A Design Space Exploration and Evaluation for Main-Memory Hash Joins in Storage Class Memory [Technical Report]

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

The long-standing debate on whether it is essential to perform partitioning for main-memory hash joins has been rigorously argued for decades. Within the scope of DRAM, however, the whole research community has yet to reach an agreement. Meanwhile, the recent upsurge of storage class memory (SCM) technologies has considerably expanded the memory hierarchy, making the above partitioning debate vastly entangled. This paper aims to revisit this debate in the context of 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.

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 [Technical Report] . PVLDB, 16(6): XXX-XXX, 2023.
doi:XX.XX/XXX.XX

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/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 [60]. 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 [13, 63]. 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 [10]. As a consequence, even though PHJ already outperforms NPHJ in some workloads [8, 9, 58, 88], the PHJ-vs-NPHJ debate is still ongoing.

Meanwhile, main memory (DRAM) technology has hit a scaling wall [23, 86]. As the logic chip size continues to drop, it is becoming increasingly difficult to shrink the DRAM cell size while maintaining enough capacity [1]. Storage Class Memory (SCM), or non-volatile memory, is the emerging memory technology that primarily targets breaking this wall [23, 40, 48]. It offers large capacity, byte-addressability, and near-DRAM access performance. Moreover, most SCM technologies support data persistence [2, 19, 27, 31, 32, 65, 80, 93, 100], making SCM an appealing alternative not only for DRAM but also for SSD. Several SCM technologies have been put into practice [2, 47, 85, 91, 94], among which NVDIMM-P [47] 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 [85].

Inspired by NVDIMM-P, leading memory manufacturers have been grinding for developing SCM products [45, 91, 94]. Up to now, Intel Optane DC Persistent Memory Module (Optane DIMM) [45]

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

[1] Memory capacity per core is expected to decrease by 30% biannually [86].

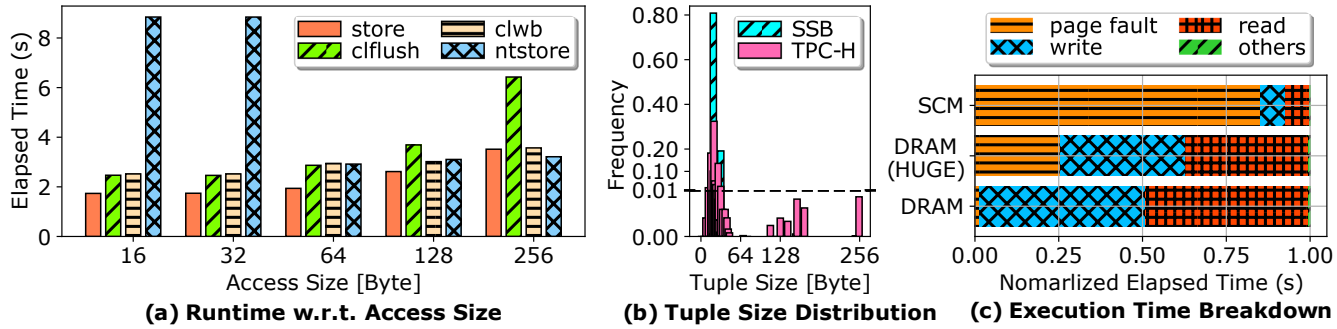


Figure 1: (a) Write performance with different access sizes. (b) The tuple size distribution of benchmark SSB [81] and TPC-H [24] joins. (c) The execution time breakdown of a write-and-read microbenckmark on a 16KB memory region ("HUGE" denotes the huge page configuration of DRAM, "others" overhead remains unnoticeable in all three memory configurations).

is the first and only industrial NVDIMM-P implementation in market. Since its release, numerous attempts have been made to harness it in developing persistent data structures [37, 38, 59, 71], fault-tolerant file systems [35, 87, 111], crash recovery mechanisms [16, 67, 109], etc. The industry community also invests huge efforts to deploy it in data centers [54] and cloud providers [33, 66]. It is expected that SCM will become a crucial building block in future data-intensive platforms.

Unfortunately, there are some salient differences between SCM and DRAM, indicating that directly translating main-memory hash joins from DRAM to SCM may lead to unsatisfactory results. Existing studies [25, 34, 62, 97, 101, 103, 104, 106] showed that SCM may have the following unique features, which have a massive impact on memory-intensive operations: (1) SCM exhibits an asymmetric read/write pattern; the write bandwidth is several times slower than the read's, making write operations much more expensive than their counterparts in DRAM. (2) There may be an access granularity mismatch between SCM DIMMs and processors [47, 85, 106], for instance, Optane DIMM has an internal granularity of 256B, which is 4× of the cacheline size, yielding detrimental read/write amplification and wastage in bandwidth. (3) The SCM internal architecture is likely to remain confidential (e.g., Intel never reveals the internal documentation to the public), and various profiling studies report contradictory results (e.g, [104] reports the discrepant buffer capacity and replacement policy with [70, 101, 102]), making it even harder to interpret SCM access behaviors. (4) The communication protocol is yet to be standardized (DDR4, DDR5, DDR-T, etc [45, 47, 70, 102, 104]), which may bring about unpredictable performance implications in practice.

Owing to the above features, existing works [25, 29, 34, 62, 64, 97, 101, 103, 104, 106] have proposed a few practical guidelines for memory-intensive operations. However, most of these guidelines focus on exploiting the persistence feature while overlooking the nature of join-related workloads. Hence, these guidelines suffer from certain limitations and do not apply to main-memory join processing. We address two fundamental limitations that have been previously overlooked.

(1) Persistence cost should be eliminated to the greatest extent possible. SCM is mostly affiliated with the appealing trait of non-volatility, and existing studies are inclined to achieve immediate persistence by employing persistent instructions (e.g., "clflush", "clwb", "ntstore"). This trend does not fade away after eADR [104, 107, 108] platform, where the cache hierachy is included in the persistence domain, is introduced, because of the higher bandwidth that persistent instructions rendered [12, 15]. However, this practice does not always boost the performance. Figure 1(a) compares the single thread write performance with different write and persist instructions at different access granularities. We can see that regular store (w/o immediate persistence guarantee) outperforms persistent stores ("clwb", "clflush", "ntstore") appreciably at smaller access sizes ($\leq$ 128B) and only loses to persistent stores at 256B granularity. It is noteworthy to mention that main-memory join processing usually operates at an access granularity of a tuple size, which is typically smaller than cacheline (cf. Figure 1(b) for the tuple size distribution in SSB [81] and TPC-H [24] joins). Hence, harnessing persistent instructions can deteriorate the join performance. Furthermore, immediate persistence is not the major concern in join processing. In light of power outage, rerunning a query has a higher gain expectation than recover-and-continue an interrupted run, not to mention the immediate persistence assurance from the recent eADR platform. As a consequence, the widely adopted persistent instructions should not be used in hash joins.
(2) Page fault overhead is more pronounced for cache-friendly algorithms in SCM. In order to demonstrate this point, we run a microbenchmark to measure the overhead of page fault. We allocate a cache-sized memory region (16KB) with different memory configurations and issue random writes followed by random reads within this region, which imitates the typical access pattern of hash table building and probing. When allocating the memory, we explicitly issue the "memset" instruction to eliminate page faults. As can be seen in Figure 1(c), the page fault overhead is negligible in DRAM and consumes no more than 25% with huge-page DRAM configuration. In contrast, page fault takes up over 80% the cost in SCM. Such phenomenon can also be observed in the join phase of PHJ (see Section 8.2 for more details). On this account, page fault in SCM is the major hindrance for cache-friendly algorithms and 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. [74] 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 join algorithms in real SCM [2] 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 (Sections 5 and 6). In order to obtain a fair comparison between NPHJ and PHJ, we also conduct a comprehensive evaluation in extensive workloads (Section 7). With a systematic experimental study and analysis, we maintain that PHJ is generally the preferable solution for SCM. In addition, we propose a set of practical analyses and several meaningful discussions to offer more insights for practitioners (Section 8). It is worth addressing that we conduct experiments with Optane DIMMs, the only available SCM hardware currently, but our findings and discussions are majorly based on NVDIMM-P [47] standard. Moreover, we do not rely on any specific persistent features. Therefore our study has strong generalizability and can be applied to any SCM that formed in a DIMM factor [2, 22, 47, 91, 94] (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 remainder of this paper is organized as follows. In Section 2, we introduce the landscape of SCM. In Section 3, we inspect the details of PHJ and NPHJ and discuss alternative implementations. Section 4 covers the details of experimental setups. We then conduct a comprehensive exploration for NPHJ and PHJ in Sections 5 and 6 respectively. In Section 7, we perform a rigorous evaluation to compare PHJ and NPHJ in wide-ranging workloads. Afterward, we discuss the experimental findings in-depth and propose practical guidelines in Section 8. We briefly review related works in Section 9 and conclude the paper in Section 10.

