How to use Persistent Memory in your Database

Dimitrios Koutsoukos, Raghav Bhartia, Ana Klimovic, Gustavo Alonso
Systems Group, Department of Computer Science
ETH Zurich, Switzerland
{dkoutsou,rbhartia,aklimovic}@ethz.ch,alonso@inf.ethz.ch

ABSTRACT

Persistent or Non Volatile Memory (PMEM or NVM) has recently become commercially available under several configurations with different purposes and goals. Despite the attention to the topic, we are not aware of a comprehensive empirical analysis of existing relational database engines under different PMEM configurations. Such a study is important to understand the performance implications of the various hardware configurations and how different DB engines can benefit from them. To this end, we analyze three different engines (PostgreSQL, MySQL, and SQLServer) under common workloads (TPC-C and TPC-H) with all possible PMEM configurations supported by Intel's Optane NVM devices (PMEM as persistent memory in AppDirect mode and PMEM as volatile memory in Memory mode). Our results paint a complex picture and are not always intuitive due to the many factors involved. Based on our findings, we provide insights on how the different engines behave with PMEM and which configurations and queries perform best. Our results show that using PMEM as persistent storage usually speeds up query execution, but with some caveats as the I/O path is not fully optimized. Additionally, using PMEM in Memory mode does not offer any performance advantage despite the larger volatile memory capacity. Through the extensive coverage of engines and parameters, we provide an important starting point for exploiting PMEM in databases and tuning relational engines to take advantage of this new technology.

1 INTRODUCTION

I/O overhead is one of the main bottlenecks in database engines. The growing DRAM capacity over the years has helped but data sizes keep increasing [28], making DRAM capacity insufficient and often too expensive. To alleviate the memory pressure and to accelerate I/O, Persistent or Non Volatile Memory (PMEM or NVM) has been proposed as a solution. NVM is both cheaper and has a higher capacity than DRAM, it is byte-addressable, and it persists data. The price is higher latency and lower bandwidth. Database researchers have extensively studied how to integrate NVM into various layers of a DBMS [17–20, 37]. Nevertheless, most of these studies are based on simulating PMEM since they were done before it was commercially available. The picture has vastly changed since the release of Intel®Optane©DC Persistent Memory Module [8], the first public commercial implementation of persistent memory. Intel®Optane©DC Persistent Memory 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 other part in Memory mode.

Optane has sparked numerous studies that benchmark and explain the behaviour of the hardware [27, 32, 34, 40]. These studies demonstrated the capabilities of Intel Optane and illustrated its basic characteristics compared to DRAM and SSDs. For example, they showed that for sequential workloads PMEM has comparable latency to DRAM while it is considerably slower for random accesses. They also demonstrated the asymmetric behaviour of PMEM between read and write bandwidth and that interference between different workloads drastically affects performance.

However, DBMSs are complex systems that behave very differently with the workload or the amount of data. Additionally, databases have many knobs that can be altered, which can affect performance significantly. Although the database community has started to evaluated PMEM on relational workloads [22, 25, 41], these studies give only part of the picture as they provide only high-level conclusions and are mostly based on micro- or small-scale benchmarks. Furthermore, they do not experiment with different database knobs and they do not correlate database characteristics with PMEM behaviour. To this end, in this paper we provide what to our knowledge is the first extensive analysis relational database engines using Intel®Optane©DC Persistent Memory: we run OLAP and OLTP workloads (TPC-H and TPC-C) on three DBMSs (PostgreSQL, MySQL, and SQLServer) under various PMEM configurations allowed by the hardware (i.e., PMEM as volatile and persistent memory). We modify the database configurations to put I/O as much as possible in the critical path to magnify the hardware differences. We also experiment with database knobs to see their effect on performance. Additionally, we give details on how query plans and different operators behave on PMEM.

Our results provide a complex picture that shows that using PMEM does not always translate into better performance, despite its hardware advantage. 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. Furthermore, in systems where there is a lot of resource contention and mixed workloads, PMEM experiences a large performance drop. That makes SSDs competitive for several scenarios, e.g. sequential queries with high selectivity. Similarly, although PMEM offers extra volatile memory capacity in Memory mode, this does not translate to a performance benefit. As of today, and based on these results, it seems that the best use of PMEM is as faster (but more expensive) storage. However, there are many caveats and researchers should carefully tune their engines and take into account the nature of the workloads to leverage the new hardware.
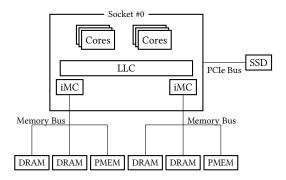
Figure 1: Socket topology

(a) Memory   (b) AppDirect   (c) Mixed

Figure 2: Memory hierarchy for different PMEM modes

2 BACKGROUND

2.1 Persistent Memory

Persistent memory, also known as non-volatile or storage class memory, is an emerging class of memory technology combining the byte-addressability and low latency of DRAM with the persistence and high capacity of disks. The technology is now available in the Intel®Optane©DC Persistent Memory Module [8] (referred to as PMEM in the rest of the paper). It is based on 3D Xpoint technology, which 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.

Every memory channel is connected directly to the CPU through an integrated Memory Controller (iMC) (Figure 1). The iMC maintains read and write pending queues for the PMEM DIMMs and it is situated within Intel's asynchronous DRAM refresh domain, which ensures persistence on power failure. Every iMC has at most 3 memory channels and every CPU socket has two iMCs, resulting in a maximum memory capacity of 6TB for a 2-socket server with 12 memory channels. The access granularity from the iMC to PMEM is 64 bytes, however PMEM's internal access granularity is 256 bytes. That may lead to read or write amplification in different access patterns. For example, in sequential access patterns, PMEM has the highest performance and it is only 2x slower than DRAM, since one internal access serves four external accesses. PMEM can be used in 3 modes: *Memory Mode*, *AppDirect Mode*, and *Mixed Mode* (Figure 2). We denote in parenthesis how the OS treats DRAM or PMEM in different modes.

