



# A Design Space Exploration and Evaluation for Main-Memory Hash Joins in Storage Class Memory

Wentao Huang  
National University of Singapore  
huang@comp.nus.edu.sg

Yunhong Ji  
Renmin University of China  
jiyunhong@ruc.edu.cn

Xuan Zhou  
East China Normal University  
xzhou@dase.ecnu.edu.cn

Bingsheng He  
National University of Singapore  
hebs@comp.nus.edu.sg

Kian-Lee Tan  
National University of Singapore  
tankl@comp.nus.edu.sg

## ABSTRACT

In this paper, we seek to perform a rigorous experimental study of main-memory hash joins in storage class memory (SCM). In particular, we perform a design space exploration in real SCM for two state-of-the-art join algorithms: partitioned hash join (PHJ) and non-partitioned hash join (NPHJ), and identify the most crucial factors to implement an SCM-friendly join. Moreover, we present a rigorous evaluation with a broad spectrum of workloads for both joins and provide an in-depth analysis for choosing the most suitable algorithm in real SCM environment. With the most extensive experimental analysis up-to-date, we maintain that although there is no one universal winner in all scenarios, PHJ is generally superior to NPHJ in real SCM.

## PVLDB Reference Format:

Wentao Huang, Yunhong Ji, Xuan Zhou, Bingsheng He, and Kian-Lee Tan. A Design Space Exploration and Evaluation for Main-Memory Hash Joins in Storage Class Memory. PVLDB, 16(6): 1249 - 1263, 2023.  
doi:10.14778/3583140.3583144

## PVLDB Artifact Availability:

The source code, data, and/or other artifacts have been made available at <https://github.com/fukien/hashjoin-scm>.

## 1 INTRODUCTION

Main-memory hash joins attracted a surge of interest in the last decade. Since I/O is no longer involved in the critical path, minimizing the cache thrashing penalty has become the main design objective. To achieve this goal, two families of hash join algorithms have been designed and extensively studied: partitioned hash join (PHJ) and non-partitioned hash join (NPHJ). In particular, PHJ borrows the idea of Grace Hash Join [53]. It introduces a preliminary partition phase to partition data into cache-sized fragments. The following join phase is performed on these fragments, which evades excessive cache thrashes.

Proponents of NPHJ, however, argue that modern parallel processors are powerful enough to hide the cache miss penalty, so the

preparatory partition phase brings little benefits but incurs excessive partitioning overhead. Moreover, the partition phase requires painstaking efforts of hardware-conscious tailoring (e.g., cache size, TLB capacity), but such efforts do not always pay off. In one embodiment, partitioning aims to transform some arbitrarily distributed data into a distribution of high locality pattern. Yet, some data already exhibit a certain level of locality, making the additional partition phase redundant [11, 55]. In another embodiment, partitioning demands meticulous tuning against the underlying hardware. Any gain in performance (after accounting for the overheads) may soon be diminished if the partitioning parameters deviate from the optimal configuration [8]. As a consequence, even though PHJ already outperforms NPHJ in some workloads [6, 7, 51, 79], the PHJ-vs-NPHJ debate is still ongoing.

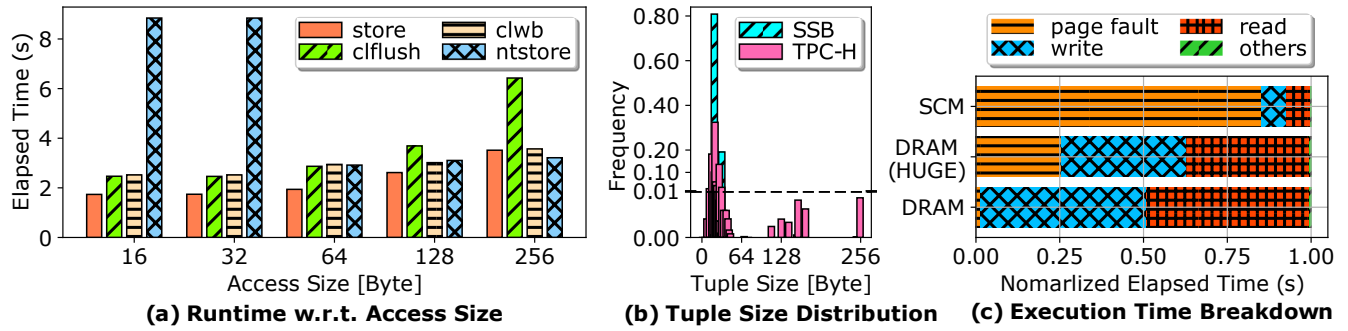
Meanwhile, main memory (DRAM) technology has hit a scaling wall [21, 77]: it is becoming increasingly difficult to shrink the DRAM cell size while maintaining enough capacity<sup>1</sup>. Storage Class Memory (SCM), or non-volatile memory, is the emerging memory technology that primarily targets breaking this wall [21, 37, 45]. It offers large capacity, byte-addressability, and near-DRAM access performance. Moreover, most SCM technologies support data persistence [1, 17, 25, 29, 57, 72, 84, 91], making SCM an appealing alternative not only for DRAM but also for SSD. Several SCM technologies have been put into practice [1, 44, 76, 82, 85], among which NVDIMM-P [44] has become the most popular endeavor. It specifies that SCM should be formed as a memory DIMM, attached to the memory bus, and communicate directly with processors through DDR interfaces. Additionally, it defines the concept of internal buffer management, standardizes the domain of persistence/visibility, and facilitates the programming paradigm [76].

Inspired by NVDIMM-P, leading memory manufacturers have been grinding for developing SCM products [42, 82, 85]. Up till now, Intel Optane DC Persistent Memory Module (Optane DIMM) is the first and only industrial NVDIMM-P implementation in market [42]. Since its release, numerous attempts have been made to harness it in developing persistent data structures [34, 35, 52, 63], fault-tolerant file systems [32, 78, 100], crash recovery mechanisms [14, 59, 98], etc. The industry community also invests huge efforts to deploy it in data centers [48] and cloud providers [30, 58]. It is expected that SCM will become a crucial building block in future data-intensive platforms.

Unfortunately, SCM differs from DRAM in a few aspects, indicating that directly translating main-memory hash joins from

<sup>1</sup>Memory capacity per core is expected to decrease by 30% biannually [77].

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit <https://creativecommons.org/licenses/by-nc-nd/4.0/> to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing [info@vldb.org](mailto:info@vldb.org). Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.  
Proceedings of the VLDB Endowment, Vol. 16, No. 6 ISSN 2150-8097.  
doi:10.14778/3583140.3583144



**Figure 1: (a) Write performance with different access sizes. (b) The tuple size distribution of SSB [73] and TPC-H [22] joins. (c) The execution time breakdown of a write-and-read microbenchmark on a 16KB memory region (“HUGE” denotes the huge page configuration of DRAM, “others” overhead remains unnoticeable in all three memory configurations).**

DRAM to SCM may lead to unsatisfactory results. For instance, SCM behaves asymmetrically in read/write bandwidth, and its internal access granularity is incompatible with cacheline (e.g., 256B vs. 64B). Although existing works [23, 31, 56, 92, 94, 95, 97] have come up with practices for developing SCM-friendly applications, they fail to notice the nature of a join workload, making these practices unavailing in SCM-based joins. We address two crucial limitations that have been previously overlooked.

(1) Persistence cost should be eliminated to the greatest extent possible. Existing studies tend to employ persistent instructions (e.g., “clflush”, “clwb”, “ntstore”) for immediate persistence [88, 97] or higher write bandwidth [10, 13]. However, persistent instructions only prevail at large access sizes and lose to regular store at small access sizes (cf. Figure 1(a)). Note that join processing typically operates at a small granularity of tuple size (cf. Figure 1(b) for the tuple size distribution in real benchmarks [22, 73]), persistent instructions, thus, can do harm to join processing. Moreover, join processing does not require immediate persistence. In light of power outage, rerunning a query has a higher gain expectation than recover-and-continue an interrupted run. Hence, persistent instructions should not be used in SCM-based joins.

(2) Page fault overhead is more pronounced for cache-friendly algorithms in SCM. We run a microbenchmark to demonstrate this point. We first allocate a cache-sized region in SCM and eliminate page faults via “memset”. Afterward, we issue random writes followed by random reads in this region and measure the runtime. We conduct the measurement in DRAM and plot the normalized time in Figure 1(c). Page faults consume no more than 25% in DRAM but takes up over 80% overhead in SCM, making page faults a bottleneck for cache-friendly algorithms in SCM (we also observe this phenomenon in PHJ). On this account, for cache-friendly algorithms, page faults should be avoided as much as possible.

As far as we know, prior SCM-related studies pay little attention to main-memory join processing. While a recent work by Maltenberger et al. [66] attempts to investigate main-memory hash joins in SCM, the work aims to compare DRAM and SCM join performance, and fails to tune the algorithms accordingly for the above limitations. Hence, their finding that NPHJ is superior over PHJ is not sufficiently conclusive (and our study shows that this is indeed the case!). We, therefore, seek to revisit the hash join problem and perform a more rigorous experimental study in SCM.

In this work, we aim to study the two families of hash joins in real SCM<sup>2</sup> to understand their relative performance. In particular, we perform a design space exploration for the implementations of PHJ and NPHJ with a particular focus on SCM-conscious tuning (Section 5). We also conduct a comprehensive evaluation in extensive workloads to compare PHJ with NPHJ fairly (Section 6). With a systematic experimental analysis, we maintain that PHJ is generally the preferable solution for SCM. Moreover, we propose several meaningful discussions to offer more insights for practitioners (Section 7). It is worth addressing that we conduct experiments with Optane DIMMs, the only available SCM hardware at present, but our findings and discussions are majorly based on NVDIMM-P standard. Moreover, we do not rely on any specific persistent features. Therefore, our study has strong generalizability and can be applied to future SCMs that formed in a DIMM factor [1, 20, 44, 82, 85] (see Section 2 for more details). To summarize, we make the following contributions:

- (1) We present, to our knowledge, the first thorough evaluation to explore the design space of main-memory hash joins in real SCM. By considering the characteristics of SCM, we scrutinize PHJ’s and NPHJ’s internal phase implementations and inspect the alternative implementations for both joins (cf. Table 1). Moreover, we attentively discuss the optimizations and identify the main bottlenecks of join processing in the real SCM environment, bridging the gap between SCM studies and main-memory hash joins.
- (2) We systematically conduct so far the most rigorous experimental study to compare PHJ and NPHJ with a wide range of workloads. Our experimental findings reveal the pros and cons of different join algorithms and answer the aforementioned question that PHJ is generally the better solution in the real SCM platform.
- (3) We propose a set of practical tips for tuning efficient join algorithms and present several analyses in a few auxiliary dimensions. These tips, along with the discussions, summarize the key insights of this paper and serve as essential guidelines for practitioners.

The rest of this paper is organized as follows. We introduce the SCM landscape in Section 2 and review PHJ and NPHJ in Section 3. Section 4 covers the details of experimental setups. We explore the design space for PHJ and NPHJ in Sections 5, and perform a comprehensive evaluation of PHJ and NPHJ in Section 6. Section 7

<sup>2</sup>Although a DRAM-SCM hybrid platform is more appealing to investigate, we seek a prerequisite to thoroughly understand the join behaviors in an SCM-only platform.

discusses the experimental findings in-depth and presents a few auxiliary analyses. We briefly review related works in Section 8 and conclude the paper in Section 9. Additional experiments and analyses are available in a technical report [38].

## 2 THE SCM LANDSCAPE

The DRAM technology is facing an acute challenge: it fails to scale to sub-20nm size [21, 77], which limits its deployment in future technology nodes. In order to break this wall<sup>3</sup>, various SCMs [17, 28, 29, 57, 72, 84] have been proposed, all of which manifest a strong ability in scaling. For instance, ReRAM was shown to scale down to the sub-5nm scale [33] and PCM was validated to shrink to the sub-2nm scale [45]. In addition to the excellent scaling ability, SCM also delivers byte-addressability, near-DRAM access speed, and low economic cost. Therefore, SCM is considered a strong alternative for DRAM.

