Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads

Björn Daase*, Lars Jonas Bollmeier*, Lawrence Benson, Tilmann Rabl
{bjoern.daase,lars.bollmeier}@student.hpi.de,{lawrence.benson,tilmann.rabl}@hpi.de
Hasso Plattner Institute, University of Potsdam

Abstract

Modern database systems for online analytical processing (OLAP) typically rely on in-memory processing. Keeping all active data in DRAM severely limits the data capacity and makes larger deployments much more expensive than disk-based alternatives. Byte-addressable persistent memory (PMEM) is an emerging storage technology that bridges the gap between slow-but-cheap SSDs and fast-but-expensive DRAM. Thus, research and industry have identified it as a promising alternative to pure in-memory data warehouses. However, recent work shows that PMEM's performance is strongly dependent on access patterns and does not always yield good results when simply treated like DRAM. To characterize PMEM's behavior in OLAP workloads, we systematically evaluate PMEM on a large, multi-socket server commonly used for OLAP workloads. Our evaluation shows that PMEM can be treated like DRAM for most read access but must be used differently when writing. To support our findings, we run the Star Schema Benchmark on PMEM and DRAM. We show that PMEM is suitable for large, read-heavy OLAP workloads with an average query runtime slowdown of 1.66x compared to DRAM. Following our evaluation, we present 7 best practices on how to maximize PMEM's bandwidth utilization in future system designs.

ACM Reference Format:
Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, Tilmann Rabl. 2021. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21), June 20–25, 2021, Virtual Event, China.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3448016.3457292

1 Introduction

Online analytical processing (OLAP) is one of the crucial applications for many companies to understand the state of their business. Since the amount of data and the need for faster data processing has increased immensely over the last decade, there has also been a shift in technology: OLAP has moved from HDDs to faster SSDs [16] and with databases like SAP HANA [26], even faster in-memory column stores [1, 48] are widely used for OLAP workloads. Many OLAP deployments need to support huge tables at good end-user performance [53], which results in a compromise between fast, but low capacity DRAM and slower, but high-capacity SSDs.

Recently, Intel® Optane™ DC Persistent Memory became commercially available. Optane DIMMs are persistent memory (PMEM) modules that fit into regular memory slots, like DRAM. PMEM promises persistent and byte-addressable storage with performance close to DRAM while providing higher density [27, 30, 51, 54].

Before the general availability of PMEM, related research on data structures and database systems was conducted based on simulations and assumptions [4, 6–9, 22, 38]. Since PMEM is available, research on the actual hardware observed that there are huge performance gaps depending on the type of I/O operation, which instruction is used, which access size is used, and how many threads access PMEM [30, 54]. Thus, it is critical to gain a solid understanding of PMEM's exact performance characteristics for future research.

In this paper, we systematically evaluate Intel's Optane DIMMs as the first commercially available PMEM and provide 7 best practices for its use in OLAP workloads. While previous work provides initial performance numbers for PMEM [51, 54], a more detailed understanding of this new technology is essential to achieve the best performance in various use-cases. With this work, we aim to provide additional insights for PMEM in OLAP workloads. To this end, we build on benchmarks of previous work (e.g., *Access Size* and *Thread Count*) but provide new and additional insights into their impact. We also extend our benchmarks beyond previous work by evaluating the use of PMEM on large-scale systems with multiple NUMA nodes and full system control. This enables us to investigate other OLAP-related parameters, such as the interaction across *NUMA Regions*, as well as the *Thread Assignment* to cores. Where applicable, we contrast PMEM with DRAM to provide insight into varying or similar device characteristics.

In this work, we focus on a PMEM-only design space. While hybrid PMEM-DRAM use is expected in the future and recent work focuses on initial performance numbers [45, 52], we see a need to fully understand PMEM characteristics before adding additional complexity through hybrid approaches.

To apply our findings in an OLAP-workload, we execute the Star Schema Benchmark (SSB) [36] on PMEM. We implement the SSB both in a handcrafted C++ version as well as in the in-memory database Hyrise [24, 25]. This allows us to show the raw impact of PMEM compared to DRAM and its expected impact in full systems, which are not PMEM-aware. Based on our results, we conclude that PMEM is a suitable alternative to expensive and capacity-limited DRAM for OLAP workloads. In summary, we make the following contributions:

(1) We extensively evaluate PMEM on a multi-socket system.
(2) We present 7 best practices to maximize PMEM bandwidth utilization for OLAP workloads on a multi-socket server.

*Both authors contributed equally.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD '21, June 20–25, 2021, Virtual Event, China
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8343-1/21/06…$15.00
https://doi.org/10.1145/3448016.3457292

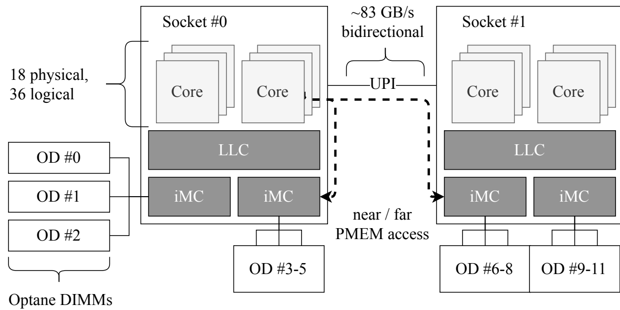

Figure 1: System Architecture.

(3) We run the Star Schema Benchmark on PMEM and DRAM in a handcrafted C++ version and the database Hyrise.

The remainder of this paper is structured as follows. Section 2 provides background on PMEM, OLAP characteristics, and describes our benchmark setup. In Section 3 and Section 4, we evaluate PMEM's sequential read and write bandwidth across several axes. Section 5 focuses on mixed workloads and random access performance. In Section 6, we present an implementation of the Star Schema Benchmark on PMEM. Section 7 condenses our insights into a set of best practices for PMEM bandwidth. We end this paper with related work in Section 8, before concluding in Section 9.

2 Background

In this section, we introduce persistent memory, describe relevant characteristics of OLAP workloads, and provide an overview of the system we perform our evaluation on.

2.1 Persistent Memory

PMEM is generally defined by its two most significant attributes: byte-addressability and persistency. These properties conceptually place it somewhere between persistent block-devices like SSDs or HDDs and volatile, but byte-addressable DRAM. The first and currently only publicly available implementation of PMEM is the Intel® Optane™ DC Persistent Memory. In the following, when speaking about the explicit behavior of PMEM, we refer to the behavior of Optane. Further, as in previous work, we assume that future implementations of PMEM will perform similarly [30, 54].

Intel Optane's integration into a system is very similar to DRAM as shown in Figure 1. It comes in the form factor of a DIMM in the sizes of 128 GB, 256 GB, and 512 GB and is connected to a memory channel. These memory channels are connected directly to the integrated memory controller (iMC) on the CPU. Each iMC serves up to three memory channels and every CPU-socket contains two iMC's, resulting in up to six memory channels connected to PMEM for each socket. The iMC and PMEM modules communicate via the DDR-T protocol, an extension of the DDR4 protocol, allowing asynchronous command- and data-timing.

Read and write accesses are inserted into a corresponding read or write pending queue (RPQ or WPQ) inside the iMC. In a system with multiple sockets, I/O operations can be inserted from one socket into the WPQs and RPQs of another socket by transmitting them over the Intel Ultra Path Interconnect (UPI), effectively allowing one socket to utilize PMEM of another socket. After insertion into the WPQ, write operations are guaranteed to be processed and thus

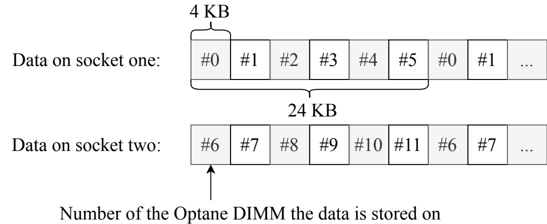

Figure 2: DIMM Interleaving. Data is interleaved at 4 KB across 6 DIMMs forming one filesystem mount point.

persisted by PMEM, even in case of a power loss. As the CPU has a cache line size of 64 Byte and Optane memory has an internal granularity of 256 Byte, multiple accesses to consecutive addresses result in a single internal read for PMEM, loading 256 Byte into a buffer and resolving following accesses from this buffer.

PMEM can be used in two different modes: *Memory Mode* and *App Direct Mode*. *Memory Mode* adds PMEM to the available DRAM as addressable main memory and DRAM becomes an inaccessible "L4"-cache in front of PMEM. Since it is not guaranteed that dirty cache lines in DRAM are persisted in case of power loss, this mode does not guarantee persistency. *App Direct Mode* makes PMEM explicitly addressable outside of DRAM and allows the application to map PMEM into its virtual address space. In App Direct, PMEM profits from *direct access*, allowing PMEM to be accessed like a filesystem (*fsdax*) or character-device (*devdax*), without filesystem-overhead or the use of the OS page cache. Both modes can be used in parallel, where parts of PMEM are in Memory Mode and extend DRAM, while other parts of PMEM in App Direct Mode act as a persistent storage device. As the App Direct Mode provides more control over PMEM, most research focuses on it instead of the Memory Mode [37, 45, 54]. However, existing applications need to be re-written to use PMEM with it, whereas Memory Mode transparently gives applications more DRAM without any changes. Data stored on PMEM is interleaved across the DIMMs of a socket in steps of 4 KB as shown in Figure 2. Following this, in a system with six PMEM DIMMs per socket, data larger than 20 KB will be striped across all DIMMs. This interleaving allows data to be accessed in parallel on multiple DIMMs.

While integrated into the system similarly to DRAM, research on Intel Optane conducted by Yang et al. [54] shows that PMEM behaves a lot more nuanced and has characteristics that differ significantly from the idea of "slow, persistent DRAM" as assumed by earlier research [5, 37, 53, 55]. Reading from PMEM yields approx. a third and writing a seventh of the bandwidth of DRAM, but is still at least an order of magnitude higher than on SSD. Like SSDs, PMEM wears out over time and offers much higher capacities than DRAM, which is typically used in smaller 32 GB DIMMs on servers [45].