[2]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.

Table 1: List of Evaluated Main-Memory Hash Joins

Taxoxomy	Join Notation	Partitioning	Hashing
Non-Partitioned Hash Join	NPHJ-SC	—	Separate Chaining
	NPHJ-LP		Linear Probing
Partitioned Hash Join	SHRll-SC	Shared Partitioning (linked list)	Separate Chaining
	SHRll-LP		Linear Probing
	SHRll-HM		Histogram Mechanism
	SHRcm-BC	Shared Partitioning (contiguous memory)	Bucket Chaining
	SHRcm-SC		Separate Chaining
	SHRcm-LP		Linear Probing
	SHRcm-HM		Histogram Mechanism
	INDll-SC	Independent Partitioning (linked list)	Separate Chaining
	INDll-LP		Linear Probing
	INDll-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

[1] " —" depicts that the algorithms do not perform partitoning;
[2] " Histogram Mechanism " represents the historgam-based re-ordering hashing scheme proposed in [58];
[3] " (contiguous memory) "-based partitioning methods apply to uniformly distributed data only.

2 THE SCM LANDSCAPE

The DRAM technology is facing an acute challenge: it fails to scale to sub-20nm size [23, 86], which limits its deployment in future technology nodes. In order to break this wall [3], various SCMs [19, 31, 32, 65, 80, 93] 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 [36] and PCM was validated to shrink to the sub-2nm scale [48]. 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 [47] standard for adopting SCM technology [4]. 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 [70,

[3]SRAM and NOR flash also have hit the scaling wall [23].
[4]JEDEC also proposes the NVDIMM-N [46], which only pairs DRAM with flash in a DIMM. Thus, it still suffers from the DRAM scaling wall and is beyond the scope of this study.

102]), 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 [106], the on-DIMM buffer and controller visit the underlying SCM media at a coarser granularity (e.g., 256B XPLine size in Optane DIMM) [5]. Thus, small-size data requests from processors will render the infamous read/write amplification. To exploit SCM's byte-addressability, NVDIMM-P suggests SCM to be accessed via DAX-mmap [3, 21, 50, 77], 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" [30, 78, 83, 84, 97]. Moreover, DAX-mmap exposes the costly page faults in SCM's critical path [21, 77] [6], which impairs the performance of cache-sensitive applications (e.g., Figure 1(c)) and leads to notorious "small files problem" [3]. Furthermmore, NVDIMM-P defines persistent instructions ("clwb", "clflush", etc.) to make use of SCM's non-volatility, and works compatibly with the prospective CXL [22] standard. In consequence, NVDIMM-P is becoming a promising building block in future large-scale analytical systems.

Thanks to the above features and the strong scaling ability, NVDIMM-P is widely acknowledged to be 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 [45] is the only available NVDIMM-P implementation up to now, 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.
- $\mathcal{P}_2$: 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) [13, 63] 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 than S [24, 88], NPHJ significantly saves the write cost, especially for the write-susceptible SCM.

Thanks to the modern parallel processors' simultaneous multithreading (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 [9, 10]. 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 join algorithms. 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.

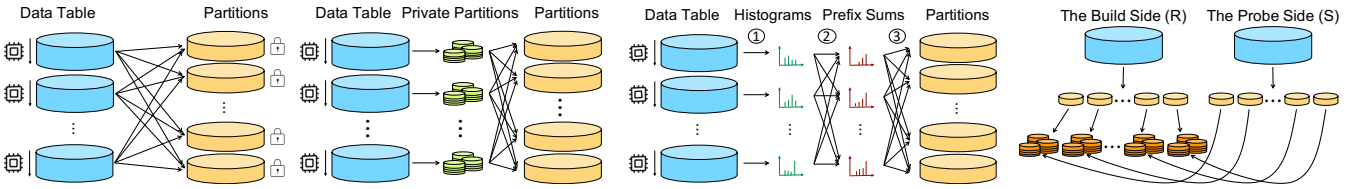
3.2.1 Partition Phase. There are numerous ways to perform partitioning [89, 110], among which radix partitioning [8] has been shown to be the best choice in main memory (DRAM) systems. A natural question to ask is whether radix partitioning still dominates in SCM. Recall that different partitioning methods 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 [13].** 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 [13].** 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 [8].** 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

[5] This granularity also represents the unit size of error-correct code (ECC) block [34].
[6] This is assumed to be a common feature for most SCM technologies [21].



(a) Shared Partitioning (b) Independent Partitioning (c) Radix Partitioning (d) Asymmetric Radix Partitioning

Figure 2: Partitioning Methods.

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. [14, 75, 76] 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 [56].** Khattab et al. [56] 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 [56]. 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 competitve profile in SCM.

It is worth mentioning that the partitioninig performance can be significantly improved with software write-combining buffers (SWWCB) and non-temporal stores ("ntstore") [6, 9, 88, 110]. SWWCB maintains a separate in-cache buffer of N-tuple capacity for each partition. During partitioning, tuples are copied to these buffers 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 SWWCB can be further enhanced with "ntstore"

Algorithm 1: Partitioning with SWWCBs and ntstore

```
1 foreach tuple t_i ∈ relation do
2     k = hash(t_i);
3     SWWCB[k][pos[k]] = t_i;      // copy t_i to SWWCB[k]
4     pos[k] ++;
5     if pos[k]%N = 0 then
          /* copy SWWCB[k] to part[k] via ntstore */
6         ntstore(part[k], SWWCB[k]);
```

(shown in Algorithm 1). 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.

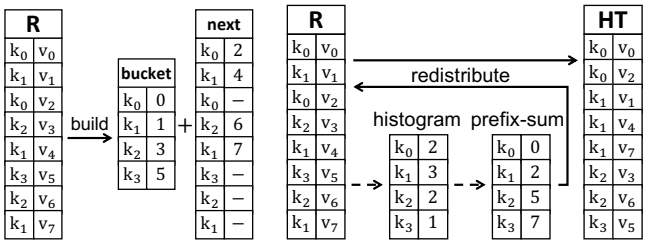
Before we proceed to demystify the join phase, we digress to discuss an alternative partition layout for shared and independent partitioning. Recall that shared and independent partitioning structure their partition layout as buffer linked lists, which may span separate memory pages. Compared to the 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

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	$(2m+1)R+(2n+k)S$	$mR+nS$

[1] "k" denotes the times for R partition number over S partition number;
[2] We do not explicitly distinguish sequential/random passes as we avail of SWWCBs' temporal sequential pattern in random page visits.



(a) Bucket Chaining **(b) Histogram Mechanism**

Figure 3: Hash Table Designs for PHJ-BC and PHJ-HM.