JEDEC specifies the NVDIMM-P [44] standard for adopting SCM technology<sup>4</sup>. In NVDIMM-P, SCM is organized as memory DIMMs and attached to memory bus as DRAM. Through an integrated memory controller (iMC), it directly communicates with processors at a cacheline granularity (64B). The DIMM equips an on-DIMM controller and a limited buffer (e.g., 16KB in Optane DIMM [62, 93]), which manage data access and buffering. The on-DIMM controller also supports prefetching, making sequential access faster than random. Due to the trade-off between address indirection and encryption [97], the on-DIMM buffer and controller visit the underlying SCM media at a coarser granularity (e.g., 256B XPLine size in Optane DIMM)<sup>5</sup>. Thus, small-size data requests from processors will result in read/write amplification. To exploit SCM's byte-addressability, NVDIMM-P suggests SCM to be accessed via DAX-mmap [2, 46], which allows data requests to be completed via efficient "load" and "store" instructions. Because of SCM's read/write asymmetry, the "load" bandwidth is superior to "store" [27, 70, 74, 75, 88]. Moreover, DAX-mmap exposes the costly page faults in SCM's critical path [19, 69]<sup>6</sup>, which impairs the performance of cache-sensitive applications (e.g., Figure 1(c)) and leads to notorious "small files problem" [2]. Furthermore, NVDIMM-P defines persistent instructions ("clwb", "clflush", etc.) to make use of SCM's non-volatility, and works compatibly with the prospective CXL [20] standard. In consequence, NVDIMM-P is becoming a promising building block in future computing systems.

Due to the above features and the strong scaling ability, NVDIMM-P is widely acknowledged as the dominant standard for future SCM devices. We, thus, seek to drill into a deeper understanding of main-memory hash joins for NVDIMM-P SCMs. As Optane DIMM is the only available NVDIMM-P implementation up to now [42], we use it to conduct our experimental study. However, our study is not limited to Optane. It can be easily generalized to any SCM technologies that conform to NVDIMM-P. For a better elaboration, we highlight the following key traits of NVDIMM-P SCM and consider them as the fundamental primitives of our study:

- $\mathcal{P}_1$ : access granularity mismatch.

<sup>3</sup>SRAM and NOR flash also have hit the scaling wall [21].

<sup>4</sup>JEDEC also proposes the NVDIMM-N [43], which pairs DRAM with flash in a DIMM. Thus, it still suffers from the DRAM scaling wall and is beyond the scope of this study.

<sup>5</sup>This granularity also represents the unit size of error-correct code (ECC) block [31].

<sup>6</sup>This is assumed to be a common feature for most SCM technologies [19].

**Table 1: List of Evaluated Main-Memory Hash Joins**

Taxonomy	Join Notation	Partitioning	Hashing
Non-Partitioned Hash Join	NPHJ-SC	—	Separate Chaining
	NPHJ-LP		Linear Probing
Partitioned Hash Join	SHRII-SC	Shared Partitioning (linked list)	Separate Chaining
	SHRII-LP		Linear Probing
	SHRII-HM		Histogram Mechanism
	SHRcm-BC	Shared Partitioning (contiguous memory)	Bucket Chaining
	SHRcm-SC		Separate Chaining
	SHRcm-LP		Linear Probing
	SHRcm-HM		Histogram Mechanism
	INDII-SC	Independent Partitioning (linked list)	Separate Chaining
	INDII-LP		Linear Probing
	INDII-HM		Histogram Mechanism
	INDcm-BC	Independent Partitioning (contiguous memory)	Bucket Chaining
	INDcm-SC		Separate Chaining
	INDcm-LP		Linear Probing
	INDcm-HM		Histogram Mechanism
	RDX-BC	Radix Partitioning	Bucket Chaining
	RDX-SC		Separate Chaining
	RDX-LP		Linear Probing
	RDX-HM		Histogram Mechanism
	ASYM-BC	Asymmetric Radix Partitioning	Bucket Chaining
	ASYM-SC		Separate Chaining
	ASYM-HM		Histogram Mechanism

<sup>1</sup> "—" depicts that the algorithms that do not perform partitioning;

<sup>2</sup> "Histogram Mechanism" represents the histogram-based re-ordering hashing scheme proposed in [51];

<sup>3</sup> "(contiguous memory)"-based partitioning methods apply to uniformly distributed data only.

- $\mathcal{P}_2$ : on-DIMM buffer/controller integrated.
- $\mathcal{P}_3$ : read/write asymmetry.
- $\mathcal{P}_4$ : costly page fault handling.
- $\mathcal{P}_5$ : persistent instructions supported.

## 3 HASH JOINS

We review PHJ and NPHJ and discuss their variants in this section. For better comprehensibility, we categorize the joins and present a taxonomy in Table 1. In addition, we refer to "the build side" and "the probe side" as  $R$  and  $S$  respectively, and use the terms "table" and "relation" interchangeably throughout the paper.

### 3.1 Non-Partitioned Hash Joins

Non-partitioned hash join (NPHJ) [11, 55] is similar to the canonical hash join. It simply comprises a build phase and a probe phase. During the build phase, all threads jointly build a shared gigantic hash table. Either separate chaining or open addressing can be employed for collision resolution. The build side is evenly divided among all threads, and each thread hashes tuples from its own chunk. Latches or compare-and-swap (CAS) atomic instructions are employed to alleviate the potential write-conflict issues in building. Typically, the hash table has far more buckets than active threads, so the lock contention cost remains low. The probe phase is conducted in a similar way but without the write-conflict protections. The algorithm incurs one read pass for both  $R$  and  $S$  but has one write pass over  $R$  only. Given that  $R$  is usually smaller

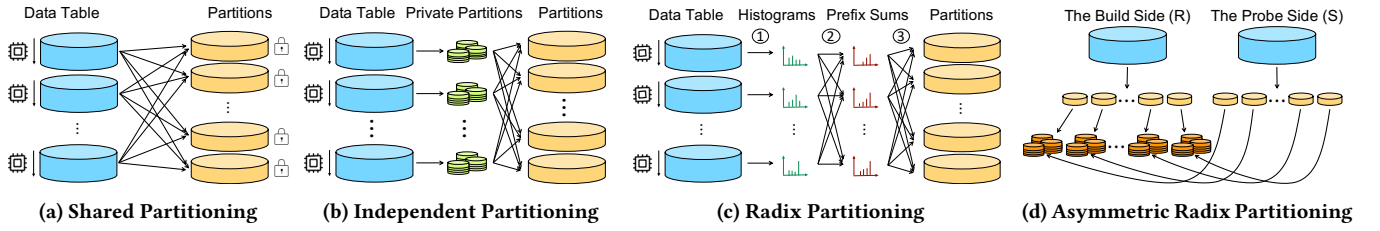


Figure 2: Partitioning Methods.

than  $S$  [22, 79]. NPHJ significantly saves the write cost, especially for the write-susceptible SCM.

Thanks to the modern parallel processors' simultaneous multi-threading (SMT) and out-of-order execution (OOE), cache miss penalties can be effectively hidden. The cache miss can be further concealed by enabling software/hardware prefetching and bucket-level alignment [7, 8]. Hence, modern parallel hardware alleviates the cache miss overhead effectively.

### 3.2 Partitioned Hash Joins

Partitioned hash join (PHJ) is another family of main-memory hash joins. In order to avoid cache thrashing during the join, it introduces a preparatory partition phase to divide relations into cache-sized sub-relations. The subsequent join phase is performed partition-wise, reducing the cache thrashing overhead by a large margin.

**3.2.1 Partition Phase.** There are numerous ways to perform partitioning [80, 99], among which radix partitioning [6] has been shown to be the best choice in main memory (DRAM) systems. A natural question to ask is whether radix partitioning still dominates in SCM. Recall that different partitioning methods induce different read/write passes and that SCM is more prone to writes than DRAM; therefore, it is necessary to reconsider the performance of partitioning algorithms in SCM environment. In the following, we revisit representative partitioning algorithms and discuss their alternative implementations with special attention to read/write passes. Without the loss of generalizability, all active threads split  $R$  and  $S$  at the beginning of the partition phase evenly.

(1) **Shared Partitioning** [11]. In shared partitioning, all threads work jointly to populate a common set of partitions, each of which is structured as a buffer linked list. In order to circumvent write-conflict issues, each partition is assigned a private lock for thread synchronization (cf. Figure 2(a)). The algorithm generates a read and a write pass on both sides.

(2) **Independent Partitioning** [11]. Independent partitioning allows each thread to create its private set of partitions (cf. Figure 2(b)), thereby eliminating the need for lock protection. Like shared partitioning, each partition is organized as a buffer linked list. After all threads finish their own jobs, their individual sets of private partitions are merged into a single set of shared partitions. Therefore, it also takes a read and a write pass to perform independent partitioning.

(3) **Radix Partitioning** [6]. Radix Partitioning is the most prominent partitioning algorithm so far (in DRAM). Unlike shared partitioning and independent partitioning, a partition here is formed as a contiguous memory region, and all partitions together also constitute a giant contiguous memory region. The algorithm operates

in three steps (cf. Figure 2(c)): ① The input relation is evenly split among all threads, where each thread scans a sub-relation and populates a histogram that counts the tuple number for every single partition. ② All threads synchronize at a barrier to modify their histograms. By computing and aggregating the prefix sum of all histograms, each thread is able to update its own histogram, where the updated values correspond to the exclusive partitioning positions for the tuples in its scanning sub-relation. ③ According to its histogram, each thread rescans its sub-relation and redistributes tuples to their respective partitioning positions. Since partitioning positions are exclusive, tuples can be efficiently written to their final destinations without write synchronization.

The above radix partitioning algorithm takes two read passes and one write pass. However, Manegold et al. [12, 67, 68] claims that the partitioning performance drops sharply once the partition fanout exceeds the TLB capacity. The radix partitioning, therefore, is modified to a multi-pass manner, each pass of which is bounded by TLB limit and thereby precluding excessive TLB thrashing. Hence, a  $m$ -pass radix partitioning requires  $2m$  read passes and  $m$  write passes for  $R$  and  $S$ , where  $m$  refers to the number of partitioning passes.

(4) **Asymmetric Radix Partitioning** [49]. Khattab et al. [49] go beyond radix partitioning and propose an idea called asymmetric radix partitioning, which targets a binary join scenario with a salient size difference, i.e.,  $S$  is much larger than  $R$ . Unlike radix partitioning that maintains same pass number for both sides, asymmetric radix partitioning applies different number of passes for partitioning  $R$  and  $S$  respectively. In particular, it takes  $m$  passes for  $R$  and  $n$  passes for  $S$  (referred to as  $m$ - $n$ -pass), where  $m > n$  (see Figure 2(d) for an example when  $m = 2, n = 1$ ). Since  $S$  is commonly larger than  $R$ , the partitioning cost should be alleviated considerably compared to  $m$ -pass radix partitioning. The algorithm, therefore, results in  $2mR + 2nS$  reads and  $mR + nS$  writes.

However, asymmetric radix partitioning has been shown to be inefficient in DRAM [49]. Fewer passes over  $S$  renders more reads during the join phase, resulting in more cache misses. The saving from partitioning quickly diminishes, suggesting the algorithm must revert to radix partitioning. Despite the disappointing profile in DRAM, we note that the join phase incurs limited write operations, which is beneficial in a write-susceptible context. Hence, asymmetric radix partitioning may exhibit a competitive profile in SCM.

It is worth mentioning that the partitioning performance can be significantly improved with software write-combining buffers (SWWCB) and non-temporal stores ("ntstore") [4, 7, 79, 99]. SWWCB maintains a separate in-cache buffer of  $N$ -tuple capacity for each partition. During partitioning, tuples are copied to these buffers

**Table 2: PHJ Passes**

Partitioning	Reads	Writes
SHRll	$2(R + S)$	$R + S$
SHRcm	$2(R + S)$	$R + S$
INDll	$2(R + S)$	$R + S$
INDcm	$2(R + S)$	$R + S$
RDX	$(2m + 1)(R + S)$	$m(R + S)$
ASYM	$(2m + 1)R + (2n + k)S$	$mR + nS$

<sup>1</sup> “k” denotes the times for  $R$  partition number over  $S$  partition number;

<sup>2</sup> We do not explicitly distinguish sequential/random passes as we avail of SWWCBS’ temporal sequential pattern in random page visits.

first. Once a buffer is full, the whole buffer is flushed to the final partition destination, combining  $N$  writes to one. Therefore, both cache thrashing and TLB thrashing are decreased by a factor of  $N$ . Partitioning with SWWCBS can be further enhanced with “ntstore”. Recall that a regular store must fetch a corresponding cacheline before writing data to it, which pollutes the cache and wastes memory bandwidth. With “ntstore”, a buffer is directly written to memory without cache pollution. As a consequence, the bandwidth utilization is significantly enhanced.