2.2 Online Analytical Processing

Today's data workloads on an industrial scale are mainly placed in two categories: Online Transaction Processing (OLTP) and Online Analytical Processing (OLAP). OLTP requires low latency reading and writing while primarily accessing only a few records per transaction [23]. OLAP is, aside from the ingestion of the data, generally read-only. The queries are more complex and often require scanning and joining multiple tables, resulting in large intermediate results and a need for higher memory capacity [46, 49]. Following this,

both the huge table sizes as well as maintaining a good end-user performance were identified as core challenges for OLAP systems [53]. Recent work on in-memory OLAP systems shows that OLAP on high-performance column stores can be orders of magnitude faster than on traditional row stores [2, 48]. However, even in these optimized column stores performance is lost due to long-latency data cache misses, which can directly be connected to the bandwidth saturation of the underlying storage medium [46]. Due to this, in this paper, we mainly focus on increasing the bandwidth of PMEM to maximize its performance for OLAP.

2.3 Experimental Setup

We perform our experiments on a dual-socket server with an architecture as shown in Figure 1. This design is very similar to the benchmark systems used in previous research [45, 54]. The CPU-sockets are Intel® Xeon® Gold 5220S with 2.70 GHz base frequency and 18 CPU cores each. Two logical cores are available per physical core due to hyperthreading. The logical cores share the physical core's L1 and L2 caches. Hyperthreading gives us 36 (18 x 2) logical cores per socket and 72 (36 x 2) logical cores in total.

There are two iMC's per socket with three memory channels, each connected to one PMEM DIMM, resulting in six (3 channels x 2 iMC's) PMEM DIMMs per socket and twelve (6 DIMMs x 2 sockets) DIMMs for the whole system. We use 128 GB Intel Optane DIMMs, resulting in a total PMEM capacity of 1.5 TB (128 GB x 6 DIMMs x 2 sockets). Besides PMEM, a 16 GB Samsung DDR4 DIMM is connected to every memory channel, totaling 186 GB (16 GB x 6 channels x 2 sockets) of DRAM.

As our server contains four NUMA nodes with different memory access distances to each other, we distinguish between the terms *NUMA region* and *NUMA node* for our benchmarks. Each NUMA node consists of 9 physical cores and one iMC connected to three PMEM DIMMs as well as three DRAM DIMMs. A NUMA region (i.e., one socket) consists of two close NUMA nodes, where the access distance within the region (intra-region) is nearly identical. The access across NUMA regions (inter-region) incurs higher overhead via the UPI. Consequently, we also distinguish between *near* and *far* PMEM, where near PMEM is any DIMM that is within the same NUMA region and far PMEM is any DIMM in the other region.

Our machine runs Ubuntu 18.04 LTS (kernel 4.15). We access PMEM in App Direct Mode via *devdax*. Our evaluation shows that the behavior and trends on *devdax* and *fsdax* are identical but *devdax* consistently achieves a 5–10% higher bandwidth in all experiments. This performance difference is attributed to the impact of page faults in fsdax. As `mmap` returns zeroed memory by default, any initial page fault in fsdax triggers the kernel to zero the requested page. Once data is present (i.e., the file was written to already or devdax is used), the memory does not need to be zeroed. We ran an experiment to verify this. The performance of devdax and fsdax is identical if all pages were pre-faulted before the measured access. We note, however, that a 2MB page fault (default in PMEM if configured with `ndctl`) takes ~0.5 ms, thus, pre-faulting 1 GB of PMEM takes at least 0.25 seconds. As we investigate large-scale OLAP workloads, we assume a system to have full hardware-control and to manage memory itself in *devdax*, which achieves higher bandwidths due to avoiding page faults.

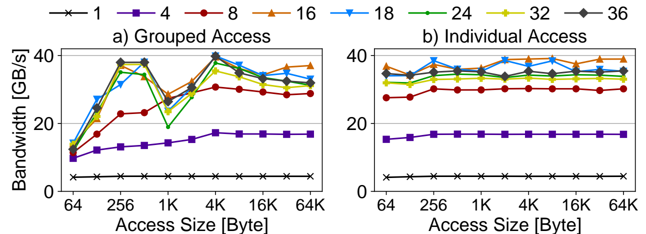

Figure 3: Read Bandwidth dependent on *Access Size* and *Thread Count*. Bandwidth peaks at 4 KB and high threads counts for grouped reads and is constant for individual sequential reads.

PMEM data is interleaved as described above and shown in Figure 2, resulting in a large PMEM character device for each socket. As a programming interface, we use the Persistent Memory Development Kit's `libpmem` library [21] without any of its high-level abstractions. All benchmarks are written in C/C++ and assembly to work as close to the hardware as possible. For additional hardware insights during the benchmarks, we use the Intel® VTune™ Profiler [20], which provides detailed information about various system components. Our code is open-source and available on GitHub[1].

3 Sequential Read Performance

Large full-table scans are the most common read pattern in OLAP workloads [14], making them highly dependent on sequential read performance. In this section, we evaluate the maximum read bandwidth of PMEM. We evaluate the impact of the *Access Size*, the *Thread Count*, *Thread Pinning*, *NUMA Effects*, and *Multi-Socket Characteristics*. We use the `vmovntdqa` instruction to read data into the CPU registers and exploit the AVX-512 capability of modern servers, as done in previous work [54]. Unless stated otherwise, all read benchmarks are performed on 70 GB raw data located in PMEM.

3.1 Read Access Size

We first evaluate the access size to determine how many bytes should be read from PMEM to achieve the highest bandwidth.

Workload. We define the access size as the number of consecutive bytes accessed by one thread. The benchmark is run for 1 to 36 threads with access sizes from 64 Byte to 64 KB. We use cores and memory of only one NUMA region, to which threads are pinned using `numactl`. As each NUMA region has 18 physical cores, we use hyperthreading only for thread counts above 18. In the *Grouped Access*, reads are interleaved across all threads, i.e., if thread 1 reads bytes 0–255, thread 2 reads from byte 256, which leads to one global sequential read. In *Individual Access*, each thread has its own disjoint memory region, i.e., thread 1 reads GB 0 to 1, thread 2 reads GB 1 to 2, which leads to #threads sequential reads.

Results. Figure 3 shows that the access size is relevant when all threads read consecutively from the same location (a) while impacting the bandwidth of individual access only marginally (b). The maximum bandwidth for grouped access ranges from 12 to 40 GB/s for 36 threads, whereas the maximum individual spans only 3 GB. It also shows that in both grouped and individual access, a peak bandwidth of ~40 GB/s is achieved. However, grouped access only achieves this peak for certain configurations and shows large variation, while individual reads constantly perform close to the

[1]https://github.com/hpides/pmem-olap

peak for high thread counts. In the grouped reads, 4 KB access results in a global maximum for various thread counts, which is not observable for individual reads. The bandwidth stays constant for access sizes larger than 64 KB.

Discussion. The reason for the impact of the access size on grouped reads is threefold. First, for small access sizes (< 256 Byte), nearly all threads operate on the same DIMM, thus not profiting from the parallelism of interleaved memory. This shows that an even distribution of threads to DIMMs is essential to achieve high PMEM bandwidths. This clearly distinguishes PMEM from DRAM. Second, 4 KB access aligns perfectly to the interleaved DIMM boundaries causing an ideal thread-to-DIMM distribution and better prefetching. Third, the L2 hardware prefetcher [17] performs poorly for 1 and 2 KB access. When running the same benchmark with the L2 prefetcher disabled, we do not observe the drop at 1 and 2K access but a more constant bandwidth for access larger than 256 Byte. However, we also observe this poor prefetching behavior in DRAM, thus, this is not a PMEM-specific anomaly but indicates that this access pattern is sub-optimal for Intel's L2 hardware prefetcher. We do not recommend disabling the prefetcher, as it is a system-wide setting that might cause unexpected performance degradation.

On the other hand, individual reads profit significantly from Optane-internal prefetching as each core accesses consecutive data in a DIMM. When deactivating the L2 prefetcher, we observe no impact regarding the access size. Thus, for individual reads, the access size is not as relevant. Accesses smaller than Optane's 256 Byte granularity still achieve 30+ GB/s, as the Optane controller can immediately answer consecutive requests from the loaded 256 Byte cache line without causing read amplification.

Insight #1: Read data from individual memory regions or in consecutive 4 KB chunks to benefit from prefetching and an even thread-to-DIMM distribution.

3.2 Read Thread Count

We now evaluate the optimal read thread count for grouped and individual reads, with the same workload as in the previous section.

Results. Figure 3 shows that the highest bandwidth (~40 GB/s) is achieved with 16 and 18 threads in both grouped and individual reads. Generally, all thread counts above 16 achieve a very high bandwidth (except for the prefetcher dip in Figure 3a). However, adding hyperthreads does not improve the bandwidth. Access with 36 threads constitutes a small outlier, as the other thread counts with hyperthreading perform worse than 18 threads and 36 threads achieve peak performance for certain access sizes. Yet, access with as few as 8 threads achieves nearly as much bandwidth utilization as 36 threads (~15% difference).

Discussion. The main insight from this benchmark is that hyperthreading is generally not needed to saturate PMEM's bandwidth. In certain cases, it even performs worse than using fewer threads. In light of the high impact of prefetching on the access size, we also investigate its impact on the thread count. When disabling the L2 prefetcher, lower thread counts (<8) perform worse but higher thread counts (>18) perform better. This indicates that the prefetcher does not work efficiently for high thread counts reading from various locations. Hyperthreading especially benefits from a disabled prefetcher, as hyperthreads share the L2 cache on one core, and thus the prefetcher constantly pollutes it. With a disabled prefetcher, 36 threads also achieve the highest bandwidth of ~40 GB/s (like 16/18 threads). However, as prefetching is most likely activated, hyperthreads will not generally increase the bandwidth.