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 noteworthy special case is asymmetric radix partitioning. Since it takes more partitoning passes over R than S, the partition fanout of R 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 pass number 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. [76] utilize a variant bucket chaining mechanism (cf. Figure 3(a)), 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. [58] proposed an alternative 2-pass hash table building method to make use of SIMD acceleration ("Histogram Mechanism" in Table 1). It first scans the partition to generate a hash value histogram. Then it uses the prefix sum of the histogram to redistribute the tuples in the second pass (cf. Figure 3(b)). 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 $< key, payload >$ pair, and both key and payload are 8-byte long. Following previous works [9, 10, 13], we set the build side cardinality (i.e., $|R|$) and the probe side cardinality (i.e., $|S|$) as 16×2^{20} and 256×2^{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 [24]. 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 evaluate the joins with two other sets of workloads: a set of skew workloads ("zipf") where the probe sides follow a zipfian distribution [7, 13, 88] with different skewnesses (Zipf factor θ), and a uniform workload set ("sel") with various join selectivities [7]. To further assess the join robustness, we synthesize another two workloads to compare the joins in "many-to-many" scenario. The first workload, named "dupfk", loosens the **PK-FK** constraint to **FK-FK**, which still holds the "foreign-key" dependency between R and S but allows duplicates in R [79]. The second workload, "dens" further increases the relation "density" on "dupfk" by narrowing down the key distribution domain [8], leading to more hashing collisions.

Due to the limited capacity of DRAM, previous works [9, 10, 13, 68] 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. Therefore, 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.

[7] The "join selectivity" is defined as $\frac{|R \bowtie |S|}{|S|}$ [92].
[8] The parameter, "density", is defined as $\frac{key\_range}{|R|}$.

Table 3: List of Evaluated Workloads

Workload	pkfk	zipf	sel	dupfk	dens	pyld	bln
$\|R\|$	$16/64/256\times2^{20}$	16×2^{20}	16×2^{20}	16×2^{20}	16×2^{20}	16×2^{20}	1×2^{30}
$\|S\|$	256×2^{20}	256×2^{20}	256×2^{20}	256×2^{20}	256×2^{20}	256×2^{20}	16×2^{30}
Tuple Size (Byte)	16	16	16	16	16	16~512	16
Distribution	uniform	zipfian	uniform	uniform	uniform	uniform	uniform
Constraint	PK-FK	PK-FK	PK-FK (selective)	FK-FK	FK-FK (dense)	PK-FK	PK-FK
Parameter Domain	—	{ 1.05, 1.25, 1.50, 1.75 }	{ 0.20, 0.40, 0.60, 0.80 }	—	{ 0.20, 0.40, 0.60, 0.80 }	—	—

[1] " PK-FK " denotes primary-key-foreign-key constraint;
[2] " FK-FK " signifies foreign-key-foreign-key constraint [79], i.e., many-to-many join.

In addition to the previous synthetic workloads, we evaluate the joins on the TPC-H benchmark [24]. We choose Query 14 to perform the evaluation, as it contains a binary join operator and an aggregation operator, which helps us to rule out other affecting factors (e.g., sorting, deduplication) and reflect the actual join performance better. We set the scale factor to 100 (i.e., $100GB$ dataset) and store the generated tables in SCM by default.

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 and 6). The remaining workloads will be evaluated rigorously in Section 7 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 [38, 59, 67, 72, 109], 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 [9] to run joins, which offers a favourable performance according to existing DRAM-based studies of main-memory hash joins [9, 13].

Following previous works of main-memory hash joins [9, 10, 13, 58, 68, 88], 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 [10], 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 [10, 11, 50, 57, 64, 73, 95, 103]. Moreover, we pre-fault SCM mappings [3, 21, 50, 77] when allocating memory for hash tables or partitions, obliterating page faults in join execution (see Section 8.2 for a discussion of page fault effects). In order to better analyze the experiments, we instrument our studies with PAPI [96] and VTune [41] to measure the hardware events of our platform. Futhermore, we employ PMWatch [43] to collect SCM's hardware-level statistics.

[9] By default, we only evaluate joins in one socket to avoid potential NUMA impacts.
[10] Previous works [9, 10, 13, 58, 68, 88] use "join throughput", i.e., $\frac{|R|+|S|}{elapsed\ time}$, for evaluation, which is equivalent to our metric, elapsed time.

5 NON-PARTITIONED HASH JOINS IN SCM

In this section, we study the implementation of NPHJ in three aspects: 1) the benefit of prefetching; 2) the effect of bucket alignment; 3) the performance w.r.t. thread scalability. The main goal of this section is to uncover the most crucial factors that contribute to a performant NPHJ implementation.

5.1 Prefetching

We start our evaluation with the prefetching analysis. Prefetching has been shown to deliver impressive improvement for hash joins [7, 9, 17, 18]. 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.

Table 4 shows the prefetching improvements for NPHJ in SCM. We also present the respective improvements in DRAM for a fair comparison. We can observe that although both NPHJ-SC and NPHJ-LP benefit from prefetching, DRAM improvements are more prominent than SCM, especially for NPHJ-SC. This improvement discrepancy is primarily attributed to the difference in memory access cost between the two memory media. SCM has higher access latency than DRAM (3~4 higher read latency [87, 106, 112]) making it harder for current processors to conceal the cache miss penalty. Meanwhile, we can see that NPHJ-SC benefits more from prefetching, either in DRAM or SCM. The reason is that the linear probing mechanism already arranges hash buckets in a close memory pattern, which yields a good cache locality and leaves little boosting space for prefetching. With prefetching, NPHJ-SC overcomes its weakness in cache locality and achieves a very close performance to NPHJ-LP (within 5% performance gap).

In order to parameterize the optimal prefetching distance in SCM, we assess the join performance with varying prefetching distances. Figure 4 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 [$\mathscr{P}_2$] . A prefetching distance of 2^{14}-tuple indicates that both prefetched tuples and hash buckets requires

Table 4: Prefetching Improvements

Memory	Algorithm	Runtime wo/ Prefetching	Runtime w/ Prefetching	Improvement
DRAM	NPHJ-SC	0.5842s	0.3206s	46.65%
	NPHJ-LP	0.3516s	0.2886s	17.91%
SCM	NPHJ-SC	2.5318s	2.1829s	13.78%
	NPHJ-LP	2.1724s	2.0836s	3.72%

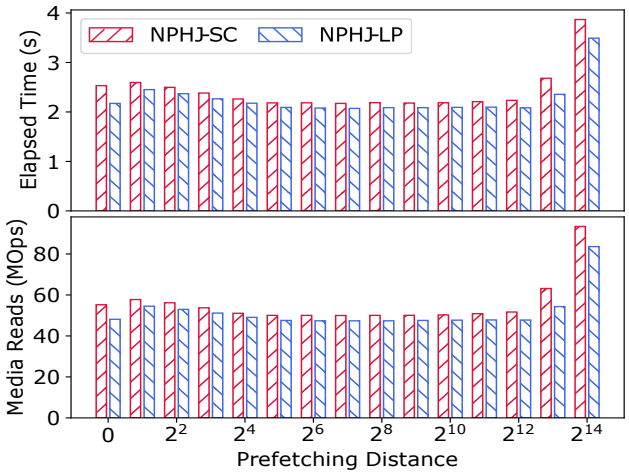


Figure 4: NPHJ execution time & internal media reads w.r.t. prefetching distance (the distance refers to the number of tuples).

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 [61]) [11]. 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 [101, 104, 106], and the total on-DIMM buffers are 96KB (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 caches [104]: 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 [$\mathscr{P}_2$] .

[11] LLC size per core is calculated as $27.5MB \div 20 = 1.375MB$.

5.2 Bucket Alignment

We now assess the impact of bucket alignment. By default, a hash bucket requires 48 bytes in our implementaion [12]. 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 DIMMSs [$\mathscr{P}_1$] . If the hash bucket spans two XPLines, the additional memory access can trigger one more SCM media read, exacerbating the bucket access overhead.

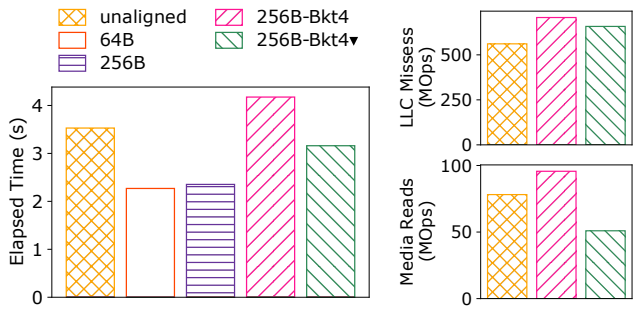
Bucket alignment aids in mitigating this issue of extra memory accesses. Bucket alignment can be set as 64B, which ensures each hash bucket be entirely stored in a single cacheline, and thereby precludes the extra memory accesses. Another rational alignment configuration is the internal granularity of 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 5).