Before we proceed to demystify the join phase, we digress to discuss an alternative partition layout for shared and independent partitioning. Recall that shared and independent partitioning structure their partition layout as buffer linked lists, which may span separate memory pages. Compared to the contiguous memory layout, scanning a buffer linked list incurs random memory reads, which could expose moderate cache misses to the join phase. Considering this factor, we restructure the partition layout from a buffer linked list to a pre-allocated contiguous memory region for shared and independent partitioning (SHRcm-\* and INDcm-\* algorithms in Table 1). This modification trades random memory accesses for sequential accesses without introducing extra passes over data, profiting not only the join execution but also the partition phase. A noteworthy issue is that we normally lack knowledge of data distribution prior to partitioning. The pre-allocated memory regions, therefore, may not well fit the actual data distribution, resulting in memory overflow for specific partitions. To alleviate this issue, we slightly enlarge the pre-allocated regions to a certain extent (since SCM has denser capacity than DRAM, we are able to allocate larger space for partitions), allowing each partition to carry more tuples than usual. However, this partitioning method may still suffer memory overflow issues with highly skewed datasets. The modification, therefore, only applies to lowly skewed datasets.

**3.2.2 Join Phase.** The join phase is executed partition-by-partition. Each active thread fetches a  $R$  partition and constructs a hash table with a distinct hash function. It then fetches the respective partition of  $S$  and probes the hash table with the same hash function. Note that different partitioning methods yield disparate partition layouts. A contiguous layout involves one sequential read pass, whereas a buffer linked list triggers one random read pass. The probing performance, thus, is varied. A notable exception is asymmetric radix partitioning. Since it takes more partitioning passes over  $R$  than  $S$ , the  $R$  partition fanout may be  $k$  times of  $S$ . A  $S$  partition should potentially find its matches in  $k$   $R$  partitions. Thus, the

asymmetric scheme takes 1 and  $k$  read passes for  $R$  and  $S$  respectively. We summarize the passes in Table 2 for ease of reference.

The major benefit of PHJ comes from hash tables of high locality, which obliterates cache thrashing during probing. In order to attain this high locality, a hash table must reside entirely within caches, indicating that a hash table across separate memory pages is not an option. Both separate chaining and bucket chaining can be employed to achieve this goal, as long as they are allocated on cache-sized memory regions. In particular, Manegold et al. [68] utilize a variant bucket chaining mechanism, where tuples are chained together via their starting offsets (in contrast to actual tuples or pointers). However, since tuples are only chained but not moved, this chaining mechanism only works for a partition of a contiguous memory layout. Additionally, Kim et al. [51] proposed a 2-pass hash table building method to make use of SIMD acceleration (“Histogram Mechanism” in Table 1). It first scans a partition to populate a hash value histogram. Then it uses the prefix sum of the histogram to redistribute tuples in the second pass. In this way, tuples with the same hash values are redistributed side-by-side, which supports SIMD lookups and thereby expedites the probing.

## 4 EXPERIMENTAL SETUP

**Testbed.** We conduct experiments on a dual-socket server machine with Linux kernel version 5.4.0-110. Each socket is equipped with an Intel Xeon Gold 6230 CPU with 20 physical cores, each of which consists of 2 logical cores (40 threads/socket). Each physical core has 32KB L1 data cache, 32KB L1 instruction cache, 1MB L2 cache, and shares 27.5MB L3 cache (last level cache) with the remaining cores in the socket. Besides, the L1 TLB capacity is 64 and 32 for 4KB-page and 2MB-page configuration respectively, and the L2 TLB entry number is 1536 for both page configurations.

The system contains 384GB DRAM and 1.5TB Optane DIMMs (2 socket  $\times$  6 channel  $\times$  128GB/DIMM). All SCM DIMMs run in app direct mode and are organized in an interleaved manner via DAX-mmap. Unless explicitly stated for NUMA effects evaluation, all memory accesses are restricted to the local socket by default.

**Workload.** We evaluate the aforementioned joins on a variety of binary-join workloads (cf. Table 3). By default, a single tuple is a 16-byte  $\langle \text{key}, \text{payload} \rangle$  pair, and both key and payload are 8-byte long. Following previous works [7, 8, 11], we set the build side cardinality (i.e.,  $|R|$ ) and the probe side cardinality (i.e.,  $|S|$ ) as  $16 \times 2^{20}$  and  $256 \times 2^{20}$  respectively. Thus, the size of the probe side is  $\times 16$  of the build side, which is a typical ratio in TPC-H benchmark [22]. Additionally, we alter the ratio to  $\times 4$  and  $\times 1$  by increasing the build side cardinality so that we are able to test the join performance with different size ratios.

By default, the build side and the probe side follow a primary-key-foreign-key (PK-FK) setting and both relations conform to the uniform join key distribution (“pkfk”). We also generate a skew workload (“zipf”) by altering the probe side distribution to zipfian with various skewnesses (Zipf factor  $\theta$ ). To further assess the join robustness, we synthesize a many-to-many join workload (“m2m”) by relaxing the PK-FK constraint to foreign-key-foreign-key (FK-FK) constraint, which allows duplicates in the build side and thereby intensifies hashing collisions.



**Table 3: List of Evaluated Workloads**

workload	pkfk	zipf	m2m	pyld	bln
$ R $	$16/64/256 \times 2^{20}$	$16 \times 2^{20}$	$16 \times 2^{20}$	$16 \times 2^{20}$	$1 \times 2^{30}$
$ S $	$256 \times 2^{20}$	$256 \times 2^{20}$	$256 \times 2^{20}$	$256 \times 2^{20}$	$16 \times 2^{30}$
tuple size	16B	16B	16B	16~512B	16B
distribution	uniform	zipfian	uniform	uniform	uniform
constraint	PK-FK	PK-FK	FK-FK	PK-FK	PK-FK
skewness ( $\theta$ )	—	{ 1.05, 1.25, 1.50, 1.75 }	—	—	—

<sup>1</sup> “PK-FK” signifies primary-key-foreign-key constraint;

<sup>2</sup> “FK-FK” denotes foreign-key-foreign-key constraint [71], i.e., many-to-many join.

Due to the limited capacity of DRAM, previous works [7, 8, 11, 60] only evaluate joins at million scale (mostly smaller than 5GB). However, SCM has much higher density than DRAM, the capacity limitation is no longer a concern. Hence, it is not only practical but also worthwhile to evaluate the joins in huge workloads. We synthesize two sorts of huge workloads for assessing join scalability:

- (1) “pyld”. We fix the cardinality of  $R$  and  $S$  but enlarge the payload size for every single tuple. The tuple size lies in the domain {16B, 32B, 64B, 128B, 256B, 512B}. Thus, the respective workload size (the sum of  $R$  and  $S$ ) ranges from 4.25GB to 136GB.
- (2) “bln”. In contrast, we keep the tuple size fixed but increase the cardinality to a billion scale (i.e.,  $|R| = 2^{30}$ ,  $|S| = 16 \times 2^{30}$ ), making the workload size 272GB in total.

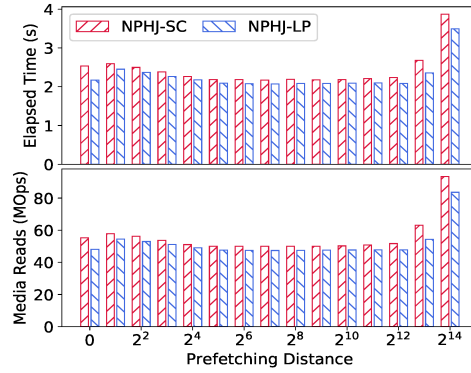
In the following experiment sections, we use the “pkfk” workload with a  $|R|:|S|$  ratio of 16 as the default workload to explore the design space for both NPHJ and PHJ (Sections 5). The remaining workloads will be evaluated rigorously in Section 6 for a fair comparison between NPHJ and PHJ.

**Implementation and Evaluation Metrics.** We implement all join algorithms listed in Table 1, and use GCC-9.3.0 to compile then with the -O3 flag enabled. Unlike existing works of persistent indices or crash recovery [35, 52, 59, 64, 98], a binary join has no need for immediate persistence, and benefits from regular stores (cf. Figure 1(a)(b)). We, thus, only issue regular stores (without cacheline flushes or memory fences) in our implementations unless otherwise stated. If not otherwise specified, we exploit all physical cores of a single socket <sup>7</sup> to run joins, which offers a favourable performance according to existing DRAM-based studies of main-memory hash joins [7, 11].

Following previous works of main-memory hash joins [7, 8, 11, 51, 60, 79], we conduct the binary join evaluation on relation  $R$  and  $S$  in the form of “SELECT COUNT (\*) FROM  $R, S$  WHERE  $R.key == S.key$ ”. We report the running elapsed time of each algorithm as the evaluation metric <sup>8</sup>, and the reported elapsed time is the median of ten consecutive runs. It is worth mentioning that before taking the ten measured runs, we warm up the SCM running pool, which is in line with previous works [8, 9, 46, 50, 56, 65, 86, 94]. Moreover, we pre-fault SCM mappings [2, 19, 46, 69] when allocating memory for hash tables or partitions, obliterating page faults in join execution. Furthermore, we instrument our studies with PAPI [87], VTune [39], and PMWatch [40] for hardware events measurement.

<sup>7</sup>By default, we only evaluate joins in one socket to avoid potential NUMA impacts.

<sup>8</sup>Previous works [7, 8, 11, 51, 60, 79] use “join throughput”, i.e.,  $\frac{|R|+|S|}{elapsed\ time}$ , for evaluation, which is equivalent to our metric, elapsed time.



**Figure 3: NPHJ runtime & SCM media reads w.r.t. prefetching distance (the distance refers to the number of tuples).**

## 5 A DESIGN SPACE EXPLORATION

We now perform a design space exploration for NPHJ and PHJ. For brevity, we only explore a few design dimensions in this section. A more detailed exploration can be found in our technical report [38]

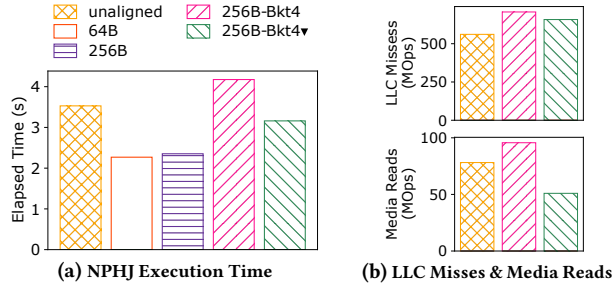
### 5.1 Non-Partitioned Hash Joins In SCM

We study NPHJ implementation in two aspects: 1) the benefit of prefetching; 2) the effect of bucket alignment. The main goal of this section is to uncover the most crucial factors that contribute to a performant NPHJ implementation.

**5.1.1 Prefetching.** We start our evaluation with the prefetching analysis. Prefetching has been shown to deliver impressive improvement for hash joins [5, 7, 15, 16]. It substantially alleviates the cache stall penalty by overlapping memory accesses with other computation instructions. The prefetched data is moved and retained in caches before its use, enhancing the cache hit rate and facilitating the join execution.