In *Memory Mode*, PMEM is used as volatile memory and DRAM becomes an L4 direct-mapped cache [9] transparent to the application. The iMC manages data traffic between DRAM and PMEM. If a read request results in a DRAM cache hit, the memory controller serves the read. In case of a DRAM cache miss, it issues a second read to PMEM. Therefore, for a cache hit the application experiences only the DRAM latency, whereas for a cache miss, it experiences the latency of both PMEM and DRAM. Applications can use PMEM in this mode with no changes in their source code.

In *AppDirect Mode*, PMEM acts as a persistent storage device much as an SSD or HDD disk. It can be configured as interleaved or standalone regions. When used as an interleaved region, PMEM provides striped read/write operations that offer increased throughput. The supported interleaved size is 4 KB. In this mode, PMEM contains one or more namespaces, which are similar to hard disk partitions.
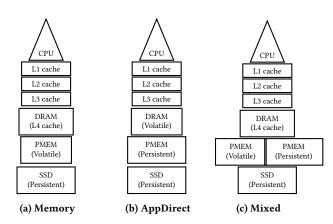
A namespace can be configured in `fsdax`, `devdax`, `sector`, or `raw` mode. Fsdax provides a file system with direct access. If we mount on the device a DAX-aware file system, we can execute load and stores from/to PMEM without using the page cache or any other OS intervention. This is the mode that is recommended by Intel. Devdax presents the device as a character device, exposed to the OS as a single file. Therefore, no filesystem can be mounted on it and read/write system calls are not supported. Sector mode configures the storage as a block device on which any non-DAX file system can be mounted. Finally, `raw` mode presents the device as a memory disk without any support for mounting a DAX filesystem.

Mixed Mode is a combination of Memory and App Direct modes. The entire DRAM is again configured as an L4 cache. The user can configure the percentage of PMEM that the system will use as volatile memory or disk, however the minimum memory DRAM to PMEM ratio recommended by Intel is 1:4.

2.2 Related work

Even before the first public hardware implementation of Non-Volatile Memory (NVM), researchers and companies were adapting DBMS components or tailoring DBMSs to NVM characteristics. Chatzistergiou et al. [23] developed a library that manages transactional updates in data structures designed for NVM and Wang et al. [39] came up with a distributed logging protocol that leverages NVM. Similarly, Arulraj and Pavlo adapted different parts of a database engine to NVM, such as the buffer manager [43], and the storage and recovery [19, 21] protocol. With their findings, they developed a DBMS, Peloton [13, 33], built to leverage NVM principles and characteristics. Similarly, SAP HANA adapted their engine to integrate NVM [16] and Liu et al. [30] built a log-free OLTP engine for NVM. Finally, van Renen et al. [37] evaluate how DBMSs perform when they use either NVM exclusively or a hybrid approach with a page-based DRAM cache in front of NVM. In addition to these efforts building new engines or parts of it to take advantage of PMEM, there has been considerable work on algorithms such as index joins [35], or data structures such as B+-Trees [24], hash tables [31], and range indexes [29].

In most of the above efforts, the authors were simulating NVM using software tools [26] to benchmark the resulting implementations. However, with the introduction of Intel Optane Persistent Memory, it is possible to execute the workloads on actual hardware.

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 $\times$ 8)
PMEM	512GB (128GB $\times$ 4)
SSD	KIOXIA KPM6XRUG1T92 (2TB)

Table 1: Server specifications

This has led to several performance analyses appearing in the last years. Outside of the database context, Swanson et. al [27] provided the first comprehensive performance measurements of Intel Optane Persistent Memory 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). Based on their findings, they propose best practices on how to leverage PMEM. This study was followed by others [34, 42] providing more in-depth analysis and elaborating on the insights of Swanson et. al. In HPC, a number of studies [32, 40] have shown promising results in hybrid DRAM-NVM configurations. Using such a configuration, the applications experience only a minimal slowdown and they can scale up to a larger amount of data, due to the larger volatile memory capacity.

In databases, the focus until now has been mainly on OLAP workloads. Benson et. al. [22, 25] use microbenchmarks or more sophisticated benchmarks (Star Schema, TPC-H) to provide best practices for using Intel Optane Persistent Memory and to investigate if it is a valid replacement for NVMe SSDs. In their experiments, they only use the AppDirect mode (both with and without fsdax). Shanbhag et. al [36] also evaluate only the AppDirect mode in an OLAP context. Complementary to that, Wu et. al [41] have run microbenchmarks, TPC-H, and TPC-C on SQL server 2019 using PMEM in Memory and AppDirect mode (with and without fsdax) and they present high-level conclusions on how DBMS can leverage PMEM. Additionally, Renen et al. [38] measure the bandwidth and latency of Intel Optane Persistent Memory and based on their findings they tune log writing and block flushing. Compared to all the aforementioned efforts, besides focusing only on more sophisticated benchmarks (TPC-H and TPC-C) instead of microbenchmarks, we also evaluate three different DBMSs in both AppDirect and Memory mode by altering various configurations and database knobs. In TPC-H, we are the first to show how query plans, specific operations, and system statistics are affected by the usage of PMEM. In TPC-C, we do a deep exploration of the available database knobs and how they affect performance.

3 EXPERIMENTAL SETUP

3.1 System specification

All the experiments in this paper were conducted on a dual-socket server with the specs mentioned in Table 1 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 use Debian 10 (kernel 5.10.46). 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. We collect system statistics with the Intel Vtune Platform Profiler [7] that provides metrics related to CPU, memory, storage, and I/O. To get more accurate statistics, we modify the source code of the profiler to increase the sampling hardware counter frequency.