Figure 5(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 5(b). We 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 [$\mathscr{P}_2$] . 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-Bkt4▼" in Figure 5. As shown in Figure 5(a), its runtime beats the unaligned configuration. We also notice that "256B-Bkt4▼" significantly lowers SCM media reads (cf. Figure 5(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 5(b) also reveals an essential finding of NPHJ in real SCM. "256B-Bkt4▼" incurs fewer media reads but more LLC misses when compared to the unaligned configuration. Given the superior performance of "256B-Bkt4▼", we argue that SCM media reads are the most significant impediment for a high-performance join. One

[12] 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.



(a) NPHJ Execution Time (b) LLC Misses & Media Reads

Figure 5: NPHJ executione time, LLC Misses and 3D-Xpoint Media Reads with different bucket configurations ("256B-Bkt4" refers to a bucket setting with a 4-tuple capacity and 256B alignment; "▾" denotes that the join is executed with 7 threads).

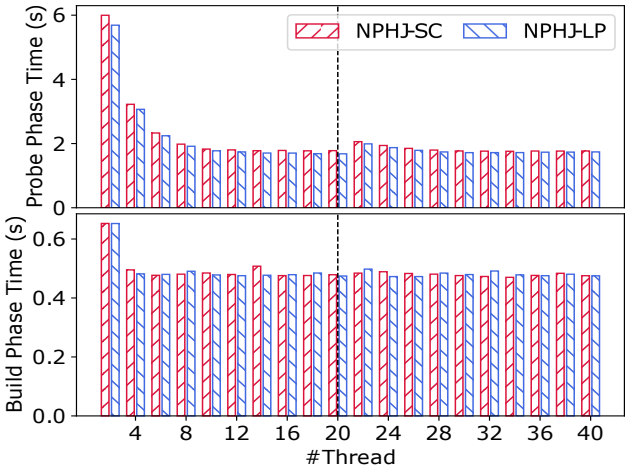


Figure 6: NPHJ execution time w.r.t. thread number (dashed lines mark the starting of simultaneous multi-threading (SMT)).

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 [$\mathscr{P}_1$,$\mathscr{P}_2$] .

5.3 Thread Scalability

NPHJ takes advantage of multi-threading to mitigate the cache miss overhead, and previous DRAM-based studies have shown that its scalable performance with the running-thread number [6, 9, 13, 63]. However, SCM is widely reckoned to behave poorly in writing and existing studies [25, 34, 62, 97, 101, 103, 104, 106] have validated its write defect. A natural question to ask is whether or not NPHJ can profit from multi-threading in SCM.

To answer this question, we evaluate NPHJ-SC and NPHJ-LP with different numbers of running threads. In particular, we separate the join processing into the build phase and the probe phase, and examine their respective performance. Figure 6 demonstrates the experimental results. The performance of the probing phase scales nicely with the number of probing threads, especially before the SMT region. After entering the SMT zone, some cores are doing more works than others, bringing about a modest increase in runtime. As the SMT number increases, the work is distributed more and more evenly over all the cores, making the runtime gradually and steadily approach a low point.

In contrast, the build phase exhibits poor thread scalability in performance. The runtime quickly converges to 0.48s when the thread number reaches 6 and increasing thread number beyond 6 does not significantly enhance performance. This poor scalability is primarily due to the deficient write bandwidth of [$\mathscr{P}_3$] SCM [25, 34, 62, 97, 101, 103, 104, 106]. Note that inserting a tuple incurs at least two reads (one access for the tuple and one access for the bucket) and one write (write the tuple to the bucket). As the thread number increases, the writes start to dominate the memory bandwidth, consequently harming the overall performance.

Tip #3: Exploit all physical cores for probing but save the cores (threads) within a certain level (e.g., 6 in Optane DCPMM) in the build phase [$\mathscr{P}_3$] .

6 PARTITIONED HASH JOINS IN SCM

Similarly, we explore the design space for PHJ in this section. In particular, we dissect PHJ into the partition phase and the join phase. Likewise, we aim to derive a comprehensive understanding of the implications of various implementation strategies.

6.1 Partition Phase

The partition phase is well known to be the dominanting phase in PHJ execution [6, 9, 88, 110]. Subsection 3.2 has presented several partitioning strategies, which generate different passes and access patterns for reads and writes. Recall that SCM is more vulnerable to writes and on-DIMM buffer makes it more susceptible to random memory access. These read/write accesses, therefore, are likely to instigate unanticipated consequences. In this subsection, we factor in scaling effects in three dimensions: 1) SWWCB size; 2) thread scalability; 2) partition fanout, and come up with a few guidelines to tune these partitoning methods for their best peroformance.

6.1.1 Effect of SWWCB and "ntstore". We first scrutinize the effect of SWWCB in "ntstore". Recall that "ntstore" delivers higher 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 [7, 10, 13, 58] 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.

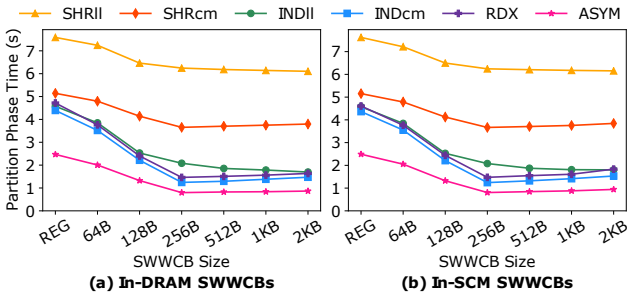


Figure 7: PHJ partitioning time using non-temporal stores with different SWWCB size ("REG" denotes the naive partitioning setting w/o "ntstore" and SWWCBs).

Figure 7 presents the partitioning results. Compared to the naive setting ("REG" in Figure 7), 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 [$\mathscr{P}_1$] .

Figure 7 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 thrashes, which again validates our finding in Section 5.2. 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 group 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 #4: Leverage "ntstore" and SWWCBs in partitioning and make SCM's internal access granularity as the SWWCB size [$\mathscr{P}_1$] .

6.1.2 Effect of thread scalability. SCM is widely reckoned to have write deficiency [26, 78, 106] [$\mathscr{P}_3$] 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 8 presents the execution time. There are generally two trends in partitioning thread scalability: (1) Shared partitioning (SHRll 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

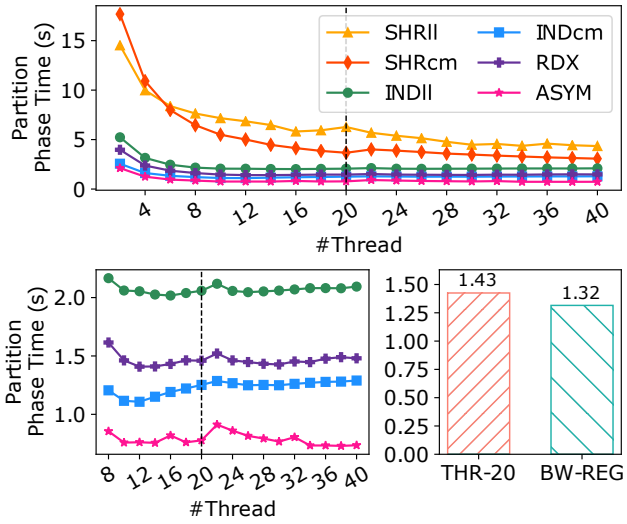


Figure 8: PHJ partitioning time w.r.t. thread number (dashed lines mark the starting of simultaneous multi-threading (SMT); the bottom left figure zooms in the partitioning time of independent partitioning, radix partitioning, and asymmetric radix partitioning; the bottom right figures compares the elapsed time between the 20-thread partitioning (THR-20) and the bandwidth-regulation partitioning (BW-REG)).