In order to parameterize the optimal prefetching distance in SCM, we assess the join performance with varying prefetching distances. Figure 3 shows the join execution time and SCM internal media read number with different prefetching distances. We can observe a strong correlation between execution time and SCM media reads. The performance first improves notably when the prefetching distance increases from 0 to  $2^4$ -tuple and then stabilizes with longer prefetching distances. Once the distance reaches  $2^{14}$ -tuple, the execution time and SCM media reads increase drastically. This phenomenon is primarily due to the limited capacity of SCM on-DIMM buffer [ $\mathcal{P}_2$ ]. A prefetching distance of  $2^{14}$ -tuple indicates that both prefetched tuples and hash buckets requires 1MB memory region (each prefetches  $2^{14}$  cachelines), which consumes 2MB space in total and exceeds the last level cache (LLC) size per core (LLC slice size [54]) <sup>9</sup>. The prefetched buckets and tuples, therefore, can no longer be buffered in LLC, rendering excessive repeated memory accesses. Moreover, they fail to reside in on-DIMM buffers either. The reason is two-fold: (1) The Optane on-DIMM buffer is believed to be 16KB [92, 95, 97], and the total on-DIMM buffers are 96KB in our platform (6 interleaved Optane DIMMs), which is far less than the size of the cache size per core. (2) The Optane on-DIMM buffer is believed to be exclusive with CPU

<sup>9</sup>LLC size per core is calculated as  $27.5MB \div 20 = 1.375MB$ .



**Figure 4: NPHJ execution time, LLC Misses and SCM Media Reads with different bucket configurations (“256B-Bkt4” refers to a 256B-alignment bucket setting with a 4-tuple capacity; “v” denotes that the join is executed with 7 threads).**

caches [95]: once a cacheline is loaded into caches, it is evicted immediately from on-DIMM buffers. Therefore, we can see a drastic rise in SCM media reads from the  $2^{14}$ -tuple prefetching distance and onwards, which exposes the long media access latency and impairs join execution.

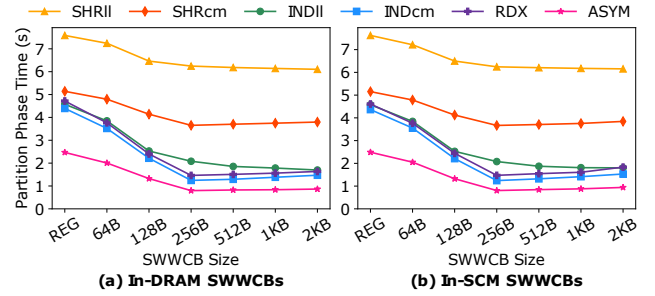
**Tip #1:** Employ prefetching for NPHJ but limit the prefetching distance within the LLC capacity. The prefetching distance can be accordingly increased if the SCM on-DIMM buffer capacity exceeds the LLC capacity or is inclusive with CPU caches [ $\mathcal{P}_2$ ].

**5.1.2 Bucket Alignment.** We now assess the impact of bucket alignment. By default, a hash bucket requires 48 bytes in our implementation<sup>10</sup>. A single hash bucket access may incur one additional memory access if the hash bucket spans two consecutive cachelines. Meanwhile, there exists a mismatch between CPU cacheline (64B) and SCM internal access granularity i.e., 256B XPLine of Optane DIMMs [ $\mathcal{P}_1$ ]. If the hash bucket spans two XPLines, the additional memory access can trigger one more SCM media read, exacerbating the bucket access overhead.

Bucket alignment aids in mitigating this issue of extra memory accesses. Bucket alignment can be set as 64B, which ensures each hash bucket be entirely stored in a single cacheline, and thereby precludes the extra memory accesses. Another rational alignment configuration is the internal granularity of SCM, i.e., 256B in our case, which eliminates the possibility of additional SCM media reads. Moreover, a 256B-aligned hash bucket is able to carry more tuples than a 64B bucket, which may also affect the NPHJ performance. With these considerations in mind, we test the performance of NPHJ with different bucket alignment configurations: unaligned, 64B-aligned, 256B-aligned, and a 256B-aligned bucket containing four tuples (denoted as “256B-Bkt4” in Figure 4).

Figure 4(a) depicts the outcomes of the experiments. As can be seen, the unaligned bucket yields an inferior result than 64B- or 256B-aligned hash bucket on account of excessive memory accesses. The 64B-aligned configuration slightly outperforms the 256B-aligned configuration because it has fewer SCM media reads. However, the “256B-Bkt4” configuration renders the worst result. In order to explicate the reason behind this phenomenon, we plot the number of LLC misses and SCM media reads in Figure 4(b). We

<sup>10</sup> A bucket comprises two 16-byte tuples, a 8-byte next pointer, a 4-byte counter, and a 1-byte latch with 3-byte padding.



**Figure 5: PHJ partitioning time using non-temporal stores with different SWWCB size (“REG” denotes the naive partitioning setting w/o “ntstore” and SWWCBs).**

can observe that, even with 256B alignment, “256B-Bkt4” still incurs more LLC misses and SCM media reads than the unaligned configuration, leading to inferior performance.

We claim that on-DIMM buffer contention is the main culprit behind this issue [ $\mathcal{P}_2$ ]. To corroborate this claim, we rerun NPHJ with “256B-Bkt4” configuration, but with only 7 running threads (cf. 20 threads of default setting), which is reflected as “256B-Bkt4v” in Figure 4. As shown in Figure 4(a), its runtime beats the unaligned configuration. We also notice that “256B-Bkt4v” significantly lowers SCM media reads (cf. Figure 4(b)). Recall that a 4-tuple hash bucket spans two consecutive cachelines. When accessing the hash bucket, the first cacheline is loaded to the caches while the second cacheline resides in on-DIMM buffers, waiting for the following read request. When executing a join with too many threads, these cachelines have to contend for the limited on-DIMM buffer space, resulting in notoriously XPLine thrashes in on-DIMM buffers. Fewer running threads, on the contrary, considerably alleviate the contention problem. Hence, the join performance can be ameliorated.

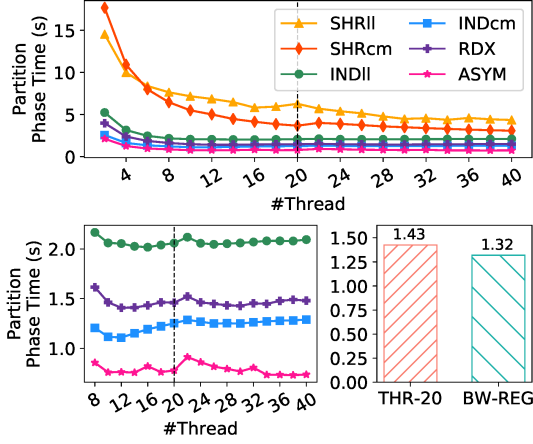
Figure 4(b) also reveals an essential finding of NPHJ in real SCM. “256B-Bkt4v” incurs fewer media reads but more LLC misses when compared to the unaligned configuration. Given the superior performance of “256B-Bkt4v”, we argue that SCM media reads are the most significant impediment for a high-performance join. One should avoid excessive media reads to the greatest extent possible; where required, trade SCM media reads for LLC misses. Overall, we maintain that the 64B-aligned bucket offers the optimal performance and provide the following configuration tip:

**Tip #2:** Align the hash buckets to the 64B-boundary for NPHJ. If the bucket size exceeds 64B, align it to the SCM internal granularity (e.g., 256B for Optane DIMM) and consider limiting the join parallelism if necessary [ $\mathcal{P}_1, \mathcal{P}_2$ ].

## 5.2 Partitioned Hash Joins In SCM

Likewise, we proceed to explore the design space for PHJ. As partitioning is well known to be the dominating factor in PHJ execution [4, 7, 79, 99], we focus on the PHJ partition phase and address two scaling effects in this subsection: 1) “ntstore” with SWWCBs; 2) thread scalability. We also investigate the effect of the partition fanout and examine the join phase implementation, but leave these details in our technical report [38] for the sake of brevity.

**5.2.1 Effect of SWWCB and “ntstore”.** We first scrutinize the effect of SWWCB in “ntstore”. Recall that “ntstore” delivers higher



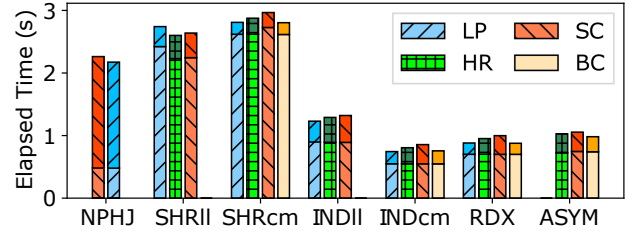
**Figure 6: Partitioning time w.r.t. thread number** (dashed lines mark the starting of simultaneous multi-threading (SMT); the bottom left figure zooms in the partitioning time of independent partitioning, radix partitioning, and asymmetric radix partitioning; the bottom right figures compares the runtime between the 20-thread partitioning (THR-20) and the bandwidth-regulation partitioning (BW-REG)).

write throughput with larger access size (cf. Figure 1(a)) and SWWCB reduces cache/TLB thrashes by combining  $N$ -tuple writes to one. We, thus, alter  $N$  value to 4~128 (SWWCB size varied from 64B to 2KB) and compare the partitioning runtime with naive setting (w/o “ntstore” and SWWCBs). We apply 2-pass partitioning to rule out potential TLB conflicts [5, 8, 11, 51] and allocate SWWCBs in SCM by default. Additionally, as DRAM has higher read performance than SCM, we conduct experiments with in-DRAM SWWCBs to see if there are any performance improvements.

Figure 5 presents the partitioning results. Compared to the naive setting (“REG” in Figure 5), all partitioning methods significantly benefit from “ntstore” and SWWCBs. The runtime scales down linearly and converges at 256B, which is equivalent to SCM internal granularity (i.e., XPLine size). As writes of this size can be directly flushed to SCM media, both read-modify-write in on-DIMM buffers and lousy write amplification in underlying media are appreciably alleviated, which accounts for the major reduction in partitioning runtime [ $\mathcal{P}_1$ ].

Figure 5 also shows that enlarging SWWCB brings no more performance gains. Although a larger SWWCB merges more writes into one and induces fewer cache/TLB thrashes, it does not affect the underlying SCM media write number. The phenomenon indicates that media-level access is more of a bottleneck than processor-level thrash, which again validates our finding in Section 5.1.2. Furthermore, in-DRAM SWWCBs do not benefit much from faster DRAM and only achieve similar results. The reason is two-fold: (1) “ntstore” retains tuples in caches, effectively mitigating cache pollution; (2) SWWCB groups  $N$  writes into one, reducing cache thrashes by a  $N$  factor. Hence, the DRAM’s superior read performance makes no difference, and we can perform a complete in-SCM partitioning without sacrificing performance.

**Tip #3:** Leverage “ntstore” and SWWCBs in partitioning and make SCM’s internal access granularity as the SWWCB size [ $\mathcal{P}_1$ ].



**Figure 7: Overall Comparison for NPHJ and PHJ** (lighter colors denote the build/partition phase while darker colors represent the probe/join phase).

**5.2.2 Effect of thread scalability.** SCM is widely reckoned to have write deficiency [24, 70, 97] [ $\mathcal{P}_3$ ] and PHJ partitioning involves intensive write operations. We, therefore, seek to cultivate an understanding of this write deficiency in PHJ partition phase.

We vary the thread count from 1 to 40 and employ “ntstore” with 256B-SWWCB for evaluation. Figure 6 presents the execution time. There are generally two trends in partitioning thread scalability: (1) Shared partitioning (SHRII and SHRCm) is highly scalable to the partitioning thread number. Share partitioning is majorly hindered by lock contention, its SCM bandwidth utilization is far from full. Thus, SCM’s limited write scalability brings no harm to shared partitioning. (2) The other partitioning methods exhibit a distinct scalability pattern. Their partitioning time drops at first and reaches a local minima at around 10~12 threads. From 14 threads onwards, the runtime rises gradually and finally converges to moderate values. Since there is no lock contention in these partitioning methods, the SCM bandwidth is exploited effectively, and the write deficiency in SCM is exposed thoroughly. In consequence, a sound configuration practice is to limit the parallelism for these independent partitioning (INDcm and INDII) and radix-based partitioning (RDX, ASYM).