Concluding, more cores lead to higher bandwidth and all physical cores are needed to saturate it. If hyperthreading is used, again, more threads achieve higher bandwidth. While the observation of *more threads equals more bandwidth* seems trivial, we observe a different behavior in write performance (see Section 4). Thus, we explicitly point out the performance implications of the thread count at this point. We evaluate the impact of the thread-ratio on mixed read-write workloads in Section 5.

Insight #2: Use all available cores for maximum read bandwidth and avoid hyperthreaded reads.

3.3 Read Thread Pinning

When multiple threads access PMEM, an efficient thread-to-core assignment (or *thread pinning*) becomes an important factor when aiming for higher bandwidths. We explore three basic approaches to assign (*pin*) threads to the available cores.

The threads are either not pinned at all (*None)*, pinned to the NUMA region near the PMEM they are accessing (*NUMA Region)*, or to individual cores in that NUMA region (*Cores)*. For *None*, the scheduler can freely assign threads to cores across all sockets and for *NUMA Region*, the scheduler assigns only to cores in that region. While the third approach is relatively independent of the scheduler, the first and second approaches rely heavily on its behavior. Relying on the scheduler has the advantage of adaptability to changing workloads, given that the scheduler is aware of PMEM characteristics. However, it also removes the ability to fine-tune the system in favor of a known workload to achieve even higher bandwidths.

Workload. In our setup, the threads are either distributed across all 36 physical cores (*None)*, pinned to the 18 cores of one NUMA region (*NUMA Region)* or pinned to one individual core (*Cores)*. In the *Cores* run, with fewer than 18 threads, we fill up the physical cores before placing threads on the logical sibling cores. All pinning variants use individual access with a fixed size of 4 KB.

Results. In Figure 4, we observe that pinning to individual cores slightly outperforms only pinning to the NUMA region on that socket when using more than 18 threads. Pinning threads to individual cores gives the highest bandwidth of ~41 GB/s with 18 threads, while only pinning them to a NUMA region results in ~40 GB/s for 18 and 36 threads. No pinning at all results in drastically worse performance, peaking at only ~9 GB/s for 8 and 36 threads.

Discussion. The higher performance with *Cores*-pinning over *NUMA Region* is explained by avoiding the scheduling overhead when having to schedule more than 18 threads onto 18 physical available cores. This is not needed when directly pinning the threads to cores. Using 18 threads or fewer does not require scheduling of multiple threads to a single core and therefore results in exactly the same bandwidth for pinning to NUMA regions or individual cores. The general bandwidth drop after 18 threads is again caused by shared L2 pollution of hyperthreads, regardless of the pinning strategy. The low bandwidth for no pinning is caused by the scheduler placing some of the threads on the far socket regarding the PMEM DIMMs we are reading from. This results in threads having

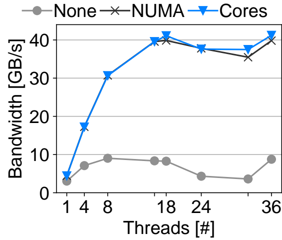

Figure 4: Read Bandwidth dependent on *Pinning*.

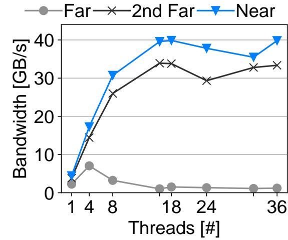

Figure 5: Read *NUMA Effects*.

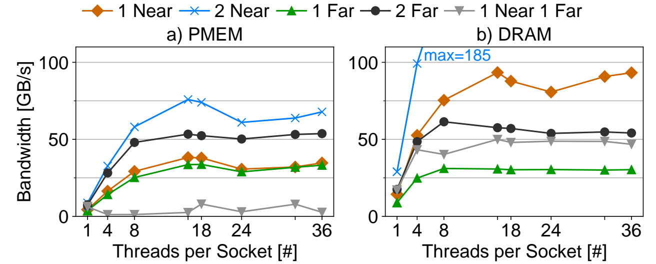

Figure 6: Read from *Multiple Sockets* on PMEM/DRAM.

to access PMEM connected to the other CPU socket. As the performance is significantly lower in this case, we evaluate the bandwidth impact of accessing far and near memory in detail in Section 3.4.

Insight #3: Pin threads to avoid far-memory access.

3.4 Read NUMA Effects

The low bandwidth of the *None*-pinning in the previous section indicates that there are NUMA effects when working with PMEM on multiple sockets. In this section, we examine them in detail.

Workload. The threads are pinned to the NUMA region on the first socket. They either read only from the PMEM connected to the same socket (*Near*), or from PMEM connected to the second socket (*Far*). We also measure a second run for far PMEM access (*2nd Far*). All access is individual with a size of 4 KB.

Results. In Figure 5, we measure the bandwidth when accessing PMEM across sockets. It shows that accessing PMEM connected to the near socket has a peak performance of ~40 GB/s. The first access on the far PMEM results in a very low bandwidth of ~8 GB/s, being worse by a factor of 5. The optimal thread count for far PMEM access also shifts from 18 threads to only 4 threads. This is the same effect we see in Figure 4, when not pinning threads at all. An interesting effect, which we observe, is the warm-up behavior when accessing far PMEM. Starting from the second run, the performance nearly matches the performance of accessing the near PMEM (~40 GB/s when accessing near PMEM vs. ~33 GB/s when accessing far PMEM in the second and consecutive runs).

Discussion. Figure 1 shows the architecture of our and other commonly used systems [45, 54]. Especially, it shows that we have two independent sets of PMEM DIMMs, which are not interleaved with each other and have to be accessed individually. Intel® Xeon® processors use a coherency protocol to manage the address space of multiple sockets that requires address space mappings. When memory is accessed by cores from another socket, mapping entries need to be reassigned [18]. If access to the same memory regions is constantly switching between sockets, constant remapping is required. The warm-up behavior results from this reassignment and strongly reduces the bandwidth. This can be verified by the fact that reading with a single thread on far memory before reading with multiple threads on far memory eliminates the warm-up behavior, which implies that this is not a core- but a NUMA region-related anomaly.

The lower bandwidth of the *2nd Far* is caused by a high cross-socket UPI utilization, which we investigate further in the following section. Considering these two effects, we recommend accessing only near PMEM, or if this is not possible, to change the assignment of address spaces to NUMA regions as rarely as possible. If this is not considered, bandwidth will be poor regardless of the access size or thread count. These effects are also observable in DRAM-based systems, albeit slightly weaker [41, 42].

Insight #4: Threads should only read data on their near socket PMEM. If this is not possible, the assignment of address spaces to NUMA regions should change as rarely as possible.

3.5 Read from Multiple Sockets

After investigating the performance of near and far PMEM from a single socket, we now evaluate the same access from multiple sockets, i.e., the interaction of multiple threads on multiple sockets concurrently reading from near and far PMEM. We also compare this to the performance of multi-socket DRAM reads.

Workload. When reading in parallel across two sockets, five combinations are possible: *i)* Reading from one socket on its near PMEM, *ii)* reading from one socket on its far PMEM, *iii)* reading from two sockets on their near PMEM respectively, *iv)* reading from two sockets on their far PMEM respectively, and *v)* reading from one socket on its near PMEM and reading from the other socket on its far PMEM (i.e., both read the same memory). The threads are pinned to NUMA regions and the access is individual 4KB.

Results. Figure 6 shows the accumulated bandwidth across all threads in these scenarios when reading across two sockets for PMEM and DRAM. The results for reading only on the near (*i*) or far (*ii*) PMEM on one socket match the results from Figure 5, peaking at ~40 GB/s and ~33 GB/s, respectively. We also see peak bandwidth for near DRAM (~100 GB/s) but a stark difference in far access, achieving only ~33 GB/s. Using 18 threads on each socket (36 total) accessing their near memory (*iii*) results in a linear speedup with the number of sockets, resulting in a bandwidth of ~80 GB/s (PMEM) and 185 GB/s (DRAM). This setting also does not use the UPI and thus, does not limit other inter-socket communication. Accessing the far memory with 36 total threads (*iv*) peaks at only ~50 GB/s (PMEM) and ~60 GB/s (DRAM). This is significantly lower than the performance of only near access on both sockets. Accessing the same PMEM from different sockets (*v*) yields a very low bandwidth on PMEM, while nearly achieving the performance of only far access on both sockets for DRAM.

Discussion. While reading from near memory performs best and scales linearly, reading from far memory does not. Since every read request and answer of far access to memory has to be conducted via the UPI, it becomes the bottleneck when both sockets access their far memory, resulting in a flattened curve compared to reading from near sockets. The UPI achieves ~40 GB/s per direction but about 25% of this is required for metadata transfer, i.e., allowing for ~30 GB/s data per direction. We verify this by running the

benchmark in VTune, showing an average UPI utilization of 90+%, including metadata. Thus, both DRAM and PMEM are limited by the UPI but the impact on DRAM is significantly higher due to otherwise higher absolute bandwidth utilization.

Reading the same memory from both sockets achieves very low bandwidth utilization compared to the respective *1 Near* configuration. This is caused by the cache coherency mapping across sockets (cf. Sec. 3.4), which incurs higher UPI utilization and writes to the memory. This is especially harmful in PMEM, as mixed read/write access heavily impacts the overall bandwidth.

A second reason for the reduced performance is the pollution of the RPQs and WPQs. On a single socket, threads insert their requests nearly sequentially into the queues. But with a second socket, these sequences are interrupted by requests inserted with latency by another socket into the same iMC. This causes read amplification of the 256 Byte PMEM cache line and thus reduces performance. Following this, adding more threads from another socket does not increase PMEM performance.

In conclusion, fully utilizing PMEM from multiple sockets requires an efficient splitting of the data across PMEM with the goal that threads on both sockets should not have to read from far PMEM. However, as providing detailed partitioning concepts is beyond the scope of this work, we refer the reader to existing research [3, 13, 31, 43, 44] and plan to investigate such PMEM-aware partitioning schemes in the future.