3.2 Basic microbenchmarks

Before evaluating database engines, we measured the latency and the bandwidth of our system for the different memory types (Tables 2 and 3). For volatile memory (DRAM and PMEM in Memory mode) we use the Intel Memory Checker Tool [6], which gives us the intrasocket idle memory latencies. For measuring latency in storage (SSD and PMEM in AppDirect mode) we use ioping [10]. Finally, for measuring bandwidth in storage we use fio [3]. For storage bandwidth measurements we follow a similar methodology to Swanson et. al [27]. We perform two sets of measurements, one with 1 thread and one with 8 threads, to observe how bandwidth scales with the number of threads. We generate 512MB files and every thread reads 1 file in 4KB blocks. We bypass the page cache and we perform synchronous I/O. For writes, we perform a fsync() call after every 4KB write. As we can observe, PMEM as volatile memory has 23% higher latency. The read bandwidth is 2% less and the write bandwidth is more than 2$\times$ lower than DRAM. For 1 thread PMEM AppDirect has more than 10$\times$ more read bandwidth for both random and sequential accesses and around 5$\times$ more write bandwidth than SSD. For 8 threads PMEM has 13-16$\times$ higher read bandwidth and 3-6$\times$ higher write bandwidth for random and sequential accesses, respectively, than SSD. Finally, we notice a large gap in bandwidth between using PMEM in Memory or AppDirect mode, but this is due to the overhead of the OS and the filesystem.

3.3 Configurations tested

We now present the different configurations that we run for the TPC-H and TPC-C benchmarks. For both TPC-H and TPC-C, we pin the database processes to socket 0 to avoid remote NUMA access to better interpret the results. Before executing each query or workload we clear the page and the database cache. For TPC-C, we use 1000 warehouses for all experiments, unless otherwise stated. We configure each user to use all the warehouses because

Memory type	Latency [us]
DRAM	0.07
PMEM-volatile	0.09
PMEM-storage (random)	9.31
PMEM-storage (sequential)	9.23
SSD (random)	68.6
SSD (sequential)	65.8

Table 2: Latency measurements for different memory types in our server

Memory type	Read bw [MB/s]	Write bw [MB/s]
DRAM	76720	57388
PMEM-volatile	75303	32094
PMEM-storage (random, 8 threads)	6128	1807
PMEM-storage (sequential, 8 threads)	8551	3694
PMEM-storage (random, 1 thread)	1829	534
PMEM-storage (sequential, 1 thread)	2449	615
SSD (random, 8 threads)	470	293
SSD (sequential, 8 threads)	504	477
SSD (random, 1 thread)	160	157
SSD (sequential, 1 thread)	191	166

Table 3: Bandwidth measurements for different memory types in our server

this ensures an even distribution of virtual users across warehouses and avoids spending a lot of time in lock contention. We set the warmup and running time to 3 minutes. That is sufficient to load the buffer pool and stabilize the database activity. Additionally, we have observed that we manage to reach the maximum tpmC by using only one of the two PMEM DIMMs available in the socket. Therefore for TPC-C unless otherwise mentioned, the AppDirect mode uses one PMEM DIMM. Although we run the Memory mode for TPC-C for many configurations, in most cases its performance is almost identical to the Default case as requests are served by the L4 DRAM cache and PMEM is not in the critical path. We therefore do not report any results involving the Memory mode. Finally, we do not run any experiments for Mixed mode, since we do not meet the minimum volatile memory ratio of 1:4 in our setup, which Intel suggests for PMEM to be cost and performance-effective over a DRAM only solution. We have 128GB of RAM and 512GB of PMEM. That leaves no PMEM capacity to be configured in AppDirect mode. Table 4 summarizes the different modes and benchmarks that have been run for all DBMSs considered.

4 POSTGRESQL

4.1 TPC-H

4.1.1 Setup. We set the buffer cache to 16GB and the effective cache size to 48GB. That way all the caches fit in socket 0 for all modes and therefore I/O is in the critical path as much as possible. We use PostgreSQL version 13.2. After importing the data, we add indexes and foreign keys with triggers disabled. Postgres does not have a prefetcher on its own and relies on the OS for this purpose.

4.1.2 General observations. The running times for the 22 TPC-H queries for PostgreSQL for the Default, Memory, and AppDirect modes are shown in Figure 3. The AppDirect mode is consistently faster than the Default mode. This happens because PMEM has

Mode Benchmark	Default	Memory	AppDirect	Mixed
TPC-H	✓	✓	✓	✗
TPC-C	✓	✓	✓	✗

Table 4: Benchmarks and modes run for all the different DBMSs for PostgreSQL, MySQL and SQLServer

lower latency than SSDs and, thus, all I/O is faster. This is more obvious in queries that have large sequential scans (queries 1, 3, 6, 7, 12, 16, 22). Although the lower latency does provide a large advantage, it is not as large as we would expect (around 8-10× according to Table 2). That happens because the AppDirect mode does not use the OS page cache and prefetcher. Therefore, in queries where these components are effectively hiding the I/O latency, the advantage is not large (e.g., query 1), contrary to queries where they do not play a big role (e.g., query 6). Another reason for not seeing all potential advantages is that, in AppDirect mode, the CPU is involved in I/O, whereas in Default mode a parallel worker thread is scheduled to deal with I/O. To verify how much the number of I/O threads affects the overall performance, we also run the AppDirect and Default modes using only one core on socket 0 and we observe that the performance difference between them reduces drastically. However, the AppDirect mode still performs better because, even with one thread, PMEM has a 10× higher read bandwidth (Table 3). Nevertheless, there are situations when AppDirect mode works much better: for parallel index lookups the latency difference is about 10× faster for the AppDirect mode (queries 2, 4, 5). Especially when the tables are large, the lookup access size is in the order of 8KB and, although the accesses are random, they have the same latency as a sequential scan for PMEM (queries 8, 9, 10, 19, 20, 21).