As stated in Section 3.2.1, radix-based partitioning involves multiple partitioning passes, and every single pass consists of three steps (①, ②, and ③ in Figure 2(c)). Note that ① and ② are read and processing dominant respectively. They only issue write requests to in-cache intermediates (histograms), which incurs no memory writes if no persistent instructions are enforced [ $\mathcal{P}_5$ ]. As read and processing exhibit strong thread scalability, parallelism limitation can generate detrimental impacts. However, as ③ is write-intensive, it can benefit from parallelism limitation. Therefore, there exists a Pareto optimal threading configuration for radix-based partitioning. Given these facts, we employ a particularized bandwidth regulation mechanism to improve radix partitioning further. Specifically, we use all physical cores to process step ① and ② but limit the threading around 10~12 for ③. We can see from Figure 6, the bandwidth regulation introduces 7.7% performance gain (1.32s vs. 1.43s). Though this improvement is not substantial in our platform (Optane DIMMs), we expect it will introduce more positive boosts in future SCM technologies, especially for SCM with larger read/write performance gap (e.g., STT-MRAM [18, 24, 27]). Overall, we provide the following partitioning tip:

**Tip #4:** Exhaust all cores for shared partitioning but enforce parallelism limitation or bandwidth regulation for independent or radix partitioning [ $\mathcal{P}_3$ ].



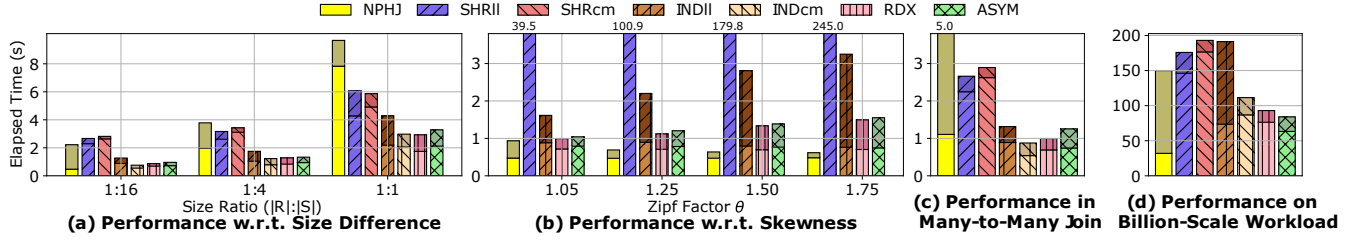


Figure 8: Join execution time across a wide range of workloads (lighter colors indicate the build/partition phase while darker colors represent the probe/join phase; as SHRII takes much longer time to complete the skewness test, we cut its bars in subfigure (b) and place its values on top of its bars).

**5.2.3 Other scaling effects.** We have also inspected the impact of the partition fanout and found that the partition fanout should be bounded by TLB limit or meticulously tailored to ensure that SWWCBS footprint is constrained within caches  $[\mathcal{P}_1, \mathcal{P}_3]$ . Besides, we examine the join phase thread scalability and discover that its throughput is highly scalable to the active physical cores  $[\mathcal{P}_3]$ . More details are available in our technical report [38].

### 5.3 Putting Everything Together

Having determined the optimal configuration for both PHJ and NPHJ, we now perform an overall comparison for all joins. In particular, we use “ntstore” with 256B-SWWCB and apply 1-pass partitioning to all PHJs (we use 2-1 pass for ASYM-\* joins), as they offer the highest write throughput in the current workload. For NPHJs, we employ a prefetching distance of  $2^4$ -tuple and align the hash bucket at 64B-boundary. Additionally, We customize each join’s ideal threading separately for each phase.

Figure 7 shows the comparison result. In general, PHJs vary a lot in performance, whereas NPHJs only obtain intermediate results. Specifically, NPHJs spend 80% runtime in the probe phase, revealing the hefty cost from random SCM accesses. Moreover, we notice that NPHJ-SC and NPHJ-LP attain comparable performance. Although their hashing schemes differ, the prefetching mechanism substantially mitigates this disparity in performance.

As for PHJs, the partition phase takes up the most time in execution. Shared partitioning joins (SHRII-\* and SHRCm-\*) lag behind others by a large margin, primarily because of their heavy lock contention. Independent partitioning joins deliver a good performance, especially INDcm-\* joins, which outperform all other joins. This superiority is mainly the result of fewer reads in partitioning and a contiguous memory layout that trades random reads for sequential reads. RDX-\* joins achieve comparable performance, even though they involve more reads in partitioning. The reason is that SCM is more susceptible to writes than reads, which weakens the impact of additional reads. Although ASYM-\* joins have one more partitioning pass over  $R$ , their performances are close to RDX-\* joins’. Recall that  $|R|$  is  $\frac{1}{16}$  of  $|S|$ , the second partitioning pass only imposes insignificant cost in partitioning. However, it incurs more pronounced overhead in the join phase. In ASYM-\* joins, each  $|S|$  partition will be processed  $k$  times. Since  $|S|$  is usually larger than  $|R|$ , the  $k$  times processing overhead can be significant. We will discuss ASYM-\* more in-depth in Section 7.2. As for the join phase, bucket chaining generally offers optimal performance. Therefore,

until otherwise stated, we leverage bucket chaining as the default PHJ hashing scheme in the following evaluation.

## 6 A RIGOROUS EVALUATION

After a design space exploration for NPHJ and PHJ, we now conduct a comprehensive evaluation for these joins in a wide range of workloads (Table 3). Additionally, we conduct many experiments to study the joins concerning NUMA, store instructions, page faults, etc., and evaluate joins in TPC-H[22], all of which are included in our technical report [38] for the space constraint.

Section 5.2 has shown that the partition phase in PHJs dominates the performance, and different hashing schemes do not substantially change the total execution time. Thus, to omit space, we only present the PHJ result with bucket chaining and take partitioning notations to represent the respective PHJs. Note that the linked list partition layout does not support bucket chaining (Section 3.2.2); we use Kim’s histogram mechanism [51] (HM) instead for it leads to solid and robust performance. Similarly, we take separate chaining as NPHJ hashing scheme. We apply all proposed implementation tips, and tune each join to its optimal configuration in respective workloads.

### 6.1 Effect of Size Difference

The previous “exploration” is conducted in a workload with the  $|R|:|S|$  ratio of 1:16. A larger size ratio incurs more writes for  $R$ , which will affect the join performance terribly in write-susceptible SCM. Thus, we raise the size ratio to 1:4 and 1:1 (cf. “pkfk” in Table 3), and report the result in Figure 8(a).

In general, the runtime increases for all joins as the size ratio grows. Specifically, NPHJ deteriorates most badly. Its build phase consumes 20% of the execution time when the size ratio is 1:16, but 80% when the ratio rises to 1:1. PHJs also endure a rising execution overhead but constantly maintain an advantage over NPHJ. As the size ratio approaches 1:1, the advantage becomes increasingly notable. SHRII and SHRCm, which lose to NPHJ at 1:16 size ratio, even surpass NPHJ from 1:4 ratio and onwards. This advantage derives from the higher throughput of “ntstore” and 256B-SWWCB. “ntstore” with 256B-SWWCB transmits data directly to the underlying SCM media, bypassing cache pollution and write amplification. However, NPHJ building writes at tuple-granularity (16B), resulting in read-modify-write and write amplification. As a result, the precious SCM bandwidth is wasted during the build phase, and NPHJ falls further behind PHJs for larger size ratios.

## 6.2 Effect of Skewness

We now evaluate joins in skew workloads (cf. “zipf” in Table 3). We vary the skewness (Zipf factor  $\theta$  [7, 8]) and plot the experimental results in Figure 8(b). Recall that the contiguous memory layout is not applicable to a skew workload (Section 3.2), we thus omit SHRCm and INDCm in this assessment.

As is evident from the figure, NPHJ benefits more from a high skew workload while PHJs degenerate. As indicated before [11], a high skew workload has better spatial and temporal locality, significantly reducing cache misses in the probing phase, which contributes to the NPHJ’s growing advantage over PHJs. However, the increasing locality induces adverse effects on PHJ’s join phase. As the data become more skew, the sizes of generated partitions will be highly imbalanced, which distributes the join works unevenly among all join threads and increase the cost of synchronization. Besides, SHRLl performs considerably worse than others because its lock contention issue becomes more intense in skew workloads.

## 6.3 Many-to-Many Join Performance

Previous works mainly focus on the **PK-FK** setting [7, 8, 11]. However, many-to-many joins (**FK-FK**) are also common in real-world queries. Hence, we proceed to evaluate joins in **FK-FK** workloads.

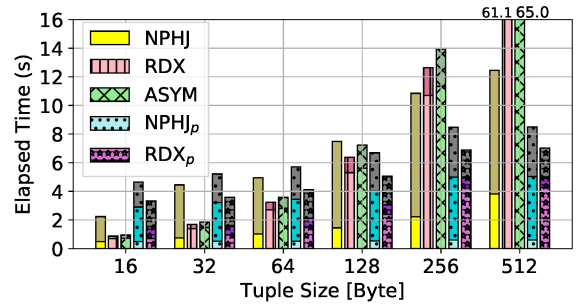
Figure 8(c) reports the experimental result. A significant runtime increase can be observed in all joins’ execution, especially NPHJ, whose higher runtime is derived from its longer probing phase. A single probing, therefore, has to visit multiple hash table entries to retrieve its potential matches. Moreover, If a hash bucket is affiliated with a long linked list, the probing must visit several separate memory pages, leading to more random SCM accesses and inducing more cache miss penalties. PHJs, however, are far less impacted. As mentioned earlier, their joining phase is executed within cache-sized partitions. The increasing hashing collision only brings about marginal overhead. Consequently, PHJs are the preferable solutions for many-to-many joins.

## 6.4 Performance in Billion-Scale Workloads

Due to the limited capacity of DRAM, existing works can only study main-memory hash joins in million-scale workloads, i.e., relations with million-scale cardinality. Since SCM can offer much denser capacity than DRAM, we are now capable of conducting a billion-scale study. We increase the cardinality to billion-scale (“bln”) and plot the evaluation result in Figure 8(d). Note that the billion-scale workload demands the re-configuration for fanouts and partitioning passes. Hence, we tune each PHJ accordingly and report their optimal performance<sup>11</sup>.

Compared with the results of million-scale experiments (Figure 7), we can see a notable time explosion for all joins. The relative performance of NPHJ is consistent with its million-scale result (Section 5.3). The build phase still accounts for 20% of the total execution time, which indicates that NPHJ’s performance is scalable to the relation cardinality.

PHJs, however, show a different view of relative performance, which is mainly due to the partition fanout re-configuration. On the one hand, a large fanout shrinks the size of each partition, precluding cache thrashes for the subsequent join phase. On the



**Figure 9: Performance w.r.t. Tuple Size** (lighter colors denote the build/partition phase while darker colors indicate the probe/join phase; NPHJ<sub>p</sub> and RDX<sub>p</sub> represent the pointer-based variant of NPHJ and RDX, and their upper gray bars denote the time of the final retrieving phase).

other hand, a large fanout exceeds the cache or TLB limit, which renders enormous cache/TLB thrashes and harms the partitioning efficiency. Thus, we see a relative performance decline in INDLl, INDCm, and RDX. Meanwhile, ASYM achieves the best result because of its 2-1 pass partitioning pattern. The reason is two-fold: (1) The 2-1 pass pattern splits the large fanout accordingly over 2 passes, preventing the first partitioning pass from sustaining excessive cache/TLB thrashes. (2) The second partitioning is only applied on the small  $R$ , which not only saves the huge re-partitioning cost of the large  $S$ , but also generates small enough  $R$  partitions for populating cache-sized hash tables. Although these asymmetric fanouts require more read passes over  $S$  in the join phase (Section 3.2.2), the saving cost from the partition phase still pays off, which makes ASYM succeed in the billion-scale workload. We will elucidate ASYM’s pros and cons more thoroughly in Section 7.2.

## 6.5 Performance with Large-Size Payloads

Aside from cardinality, large-size payloads also affect the workload size considerably. To assess the payload impact, we vary the payload size from 16B to 512B while fixing the relation cardinality. Similarly, all joins are carefully tailored and are compared with their optimal configurations.

Before we analyze the experiment, we briefly digress to describe a pointer-based version of join implementation. Since SCM supports byte-addressability, in-SCM tuples can be accessed with pointer-indirection. Instead of directly manipulating the full tuples, a join can be conducted by processing  $\langle key, pointer \rangle$  pairs, which considerably saves the bytes read/written in execution. In order to retrieve the join results, the pointer-based implementation requires an additional round of random reads to retrieve the tuples via pointer-indirection. In a nutshell, pointer-based implementation makes a trade-off between processing and retrieval.