Insight #5: If possible, stripe data into independent and evenly distributed data sets across the PMEM of all sockets and ensure that sockets read only from near PMEM.

4 Sequential Write Performance

In order to analyze large volumes of data, it first needs to be ingested into the system that analyzes it. Thus, an important feature of data warehouses is an efficient data import [14]. Additionally, large intermediate results are often produced in complex analytical queries, e.g., when data is sorted or joins are performed. Again, these intermediate writes should be as efficient as possible to achieve higher throughput within OLAP systems. Both data ingestion, as well as intermediate result creation, are often expressed in sequential writes. Therefore, we investigate the performance of sequential writes in OLAP systems in this section.

We perform our analysis analogously to that of sequential reads in Section 3, covering the same dimensions *Access Size*, *Thread Count*, *Thread Pinning*, *NUMA Effects*, and *Multi-Socket Characteristics*. For all our stores, we use the non-temporal `vmovntdq` instruction, which exploits AVX-512 of modern systems, as done in previous work [54]. All writes are followed by an `sfence` instruction, which is required to guarantee persistence. Unless stated otherwise, all write benchmarks write 70 GB of raw data to PMEM.

4.1 Write Access Size

A crucial factor for the overall write bandwidth to PMEM is the size in which data is written. To better understand the impact this size has, we evaluate it for various thread counts.

Workload. We run the benchmark for 1 to 36 threads with access sizes from 64 Byte to 32 MB. The access size is defined as the number of consecutive bytes that a thread writes in one operation.

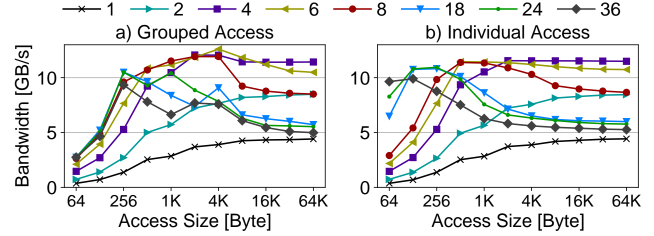

Figure 7: Write Bandwidth dependent on *Access Size* and *Thread Count*. Bandwidth is heavily impacted by the combination of access size and thread count.

In the *Grouped Access*, writes are interleaved across all threads, i.e., if thread 1 writes bytes 0–255, thread 2 writes from byte 256, which leads to one global sequential write. In *Individual Access*, each thread has its own disjoint memory region, i.e., thread 1 writes GB 0 to 1, thread 2 writes GB 1 to 2, which leads to #threads sequential writes. All threads are pinned to a single socket via `numactl`.

Results. We show the results in Figure 7. We observe that the access size has a significant impact on the bandwidth for both grouped (a) and individual (b) access. However, grouped writes behave non-monotonically in that they increase, decrease, and increase again for certain access sizes, which is not observable as strongly for individual writes. Overall, two trends are visible, *i)* bandwidth increases significantly from 64 Byte to 256 Byte or 4 KB and *ii)* bandwidth decreases after 256 Byte for thread counts above 18. Thus, we observe the counterintuitive trend to achieve the highest bandwidth: *the higher the thread count, the lower the access size must be.* Unlike read operations, this indicates a strong entanglement between the access size and the number of threads used. In this section, we focus on the access size alone that yields the highest performance and we investigate the impact of the thread count in the following section. We also distinguish between grouped and individual writes, as they differ significantly for small access sizes, e.g., 2.6 GB/s compared to 9.6 GB/s with 64 Byte and 36 threads. Writes larger than 1 KB achieve the highest overall bandwidth with a global maximum of 12.6 GB/s for grouped 4 KB access.

Discussion. Similar to sequential read access, there are two peaks at around 256 Byte and 4 KB. The 256 Byte peak indicates a beneficial write-size equivalent to Optane's internal access granularity. The 4 KB peak, on the other hand, indicates a beneficial write-size equivalent to the DIMM-interleaving size.

To understand this behavior in more detail, we look at the underlying components in the Optane DIMMs. Each DIMM contains an internal write-combining buffer that groups neighboring writes before flushing them to the underlying storage medium. A write is not necessarily flushed directly but held in the buffer, which increases the number of write requests the DIMM can process as fewer expensive PMEM flushes are needed. This buffer is required, as the CPU transfers only 64 Byte cache lines but Optane works with 256 Byte "cache lines". Thus, for writes smaller than 256 Byte, Optane must perform a read-modify-write operation. 256 Byte writes do not cause any write amplification and can be written directly by the controller and they are small enough to not be interrupted by other threads. As each DIMM contains its own write-combining buffer, aligned 4 KB writes target exactly one DIMM and lead to an ideal thread-to-DIMM distribution with minimal interruption by other threads.

The most significant difference between grouped and individual writes is the stark contrast for <128 Byte writes at high thread counts and the non-monotonic behavior of grouped writes. If each thread receives its own memory region, the other threads do not interfere with the write-combining logic as strongly. This indicates that the buffer cannot perform write-combining efficiently across threads. Secondly, similar to grouped sequential reads, smaller access sizes all target the same DIMM. This pattern does not fully utilize the parallelism of all available DIMMs with a poor thread-to-DIMM distribution. For individual writes, the threads naturally distribute across the DIMMs, leading to a higher DIMM-parallelism.

In general, PMEM writes behave significantly differently than DRAM writes. In DRAM, more threads result in higher bandwidth and we do not observe any decrease in performance for larger access sizes. This must be taken into account when designing write-heavy, memory-bound systems, as the favorable access sizes known from DRAM might have a negative impact on the overall bandwidth utilization in PMEM.

This evaluation provides two main insights: *i)* workloads requiring many small writes, e.g., appending to a log file, should be performed on individual memory locations, e.g., one log per worker and *ii)* grouped writes should be either 4 KB or 256 Byte operations. As we are investigating the maximum performance for a given access size at this point, we recommend using 4 KB, as this performs best for both grouped and individual writes. However, as shown above, the performance is tightly coupled to the access size in combination with the thread count. Thus, we evaluate the impact of the thread count in the following section.

Insight #6: Write data in 4 KB chunks to achieve the highest bandwidth or in 256 Byte chunks if smaller consecutive writes are necessary.

4.2 Write Thread Count

Having determined a reliable access size, we proceed to evaluate the optimal thread count for writing to PMEM.

Workload. We use the identical workload as in the previous section with 1 to 36 threads on one socket.

Results. Figure 7a shows the highest bandwidth of ~12.5 GB/s for 4, 6 and 8 threads at 4 KB. However, only the 4- and 6-thread configurations achieve ~12 GB/s with an increasing access size while the 8-thread configuration drops to ~8 GB/s. A similar trend is observable for the individual writes. A second peak is visible around 256 Byte, where all thread counts above 18 achieve ~10 GB/s. However, for access sizes larger than 256 Byte, the high thread count performance decreases significantly, stabilizing at around 5–6 GB/s. Generally, for thread counts up to 4 (and 6), larger access sizes result in higher bandwidth but for higher thread counts the performance drops after a peak. This is visible for larger writes, where 2 threads achieve the same bandwidth as 8 threads.

To further investigate this behavior, we provide a combined measurement of access size and thread count in Figure 8. It depicts a boomerang-shaped peak-bandwidth pattern along top left – bottom left – bottom right with a bandwidth above 10 GB/s. There are two major observations to be made in this figure. First, the bandwidth does not drop when increasing the access size but keeping the number of threads constant at around 4 to 8. Second, the performance

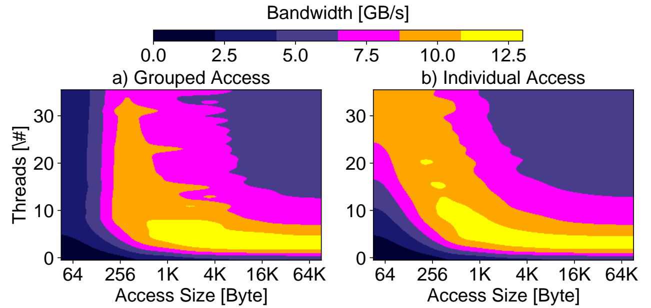

Figure 8: Write Bandwidth dependent on *Access Size* and *Thread Count*. Write bandwidth peaks at 1–4 KB access size and 4–6 threads. Increasing only one of them shows a slight decrease, increasing both at the same time significantly decreases bandwidth.

does not drop significantly when increasing the threads but keeping the access size constant between 256 Byte and 1 KB.

Discussion. Our measurements show that write bandwidth displays a considerable variation in performance. Write bandwidth is tightly coupled to two parameters, i.e., the access size and thread count. This differs significantly from the equivalent read bandwidth results, where an increase in threads or access size leads to an increase in bandwidth. A first major insight is that 4 threads are sufficient to fully saturate the PMEM bandwidth. Also, while 4, 6, and 8 write threads all reach a maximum performance at 4 KB, only 4 and 6 threads can maintain this bandwidth for larger access sizes. Compared to reads, we observe far less variance in the bandwidth, as writes are not affected by caching and prefetching, which might lead to unexpected performance as in reads (e.g., bad 1 KB access).

Two further related insights are that *i)* more threads harm the overall bandwidth of the system for large accesses and *ii)* scaling both the access size and thread count reduces the bandwidth while scaling only one of them can improve it. This behavior contrasts that of DRAM and leads to counterintuitive access patterns for PMEM users.

The effect that the bandwidth degrades beyond 256 Byte access size in certain thread configurations has two reasons. The first of which is contention at the iMC-level. As write operations are masked by the iMC, the application can issue a significantly higher number of requests than the underlying storage medium supports. Additional threads will in turn issue more requests. If the storage cannot complete the write operations as fast as they are received, this leads to full WPQs in the iMC and, thus, slower write calls.

The second and more relevant cause for a decreased bandwidth with large access sizes and higher thread counts is the internal write-combining buffer. As an application can only write a maximum of 64 Byte in a single request (CPU cache line size), the buffer receives many requests to write the given access size, which needs to be held in the buffer for efficient flushing. However, a higher number of threads issue many requests spread across the memory range, thus requiring the buffer to flush more often as it cannot hold all data. This leads to write amplification and decreased bandwidth.