The Memory mode has slightly worse running times for most queries. To understand where the overhead comes from, in Figures 4 and 5 we show the DRAM read and write bandwidth consumed by each query for the Default and Memory modes, respectively. For all the queries, we read and write more data from DRAM in Memory Mode compared to the Default mode. 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 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. Finally, another reason for the increased DRAM bandwidth for both reads and writes are the collisions in the L4 direct-mapped cache, since multiple PMEM memory lines can map to the same DRAM memory line. We verify the last assumption by allocating 64GB instead of 256GB of volatile memory in Memory mode. This way, two PMEM memory lines always map to different
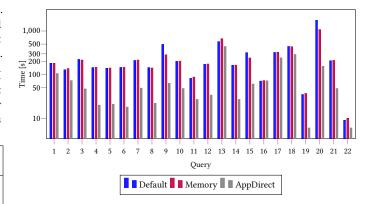

Figure 3: Running time (log-scale) for TPC-H SF-100 on PostgreSQL for different system configurations
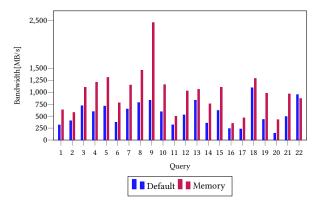
Figure 4: DRAM average read bandwidth on PostgreSQL for different system configurations

DRAM memory lines. This decreases the cache conflicts and the DRAM read/write bandwidth and we observe that the Memory mode has the same running time as the Default mode. We also see in Figure 6 that the Memory mode consistently reads fewer or equal amounts of data because of the larger page cache available, whereas the AppDirect mode consistently reads an equal or bigger amount of data because of the disabled page cache. Exceptions to that are queries 4, 8, 10 and 11, because of the inability to take advantage of the OS cache and prefetching.

We now analyze the differences between the Default and the Memory mode in more detail. Queries 1, 4, 5, 6, 12, 14 and 17 are less than 2% slower. This happens because their working set is less than 64GB and therefore most of the accesses are served by the L4 DRAM cache. On the other hand, queries 2, 11, 13, and 19 are slower by more than 5%, because their working set is larger than the L4 and/or the OS page cache. Finally, queries 9, 15, and 20 are faster in the Memory mode. These queries benefit from the larger OS page cache that the OS can allocate, since socket 0 with PMEM in Memory mode has 256GB of volatile memory, compared to the Default mode which has 64GB.

4.1.3 Query-by-query. In this section, we take a more detailed look at the queries to gain additional insights. Note that we only compare the Default with the AppDirect mode.
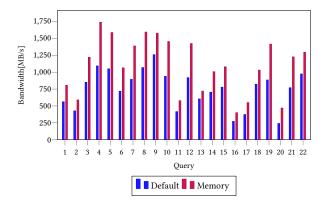

Figure 5: DRAM average write bandwidth on PostgreSQL for different system configurations
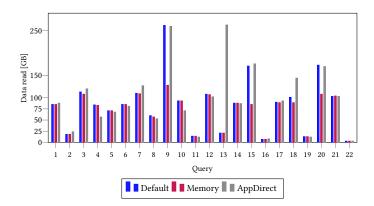

Figure 6: Data from storage read for TPC-H SF-100 on PostgreSQL for different system configurations

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 computing the aggregations, and only 17 seconds are spent in scanning the data. PMEM has a latency of 10us for 4KB application reads [2]. 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 because 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 Intel Optane requires more CPU and memory resources and provides lower bandwidth [25, 27]

In query 5, there is a multibatch parallel hash join where the table orders is scanned. The join does not start executing until the entire table has been scanned. Therefore, the inter-arrival times of read requests are very close and the prefetcher does not work effectively in the Default mode.

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.

Query 14 involves a hash join between lineitem and part with lineitem on the build side. The hash table has to be built completely before the rest of the query plan is executed and therefore we

have a sequential scan on `lineitem`. For the scanning, 6 workers are allocated. The SSD reaches its maximum read bandwidth, but this is not the case for PMEM, which does not reach its maximum read bandwidth even with 9 workers. 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.

4.1.4 Summary. The AppDirect mode is largely beneficial for sequential scans that select very little data. When the majority of the data has to be processed, then the lack of the page cache and prefetching in AppDirect mode eliminates most of the advantages of PMEM. The Default mode is also competitive in queries involving CPU-intensive operations (hash aggregations and joins) when I/O requests are not the bottleneck. Finally, because the CPU is directly involved in I/O in AppDirect mode, if we do not use a high number of threads, the bandwidth utilization is not optimal and the performance is comparable to the Default mode. The Memory mode is slightly slower than the Default mode, except for queries that have a large working set and can benefit from the larger OS page cache offered by PMEM. In addition, the direct-mapped L4 cache leads to larger DRAM utilization due to conflict misses. Applications could allocate volatile memory space contiguously to avoid them.

4.2 TPC-C

4.2.1 Setup. We use PGTune [15] to tune PostgreSQL to our hardware configuration. We set the `checkpoint_timeout` to 30 seconds and the `max_wal_size` to 5GB to force checkpoints very often. It is known that writes have limited bandwidth in PMEM (as seen in Table 3). By checkpointing aggressively, we put I/O in the critical path and we can saturate the PMEM device more easily. This configuration, though not very common, is useful for meeting a very low downtime requirement. We also use a large buffer pool to reduce the proportion of reads to writes. After the warmup time, we checkpoint and run a `VACUUM` command to remove potential sources of performance variations.

4.2.2 Different modes and WAL compression. In Figure 7 we present the result of running TPC-C for different numbers of concurrent users for different system configurations. Besides the Default and the AppDirect mode, we run both these modes with an option that allows as to compress the Write-Ahead Log (WAL) before writing. We denote them with `Default-compress` and `AppDirect-compress` in the figure. We also run TPC-C in AppDirect mode but with DAX disabled (denoted as `AppDirect-nodax` in the figure). As we see, the AppDirect mode reaches a plateau after 32 concurrent users, while
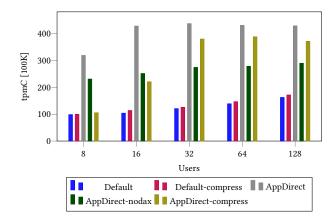

Figure 7: tpmC for TPC-C with 1000 warehouses on Postgres for different system configurations
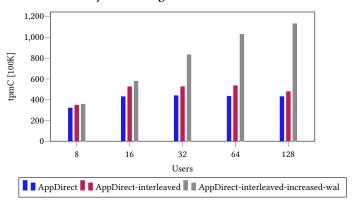

Figure 8: tpmC for TPC-C on Postgres using different number of DIMMs

the Default mode slowly increases. Postgres utilizes the PMEM's bandwidth very efficiently, because the checkpoint, background, and WAL writer processes use only one thread. Additionally, when the client workers get involved in flushing dirty pages, we see a slight tpmC drop.