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 INDll) 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 [$\mathscr{P}_5$]. 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 8, 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 [20, 26, 30]). Overall, we provide the following partitioning tip:

Tip #5: Exhaust all cores for shared partitioning but enforce parallelism limitation or bandwidth regulation for independent or radix partitioning [$\mathscr{P}_3$] .

6.1.3 Effect of partition fanout. Previous studies [9, 75, 76] have found that partitioning with too many fanouts will generate excessive TLB thrashes, which becomes the predominant overhead in partitioning. Multi-pass partitioning, which splits fanouts across passes, is therefore proposed and is validated to outpower 1-pass partitioning [9, 10]. In order to determine the optimal partitioning configuration, we perform evaluations with different fanouts and different passes. Specifically, we vary the fanout from 2^4 to 2^{16} (4~16 bits) and compare 1-pass partitioning with 2-pass partitioning. Similarly, we use "ntstore" with 256B-SWWCB and enforce the aforementioned bandwidth regulation mechanism to improve throughput. Recall that the L2 TLB capacity is 1536 (Section 4), so TLB thrashes will occur more frequently from 2^{11} fanouts and onwards.

Figure 9 depicts the experimental results. We can see that shared partitioning benefits from a larger fanout, which eases the lock contention more effectively, and 2-pass is always inferior to 1-pass as the second pass introduces another round of lock contention. The remaining partitioning methods, however, exhibit two different trends. The 1-pass configurations yield an outstanding and steady performance within 2^{11} fanouts but worsen with larger fanouts. 2-pass configurations show a stable and moderate performance across all fanout configurations and surpass their 1-pass counterparts around $2^{13\sim15}$ fanouts. The performance discrepancy is credited to TLB thrashes and cache thrashes. In every single partitioning pass, larger fanout will cause more page accesses; thus, TLB thrashes become increasingly frequent once the fanout exceeds the TLB limit (i.e., 1536 in our case). Moreover, since we maintain a separate SWWCB to buffer writes to every single partition, a large fanout will require more SWWCB footprint, which may introduce excessive SCM accesses once the size exceeds the cache capacity. From 2^{12} fanouts and onwards, a single partitioning pass exceeds the TLB limit (1536) and cache capacity (1MB per core); thus 1-pass partitioning becomes increasingly inferior. However, 2-pass partitioning does not exceed both passes' TLB and the cache limit. Thus, it shows a strong robustness and outperforms 1-pass partitioning in large fanouts. Overall, in the "pkfk" workoad, 1-pass partitioning within 2^{11} fanouts delivers the best performance, and we employ 1-pass partitioning in the remaining evaluations until otherwise stated.

As TLB thrash renders page table walk and cache thrash incur excessive SCM accesses, it is hard to distinguish which factor is the more major overhead in partitioning. We hypothesize that cache thrash is likely to play a crucial role. The reason is two-fold: (1) Cache thrashes evict the stale cachelines randomly, leading to random SCM writes. The page table walk from TLB thrashes only incur SCM reads. Therefore, cache thrashes are more inclined to incur heavy overhead in write-susceptible SCM [$\mathscr{P}_3$] . (2) Due to the concern of indirection, encryption, and power consumption, the SCM internal access granularity is not likely to decrease [34, 106] [$\mathscr{P}_1$] . Other SCM may even try larger granularity, motivating us to increase SWWCB size. Moreover, there is also a scaling wall in SRAM manufacturing [23]. The cache size per core is to reduce. In consequence, we expect cache will be easier to thrash for the upcoming NVDIMM-P SCM technologies in the near future and present the following tip:

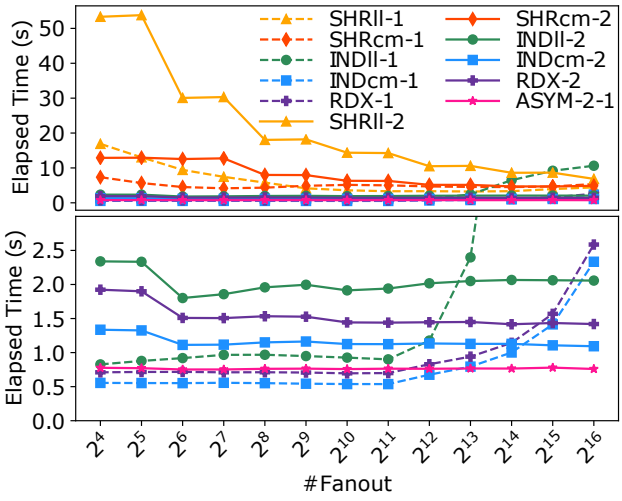


Figure 9: PHJ partitioning time with various fanouts (dashed lines denote the 1-pass partitioning while solid lines represent the 2-pass partitioning; the bottom figure zooms in the partitioning time of independent partitioning, radix partitioning, and asymmetric radix partitioning).

Tip #6: Limit the partition fanout within TLB capacity and ensure that the SWWCBs footprint does not exceed the cache limit [$\mathscr{P}_1$, $\mathscr{P}_3$] .

6.2 Join Phase

We now evaluate various hashing schemes (cf. Table 1) for PHJ in two dimensions, fanout and thread scalability. Because the join phase is subject to partition layouts, we perform the evaluation with three partition layouts, linked list (for SHRll and INDll), contiguous memory (SHRcm and INDcm), and radix (RDX and ASYM).

6.2.1 Effect of partition fanout. Figure 10 depicts the join phase execution time for $2^{4\sim16}$ fanouts. The runtime of all hashing schemes improve with increasing fanouts and reach their optimal performances from 2^8 fanouts and onwards, validating the efficacy of partitioning. Comparing the performane of different partition layouts, the linked list layout offers the worst result. This is because the linked list layout organizes partitions in separate memory fragments, which incurs more L1/L2 cache misses. The same reason accounts for the worst result of separate chaining among all hashing schemes. The remaining three hashing schemes (bucket chaining, linear probing, and histogram mechanism [58]) build the hash tables more compactly; thus they result in comparable performance.

6.2.2 Effect of thread scalability. Figure 11 illustrates the thread scalability in the join phase. We leave out the notations of partition layouts as they all exhibit similar trends in performance. Since the join phase incurs sequential SCM reads, the throughput is highly scalable to the thread number. We also notice that SMT threads do not bring any performance gains. As PHJs are primarily tailored to be cahce-efficient, there is little room for SMT to conceal the cache

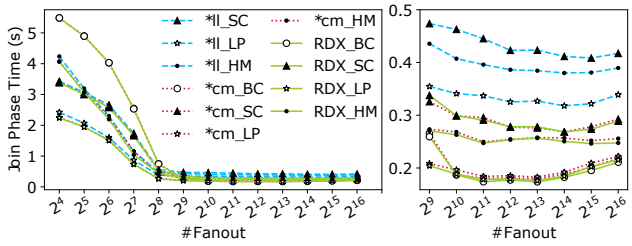


Figure 10: The join phase time of radix partitioning with different fanouts (the right figure zooms in the join phase time for 2^9 –2^{16} fanouts).

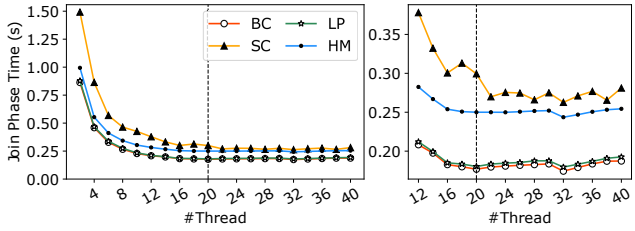


Figure 11: PHJ join phase time w.r.t. thread number (dashed lines mark the starting of simultaneous multi-threading (SMT); the right figure zooms in the join phase time from 12 threads and onwards).

miss penalty further. In summary, satisfactory performance can be achieved by running all physical cores.

6.3 Putting Everything Together

Having determined the optimal configuration for both the partition and the join phase, we now perform an overall comparison for all PHJs. In particular, we use "ntstore" with 256B-SWWCB and apply 1-pass partitioning to all joins (we use 2-1 pass for ASYM-∗ joins), as they deliver the highest write throughput in the current workload. We also configure each algorithm with its optimal threading and fanout configurations for the partition and the join phase separately.