Figure 9 depicts the result. For brevity, we only present results of NPHJ, RDX, ASYM, and two pointer-based implementations, NPHJ<sub>p</sub> with RDX<sub>p</sub>. Other PHJs exhibit similar performance trends but deliver suboptimal results. We make the following observations. First, PHJs (RDX, ASYM) beat NPHJ with small payloads but lose to NPHJ with large payloads (256B and onwards). This is because large payloads raise the partitioning overhead, and the join phase gains will soon be reduced. Second, pointer-based implementations perform poorly with small-size payloads. However, they

<sup>11</sup>The 1-pass partitioning still overcomes the 2-pass partitioning for all PHJs

outperform others with larger payloads, especially  $RDX_p$ , which dominates from 128B-payload onwards. The reason behind this success is the colossal partitioning saving by using  $\langle key, pointer \rangle$  pairs. The saving becomes more and more pronounced when payloads get larger and larger, which renders a broader winning margin for pointer-based versions. Third,  $NPHJ_p$ 's retrieving time keeps growing with increasing payload size, while  $RDX_p$ 's retrieving time remains almost constant across all sizes. This is because  $NPHJ_p$  stores the intermediate join result ( $\langle key, pointer \rangle$  pairs) randomly. The increasing payload size raises the amount of random reads, impairing the retrieval performance.  $RDX_p$ , however, stores the intermediate join result partition-wise. If a tuple is joined multiple times, it will only be called when retrieving its partition. Moreover, as long as the partition is well cache-sized, the tuple will reside in caches all the time until another partition retrieval starts. Thus, excessive cache misses can be eliminated in the retrieving phase. Overall, PHJs, or their pointer-based implementations, are better solutions for various payload sizes.

## 7 DISCUSSION

We now summarize our experiment findings and bring about a few auxiliary discussions for SCM-based joins.

### 7.1 Locality is All You Need

Previous experiments (Section 6) suggest that PHJs are generally better than NPHJs (PHJs also prevail in other experiments; see our technical report [38] for more details). By partitioning, PHJs arrange arbitrary distributed relations into a set of high-locality sub-relations, which buffers the following hash table accesses within caches, and thereby drastically reduces the expensive SCM accesses. Despite the fact that partitioning entails penalties from additional relation passes, its ensuing high locality provides significant performance gains in the subsequent join phase. Even though SCM offers slower reads/writes than DRAM/SRAM, the gain-over-penalty does not compromise. Hence, the preliminary partition phase is well worth a shot.

In contrast, though NPHJs incur fewer read/write passes, they fail to yield such locality, and hence suffer from massive random SCM accesses during execution. A notable exception is the skew workload (Section 6.2), in which NPHJs surpass PHJs. However, this is because a skew workload exhibits a high locality inherently, which prevents NPHJ probing from random SCM accesses and thereby makes PHJ partitions redundant. As a consequence, this performance exception confirms the effectiveness of high locality.

Thanks to the high locality of the compact partition layout,  $RDX$ ,  $ASYM$ , and  $INDcm$  generally deliver better performances among all PHJs (Section 5.3). Moreover, the superiority of bucket chaining (BC), histogram mechanism (HR), and linear probing (LP) against separate chaining (SC) also validates the efficacy of high locality (Section 5.3). In a nutshell, high locality is the dominating factor that contributes to an efficient SCM-based join implementation.

### 7.2 Read/Write Asymmetry in PHJ

Since read/write asymmetry  $[\mathcal{P}_3]$  is widely acknowledged as an inherent SCM primitive [24, 70], write-limited algorithms [26, 61, 89,

90] have become a principle guideline for performance improvements. However, as Section 6 reports,  $ASYM$  joins, which save writes by reducing  $S$  partitioning passes, do not always render minimal runtime. Hence, we now give an in-depth analysis to determine the predominating conditions for  $ASYM$  joins.

We take  $m$ -pass  $RDX$  and  $m$ -pass  $ASYM$  for comparison, as they deliver not only magnificent performance but also great applicability for wide-ranging workloads<sup>12</sup>. Let  $read$  and  $write$  be the SCM bandwidth of read and write respectively, and  $\lambda$  denotes  $\frac{read}{write}$ . Following the pass number in Table 2,  $m$ -pass  $RDX$  has a cost model of:  $\frac{(2m+1)(R+S)}{read} + \frac{m(R+S)}{write}$ . Similarly, we can derive a cost model for  $m$ -pass  $ASYM$  if  $R$  partition number is  $k$  times of  $S$ :  $\frac{(2m+1)R+(2n+k)S}{read} + \frac{mR+nS}{write}$ . We further assume that  $S$  is  $x$  times of  $R$  and refer to the cost ratio of  $RDX$  over  $ASYM$  as  $\epsilon$ . Given that 2-pass  $RDX$  is able to partition a 4TB relation without thrashing cache/TLB, we set  $m = 2$ ,  $n = 1$  and derive the following function:

$$\epsilon = f(k, x, \lambda) = \frac{(x+1)(2\lambda+5)}{(x+2)\lambda + (k+2)x+5}. \quad (1)$$

In our platform,  $\lambda$  is close to 4.36<sup>13</sup> if partitioning is properly configured without cache/TLB thrashes. We thus parameterize  $f(k, x, 4.36)$  on  $k$  and  $x$  in Figure 10(a). As can be seen, the cost ratio  $\epsilon$  ranges from 0.50 to 1.50, and it gets higher when  $x$  becomes larger and  $k$  gets smaller. In particular, for  $k \leq 4$ ,  $\epsilon$  becomes larger than 1, indicating that  $ASYM$  starts to surpass  $RDX$ . The  $\epsilon$  is close to 1.50 for  $x \geq 4$ , which suggests that  $ASYM$  is at least 50% better than  $RDX$ . We thus conclude that 2-1-pass  $ASYM$  should be applied on a workload with a large size difference ( $x \geq 4$ ) and parameter  $k$  should be limited within 4.

In order to validate the above cost model, we synthesize a microbenchmark, with cardinality ranges from 64 to 16384 million (5~320GB) and size ratio  $x$  within 4~32. Figure 10(b) compares the results between 2-pass  $RDX$  ( $RDX-2$ ) and 2-1-pass  $ASYM$  ( $ASYM$ ). We also plot the 1-pass  $RDX$  result ( $RDX-1$ ) for a comprehensive comparison. As expected,  $RDX-1$  is competitive for small cardinalities whereas  $RDX-2$  dominates in large-scale datasets, which validates the efficacy of multi-pass partitioning. Meanwhile,  $ASYM$  generally offers robust and competitive results across all workloads. In particular, it beats  $RDX-2$  by a large margin, except that cache/TLB excessively thrash in  $S$  partitioning ( $|R| \geq 2048M$ ), which corroborates the correctness of our cost ratio function. Also,  $ASYM$  delivers comparable results as  $RDX-1$  in small-scale datasets and prevails when  $|R| \geq 256M$ , which confirms its moderate write cost in producing cache-sized partitions. Specifically, we notice that  $ASYM$  achieves the best result when  $256M \leq |R| \leq 1024M$  and  $16 \leq x \leq 32$ . Given that most large-scale queries fit in this size ratio range [8, 22, 73] and can be reduced to this scale by selection pushdown [71, 79], we maintain that  $ASYM$  can be incorporated to query plans for upcoming SCM-based DBMSs.

Since other SCM may have disparate read/write asymmetries (i.e.,  $\lambda$ ) [24], we derive the partial derivative of  $\epsilon$  with respect to  $\lambda$ :

$$\frac{\partial \epsilon}{\partial \lambda} = \frac{\partial f(k, x, \lambda)}{\partial \lambda} = \frac{(2k-1)(x+1)x}{[(x+2)\lambda + (k+2)x+5]^2}. \quad (2)$$

<sup>12</sup> $INDcm$  joins require larger memory footprint and are not applicable to skew workloads (Section 3.2.1).

<sup>13</sup> $read$  is 2.31GT/s while  $write$  is 0.53GT/s, where GT/s denotes the Giga tuples per second.

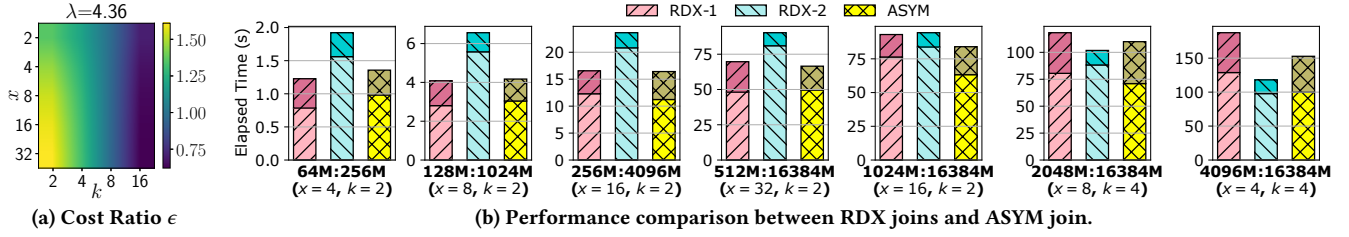


Figure 10: (a) is annotated with the cost ratio  $\epsilon$  of 2-pass RDX over 2-1-pass ASYM (a lighter shade denotes that ASYM is far better than RDX while a darker shade indicates the opposite). (b) Join execution time for 1-pass RDX (RDX-1), 2-pass RDX (RDX-2), and 2-1-pass ASYM (ASYM) with varying cardinalities ( $|R|:|S|$ ) and size ratio  $x$  (lighter colors represents the partition phase while darker colors stands for the join phase).

Given that  $k > 1$  forever holds, the above partial derivative is always positive. Therefore, for SCM with larger read/write asymmetry, ASYM will render more performance gains over RDX<sup>14</sup>.

### 7.3 Future SCM and Beyond

Through extensive experiments, we conclude that PHJ is generally the better solution (Section 6). We also provide practical tips (Section 5) for configuring efficient join implementations. Unfortunately, Intel shuts down the Optane business [41] out of financial issues, so it is natural to question the value of our conclusions. We, however, do not reckon that this marks the end of SCM and maintain that our study will remain valuable for the following reasons.

First, SCMs are inevitable. As SCM technology is initially proposed to break the DRAM scaling wall [37, 77], its necessity is not going to die. Meanwhile, SRAM and flash have their own scaling challenges, which can also be resolved by deploying SCM in various storage tiers [21, 70]. As widespread deployment leads to high production volume and high volume drives down the production cost [21], Optane’s financial concerns are expected to be resolved.

Second, our tips in Section 5 rely on the primitives of NVDIMM-P, a predominant SCM standard that future manufacturers will likely adhere to. While some primitives may vary in SCM prototypes (e.g., different internal granularity  $[\mathcal{P}_1]$ ), it is simple to accordingly adjust our tips to function in these devices.

Third, future SCMs will probably conform to the emerging CXL standard [20], which sacrifices access latency to avoid bandwidth contention [47]. On this account, the latency-bound NPHJs [8] are likely to be more bottlenecked. Hence, the PHJ-over-NPHJ conclusion is unlikely to fade away.

## 8 RELATED WORK

**Main-Memory Hash Joins.** Main-memory hash joins have been rigorously studied for almost thirty years. Shatdal et al. [83] open up the research of PHJ. They note that the cache miss penalty accounts for most join overhead, and partitioning can help reduce this overhead considerably. Subsequently, Boncz et al. [12, 67, 68] confirm this idea and add that TLB thrashes impair partitioning terribly. The partitioning, therefore, should be done in a multi-pass manner where every pass fanout should not exceed the TLB capacity. Follow-up works [4–7, 51] extend their idea to parallel query

processing and develop a performant PHJ implementation. Meanwhile, Blanas et al. [11] argue that modern hardware effectively conceal the cache miss overhead, which makes partitioning unnecessary and leads NPHJ in beating PHJ. Afterward, Schuh et al. [79] compare PHJ with NPHJ in microbenchmarks and proclaim that PHJ generally outperforms NPHJ. However, Bandle et al. [8] later conduct an evaluation in TPC-H [22] and show that NPHJ is a better solution. As a result, the PHJ-vs-NPHJ debate is still ongoing.