WAL compression is widely used for Postgres to close the gap between CPU and storage. 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 Postgres 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 7. Therefore, although Postgres 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.

4.2.3 Interleaved vs non-interleaved. We compare the AppDirect mode using only one PMEM DIMM with the AppDirect Mode

using both PMEM DIMMs (denoted with AppDirect-interleaved) in Figure 8. The interleaving access block size is 4KB and any read/write for Postgres 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 `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.

4.2.4 Data and WAL placement. Finally, we evaluate the effectiveness of PMEM as data or a WAL store. We present the results in Figure 18. We put both the WAL and the data in SSD (SSD-both), the data in SSD and the WAL in PMEM (SSD-data-PMEM-wal), the data in PMEM and the WAL in SSD (PMEM-data-SSD-wal), and both the data, and the WAL in PMEM (PMEM-both). 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. We therefore conclude that PMEM should be used as a data rather than just as a WAL store, because databases can utilize PMEM's read bandwidth more effectively than its write bandwidth.

4.2.5 Summary. We see that compressing the WAL as well as placing it in faster storage (e.g. PMEM) does not offer us a large performance advantage. In general, although TPC-C involves many I/O writes and reads, these are still done on the same data group and that makes the workload more CPU than I/O bound. Finally, Postgres utilizes the PMEM bandwidth very effectively in DAX mode but, when the page cache is enabled, the tpmC drops drastically. This happens because the PMEM latency is low compared to that
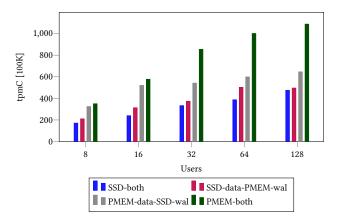
of SSDs/HDDs and since Postgres relies on the OS for prefetching, keeping/updating the page cache is an additional overhead.

5 MYSQL

5.1 TPC-H

5.1.1 Setup. We use MySQL version 8.0.23. We set the buffer pool again to 16GB for the reasons stated in Section 4.1.1 and to have the same parameters across databases. MySQL does not support multiple threads per query, except for particular type of queries, e.g., SELECT COUNT(*) [5].

5.1.2 General observations. In Figure 10, we present the runtimes for TPC-H SF 100 for MySQL for different configurations. Regarding the query times, we observe similar trends to Postgres. The difference between the AppDirect and the Default mode is not as big, because we do not reach the bandwidth limitations of the SSD with a single thread. In addition, compared to Postgres, MySQL has two main differences. First, it prefers nested loop joins over hash or merge joins. Therefore, in queries involving joins, there are random accesses with large block sizes (i.e. 16KB), where PMEM has similar bandwidth to sequential accesses. In contrast, prefetching and the OS page cache cannot help in Default mode for random accesses. Second, 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.

In Memory mode queries 3, 7, 9, 20, and 21 have a smaller running time than the Default mode, because of the larger page cache available. All these queries have a large working set and the data does not fit in the buffer cache. For the other queries, the two modes are either comparable or the Memory mode is slightly worse for the reasons already mentioned in Section 4.1.2.

5.1.3 Query-by-query. We now take a deeper look into a subset of the queries . We again compare only the AppDirect and the Default modes, as those are the ones that have the largest runtime difference. 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

Figure 9: tpmC for TPC-C on Postgres using PMEM as WAL or data store
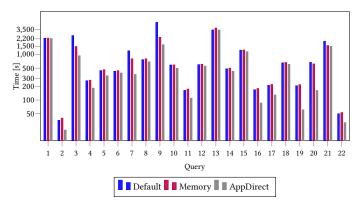

Figure 10: Running time for TPC-H SF-100 on MySQL for different system configurations

has two reasons. First, prefetching and page caching by the OS 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 for the reasons explained in the previous section.

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.

Finally, query 21 has 3 joins with `lineitem`. The first join has a barrier and there is little time difference between the modes, because the index lookup happens in the clustered primary key (i.e. `order_key`) and therefore there are a lot of page cache hits and prefetching is effective. Then, there is a second hash join with `nation` that breaks the ordering of `order_key`. That makes the second join very slow in Default mode because the index lookups on the primary key are random and the working set is very large (i.e. the entire `lineitem`). Since the second join has no barrier, the third join happens simultaneously with all the pages needed available in main memory. As a result, for this join both modes take the same amount of time.

5.1.4 Summary. We observe that operations that have large block accesses ($\geq$ 16KB) index lookups with random accesses have a large performance advantage in AppDirect mode compared to the Default mode. These accesses cannot be effectively prefetched and the PMEM bandwidth is very similar to a sequential access. Additionally, whenever possible, we should use clustered indexes, since the additional lookup in a non-clustered index has a performance penalty for PMEM due to the lack of an OS page cache. For the Memory mode, we make similar conclusions to PostgreSQL, i.e. that we should use it for queries with working sets larger than the DRAM available to take advantage of the larger page cache.
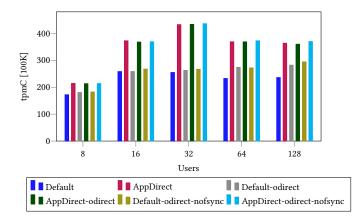

Figure 11: tpmC for TPC-C on MySQL with different flushing methods

5.2 TPC-C

5.2.1 Setup. We use a large buffer pool (48GB unless otherwise mentioned) with similar reasoning as in Section 4.2.1. We disable binary logs and we leave the log buffer size and the log file size at their default values, which is very small. This causes checkpointing every second.