Figure 12 shows the comparison result. We can see that the partition phase takes up the most time in join execution. Shared partitioning joins (SHRll-∗ 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 $|S|$

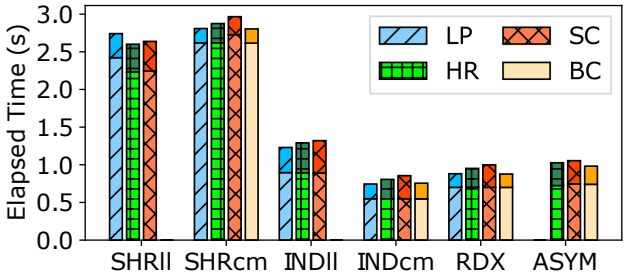


Figure 12: Overall Comparison for Partitioned Hash Joins (lighter colors indicate the partition phase while darker colors represent the join phase) .

partition will be processed k times. Since $|S|$ is usually larger than $|R|$, the k times processing overhead can be significant. We will present a more detailed discussion for ASYM-∗ in Section 8.3. As for the join phase, bucket chaining generally delivers optimal performance. Therefore, until otherwise stated, we leverage bucket chaining as the default hashing solution in the following evaluation.

7 PARTITIONED VS. NON-PARTITIONED

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). Section 6 has shown that the partition phase in PHJs dominates performance, and different hashing schemes do not fundamentally change the total execution time. Due to space constraints, we only present the PHJ result with bucket chaining and take the partitioning notations to represent the respective PHJs. Other hashing schemes also deliver a similar performance. Note that the linked list partition layout does not support bucket chaining (Section 3.2.2); we use Kim's histogram mechanism [58] (HM) instead for it leads to solid and robust performance. Similarly, we take the separate chaining as the default hashing scheme for NPHJ. We apply all proposed tips in joins implementation, and each join is tuned to its optimal configuration in the respective workload.

7.1 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 raio to 1:4 and 1:1 (cf. "pkfk" in Table 3), and report the result in Figure 13(a).

In general, the runtime increases for all joins as the size ratio grows. Specifically, NPHJ deteriorates most badly. Its build phase consumes only 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. SHRll and SHRcm, which lose to NPHJ at 1:16 size ratio, even surpass NPHJ from 1:4 ratio and onwards. This advantage comes from the higher throughput of "ntstore" and 256B-SWWCB. "ntstore" with 256B-SWWCB transmits data directly to

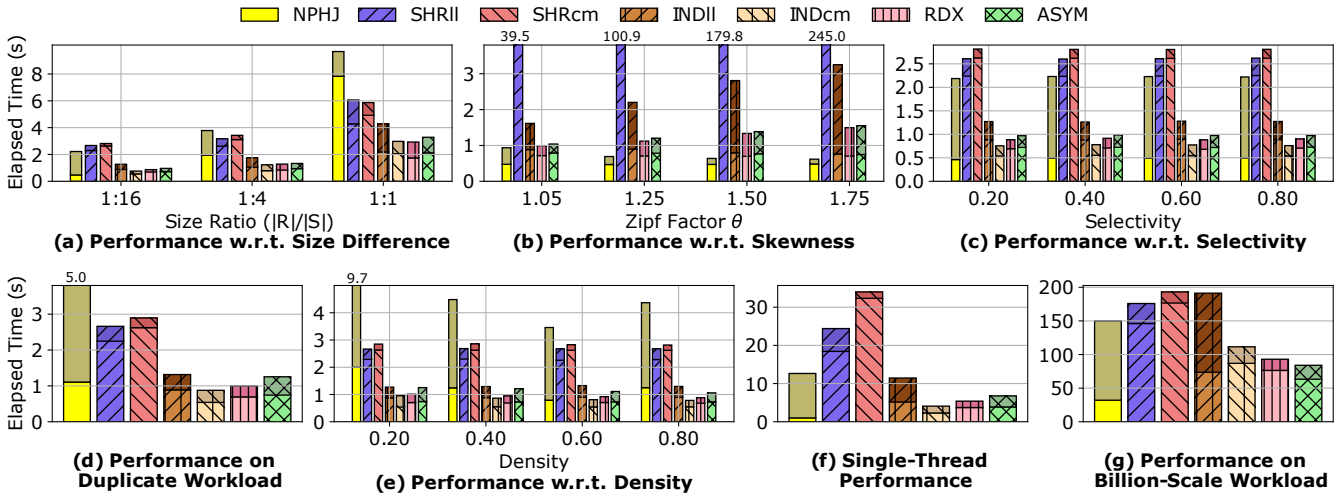


Figure 13: Join execution time across a wide range of workloads (lighter colors indicate the build/partition phase while darker colors represent the probe/join phase; as some algorithms take much longer time to complete certain tests, we cut their bars in the respective subfigures and place their values on top of their bars).

the underlying SCM media, bypassing cache pollution and write-amplification. However, NPHJ building performs write 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.

7.2 Effect of Skewness

We now evaluate joins in skew workloads (cf. "zipf" in Table 3). We vary the skewness (Zipf factor θ [9, 10]) and plot the experimental results in Figure 13(b). Recall that the contiguous memory layout is not applicable to a skew workload (Section 3.2.1), 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 [13], 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 high-skew workloads.

7.3 Effect of Selectivity

We proceed with our evaluation by varying join selectivities (cf. "sel" in Table 3) and report the result in Figure 13(c). We find that the join selectivity does not affect the join performance substantially. Note that a low selectivity evinces that the probe side (S) can only find matches in limited R tuples, which indicates a high locality in probing. However, both sides still follow a uniform distribution. The high locality, therefore, is not able to buffer the probed hash entries entirely in caches. As a consequence, NPHJ's excessive cache miss penalty is not alleviated. As for PHJs, the partitioning already help buffer the probing entries in caches. Hence, the higher locality from a lower selectivity brings no more improvements.

7.4 Many-to-Many Join Performance

Previous works mainly focus on the **PK-FK** setting [9, 10, 13]. However, many-to-many joins (**FK-FK**) are also common in real-world queries. Hence, we evaluate these joins in **FK-FK** workloads.

The first workload ("dupfk") allows duplicates on both sides and Figure 13(d) shows the experimental result. A significant increase in runtime can be observed in all joins' execution time, especially NPHJ, whose higher runtime is derived from its longer probing phase. This is because allowing duplicates will intensify the hash collision problem. 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 addresses, leading to more random memory 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. We further amplify this hash collision effect by narrowing down the key domain ("dens"), resulting in more repeated duplicates. NPHJ's performance is more exacerbated with a lower density, whereas PHJs are only slightly degraded. Therefore, PHJs are superior and more robust than NPHJ in many-to-many join.

7.5 Single Thread Join Performance

Up till now, we employ multi-threading in join evaluation, whereas some database systems only support the single thread query processing [51, 82]. On this count, we execute these joins with a single thread and report the outcome in Figure 13(f). Compared to the previous multi-threading experiment (cf. Figure 13(a)), the performance of all joins degrade drastically. Nonetheless, RDX, ASYM, and INDll still achieve the optimal result for their partitioning reducing cache misses successfully during probing. Meanwhile, NPHJ's probing phase is seriously slowed down as the cache thrash overhead is directly exposed in single-thread execution. SHRll and SHRcm produce the worst result for the same reason. Excessive cache/TLB thrashes are exposed when a single thread operates over a gigantic memory region for partitioning. Besides, unlike the other PHJs, SHRll and SHRcm must acquire/release locks, which prevents them from using the efficient SWWCB-based "ntstore" for partitioning. As a result, shared partitioning's performance is far lagged behind.

7.6 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 $|R|$ and $|S|$ to 2^{30} and 16×2^{30} respectively, and plot the result in Figure 13(g). Note that the billion-scale workload demands the re-configuration for fanout and partitioning passes. Hence, we tune each PHJ accordingly and report their optimal performance [13].