**SCM System Studies.** Since the commercialization of Optane DIMMs, numerous studies have been conducted to study its impact in various research fields. Several works [10, 23, 31, 88, 92, 95, 97] characterize its access profile, providing a few practices for better utilizing the hardware. Some other researchers develop SCM-friendly data structures [34, 36, 52, 63], which exploit SCM’s non-volatility for fast recovery. Other studies focus more on general SCM technologies rather than Optane DIMMs; they mostly follow the NVDIMM-P specification and propose designs for logging [3], file system [96], memory security [32], etc.

Unfortunately, few efforts have been made for SCM-based join processing. Viglas [89] first studies the read/write asymmetry impact in join and Shanbhag et al. [81] revisit his findings in Optane DIMMs. Besides, Daase et al. [23] and Lasch et al. [56] re-examine query benchmarks in Optane DIMMs. Nonetheless, they target traditional external joins, failing to exploit the SCM’s byte-addressability in join processing. Maltenberger et al. [66] take the advantage of byte-addressability and evaluate main-memory hash joins in SCM recently. However, they overlook a few SCM primitives and end up with a misleading conclusion, which our experimental study seeks to rectify.

## 9 CONCLUSION

This paper revisits main-memory hash joins in SCM. In particular, we explore the design space for PHJ and NPHJ and provide a few tips for a performant join implementation. Through a comprehensive evaluation, we demonstrate that PHJ is generally the preferred solution in SCM. Our study, along with discussions, are not limited to current SCM hardware. They can be easily extended and applied to future NVDIMM-P SCM technologies and beyond.

## ACKNOWLEDGMENTS

This project is partially supported by a grant funded by the Ministry of Education (Title: inPMdb: An in-Persistent Memory Database System; WBS No: A8000082-00-00) and Shanghai Engineering Research Center of Big Data Management.

<sup>14</sup>We also have the same conclusion for 1-pass RDX and ASYM but omit the function/derivative for brevity.



## REFERENCES

- [1] AgigaTech. 2022. AGIGARAM® NVDIMM-N. <http://agigatech.com/products/agigaram-nvdimms/>
- [2] Chloe Alverti, Vasileios Karakostas, Nikhita Kunati, Georgios Goumas, and Michael Swift. 2022. DaxVM: Stressing the Limits of Memory as a File Interface. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 369–387.
- [3] Joy Arulraj, Matthew Perron, and Andrew Pavlo. 2016. Write-Behind Logging. *Proc. VLDB Endow.* 10, 4 (2016), 337–348.
- [4] 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 (2013), 85–96.
- [5] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2012. Main-memory hash joins on multi-core CPUs: Tuning to the underlying hardware [Technical Report]. <ftp://ftp.inf.ethz.ch/pub/publications/tech-reports/7xx/779.pdf>
- [6] 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.
- [7] Cagri Balkesen, Jens Teubner, Gustavo Alonso, and M. Tamer Özsu. 2015. Main-Memory Hash Joins on Modern Processor Architectures. *IEEE Trans. Knowl. Data Eng.* 27, 7 (2015), 1754–1766.
- [8] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To Partition, or Not to Partition, That is the Join Question in a Real System. In *SIGMOD Conference*. ACM, 168–180.
- [9] Ronald Barber, Guy M. Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi K. Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. 2014. Memory-Efficient Hash Joins. *Proc. VLDB Endow.* 8, 4 (2014), 353–364.
- [10] Lawrence Benson, Leon Papke, and Tilmann Rabl. 2022. PerMA-Bench: Benchmarking Persistent Memory Access. *Proc. VLDB Endow.* 15, 11 (2022), 2463–2476.
- [11] Spyros Blanas, Yanan Li, and Jignesh M. Patel. 2011. Design and evaluation of main memory hash join algorithms for multi-core CPUs. In *SIGMOD Conference*. ACM, 37–48.
- [12] Peter A. Boncz, Stefan Manegold, and Martin L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *VLDB. Morgan Kaufmann*, 54–65.
- [13] Wentao Cai, Haosen Wen, H. Alan Beadle, Chris Kjellqvist, Mohammad Hedayati, and Michael L. Scott. 2020. Understanding and optimizing persistent memory allocation. In *ISMM*. ACM, 60–73.
- [14] Daniel Castro, Alexandro Baldassin, João Barreto, and Paolo Romano. 2021. SPHT: Scalable Persistent Hardware Transactions. In *FAST*. USENIX Association, 155–169.
- [15] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2004. Improving Hash Join Performance through Prefetching. In *ICDE*. IEEE Computer Society, 116–127.
- [16] Shimin Chen, Anastassia Ailamaki, Phillip B. Gibbons, and Todd C. Mowry. 2007. Improving hash join performance through prefetching. *ACM Trans. Database Syst.* 32, 3 (2007), 17.
- [17] Ping Chi, Shuangchen Li, Yuanqing Cheng, Yu Lu, Seung H Kang, and Yuan Xie. 2016. Architecture design with STT-RAM: Opportunities and challenges. In *2016 21st Asia and South Pacific design automation conference (ASP-DAC)*. IEEE, 109–114.
- [18] Yu-Der Chih, Yi-Chun Shih, Chia-Fu Lee, Yen-An Chang, Po-Hao Lee, Hon-Jarn Lin, Yu-Lin Chen, Chieh-Pu Lo, Meng-Chun Shih, Kuei-Hung Shen, et al. 2020. 13.3 a 22nm 32Mb embedded STT-MRAM with 10ns read speed, 1M cycle write endurance, 10 years retention at 150 c and high immunity to magnetic field interference. In *2020 IEEE International Solid-State Circuits Conference (ISSCC)*. IEEE, 222–224.
- [19] Jungsik Choi, Jiwon Kim, and Hwansoo Han. 2017. Efficient Memory Mapped File I/O for In-Memory File Systems. In *HotStorage*. USENIX Association.
- [20] CXL Consortium. 2022. Compute Express Link (CXL) Specification. [https://www.computeexpresslink.org/\\_files/ugd/0c1418\\_1798ce97c1e6438fba818d760905e43a.pdf](https://www.computeexpresslink.org/_files/ugd/0c1418_1798ce97c1e6438fba818d760905e43a.pdf)
- [21] Thomas Coughlin and Objective Analysis Jim Handy. 2022. Persistent Memories: Without Optane, Where Would We Be? <https://storagedeveloper.org/events/sdc-2022/agenda/session/324>
- [22] Transaction Processing Performance Council. 2021. TPC BENCHMARK H (Decision Support) Standard Specification Revision 3.0.0. [https://www.tpc.org/tpc\\_documents\\_current\\_versions/pdf/tpc-h\\_v3.0.0.pdf](https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf)
- [23] Björn Daase, Lars Jonas Bollmeier, Lawrence Benson, and Tilmann Rabl. 2021. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *SIGMOD Conference*. ACM, 339–351.
- [24] Tim Daulby, Anand Savanth, Alex S. Weddell, and Geoff V. Merrett. 2020. Comparing NVM Technologies through the Lens of Intermittent Computation. In *ENSys@SenSys*. ACM, 77–78.
- [25] CA de Araujo, Jolanta Celinska, Chris R McWilliams, Lucian Shifren, Greg Yeric, XM Huang, Saurabh Vinayak Suryavanshi, Glen Rosendale, Valeri Afanas'ev, Eduardo C Marino, et al. 2022. Universal Non-Polar Switching in Carbon-doped Transition Metal Oxides (TMOs) and Post TMOs. *arXiv preprint arXiv:2204.07656* (2022).
- [26] Laxman Dhulipala, Charles McGuffey, Hongbo Kang, Yan Gu, Guy E. Blelloch, Phillip B. Gibbons, and Julian Shun. 2020. Sage: Parallel Semi-Asymmetric Graph Algorithms for NVRAMs. *Proc. VLDB Endow.* 13, 9 (2020), 1598–1613.
- [27] Ivan Fernandez, Aditya Manglik, Christina Giannoula, Ricardo Quislan, Nika Mansouri-Ghiasi, Juan Gómez-Luna, Eladio Gutiérrez, Oscar G. Plata, and Onur Mutlu. 2022. Accelerating Time Series Analysis via Processing using Non-Volatile Memories. *CoRR abs/2211.04369* (2022).
- [28] Bill Gervasi. 2019. Will Carbon Nanotube Memory Replace DRAM? *IEEE Micro* 39, 2 (2019), 45–51.
- [29] Seyed Ali Ghasemi, Belal Jahannia, and Hamed Farbeh. 2022. GraphA: An efficient ReRAM-based architecture to accelerate large scale graph processing. *Journal of Systems Architecture* (2022), 102755.
- [30] Caixin Gong, Chengjin Tian, Zhengheng Wang, Sheng Wang, Xiyu Wang, Qilei Fu, Wu Qin, Qian Long, Rui Chen, Jiang Qi, Ruo Wang, Guoyun Zhu, Chenghu Yang, Wei Zhang, and Feifei Li. 2022. Tair-PMem: a Fully Durable Non-Volatile Memory Database. *Proc. VLDB Endow.* 15, 12 (2022), 3346–3358.
- [31] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the Idiosyncrasies of Real Persistent Memory. *Proc. VLDB Endow.* 14, 4 (2020), 626–639.
- [32] Xijing Han, James Tuck, and Amro Awad. 2022. Horus: Persistent Security for Extended Persistence-Domain Memory Systems. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1255–1269.
- [33] Preetam Hazra and KB Jinesh. 2018. Scaling of resistive random access memory devices beyond 100 nm2: influence of grain boundaries studied using scanning tunneling microscopy. *Nanotechnology* 29, 49 (2018), 495202.
- [34] Yuliang He, Duo Lu, Kaisong Huang, and Tianzheng Wang. 2022. Evaluating Persistent Memory Range Indexes: Part Two. *Proc. VLDB Endow.* 15, 11 (2022), 2477–2490.
- [35] Daokun Hu, Zhiwen Chen, Wenkui Che, Jianhua Sun, and Hao Chen. 2022. Halo: A Hybrid PMem-DRAM Persistent Hash Index with Fast Recovery. In *SIGMOD Conference*. ACM, 1049–1063.
- [36] Daokun Hu, Zhiwen Chen, Jianbing Wu, Jianhua Sun, and Hao Chen. 2021. Persistent Memory Hash Indexes: An Experimental Evaluation. *Proc. VLDB Endow.* 14, 5 (2021), 785–798.
- [37] Kaisong Huang, Yuliang He, and Tianzheng Wang. 2022. The Past, Present and Future of Indexing on Persistent Memory. *Proc. VLDB Endow.* 15, 12 (2022), 3774–3777.
- [38] Wentao Huang, Yunhong Ji, Xuan Zhou, Bingsheng He, and Kian-Lee Tan. 2022. A Design Space Exploration and Evaluation for Main-Memory Hash Joins in Storage Class Memory [Technical Report]. <https://www.comp.nus.edu.sg/~huang/assets/works/VLDB-2023/hashjoin-scm/main-tr.pdf>
- [39] Intel. 2014. Intel® VTune Profiler. <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>
- [40] Intel. 2020. Intel® PMWatch. <https://github.com/intel/intel-pmwatch/>
- [41] Intel. 2022. Intel Reports Second-Quarter 2022 Financial Results. <https://www.intc.com/news-events/press-releases/detail/1563/intel-reports-second-quarter-2022-financial-results?text=Second%2Dquarter%20GAAP%20revenue%20of%20billion%2C%20down%2017%25%20YoY.&text=Intel's%20Client%20Computing%20and%20Datacenter,Mobility%20achieved%20record%20quarterly%20revenue>
- [42] Intel. 2022. Intel® Optane™ Persistent Memory. <https://www.intel.sg/content/www/xa/en/architecture-and-technology/optane-dc-persistent-memory.html>
- [43] JEDEC. 2018. DDR4 NVDIMM-N DESIGN SPECIFICATION. <https://www.jedec.org/standards-documents/docs/jesd248>
- [44] JEDEC. 2021. DDR4 NVDIMM-P BUS PROTOCOL. <https://www.jedec.org/system/files/docs/JESD304-4-01.pdf>
- [45] Rakesh Gnana David Jeyasingh, Jiale Liang, Marissa Caldwell, Duygu Kuzum, and H.-S. Philip Wong. 2012. Phase Change Memory: Scaling and applications. In *CICC*. IEEE, 1–7.
- [46] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. 2019. SplitFS: reducing software overhead in file systems for persistent memory. In *SOSP*. ACM, 494–508.
- [47] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald G. Dreslinski. 2021. Improving Performance of Flash Based Key-Value Stores Using Storage Class Memory as a Volatile Memory Extension. In *2021 USENIX Annual Technical Conference, USENIX ATC 2021, July 14-16, 2021, Irina Calciu and Geoff Kuenning (Eds.)*. USENIX Association, 821–837. <https://www.usenix.org/conference/atc21/presentation/kassa>
- [48] Hiwot Tadese Kassa, Jason Akers, Mrinmoy Ghosh, Zhichao Cao, Vaibhav Gogte, and Ronald G. Dreslinski. 2022. Power-optimized Deployment of Key-value Stores Using Storage Class Memory. *ACM Trans. Storage* 18, 2 (2022),

13:1–13:26.