5.2.2 Flushing methods. We first experiment with different flushing methods of InnoDB. More specifically, we enable the `O_DIRECT` flag, which uses direct I/O for data files and buffered I/O for logs. We present the results in Figure 11. 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 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. Based on these, we use the `O_DIRECT` flag for the rest of our experiments.

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 tries to take away these calls. As we can see in Figure 11, when we use this flag, the performance increase is negligible, especially in AppDirect mode. This is a result of mainly two reasons. Firstly, `fsync` calls are not totally eliminated and they are still used for synchronizing file system metadata changes, for example during appends [12]. Secondly, writing dirty pages to the tablespace is done in batches that require a single `fsync` call that happens in the
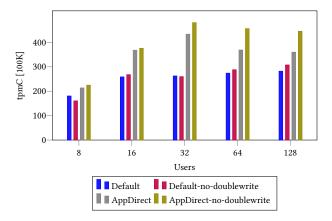
Figure 12: tpmC for TPC-C on MySQL for different number of clients with and without the double-write buffer

background. Therefore, this does not affect transactions directly [4]. 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 the writes are complete.

5.2.3 Concurrent and double-write buffer. We now experiment with the number of concurrent clients and the use of the double-write buffer. By default, the double-write buffer is enabled both in the Default and the AppDirect mode. We present the results in Figure 12. For the Default mode, as the number of concurrent clients increases, the tpmC first increases and then reaches a plateau. Since the CPU utilization is low, the bottleneck is the SSD. The latency keeps increasing due to increasing I/O and CPU scheduling delays. In AppDirect mode, the tpmC increases and reaches a peak before dropping off again. Given that we don't observe this behaviour in the Default mode, the drop is not because of MySQL, but due to PMEM. This behaviour is consistent with other studies [25], which notice a decrease in read and write performance as the number of concurrent accesses increases. When the number of read/writes operations increases, the read and write combining cache does not perform so well, as its size is limited. Another reason for the drop in tpmC is the increasing contention at the iMC level. When clients increase, there are more L3 cache misses, which in turn lead to more DRAM accesses and I/O between DRAM and PMEM.

The double-write buffer is a storage area where dirty pages are flushed to, before writing to their respective positions in the data files [11]. This buffer does not incur any additional cost, because InnoDB writes the 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 12 for the two modes. We see in the figure 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 for the AppDirect mode when double-writes are disabled, confirming once more that concurrent writes are a pain point for PMEM.
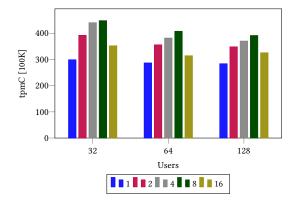

Figure 13: tpmC for TPC-C on MySQL for different number of write threads

5.2.4 I/O threads. By default, MySQL uses async I/O to perform reads and writes. This removes the 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 don't have a significant role for the TPC-C workload. We vary the write threads for different number of concurrent users and we present the results in Figure 13. The throughput reaches a maximum for 8 threads for different numbers of concurrent users and decreases after that, confirming once again that PMEM is not very effective at performing many concurrent accesses.

5.2.5 ext4 block size. We also evaluate how the ext4 block size affects PMEM's performance. We present the results in Figure 14. 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 [25].
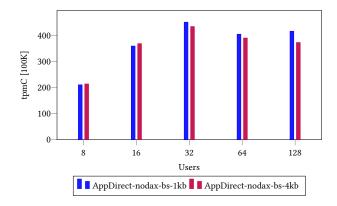

Figure 14: tpmC for TPC-C on MySQL for different block sizes on AppDirect mode

5.2.6 Log placement. To confirm that placing the logs in PMEM does not offer a significant performance advantage irrespective of the DBMS, we place again the log directory in PMEM instead of the SSD and we present the results in Figure 15. We can again confirm that redo logs are not the bottleneck and buffered I/O hides the latency for the Default mode.

5.2.7 Buffer pool size and app direct modes. We finally experiment with the buffer pool size and different modes in AppDirect mode (no-dax, sector). For brevity, we omit the figures and we present only the high-level conclusions. With a smaller buffer pool (16GB), we have more reads per transaction, while writes remain constant because of the aggressive flushing mode. Thus, the AppDirect mode remains unaffected since its read latency (10us) is negligible compared to the overall query latency (10ms). On the other hand, for the Default mode when we have a small number of clients ($\leq$ 16) and a small buffer pool (16GB) the tpmC drops around 20%. This happens due to memory copies between the page cache and the buffer pool. For large number of clients, having a larger buffer pool (48GB) still increases tpmC but not as significantly, since time is split more evenly between I/O and CPU. Regarding different configurations in the AppDirect mode, we experiment with the no-dax and the sector mode. Both perform worse than the dax mode, because we have more levels of indirection. For the no-dax mode, reads and writes still go through the block layer and this requires alignment verification. For the sector mode, besides the block layer, we also have to go through the block translation table.

5.2.8 Summary. Similar to Postgres, log placement in PMEM gives a negligible performance advantage. Since concurrent writes are a bottleneck for PMEM, we should finetune the respective parameters carefully. Specifically, increasing the number of write threads above a threshold decreases performance since the hardware is not as effective in write combining. We can further increase performance by disabling the double-write buffer, which reduces concurrent writes. Performance also drops when there is a lot of interference between reads and writes, e.g. when the buffer pool has a small size. Finally, we see again that in AppDirect mode no-dax performs worse than fsdax because there are more levels of indirection.
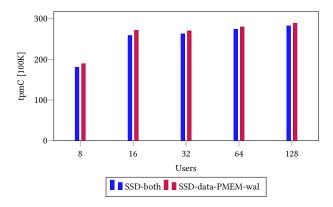

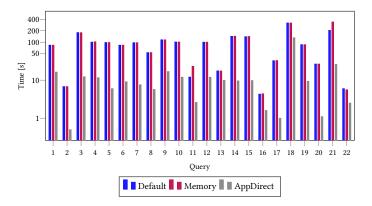**Figure 15: tpmC for TPC-C on MySQL with placing the log directory in PMEM**

Figure 16: Running time (log-scale) for TPC-H SF-100 on SQLServer for different system configurations

6 SQLSERVER

6.1 TPC-H

6.1.1 Setup. We use SQLServer version 2019-15.0.4178. We set the buffer cache to 16GB to put I/O in the critical path and have a consistent configuration across DBMSs. We add non-clustered indexes to foreign keys as SQLServer does not do this automatically. For the AppDirect mode we also place the tempdb that stores intermediate results in PMEM. However, SQLServer does not allow placing logs in a fsdax filesystem [1]. Thus, in AppDirect mode we leave them in the SSD. For OLAP workloads this does not affect performance.