Compared with the results of million-scale experiments, we can see a notable time explosion for all joins. The relative performance of NPHJ is consistent with its million-scale result (Section 7.1). 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 attributed to the re-configuration of partitioning fanouts. On the one hand, a large fanout shrinks the size of each partition, precluding cache thrashes for the subsequent join phase. On the other hand, a large fanout exceeds the cache or TLB capacity, which renders enormous cache/TLB thrashes and degenerates 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 the cache-sized hash table. Although these asymmetric fanouts require more read passes over S in the join phase (Section 3.2.2), the cost saving 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 8.3.

[13] The 1-pass partitioning still overcomes the 2-pass partitioning for all PHJs

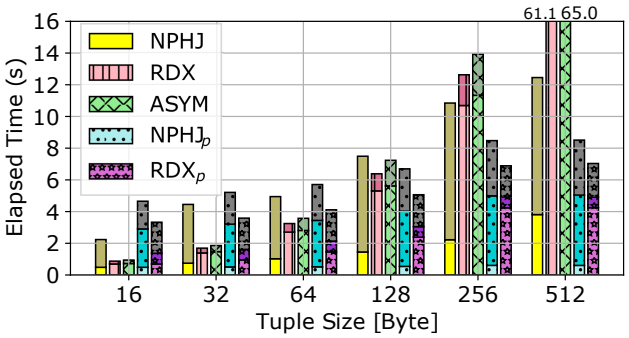


Figure 14: Performance w.r.t. Tuple Size (lighter colors indicate the build/partition phase while darker colors represent the probe/join phase; NPHJ$_p$ and RDX$_p$ represent the pointer-based variant of NPHJ and RDX, and their upper gray bars denote the time of the final retrieving phase) .

7.7 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 $< key, pointer >$ 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 14 depicts the result. For brevity, we only present results of NPHJ, RDX, ASYM, and two pointer-based implementations, NPHJ$_p$ with RDX$_p$. 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 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 $< key, pointer >$ 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 ($< key, pointer >$ 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.

7.8 Effect of NUMA

Recall that previous experiments are evaluated within a single socket, we now conduct an experiment to investigate the NUMA impact. In particular, we place the original relations, partitions, and hash tables in the first socket SCM while using the cores of the second socket (referred to as "remote"). For brevity, we take RDX as the PHJ representative, and other PHJs demonstrate similar performances.

Figure 15(a) shows the runtime of join execution. For better comparison, we also report the result of non-NUMA execution (referred to as "local"). We can see that both joins experience a runtime explosion in "remote" setting. To demystify this fact, we boil down to the hardware events. We first compare the LLC and TLB misses between "local" and "remote" but find no discernible difference. Hence, the main culprit must lie within SCM.

We proceed to measure the read/write requests of SCM and report the result in Figure 15(b)(c). We notice that "remote" causes a slight increase in SCM reads, but much more requests of SCM writes. Moreover, the increasing write requests can be observed in every phase of join execution, including NPHJ's probe phase, which is supposed to involve no write requests (cf. Figure 15(c) for dark yellow bars). The root cause of this phenomenon is the directory coherence protocol of our Testbed [42]. Xeon processors must maintain cache coherence directories when accessing memory. They keep the "local" directories within caches but leave the "remote" directories in memory. Any "remote" memory accesses, even "remote" reads, will generate writes to the "remote" memory for updating directories. Hence, the "remote" setting incurs costly SCM writes, resulting in a massive decline in join performance. Existing SCM-based NUMA studies [59] report similar findings and discover that the cache coherence writes can be significantly reduced under the snoop-based protocol. Thus, we decouple this performance decline from SCM technologies or join algorithms and maintain it will no longer exist in snoop-based platforms. Regardless of NUMA settings, RDX offers superior performance over NPHJ, revealing its strong robustness against NUMA.

7.9 Evaluation in Real Benchmarks

Now that we have developed a good understanding of joins in synthesized workloads, we can proceed to evaluate these joins in real benchmarks. We use TPC-H Query 14 with a 100 scaling factor for evaluation. In accordance with existing studies [10, 88], we use HyperDB [55] to generate the query plan and enforce selection pushdown and pipeline in join execution. Similarly, we take RDX as the PHJ representative for comparing NPHJ.

Figure 15(d) compares NPHJ with RDX. A notable observation is that the selection pushdown accounts for the majority (over 75%)

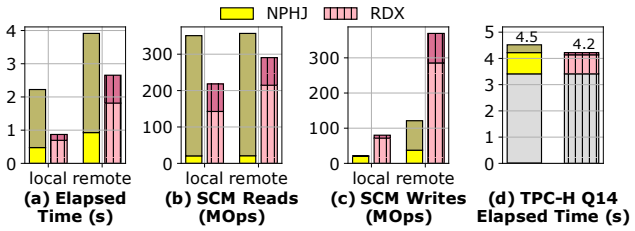


Figure 15: (a), (b), and (c) compares the performance of NUMA, (d) shows the elapsed time on TPC-H Query 14 with scale factor of 100. (lighter colors indicate the build/partition phase while darker colors represent the probe/join phase; for (d), the lower gray color denote the runtime of selection pushdown).

of runtime, which suggests that join processing may not be a bottleneck in a real query benchmark. The pushdown filters out over 98% of the build side tuples and downsizes tuples from 192B to 64B, considerably reducing the build and the probe side size. Aside from the pushdown, RDX still beats NPHJ by 27% (0.81s vs. 1.11s). The experiment demonstrates that RDX not only dominates in microbenchmarks but also ourperforms NPHJ in real queires. However, real queries can be affected by many factors. Compared to the join algorithm, it is more recommended to develop a good query plan or adopt an advanced query optimizer, which may generate more pronounced effects in real queries.

8 DISCUSSION

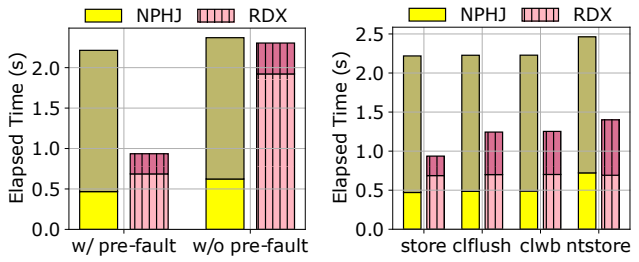
We now summarize our experiment findings and bring about a few auxiliary discussions for SCM-based joins.

8.1 Locality is All You Need

Previous experiments (Section 7) suggest that PHJs are generally better solutions than NPHJs. 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 7.2), 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 6.3). Moreover, the superiority of bucket chaining



(a) Effect of Page Faults **(b) Effect of Store Instructions**

Figure 16: Performance w.r.t. (a) Pre-Fault Mechanism (b) Different Store Instructions.

(BC), histogram mechanism (HR), and linear probing (LP) against separate chaining (SC) also validates the efficacy of high locality (Section 6.3). In a nutshell, high locality is the dominating factor that contributes to an efficient SCM-based join implementation.

8.2 SCM Overhead Analysis

We now address two NVDIMM-P primitives that fundamentally change the SCM-based join performance: costly page faults ($\mathscr{P}_4$) and persistent instructions ($\mathscr{P}_5$).

Costly page faults ($\mathscr{P}_4$). Page fault handling accounts for the major overhead in cache-friendly applications (cf. Figure 1(c)). It involves multiple hefty operations (e.g., CLR, CMPT [77]) and is automatically triggered whenever a page table entry is created. Note that page faults are inherent to Linux *mmap* mechanism [77] and are widely reckoned to be costly across various SCM technologies [3, 21, 49].

Although page faults are considered attentively in operating system designs [49, 52, 77], they are mostly overlooked by the query processing community [10, 56, 74, 88]. This is probably because page faults only consume minor or moderate overhead in DRAM-based query executors [50]. However, in SCM environment, page fault overhead can no longer be ignored, and our study is the first work to address its impact in SCM-based query processing.