Regarding the thread count, PMEM writes behave significantly different than DRAM writes. In PMEM, adding threads beyond 8 harms the bandwidth, while DRAM benefits from more threads and even scales nearly linearly when using hyperthreading. Therefore, we advise against scaling up both the number of threads and the

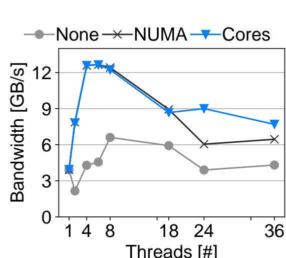

Figure 9: Write Bandwidth dependent on the *Pinning*.

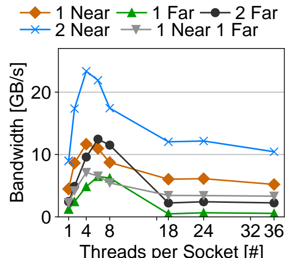

Figure 10: Writing data to *Multiple Sockets*.

access size. If the access is strictly sequential for all threads (e.g., appending to a log) and writes are small (< 256 Byte), adding threads will not harm the performance as severely. However, for larger access sizes (> 1 KB) an increase in threads causes stronger write amplification and reduced performance.

Insight #7: Use 4 – 6 threads to write to PMEM in large blocks or keep the access small when scaling the number of threads.

4.3 Write Thread Pinning

We now explore the effect that thread pinning has on the write bandwidth of PMEM.

Workload. The threads are either not pinned at all (*None*), pinned to the NUMA region near the PMEM they are accessing (*NUMA Region*), or to explicit cores in that NUMA region (*Cores*). All pinning variants use individual access with a size of 4 KB.

Results. In Figure 9, we observe the same pattern as for reading (see Section 3.4). No pinning performs worse than pinning and pinning to individual cores outperforms pinning to NUMA regions when using more than 18 threads. During writing, no pinning peaks at ~7 GB/s while pinning to individual cores peaks at ~13 GB/s. We also see that no pinning has a weaker relative impact on the write bandwidth than it does on reading. No pinning is 2x worse for writing, while we see that no pinning is 4x worse for reading. We also observe the same effect as in Figure 7, where the bandwidth drops for a 4 KB access after 8 threads.

Discussion. The increased bandwidth of explicit over NUMA-region pinning has the same reason as for reading. The scheduler must place more hyperthreads than available cores, causing scheduling overhead and more frequent thread-to-core changes. As each NUMA region contains two NUMA nodes, intra-region placements might still cause inter-node assignments, which write data via different iMCs. Thus, the write-combining buffer cannot combine writes as efficiently, causing higher write amplification and reduced bandwidths. No pinning again shows a very unfavorable performance. As threads can be placed on both sockets, NUMA impacts overall bandwidth. To further investigate this, we evaluate the impact of writing to different NUMA regions in the following section.

If you need more than 18 threads (e.g., when using multiple workers) and exceed the NUMA regions for writing, you should always pin the threads to individual cores. Furthermore, when read threads are pinned to individual cores one should also pin write threads to avoid cache conflicts or bad placements.

Insight #8: Pin write-threads to individual cores if you have full system control. Otherwise, pin them to NUMA regions.

4.4 Write NUMA Effects

Section 4.3 shows that writing is also impacted by far PMEM access, resulting in low bandwidth. Therefore, we evaluate how to efficiently write in a multi-socket environment.

Workload. The 36 threads (one per physical core) are pinned to the NUMA regions of their respective socket. We run two experiments, one where all threads only write to their near PMEM and one where all threads only write to their far PMEM. The access size is fixed to 4 KB.

Results. The two single-socket configurations in Figure 10 (i.e., *1 Far* and *1 Near*) show the write bandwidth dependent on the number of threads. They show that writing to only near PMEM has a peak performance of ~12.5 GB/s (4 threads) but only ~7 GB/s (with 8 threads) when accessing PMEM connected to the far socket. Additionally, we observe that at least 6 threads are needed to maximize the bandwidth when accessing the far memory compared to 4 in near memory accesses. Unlike reading, we do not observe any warm-up effect when writing.

Discussion. The cause for the bandwidth drop when writing to far PMEM is twofold, *i)* higher latency for individual writes via the UPI and *ii)* read-modify-write behavior instead of write-only. If a write accesses far PMEM, all data needs to pass through the UPI, the latency for the blocking write operation increases and, thus, the bandwidth is reduced. Generally, the UPI utilization is very low when writing, as far writes do not achieve high absolute bandwidths. However, this is not the only cause for reduced bandwidth. We observe data being read from the PMEM that should only be written via a non-temporal store. In our case, some `ntstore` operations behave like a cache line read-modify-write (or `clwb`) operation. Intel's Architecture Guide [19] states that an `ntstore` only provides a non-temporal "hint" and is not necessarily executed as such. This read-modify-write pattern requires more bandwidth and is essentially a mixed read-write workload, thus reducing the available write bandwidth. In this benchmark, we observed a write amplification of up to 10x, e.g., ~500 MB/s actual data with 18 far threads but an internal write bandwidth consumption of 5 GB/s. To further understand these effects, we investigate multi-socket writes in Section 4.5 and mixed read-write workloads in Section 5.1.

Insight #9: Threads should only write data to their near PMEM.

4.5 Writing to Multiple Sockets

Section 4.4 shows that writing to far memory results in low bandwidth. Therefore we evaluate how to properly utilize the multi-socket characteristics of our server for writing data.

Workload. To optimize our workload for the dual-socket characteristics, we use the same combinations as in Section 3.5. When writing in parallel across two sockets these are, *i)* writing from one socket to its near PMEM, *ii)* writing from one socket to its far PMEM, *iii)* writing from one socket to its near PMEM and writing from the other socket on the same PMEM (its far PMEM), *iv)* writing from two sockets to their respective near PMEM, and *v)* writing from two sockets to their respective far PMEM. The configurations that write from both sockets in parallel use up to 72 threads, while the other ones use a maximum of 36 threads. Threads are pinned to the NUMA region of their respective socket and the access size is fixed to 4 KB.

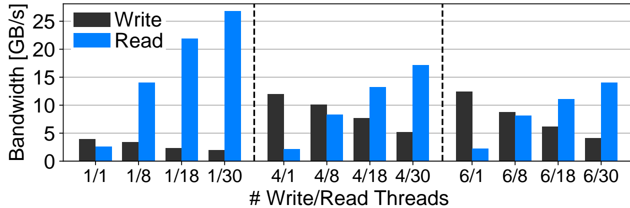

Figure 11: Mixed Workload Performance.

Results. Figure 10 shows that accessing near PMEM *(i)* achieves the highest bandwidth and doubles when using both sockets *(iv)*. The same 2x speed-up holds, though significantly lower, for writing to the respective far PMEM *(v)*, peaking at ~13 GB/s for 8 threads per socket. When both sockets write to the same PMEM *(iii)*, the performance is low as in the equivalent reading operations (see Section 3.5). This peaks at ~8 GB/s, again showing worse results than using only near threads.

Discussion. Accessing far memory, regardless of whether from one or both sockets, achieves at most 50% of the bandwidth compared to their near PMEM access. As discussed in the previous section, higher latency and read-modify-write behavior is the main cause for the poor far memory write performance. The more threads write to far memory, the bigger is the performance drop due to write amplification. Also, writing to far memory that is also being written to from the near socket, reduces the bandwidth by up to 25%. Writing data in this pattern should be avoided whenever possible.

Overall, writing across NUMA regions performs considerably worse than reading across them. We recommend writing data only to near memory to avoid significant performance hits. Similar to reading, systems should use efficient partitioning schemes, which take multi-socket PMEM into account (see Section 3.5). This is especially relevant for the few write-intensive workloads in OLAP, e.g., data import.

Insight #10: Avoid contending cross-socket writes.

5 Mixed and Random Access

Although large sequential read operations (e.g., table scans) are the predominant mode of data access in OLAP workloads, random access and mixed read-writes need to be supported efficiently to allow for parallel aggregations and joins, as well as multi-user systems. We evaluate mixed read-write workloads in Section 5.1 and random data access in Section 5.2.

5.1 Mixed Workload Performance

Even though OLAP workloads consist of mostly independent large reads and writes, they are usually run in parallel to better utilize the system. Queries should be able to run while data is ingested to not halt the entire system. Also, intermediate results are created while other queries are running. They both require efficient reads while writing. We now investigate the overall bandwidth in mixed read-write workloads.

Workload. We run a benchmark with x write threads and y read threads in different $x - y$ combinations. We evaluate 1, 4, and 6 write threads, each with 1, 8, 18, and 30 read threads, which gives us at most 36 threads on one socket. Both workloads read/write different data (each 40 GB) on the same PMEM DIMMs. We measure the write and read bandwidth for each run. All combinations use 4KB individual access and are pinned to NUMA regions.

Results. The results are shown in Figure 11. The entries are denoted as x/y, i.e., write/read threads. For comparison, we achieve a read performance of ~31 GB/s with 30 threads without contending workloads and a peak write performance of ~13 GB/s with 6 threads. Adding a single write thread to the 30 read threads already reduces the achieved read bandwidth to ~26 GB/s. Adding more write threads further reduces the read bandwidth. Also, the combined read and write bandwidth does not exceed the non-contended maximum read bandwidth for any combination. More contending write threads reduces the bandwidth further to ~45% of the maximum read bandwidth with 6 write threads.

On the other hand, the write performance, while still reduced, does not get hit as hard initially. Using a write thread count of 4 with 1 read thread, only reduces the bandwidth from ~13 GB/s to ~12 GB/s, nearly matching the maximum write bandwidth. However, when running with 30 read threads the write bandwidth drops to just above ~40% of the maximum bandwidth, similar to the read performance drop. 6 write threads also reach ~12 GB/s with one contending read thread but perform slightly worse than 4 write threads for the other configurations. Generally, increasing the number of read threads harms the write performance and vice versa, even if the maximum bandwidth is not yet reached.