- [49] Omar Khattab, Mohammad Hammoud, and Omar Shekfeh. 2018. PolyHJ: A Polymorphic Main-Memory Hash Join Paradigm for Multi-Core Machines. In *CIKM*. ACM, 1323–1332.
- [50] Ana Khorguani, Thomas Ropars, and Noel De Palma. 2022. ResPCT: fast checkpointing in non-volatile memory for multi-threaded applications. In *EuroSys*. ACM, 525–540.
- [51] Changkyu Kim, Eric Sedlar, Jatin Chhugani, Tim Kaldewey, Anthony D. Nguyen, Andrea Di Blas, Victor W. Lee, Nadathur Satish, and Pradeep Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proc. VLDB Endow.* 2, 2 (2009), 1378–1389.
- [52] Wook-Hee Kim, Madhava Krishnan Ramanathan, Xinwei Fu, Sanidhya Kashyap, and Changwoo Min. 2021. PACTree: A High Performance Persistent Range Index Using PAC Guidelines. In *SOSP*. ACM, 424–439.
- [53] Masaru Kitsuregawa, Hidehiko Tanaka, and Tohru Moto-Oka. 1983. Application of Hash to Data Base Machine and Its Architecture. *New Gener. Comput.* 1, 1 (1983), 63–74.
- [54] Tomohiro Korikawa, Akio Kawabata, Fujun He, and Eiji Oki. 2020. Packet processing architecture using last-level-cache slices and interleaved 3D-stacked DRAM. *IEEE Access* 8 (2020), 59290–59304.
- [55] Harald Lang, Viktor Leis, Martina-Cezara Albutiu, Thomas Neumann, and Alfons Kemper. 2013. Massively Parallel NUMA-Aware Hash Joins. In *IMDM@VLDB (Revised Selected Papers) (Lecture Notes in Computer Science, Vol. 8921)*. Springer, 3–14.
- [56] Robert Lasch, Thomas Legler, Norman May, Bernhard Scheirle, and Kai-Uwe Sattler. 2022. Cost Modelling for Optimal Data Placement in Heterogeneous Main Memory. *Proc. VLDB Endow.* 15, 11 (2022), 2867–2880.
- [57] Benjamin C. Lee, Ping Zhou, Jun Yang, Youtao Zhang, Bo Zhao, Engin Ipek, Onur Mutlu, and Doug Burger. 2010. Phase-Change Technology and the Future of Main Memory. *IEEE Micro* 30, 1 (2010), 143.
- [58] Sekwon Lee, Soujanya Ponnappalli, Sharad Singhal, Marcos K Aguilera, Kimberly Keeton, and Vijay Chidambaram. 2023. DINOMO: An Elastic, Scalable, High-Performance Key-Value Store for Disaggregated Persistent Memory (Extended Version). *Proc. VLDB Endow.* 15, 13 (2023).
- [59] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. 2019. Recipe: converting concurrent DRAM indexes to persistent-memory indexes. In *SOSP*. ACM, 462–477.
- [60] Lucas Lersch, Xiangpeng Hao, Ismail Oukid, Tianzheng Wang, and Thomas Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proc. VLDB Endow.* 13, 4 (2019), 574–587.
- [61] Yu-Pei Liang, Tseng-Yi Chen, Yuan-Hao Chang, Shuo-Han Chen, Hsin-Wen Wei, and Wei-Kuan Shih. 2020. B\*-Sort: Enabling Write-Once Sorting for Non-volatile Memory. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 39, 12 (2020), 4549–4562.
- [62] Sihang Liu, Suraj Kanniwadi, Martin Schwarzl, Andreas Kogler, Daniel Gruss, and Samira Khan. 2023. Side-Channel Attacks on Optane Persistent Memory. In *32th USENIX Security Symposium (USENIX Security 23)*.
- [63] Baotong Lu, Jialin Ding, Eric Lo, Umar Farooq Minhas, and Tianzheng Wang. 2021. APEX: A High-Performance Learned Index on Persistent Memory. *Proc. VLDB Endow.* 15, 3 (2021), 597–610.
- [64] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (2020), 1147–1161.
- [65] Shaonan Ma, Kang Chen, Shimin Chen, Mengxing Liu, Jianglang Zhu, Hongbo Kang, and Yongwei Wu. 2021. ROART: Range-query Optimized Persistent ART. In *FAST*. USENIX Association, 1–16.
- [66] Tobias Maltenberger, Till Lehmann, Lawrence Benson, and Tilmann Rabl. 2022. Evaluating In-Memory Hash Joins on Persistent Memory. In *EDBT*. OpenProceedings.org, 2:368–2:372.
- [67] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2000. What Happens During a Join? Dissecting CPU and Memory Optimization Effects. In *VLDB*. Morgan Kaufmann, 339–350.
- [68] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Trans. Knowl. Data Eng.* 14, 4 (2002), 709–730.
- [69] Mark Mansi, Bijan Tabatabai, and Michael M. Swift. 2022. CBMM: Financial Advice for Kernel Memory Managers. In *USENIX Annual Technical Conference*. USENIX Association, 593–608.
- [70] Sparsh Mittal and Jeffrey S. Vetter. 2016. A Survey of Software Techniques for Using Non-Volatile Memories for Storage and Main Memory Systems. *IEEE Trans. Parallel Distributed Syst.* 27, 5 (2016), 1537–1550.
- [71] Yoon-Min Nam, Donghyoung Han, and Min-Soo Kim. 2020. SPRINT: A Fast n-ary Join Query Processing Method for Complex OLAP Queries. In *SIGMOD Conference*. ACM, 2055–2070.
- [72] Dimin Niu, Yiran Chen, and Yuan Xie. 2010. Low-power dual-element memristor based memory design. In *Proceedings of the 16th ACM/IEEE international symposium on Low power electronics and design*. 25–30.
- [73] 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 (Lecture Notes in Computer Science, Vol. 5895)*. Springer, 237–252.
- [74] Ismail Oukid. 2018. *Architectural Principles for Database Systems on Storage-Class Memory*. Ph.D. Dissertation. Dresden University of Technology, Germany.
- [75] Ismail Oukid. 2019. Architectural Principles for Database Systems on Storage-Class Memory. In *BTW (LNI, Vol. P-289)*. Gesellschaft für Informatik, Bonn, 477–486.
- [76] SNIA Technical Position. 2017. NVM Programming Model (NPM) Version 1.2. <https://www.snia.org/sites/default/files/technical-work/npm/release/SNIA-NVM-Programming-Model-v1.2.pdf>
- [77] Moinuddin K. Qureshi. 2014. Memory Scaling is Dead, Long Live Memory Scaling. [https://hps.ece.utexas.edu/yale75/qureshi\\_slides.pdf](https://hps.ece.utexas.edu/yale75/qureshi_slides.pdf)
- [78] Amanda Raybuck, Tim Stamler, Wei Zhang, Mattan Erez, and Simon Peter. 2021. HeMem: Scalable Tiered Memory Management for Big Data Applications and Real NVM. In *SOSP*. ACM, 392–407.
- [79] Stefan Schuh, Xiao Chen, and Jens Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *SIGMOD Conference*. ACM, 1961–1976.
- [80] Felix Martin Schuhknecht, Pankaj Khanchandani, and Jens Dittrich. 2015. On the Surprising Difficulty of Simple Things: the Case of Radix Partitioning. *Proc. VLDB Endow.* 8, 9 (2015), 934–937.
- [81] 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.
- [82] Simon Sharwood. 2022. Last week Intel killed Optane. Today, Kioxia and Everspin announced comparable tech. [https://www.theregister.com/2022/08/02/kioxia\\_everspin\\_persistent\\_memory/](https://www.theregister.com/2022/08/02/kioxia_everspin_persistent_memory/)
- [83] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. 1994. Cache Conscious Algorithms for Relational Query Processing. In *VLDB*. Morgan Kaufmann, 510–521.
- [84] Ali Sheikholeslami and P. Glenn Gulak. 2000. A survey of circuit innovations in ferroelectric random-access memories. *Proc. IEEE* 88, 5 (2000), 667–689.
- [85] Anton Shilov. 2022. Samsung’s Memory-Semantic CXL SSD Brings a 20X Performance Uplift. <https://www.tomshardware.com/news/samsung-memory-semantic-cxl-ssd-brings-20x-performance-uplift>
- [86] Dixin Tang, Zechao Shang, Aaron J. Elmore, Sanjay Krishnan, and Michael J. Franklin. 2020. Thrifty Query Execution via Incrementability. In *SIGMOD Conference*. ACM, 1241–1256.
- [87] Daniel Terpstra, Heike Jagode, Haihang You, and Jack J. Dongarra. 2009. Collecting Performance Data with PAPI-C. In *Parallel Tools Workshop*. Springer, 157–173.
- [88] 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.
- [89] Stratis Viglas. 2014. Write-limited sorts and joins for persistent memory. *Proc. VLDB Endow.* 7, 5 (2014), 413–424.
- [90] Lukas Vogel, Alexander van Renen, Satoshi Imamura, Jana Giceva, Thomas Neumann, and Alfons Kemper. 2022. Plush: A Write-Optimized Persistent Log-Structured Hash-Table. *Proc. VLDB Endow.* 15, 11 (2022), 2895–2907.
- [91] Weier Wan, Rajkumar Kubendran, Clemens Schaefer, Sukru Burc Eryilmaz, Wenqiang Zhang, Dabin Wu, Stephen Deiss, Priyanka Raina, He Qian, Bin Gao, et al. 2022. A compute-in-memory chip based on resistive random-access memory. *Nature* 608, 7923 (2022), 504–512.
- [92] Zixuan Wang, Xiao Liu, Jian Yang, Theodore Michailidis, Steven Swanson, and Jishen Zhao. 2020. Characterizing and Modeling Non-Volatile Memory Systems. In *MICRO*. IEEE, 496–508.
- [93] Zixuan Wang, Mohammadkazem Taram, Daniel Moghimi, Steven Swanson, Dean Tullsen, and Jishen Zhao. 2023. NVLeak: Off-Chip Side-Channel Attacks via Non-Volatile Memory Systems. In *32th USENIX Security Symposium (USENIX Security 23)*.
- [94] 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 *DaMoN*. ACM, 14:1–14:3.
- [95] Lingfeng Xiang, Xingsheng Zhao, Jia Rao, Song Jiang, and Hong Jiang. 2022. Characterizing the performance of intel optane persistent memory: a close look at its on-DIMM buffering. In *EuroSys*. ACM, 488–505.
- [96] Jian Xu and Steven Swanson. 2016. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In *FAST*. USENIX Association, 323–338.
- [97] 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.
- [98] Wen Zhang, Scott Shenker, and Irene Zhang. 2020. Persistent State Machines for Recoverable In-memory Storage Systems with NVRam. In *OSDI*. USENIX Association, 1029–1046.

- [99] Zuyu Zhang, Harshad Deshmukh, and Jignesh M. Patel. 2019. Data Partitioning for In-Memory Systems: Myths, Challenges, and Opportunities. In *CIDR*. [www.cidrdb.org](http://www.cidrdb.org).
- [100] Diyu Zhou, Yuchen Qian, Vishal Gupta, Zhifei Yang, Changwoo Min, and Sanidhya Kashyap. 2022. ODINFS: Scaling PM Performance with Opportunistic Delegation. In *OSDI*. USENIX Association, 179–193.