smart, Christian Herold, Antonino Bonanni, Adrian Jackson, and Mark Parsons. 2019. An early evaluation of Intel's optane DC persistent memory module and its impact on high-performance scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–19.

[58] Kan Wu, Andrea Arpaci-Dusseau, Remzi Arpaci-Dusseau, Rathijit Sen, and Kwanghyun Park. 2019. Exploiting intel optane ssd for microsoft sql server. In *Proceedings of the 15th International Workshop on Data Management on New Hardware*. 1–3.

[59] Yinjun Wu, Kwanghyun Park, Rathijit Sen, Brian Kroth, and Jaeyoung Do. 2020. Lessons learned from the early performance evaluation of Intel Optane DC Persistent Memory in DBMS. In *Proceedings of the 16th International Workshop on Data Management on New Hardware*. 1–3.

[60] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the performance of intel optane persistent memory: a close look at its on-DIMM buffering. In *Proceedings of the Seventeenth European Conference on Computer Systems*. 488–505.

[61] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. 2020. An empirical guide to the behavior and use of scalable persistent memory. In *18th {USENIX} Conference on File and Storage Technologies ({FAST} 20)*. 169–182.

[62] Jun Yang, Qingsong Wei, Cheng Chen, Chundong Wang, Khai Leong Yong, and Bingsheng He. 2015. NV-Tree: Reducing Consistency Cost for NVM-Based Single Level Systems. In *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST 2015)*. 167–181.

[63] Zhou Zhang, Zhaole Chu, Peiquan Jin, Yongping Luo, Xike Xie, Shouhong Wan, Yun Luo, Xufei Wu, Peng Zou, Chunyang Zheng, Guoan Wu, and Andy Rudoff. 2022. PLIN: A Persistent Learned Index for Non-Volatile Memory with High Performance and Instant Recovery. *Proceedings of the VLDB Endowment* (2022), 243–255.

[64] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *Proceedings of the 2021 International Conference on Management of Data*. 2195–2207.

[65] Yoav Zuriel, Michal Friedman, Gali Sheffi, Nachshon Cohen, and Erez Petrank. 2019. Efficient Lock-Free Durable Sets. In *Proceedings of the ACM on Programming Languages (PACMPL 2019)*.