Discussion. The read bandwidth already shows a major reduction with one write thread accessing the same PMEM DIMMs. Our experiments show that replacing the one contending write thread with a contending read thread only reduces the bandwidth to ~29 GB/s instead of ~26 GB/s. This has two reasons, *i)* the bandwidth drop observed with the contending read thread is caused by the L2 Hardware Prefetcher again. Having already only one contending read thread, makes the L2 Hardware Prefetcher prefetch data from two locations, yielding suboptimal results. *ii)* The difference between having a contending read or write thread is caused by the read/write imbalance on PMEM. Due to write operations being significantly slower and therefore blocking the iMCs for a longer time than read operations, the bandwidth with a contending write thread is lower than with a contending read thread. To test this, we replicate this experiment on DRAM, which shows that the gap between an added contending read or write thread is not as significant. The read/write imbalance is considerably smaller on DRAM and therefore this effect is only moderately observable.

From these results, no clear recommendation is evident. As mixing read and write workloads harms the overall bandwidth, the reader must make assumptions about their workloads and adapt accordingly. However, we suggest using the previously recommended number of threads for sequential reads and writes. This combination keeps the balance between read and write bandwidth, as both drop to ~1/3 of their respective maximums. Configurations resulting in a higher accumulated bandwidth result in a significantly higher read/write imbalance, due to a very low write bandwidth. As the bandwidth is impacted notably, for latency insensitive workloads it might be beneficial to execute them sequentially instead of parallel. However, this is highly workload-dependent and cannot be generalized.

Insight #11: Serialize PMEM access when possible.

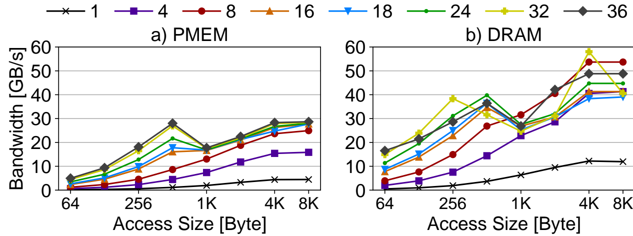

Figure 12: Random PMEM/DRAM Read Bandwidth

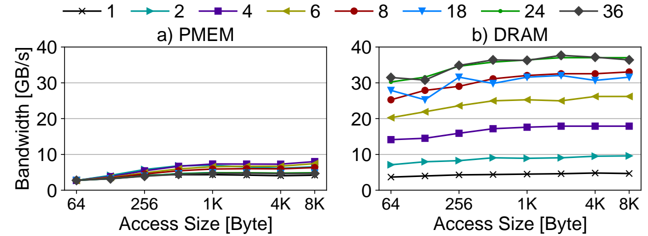

Figure 13: Random PMEM/DRAM Write Bandwidth

5.2 Random Access Performance

While sequential access generally dominates in OLAP workloads, random access occurs, e.g., during hash probing or point-lookups [46]. Thus, we investigate the bandwidth of random reads and writes.

Workload. We run the benchmark with 1 – 36 threads for access sizes ranging from 64 Byte to 8 KB, as we do not consider larger access sizes to be *random* anymore. As random access most often occurs in joins or aggregations, we limit the memory range to 2 GB, representing, e.g., a hash index.

Results. In Figure 12, we show the impact of random access on the read bandwidth. The maximum random read bandwidth is worse than its corresponding sequential bandwidth, reaching only up to ~2/3 of the maximum for larger access sizes above 4 KB, while DRAM reads achieve only 50% of their sequential maximum. For common smaller access sizes around 256/512 Byte[2] PMEM and DRAM both achieve ~50% of sequential performance. Generally, more threads achieve a higher bandwidth and hyperthreading improves the PMEM bandwidth, unlike sequential reads.

Figure 13 presents the bandwidth for random writes. Similar to reads, the maximum bandwidth is about 2/3 and 50% of the maximum sequential performance for PMEM and DRAM, respectively We again observe the highest PMEM bandwidth for 4–6 threads and larger access sizes generally improve the bandwidth utilization. On the other hand, the access size has little impact on the DRAM bandwidth and more threads achieve higher bandwidths.

Discussion. Generally, the trends of sequential access are also present in random access. More threads yield higher read bandwidth utilization but decrease write performance. Random reads do not achieve peak sequential performance, as they cannot benefit from prefetching and random writes suffer from reduced write-combining. Very small PMEM access is additionally impacted by read and write amplification, as Optane works in 256 Byte granularity. These results indicate that PMEM behaves similarly to typical *random access memory*, once a minimum access size is reached, i.e., ~512 Byte in this experiment. DRAM similarly profits from larger access sizes and does not reach its peak bandwidth until 4 KB, but the absolute bandwidth is higher than PMEM's, making it more suitable for random-access workloads.

However, we note that DRAM and PMEM are impacted differently by the size of the memory region. In our evaluation, we discovered that a 2 GB DRAM allocation is present on only one NUMA node within the socket, i.e., only 3/6 channels process requests. When operating on large memory regions, e.g. 90 GB (= all DRAM per socket), DRAM's random performance nearly doubles, as all channels are active. This scaling reaches 90% of DRAM's sequential performance and exhibits, e.g., 4x bandwidth over PMEM for 512 Byte. A larger memory region does not impact PMEM, as it is already interleaved across all channels at 4 KB granularity. Hence, in large OLAP systems DRAM scales significantly better when in full use. This difference in performance shows that hybrid designs are essential in future OLAP designs [45, 52].

Concluding, sequential PMEM access clearly outperforms random access. Yet, (*random-access-*) DRAM also benefits from larger access sizes and does not reach its full bandwidth for small operations, albeit with considerably higher absolute performance. Therefore, we recommend treating PMEM more as *sequential access memory* than *random access memory* for I/O-heavy workloads and investigate the co-design of hybrid memory models in future work. If random access is required, choose the access size as large as possible but at least 256 Byte.

Insight #12: Access PMEM sequentially or use the largest possible access for random workloads.

6 OLAP on Persistent Memory

To determine whether PMEM is suitable for OLAP workloads, we implement the Star Schema Benchmark (SSB) as a commonly used data warehouse benchmark [36]. The SSB has a star schema with one fact table *lineorder*, and four dimension tables *date, supplier, customer, part*. The SSB introduces 13 queries (Q) that are grouped into 4 query flights (QFs), where the fact table is joined with a varying number of dimension tables. Queries inside of the same flight always join the same tables but vary both in selectivity and aggregation. The SSB defines a scaling factor (*sf*) to determine the data size, where a scaling factor of 1 equals 6 million tuples in the fact table. To investigate the use of PMEM in existing OLAP systems, in Section 6.1 we execute the SSB on the in-memory database Hyrise [24, 25]. In Section 6.2, we handcraft an SSB implementation and apply the insights from our evaluation to show the potential of PMEM in future PMEM-aware OLAP workloads.

6.1 Hyrise SSB

For this experiment, we run the SSB on Hyrise, an open-source, columnar, in-memory database. All tables and intermediates are stored either completely in PMEM or in DRAM. We use *sf* 50, which results in a fact table with 300 million entries. Higher scaling factors exceed the available memory per socket on our server. As Hyrise does not support NUMA-aware allocation of intermediates and data structures, we run Hyrise on a single socket to avoid undesirable remote memory access. In Figure 14a, we show the SSB query execution times for PMEM and DRAM. On average, PMEM-Hyrise is 5.3x slower than on DRAM, with a maximum difference of 7.7x for Q2.3 and a minimum of 2.5x for Q3.1.

[2] Recent PMEM data structures work on internal 256 Byte access granularity [15, 34]

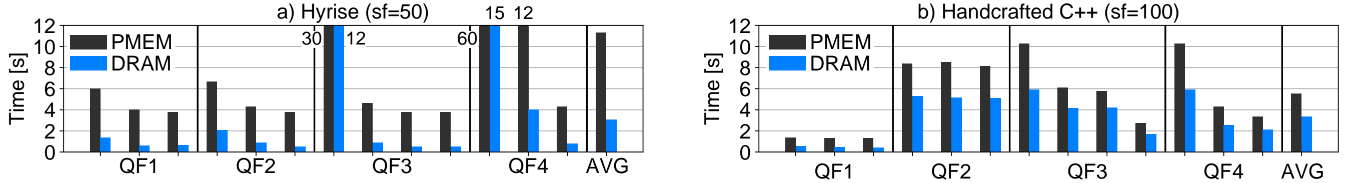

Figure 14: Star Schema Benchmark Performance.

As the results show, using PMEM as a drop-in replacement for DRAM results in low performance. For most queries, we found that hash-operations (join/aggregate) take over 90% of the execution time. This indicates dominant random access operations, which yield lower performance in PMEM than in DRAM. Thus, Hyrise's PMEM-unaware hash index implementation performs worse in PMEM than in DRAM. To overcome this performance gap, OLAP systems should use PMEM-optimized data structures.

6.2 Handcrafted SSB

To demonstrate the potential of PMEM-aware OLAP systems over existing unaware ones, we implement a handcrafted SSB in C++. The goal of this version is to illustrate how certain optimizations affect OLAP queries and not to achieve the highest possible performance with complex join implementations of query-specific tweaks. We note that optimizing PMEM for joins and related data structures [27, 33, 45] is still an open research question, requiring similar investigation as previous in-memory join algorithms [10–12], which is beyond the scope of this paper. Data is stored in a row format with a custom schema in one file per table. To avoid parsing overhead for each tuple, we align all fields to 128 Byte, which is slightly larger than the size of a tuple (< 10%) and we access only the columns that are relevant to the given query. We implement a basic hash join with the PMEM-optimized hash index *Dash* [34].