6.1.2 General observations. We present the running times for the 22 TPC-H queries for SQLServer for the Default, Memory and AppDirect modes in Figure 16. SQLServer uses 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. It is also very efficient at optimizing the queries and pipelining I/O and processing. Therefore it reads only once the necessary tables for each query and does not incur redundant I/O. Additionally, SQLServer prefers to write through to physical hardware bypassing the OS page cache [14].

The Default and the Memory mode have almost identical running times for all but two queries. The average CPU utilization is very low for these two modes, indicating that most of the time is spent transferring data from disk to main memory. Prefetching is done by the iMC and processing is interleaved with I/O very efficiently, making the working set of almost all the queries to fit in the L4 cache, making essentially the two modes running under the same hardware. This comes in contrast with the findings of Wu et al. [41], where the Memory mode was slower than the Default mode. However, the authors of this paper perform warm runs, whereas in our case we clear all the caches before performing measurements. That indicates that the Memory mode may be advantageous when I/O is involved and PMEM is effective at hiding the latency difference with DRAM in these cases. We are also not aware of what other optimizations the authors of this paper might have used leading to the observed performance difference. The two queries where Memory mode is slower is queries 11 and 21, where there are many conflict misses in the L4 DRAM cache and a significant data transfer between DRAM and PMEM, compared to the Default

mode. In addition to that, as mentioned, SQLServer doesn't utilize the OS page cache and therefore does not take advantage of the larger volatile memory available in the Memory mode.

Compared to the Default mode, the AppDirect mode is much faster again except for one query. The difference is due to the lower latency that PMEM offers compared to SSDs. Furthermore, since SQLServer uses many threads, it takes advantage of the full PMEM bandwidth and it can reach read bandwidth up to 8 GB/s for some queries. The difference is more obvious for queries involving the `lineitem` table, since it is the larger table in the benchmark (around 100GB of storage together with indexes). We also notice that due to the very high bandwidth SQL server had zero read-ahead reads, i.e. it doesn't place any pages into the cache for the query.

6.1.3 Query-by-query. Since SQLServer is not open-source as the other two databases, we only have estimates of the I/O costs and not the exact runtimes as in PostgreSQL and MySQL. Thus, we cannot provide details about the application memory latency. We choose again a subset of the queries with interesting insights and we compare the Default with the AppDirect mode.

Query 5 has a clustered index seek on `lineitem` on `orderkey`. `Lineitem` and `orders` are the two largest tables of TPC-H and therefore after this seek the working set is reduced substantially. PMEM has 6× more read bandwidth than the SSD for this particular query and it processes tuples as they come, without the need to put additional pages to the buffer cache. The joins of the query are either nested loop joins or hash joins. Since I/O is out of the critical path, there are more resources to efficiently execute the joins resulting in an overall 16× faster runtime for the AppDirect mode compared to the Default mode.

Query 8 has a nested loop join with `lineitem` resulting in a lot of data movement from/to the buffer cache in AppDirect mode. In contrast, this plan puts the Default mode in a disadvantage as the DBMS spends most of the time doing read-ahead reads to the buffer cache. Compared to the other queries the CPU utilization of the AppDirect mode is low (around 60%, where for most of the other it is around 80%). Finally, query 18 is only 3× faster in AppDirect than in the Default mode. In the query plan, we can see that there are 3 nested loop joins and 2 stream aggregations on `lineitem` that expect sorted data. Therefore the query is more CPU than I/O intensive, and that is the reason of the smaller proportional time difference compared to the rest of the benchmark.

6.1.4 Summary. We see that the Memory mode behaves very similarly to the Default mode, except for queries that cause conflict misses in the DRAM L4 cache. Since SQLServer avoids using the OS page cache, it cannot take advantage of the larger capacity of PMEM. On the other hand, the DBMS uses many threads for I/O and processing and utilizes the full read bandwidth of PMEM, making the AppDirect mode much faster than the other two modes.

6.2 TPC-C

6.2.1 Setup. We set the buffer cache to 48GB for similar reasons as in Section 4.2.1. We do automatic checkpoints every minute (the lowest possible value allowed by SQLServer). To use PMEM, SQLServer requires `fsdax` mode to store the data and `sector` mode to store the logs [1]. We therefore have to use both memory sockets,
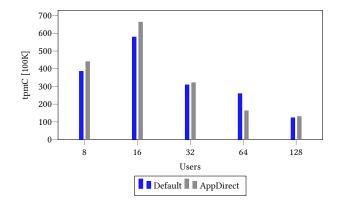

Figure 17: tpmC for TPC-C on SQLServer for different number of concurrent users

as it is not possible to create a mixed namespace in one socket that has the capacity for 1000 warehouses. We have observed that using PMEM in interleaved mode increases performance for PMEM. Thus, we use only this mode in this section. Finally, We also place the `temp` database and logs, which store intermediate results, to the different storage mediums.

6.2.2 Different modes and concurrent users. We first experiment with the number of concurrent clients and we present the results in Figure 17. In general, SQLServer uses all the available cores in socket 0. As we can observe, for small number of concurrent users, the higher bandwidth that PMEM offers gives a small performance advantage compared to the Default mode, due to the lower latency it provides 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. For such a high number of users, there is a lot of interference between I/O and processing in the system. Additionally, there is a large number of concurrent reads and writes, which causes a significant performance drop for PMEM [25]. This performance reduction is so significant that SSD outperforms PMEM for 64 users. 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.