The benchmark is conducted with *sf* 100, i.e., 600 million *lineorder* entries in 70GB. In our implementation, we assume all data to fit into DRAM and PMEM and we use the same index implementation for both DRAM and PMEM. To improve the performance of our SSB in DRAM and PMEM, we use 36 threads, explicitly pinned to all 36 physical cores across both sockets. The SSB experiments are performed on *fsdax*, as Dash requires a filesystem interface.

The fact table is shuffled and striped across PMEM on both sockets and threads access only their *near* data in individual chunks (cf. *Individual Access* in Sec. 3.1). To achieve this striping in a full OLAP system, NUMA-aware partitioning schemes could be used. In DRAM, we analogously bind memory to both sockets using `libnuma`. Since the dimension tables are very small in comparison to the fact table, we replicate them on both sockets to avoid far random access, which would drastically decrease the bandwidth utilization. This data layout results in an approximately equal workload on both sockets.

We note that storing data in such a manner and creating optimal partitions is not always possible and generally hard to achieve, e.g., due to skewed data. Full NUMA-awareness in OLAP systems and its trade-offs pose interesting questions for future work.

Figure 14b shows the performance of PMEM in comparison to DRAM. On average, PMEM performance is 1.66x times worse than that of DRAM. PMEM achieves its best performance in comparison to DRAM in query Q3.3, being only 1.4x as slow as DRAM and worst in Q1.3, being 3x as slow.

QF1 differs from the other queries since it focuses on sequential read performance with only minimal joins, filters, and intermediates. On PMEM, the queries finish in ~1.3 seconds, while completing in only ~0.5 seconds in DRAM, matching the 2-3x difference in bandwidth between DRAM and PMEM as discussed in Section 3.5.

The other query flights consist of joins with multiple dimension tables. Furthermore, they require storing intermediates for the group aggregate. In these complex queries, PMEM performs better than in QF1, as they require more processing than raw table scan speed. We observe that the use of a PMEM-optimized hash index is beneficial in the joins compared to Hyrise, as PMEM achieves much closer performance than in the unoptimized version. Overall, the average query performance in QF2–4 is only 1.6x slower on PMEM than on DRAM.

Table 1: Optimization of Q2.1

	1 Thr.	18 Thr.	2-Socket	NUMA	Pinning
PMEM	306.7 s	25.1 s	12.3 s	9.4 s	8.6 s
DRAM	221.2 s	15.2 s	9.2 s	5.2 s	5.2 s

To show the impact of different optimizations, we break down the performance of QF2.1, which is representative of the other queries. In Table 1, we list the query completion time for PMEM and DRAM in seconds per query. We present the performance improvements of increasing the number of threads (*18 Thr.*), using memory and CPUs on both sockets (*2-Socket*), pinning threads to NUMA regions (*NUMA*), and pinning threads to physical cores (*Pinning*).

Increasing the number of threads improves the performance of DRAM more than that of PMEM (14x vs. 12x). Additionally, the runtime of both PMEM and DRAM can be further reduced by 3x when utilizing the dual-socket architecture. In contrast to QF1, in QF2 and later query flights, the join and hash operators limit the query performance, narrowing the gap between PMEM and DRAM. This is supported by the measured bandwidth utilization of ~15 GB/s for PMEM and ~30 GB/s for PMEM. Further investigation shows that the benchmark is memory bound over 70% of the time, indicating that the random-accesses-heavy hash lookups constitute a major performance bottleneck. Following this insight, conducting future research on random access data structures and operations is essential to achieve a good OLAP performance.

To provide an intuition of the performance of a "traditional" OLAP system, we also run Q2.1 on an NVMe SSD[3], while storing the hash indexes and intermediates in DRAM. The benchmark completes in 22.8 seconds, as it is limited by the table scan bandwidth. Compared to the traditional setup, PMEM outperforms SSDs by over a factor of 2.6x while not using any DRAM. This shows

[3]Intel® SSD DC P4610 Series, 3.20 GB/s Sequential Read 2.08 GB/s Sequential Write

that PMEM shifts the bottleneck from traditional table scan disk IO-bound processing to memory-bound operator processing and should be treated like modern in-memory systems when optimizing.

7 Best Practices for OLAP

In this section, we gather our insights and discuss them in a larger context. Following our evaluation, we propose the following generalized best practices to maximize bandwidth when working with persistent memory in OLAP workloads. For the individual insights, please refer to the corresponding sections.

(1) Read and write to PMEM in distinct memory regions (#1, #6).
(2) Scale up the number of threads when reading but limit the threads to 4 – 6 per socket when writing (#2, #7).
(3) Pin threads (explicitly) within their NUMA regions for maximum bandwidth (#3, #8).
(4) Place data on all sockets but access it only from near NUMA regions (#4, #5, #9, #10).
(5) Avoid large mixed read-write workloads when possible (#11).
(6) Access PMEM sequentially or use the largest possible access for random workloads. (#12).
(7) Use PMEM in *devdax* mode for maximum performance.

Overall, our evaluation has shown that PMEM should be optimized like DRAM for reads. They behave similarly and benefit from the same scaling/access optimizations that are applied to DRAM, e.g., 4KB access in DRAM is aligned with OS cache sizes while 4KB access in PMEM aligns directly with the underlying DIMM-interleaving. Writes, on the other hand, are more complex and impacted unexpectedly when compared to DRAM. To achieve high write bandwidths, it is essential to optimize for write-combining. While NUMA-awareness is critical in both DRAM and PMEM, it should especially be considered for PMEM as it cannot mask poor access patterns with high absolute performance. Generally, unless an application is memory-bound, bad DRAM access is less likely to be noticed due to its overall performance, whereas PMEM can quickly become a bottleneck.

While DRAM is required to achieve the best performance in the SSB and hybrid designs must be considered in future systems, PMEM-only systems offer a good price/performance. We briefly demonstrate this by comparing the current cost of DRAM and PMEM.[4] A 128 GB PMEM DIMM, as used in our system, costs ~$575 [29]. Our system contains 12 DIMMs, totaling at ~$6900 for 1.5 TB PMEM. On the other hand, DRAM costs approx. $700 for a 64 GB module. The cost of 1.5 TB DRAM[5] would be ~$16800, i.e., 2.4x higher with the average SSB query performance of DRAM being only 1.6x better than PMEM. Should this price difference remain in hourly cloud costs, PMEM offers a viable price/performance alternative to DRAM.

8 Related Work

With the availability of Intel's Optane DIMMs, first papers characterizing PMEM performance have been published. At the same time, researchers have already ported existing DBMS to PMEM, serving OLAP and OLTP workloads. Even others continuously optimize OLAP transaction processing of data warehouses for many years.

PMEM Performance Characteristics Even before PMEM was available, researchers have already written papers speculating about its performance and characteristics [39, 50]. With its public availability in 2019, researchers started investigating its actual performance. This work ranges from deriving a general understanding of PMEM covering a wide variety of PMEM behaviors [28, 30, 54] over examining its interaction in hybrid memory systems [45] to designing and evaluating specific PMEM data and index structures [27, 33, 51].

However, recent work on PMEM's performance characteristics does not focus on large multi-socket systems, which are required for OLAP workloads. Yang et al. [54] also show that certain assumptions in previous work on PMEM data structures, e.g., the assumed access pattern's behaviors, do not hold for real PMEM hardware. Both these factors show that there are still open questions regarding the low level, hardware-near PMEM characteristics, and performance. Therefore, our evaluation is independent of exact data structures providing general best practices for standalone, PMEM-only performance on large, multi-socket servers.

OLAP Workloads Recent studies show that OLAP workloads are bandwidth-intensive with various data access patterns focusing on sequential and random accesses [32, 46]. Furthermore, Sirin et al. argue that bandwidth utilization is more critical than latency, which can be masked by parallel workloads [40, 46]. Research has also shown that efficiently using the characteristics of modern hardware can improve the performance of OLAP systems by orders of magnitude [35, 47]. We therefore focus on improving PMEM's bandwidth in OLAP typical workloads on large multi-socket systems.

9 Conclusion

For many years researchers have speculated about the characteristics of persistent memory. Recent work has shown that many of the assumptions research was based on do not hold for the only recently available actual hardware. In this paper, we present an in-depth analysis of the performance of persistent memory for OLAP workloads. Our evaluation shows that it is crucial to understand PMEM's performance characteristics in large, multi-socket systems to maximize its bandwidth. While PMEM can be treated like DRAM for most read access, it must be used differently when writing. We propose a set of 7 best practices for application designers to fully utilize PMEM's bandwidth in future systems. We implement the SSB with our proposed optimizations and show that the performance gains apply to real-world OLAP workloads. In the SSB, PMEM is only 1.66x slower than DRAM on average for large OLAP queries. In conclusion, we show that PMEM is suitable for data-intensive systems and provides a higher capacity at lower prices than DRAM while performing only slightly worse. In future work, we plan to transfer our insights to hybrid PMEM-DRAM setups.

Acknowledgments

We thank Markus Dreseler for his help with Hyrise and the reviewers for their valuable feedback. This project has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 957407.

[4]This calculation is illustrative, as PMEM prices are not yet stable and cloud providers do not offer PMEM.
[5]This is not possible with most common DRAM configurations.

References

[1] Daniel J. Abadi, Peter A. Boncz, and Stavros Harizopoulos. 2009. Column oriented Database Systems. *Proc. VLDB Endow.* 2, 2, 1664–1665.

[2] Daniel J. Abadi, Samuel Madden, and Nabil Hachem. 2008. Column-stores vs. row-stores: how different are they really?. In *SIGMOD*. ACM, 967–980.

[3] Tom J. Ameloot, Gaetano Geck, Bas Ketsman, Frank Neven, and Thomas Schwentick. 2016. Data partitioning for single-round multi-join evaluation in massively parallel systems. *SIGMOD Rec.* 45, 1, 33–40.

[4] Mihnea Andrei, Christian Lemke, Günter Radestock, Robert Schulze, Carsten Thiel, Rolando Blanco, Akanksha Meghlan, Muhammad Sharique, Sebastian Seifert, Surendra Vishnoi, Daniel Booss, Thomas Peh, Ivan Schreter, Werner Thesing, Mehul Wagle, and Thomas Willhalm. 2017. SAP HANA Adoption of Non-Volatile Memory. *Proc. VLDB Endow.* 10, 12, 1754–1765.

[5] Joy Arulraj, Justin J. Levandoski, Umar Farooq Minhas, and Per-Åke Larson. 2018. BzTree: A High-Performance Latch-free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5, 553–565.

[6] Joy Arulraj, Andrew Pavlo, and Subramanya Dulloor. 2015. Let's Talk About Storage & Recovery Methods for Non-Volatile Memory Database Systems. In *SIGMOD*. ACM, 707–722.

[7] Joy Arulraj, Andrew Pavlo, and Prashanth Menon. 2016. Bridging the Archipelago between Row-Stores and Column-Stores for Hybrid Workloads. In *SIGMOD*. ACM, 583–598.

[8] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *Proc. VLDB Endow.* 10, 4, 337–348.

[9] Peter Bailis, Camille Fournier, Joy Arulraj, and Andrew Pavlo. 2016. Research for Practice: Distributed Consensus and Implications of NVM on Database Management Systems. *Commun. ACM* 59, 11, 52–55.

[10] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M. Tamer Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1, 85–96.

[11] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2013. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware. In *ICDE*. IEEE Computer Society, 362–373.

[12] Spyros Blanas, Yinan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD*. ACM, 37–48.

[13] Stefano Ceri, Mauro Negri, and Giuseppe Pelagatti. 1982. Horizontal Data Partitioning in Database Design. In *SIGMOD*. ACM Press, 128–136.

[14] Surajit Chaudhuri and Umeshwar Dayal. 1997. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Rec.* 26, 1, 65–74.

[15] Youmin Chen, Youyou Lu, Fan Yang, Qing Wang, Yang Wang, and Jiwu Shu. 2020. FlatStore: An Efficient Log-Structured Key-Value Storage Engine for Persistent Memory. In *ASPLOS*. ACM, 1077–1091.

[16] Zhibo Chen and Carlos Ordonez. 2013. Optimizing OLAP cube processing on solid state drives. In *DOLAP*. ACM, 79–84.

[17] Intel Corporation. 2014. Disclosure of Hardware Prefetcher Control on Some Intel® Processors. https://software.intel.com/content/www/us/en/develop/articles/disclosure-of-hw-prefetcher-control-on-some-intel-processors.html.

[18] Intel Corporation. 2019. Intel® Xeon® Processor Scalable Family Technical Overview. https://software.intel.com/content/www/us/en/develop/articles/intel-xeon-processor-scalable-family-technical-overview.html.

[19] Intel Corporation. 2020. Intel 64 and IA-32 Architectures Developer's Manual: Vol. 1. https://www.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-software-developer-vol-1-manual.html.

[20] Intel Corporation. 2020. Intel® VTune™ Profiler. https://software.intel.com/content/www/us/en/develop/tools/vtune-profiler.html.

[21] Intel Corporation. 2020. Persistent Memory Development Kit. https://pmem.io/pmdk/.

[22] Justin DeBrabant, Joy Arulraj, Andrew Pavlo, Michael Stonebraker, Stanley B. Zdonik, and Subramanya Dulloor. 2014. A Prolegomenon on OLTP Database Systems for Non-Volatile Memory. In *ADMS*. VLDB Endowment Inc., 57–63.

[23] Conor Doherty, Gary Orenstein, Steven Camiña, and Kevin White. 2015. *Building real-time data pipelines: unifying applications and analytics with in-memory architectures.* O'Reilly Media.

[24] Markus Dreseler, Martin Boissier, Tilmann Rabl, and Matthias Uflacker. 2020. Quantifying TPC-H Choke Points and Their Optimizations. *Proc. VLDB Endow.* 13, 8, 1206–1220.

[25] Markus Dreseler, Jan Kossmann, Martin Boissier, Stefan Klauck, Matthias Uflacker, and Hasso Plattner. 2019. Hyrise Re-engineered: An Extensible Database System for Research in Relational In-Memory Data Management. In *Advances in Database Technology - 22nd International Conference on Extending Database Technology, EDBT 2019, Lisbon, Portugal, March 26-29, 2019.* OpenProceedings.org, 313–324.

[26] Franz Färber, Sang Kyun Cha, Jürgen Primsch, Christof Bornhövd, Stefan Sigg, and Wolfgang Lehner. 2011. SAP HANA database: data management for modern business applications. *SIGMOD Rec.* 40, 4, 45–51.

[27] Philipp Götze, Arun Kumar Tharanatha, and Kai-Uwe Sattler. 2020. Data structure primitives on persistent memory: an evaluation. In *DaMoN*. ACM, 15:1–15:3.

[28] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2021. Understanding the Idiosyncrasies of Real Persistent Memory. *Proc. VLDB Endow.* 14, 4, 626–639.

[29] Jim Handy. 2020. Intel's Optane DIMM Price Model. https://thememoryguy.com/intels-optane-dimm-price-model/.

[30] Joseph Izraelevitz, Jian Yang, Lu Zhang, Juno Kim, Xiao Liu, Amirsaman Memaripour, Yun Joon Soh, Zixuan Wang, Yi Xu, Subramanya R. Dulloor, Jishen Zhao, and Steven Swanson. 2019. Basic Performance Measurements of the Intel Optane DC Persistent Memory Module. *CoRR* abs/1903.05714.

[31] Qifa Ke, Vijayan Prabhakaran, Yinglian Xie, Yuan Yu, Jingyue Wu, and Junfeng Yang. 2011. Optimizing Data Partitioning for Data-Parallel Computing. In *HotOS*. USENIX Association.

[32] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter A. Boncz. 2018. Everything You Always Wanted to Know About Compiled and Vectorized Queries But Were Afraid to Ask. *Proc. VLDB Endow.* 11, 13, 2209–2222.

[33] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proc. VLDB Endow.* 13, 4, 574–587.

[34] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8, 1147–1161.

[35] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. Knowl. Data Eng.* 14, 4, 709–730.

[36] Patrick E. O'Neil, Elizabeth J. O'Neil, Xuedong Chen, and Stephen Revilak. 2009. The Star Schema Benchmark and Augmented Fact Table Indexing. In *TPCTC*, Vol. 5895. Springer, 237–252.

[37] Ismail Oukid, Johan Lasperas, Anisoara Nica, Thomas Willhalm, and Wolfgang Lehner. 2016. FPTree: A Hybrid SCM-DRAM Persistent and Concurrent B-Tree for Storage Class Memory. In *SIGMOD*. ACM, 371–386.

[38] Andrew Pavlo. 2015. Emerging Hardware Trends in Large-Scale Transaction Processing. *IEEE Internet Comput.* 19, 3, 68–71.

[39] Steven Pelley, Thomas F. Wenisch, Brian T. Gold, and Bill Bridge. 2013. Storage Management in the NVRAM Era. *Proc. VLDB Endow.* 7, 2, 121–132.

[40] 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 *DaMoN*. ACM, 13:1–13:8.

[41] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2015. Scaling Up Concurrent Main-Memory Column-Store Scans: Towards Adaptive NUMA-aware Data and Task Placement. *Proc. VLDB Endow.* 8, 12, 1442–1453.

[42] Iraklis Psaroudakis, Tobias Scheuer, Norman May, Abdelkader Sellami, and Anastasia Ailamaki. 2016. Adaptive NUMA-aware data placement and task scheduling for analytical workloads in main-memory column-stores. *Proc. VLDB Endow.* 10, 2, 37–48.

[43] Tilmann Rabl and Hans-Arno Jacobsen. 2017. Query Centric Partitioning and Allocation for Partially Replicated Database Systems. In *SIGMOD*. ACM, 315–330.

[44] Peter Scheuermann, Gerhard Weikum, and Peter Zabback. 1998. Data Partitioning and Load Balancing in Parallel Disk Systems. *Proc. VLDB Endow.* 7, 1, 48–66.

[45] Anil Shanbhag, Nesime Tatbul, David Cohen, and Samuel Madden. 2020. Large-scale in-memory analytics on Intel® Optane™ DC persistent memory. In *DaMoN*. ACM, 4:1–4:8.

[46] Utku Sirin and Anastasia Ailamaki. 2020. Micro-architectural Analysis of OLAP: Limitations and Opportunities. *Proc. VLDB Endow.* 13, 6, 840–853.

[47] Juliusz Sompolski, Marcin Zukowski, and Peter A. Boncz. 2011. Vectorization vs. compilation in query execution. In *DaMoN*. ACM, 33–40.

[48] Michael Stonebraker, Daniel J. Abadi, Adam Batkin, Xuedong Chen, Mitch Cherniack, Miguel Ferreira, Edmond Lau, Amerson Lin, Samuel Madden, Elizabeth J. O'Neil, Patrick E. O'Neil, Alex Rasin, Nga Tran, and Stanley B. Zdonik. 2005. C-Store: A Column-oriented DBMS. In *Proc. VLDB Endow.* ACM, 553–564.

[49] Alejandro A. Vaisman and Esteban Zimányi. 2014. *Data Warehouse Systems - Design and Implementation.* Springer.

[50] 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 *SIGMOD*. ACM, 1541–1555.

[51] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2019. Persistent Memory I/O Primitives. In *DaMoN*. ACM, 12:1–12:7.

[52] Alexander van Renen, Lukas Vogel, Viktor Leis, Thomas Neumann, and Alfons Kemper. 2020. Building blocks for persistent memory. *VLDB J.* 29, 6, 1223–1241.

[53] Shivaram Venkataraman, Niraj Tolia, Parthasarathy Ranganathan, and Roy H. Campbell. 2011. Consistent and Durable Data Structures for Non-Volatile Byte-Addressable Memory. In *FAST*. USENIX, 61–75.

[54] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steven Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *FAST*. USENIX Association, 169–182.

[55] 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 *FAST*. USENIX Association, 167–181.