6.2.3 Log placement. We experiment with log placement as we did for the other databases. Generally, placing the log in PMEM offers only a small performance advantage, compared to placing the data in PMEM. This shows that I/O is more on the critical path than the redo logs. As the number of clients increases, we can see that log placement does not make a significant difference. 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).

6.2.4 Summary. Since SQLserver utilizes all the available cores in the socket, the performance drop when there are many concurrent reads and writes in PMEM is very obvious compared to the other databases. We again confirm that the log placement does not play a significant role in the overall performance, showing that PMEM is more attractive for its lower latency rather than its persistence.
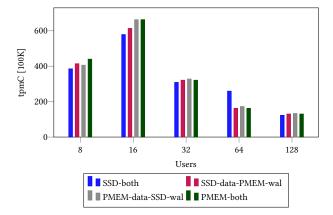
Figure 18: tpmC for TPC-C on SQLServer using PMEM as WAL or data store

7 DISCUSSION

We summarize the main insights from all the previous sections and provide best practices for the different configurations that a DBMS should follow to maximize the PMEM utilization.

(i) **Memory mode does not offer a significant performance advantage**: As we have observed for all databases and benchmarks, for the vast majority of cases, Memory mode with its larger volatile memory capacity does not offer any performance advantage. On the other hand, there are a lot of cases, where performance is slightly worse. This happens due to the conflict misses in the DRAM L4 cache, which also causes additional memory traffic between PMEM and DRAM.

(ii) **Memory mode is useful for queries with sequential accesses and low number of reads/writes to memory**: In the case of queries with sequential accesses, the OS can take advantage of the larger filesystem cache offered by PMEM and together with prefetching these queries can have a significant speedup compared to configurations that do not use PMEM at all. If there is significant memory traffic between the different storage layers (e.g. when the workload performs many writes or the working set is larger than the OS page cache) the Memory mode can experience a small performance overhead.

(iii) **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 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.

(iv) **The lower latency of PMEM in AppDirect mode does not translate to an advantage equal to the hardware characteristics**: From our microbenchmarks, PMEM has 7-8× lower latency from an SSD and in some cases more than 10× higher read and write bandwidth. This difference does not translate to the expected runtime decrease as the I/O path is not fully optimized. The AppDirect mode with fsdax, which is the one recommended by Intel, does not have the advantage of an OS page cache. This feature is largely beneficial for sequential queries

with high selectivity where prefetching is very efficient. It is also useful in the case of secondary indexes, where data from tables is stored in the page cache.

(v) **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 noticed this behaviour in all three databases (in MySQL for TPC-H, when we restricted PostgreSQL to one core and in SQLServer for TPC-C).

(vi) **PMEM requires fine-tuning for write-heavy workloads**: As noticed by previous studies [25, 27] heavy-write workloads and read/write interference decrease the performance of PMEM dramatically. We noticed the same behaviour when running TPC-C. Therefore, we should carefully tune the number of write threads to avoid interference. Especially in the context of databases, several configurations that increase additional writes should be turned off to increase performance (e.g. the double-write buffer)

(vii) **Optimizations made for SSDs/HDDs need to be re-designed when using PMEM**: Many traditional optimizations try to avoid I/O as much as possible due to the latency gap between main memory 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.

(viii) **Log placement in PMEM does not increase performance significantly**: As we have observed across databases, log placement in PMEM does not increase the transaction rate for TPC-C significantly. Placing data in PMEM increases throughout for TPC-C due to the lower latency and higher bandwidth of PMEM compared to SSD/HDD.

8 CONCLUSION

In this paper, we benchmarked PMEM under three different relational engines (PostgreSQL, MySQL and SQLServer) and two popular benchmarks (TPC-H and TPC-C). Our study sheds light on how to efficiently integrate PMEM into the memory hierarchy and how to tune DBMSs to get the best performance out of each configuration. 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 and because the I/O path is not fully optimized in the case of PMEM, SSDs still remain a competitive solution for a number of workloads.

REFERENCES

[1] Accessed 2021. 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.

[2] Accessed 2021. 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.

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

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

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

[6] Accessed 2021. Intel memory latency checker. https://www.intel.com/content/www/us/en/developer/articles/tool/intelr-memory-latency-checker.html.

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

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

[9] Accessed 2021. Intel®Optane©Technology FactSheet. https://www.intel.com/content/dam/www/public/us/en/documents/fact-sheets/nsg-dc-solution-selection-battlecard-factsheet.pdf.

[10] Accessed 2021. Ioping. https://github.com/koct9i/ioping.

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

[12] Accessed 2021. 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.

[13] Accessed 2021. Peloton. https://pelotondb.io/.

[14] Accessed 2021. 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.

[15] Accessed 2021. PGTune. https://pgtune.leopard.in.ua/.

[16] 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* 10, 12 (2017), 1754–1765.

[17] 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.

[18] Joy Arulraj and Andrew Pavlo. 2019. Non-volatile memory database management systems. *Synthesis Lectures on Data Management* 11, 1 (2019), 1–191.

[19] 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.

[20] 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).

[21] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-behind logging. *Proceedings of the VLDB Endowment* 10, 4 (2016), 337–348.

[22] 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.

[23] 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* 8, 5 (2015), 497–508.

[24] Shimin Chen and Qin Jin. 2015. Persistent b+-trees in non-volatile main memory. *Proceedings of the VLDB Endowment* 8, 7 (2015), 786–797.

[25] 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.

[26] 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.

[27] 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).

[28] 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.

[29] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating persistent memory range indexes. *Proceedings of the VLDB Endowment* 13, 4 (2019), 574–587.

[30] 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* 14, 5 (2021), 835–848.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] Tianzheng Wang and Ryan Johnson. 2014. Scalable logging through emerging non-volatile memory. *Proceedings of the VLDB Endowment* 7, 10 (2014), 865–876.

[40] 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.

[41] 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.

[42] 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.

[43] 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.