Given that DBMSs tend to warm up buffers before query execution [10, 11, 50, 57, 64, 73, 95, 103], we thus pre-handle page faults in this phase with the SCM pre-fault [52, 77] mechanism. To elucidate this effect, we make an ablation comparison of the pre-fault mechanism. Figure 16(a) depicts the result. We can see that both joins are compromised if we do not pre-fault SCM pages. In particular, RDX's partition phase is compromised the most. Since it requires a large memory region for partitioning, its memory allocation generates far more page faults than other join phases. In contrast, the pre-fault mechanism eliminates these page faults ahead of join execution. The join throughput, therefore, circumvents the costly page fault handling on the fly. In consequence, the experimental comparison confirms the vast benefits of page fault pre-handling.

Persistent instructions ($\mathscr{P}_5$). Predominant SCM standards (e.g., NVDIMM-P [47], NPM [85]) have defined persistent instructions, which not only guarantee immediate persistence but also deliver high store bandwidth [12, 34]. However, as Figure 1(a)(b) indicate, persistent instructions are inappropriate in join processing. We thus perform an experiment to investigate the store instruction impact on SCM-based join processing.

Figure 16(b) demonstrates the results. We can see that persistent instructions ("clflush", "clwb", "ntstore") yield a runtime increase for both joins. Specifically, RDX's join phase is more sensitive to store instructions. As persistent instructions explicitly evict cachelines to SCM, RDX's join phase fails to buffer probing within caches, resulting in excessive cache thrashes and limiting join execution. Moreover, persistent instructions cause write amplification during writing, which wastes the precious resource of SCM bandwidth and compromises the join throughput. Overall, persistent instructions should not be employed in SCM-based join processing.

8.3 Read/Write Asymmetry in PHJ

As read/write asymmetry [$\mathscr{P}_3$] is widely acknowledged as an inherent SCM primitive [26, 78], write-limited algorithms [28, 69, 98, 99] have become a principle guideline for performance improvements. However, as Section 7 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-n-pass ASYM for comparison, as they deliver not only magnificent performance but also great applicability for a wide range of workloads [14]. Let $read$ and $write$ be the SCM bandwidth of read and write respectively, and λ 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-n-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 ϵ. We can derive the following function:

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

Since 2-pass RDX is able to partition a 4TB relation without thrashing cache/TLB, we set $m = 2$, $n = 1$ and convert the above function to:

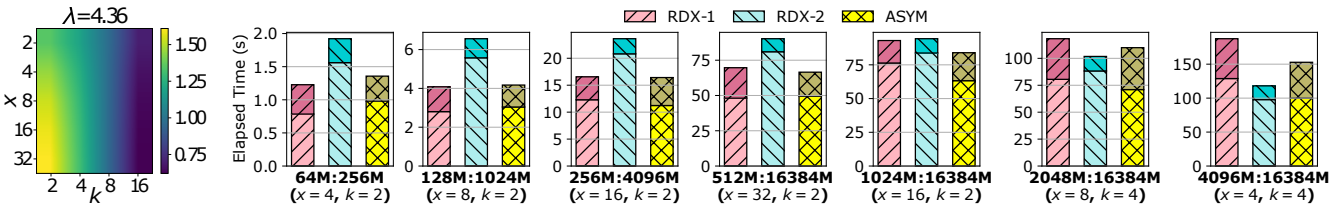
$$\epsilon = f(k, x, \lambda) = \frac{(x+1)(2\lambda + 5)}{(x+2)\lambda + (k+2)x + 5}. \quad (2)$$

In our platform, λ is close to 4.36 [15] if partitioning is properly configured without cache/TLB thrashes. We thus parameterize $f(k, x, 4.36)$ on k and x in Figure 17(a). As can be seen, the cost ratio ϵ ranges from 0.50 to 1.50, and it gets higher when x becomes larger and k gets smaller. In particular, for $k \leq 4$, ϵ becomes larger than 1, indicating that ASYM starts to surpass RDX. The ϵ is close to 1.50 for $x \geq 4$, which suggests that ASYM is at least 50% superior to RDX. We thus conclude that 2-1-pass ASYM should be applied on a workload with a large size ratio ($x \geq 4$) 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 17(b) compares the

[14] INDcm joins require larger memory footprint and are not applicable to skew workloads (Section 3.2.1).

[15] $read$ is 2.31GT/s while $write$ is 0.53GT/s, where GT/s denotes the Giga tuples per second.



(a) Cost Ratio ϵ

(b) Performance comparison between RDX joins and ASYM joins.

Figure 17: (a) is annotated with the cost ratio ϵ 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 cadinalities ($|R|$:$|S|$) and size ratio x (lighter colors represents the partition phase while darker colors stands for the join phase).

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. We make the following observations: First, RDX-1 is competitive for small cardinalities but gradually lags in large-scale workloads. The reason is that a high cardinality requires large partition fanouts, which makes RDX-1 hindered by excessive cache/TLB thrashes. Second, RDX-2 performs poorly in million-scale workloads but starts to dominate from 2048M cardinality onwards. This is because of RDX-2's additional partitioning overhead for splitting fanouts, which only pays off in large-scale datasets. Third, 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) [16], which corroborates the correctness of our cost ratio function. Compared to RDX-1, ASYM delivers comparable results in small-scale datasets and predominates when $|R| \geq$ 256M, which confirms its superiority in producing cache-sized partitions with moderate write cost. Specifically, we can also 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 [10, 24, 81] and can be reduced to this scale by selection pushdown [79, 88], we maintain that ASYM can be incorporated to query plans for upcoming SCM-based DBMSs.

Since other SCM may have different read/write asymmetries (λ) [26], we derive the partial derivative of ϵ with respect to λ:

$$\frac{\partial \epsilon}{\partial \lambda} = \frac{\partial f(k,x,\lambda)}{\partial \lambda} = \frac{(2k-1)(x+1)x}{[(x+2)\lambda + (k+2)x + 5]^2}. \quad (3)$$

Given that $k > 1$ forever holds, the above partial derivative is always positive. To cut a long story short, for SCM with larger read/write asymmetry, ASYM will render more performance gains over RDX [17].

8.4 Future SCM and Beyond

Through extensive experiments, we conclude that PHJ is generally the better solution (Section 7). We also provide practical tips (Section 5, 6) for configuring efficient join implementations. Unfortunately, Intel shuts down the Optane DIMM business [44] 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. Although there are currently no commercial alternatives, we address that our study will remain valuable for the following reasons.

First, SCMs are inevitable. As SCM technology is initially proposed for breaking the DRAM scaling wall [40, 86], its necessity is not going to die. Meanwhile, SRAM and flash have their own scaling challenges, which can be resolved by deploying SCM in various storage tiers [23, 78]. As widespread deployment leads to high production volume and high volume drives down the production cost [23], the financial problem that fails Optane will no longer be a concern.

Second, our conclusion that PHJ-over-NPHJ is based on the evaluations in PCM [48], which is slightly slower than DRAM [26, 30]. Given that existing DRAM-based studies [6, 9, 88] also prefer PHJ to NPHJ and most SCMs offer the access speed in the range of DRAM and SCM access speed, the PHJ-over-NPHJ conclusion shall never fade away.

Third, the proposed tips (Section 5, 6) are mainly based on the fundamental primitives of NVDIMM-P, a predominant SCM standard that most manufacturers adhere. As future SCM products will presumably follow this standard, these primitives are likely to remain in place. Even though some primitives may get altered by some SCM prototypes (e.g., different internal granularity [$\mathscr{P}_1$], disparate on-DIMM buffer size [$\mathscr{P}_2$]), it is still easy to extend our tips accordingly and make them function in these devices.

Last but not least, our studies serve as valuable references to other modern storage technologies. In particular, modern SSD products intend to support byte-addressability and fast random access by encapsulating internal byte-addressable buffers [1, 5], and would benefit from the intrinsic ideas of our configuration tips (Section 5, 6). Moreover, future memory technologies will probably conform to the emerging CXL standard [22], which sacrifices access latency to avoid bandwidth contention [53]. Hence, the latency-bound NPHJs [10] are likely to be more bottlenecked, leaving PHJs the preferred approaches in CXL. Furthermore, the read/write asymmetry is universal across most storage devices, and it even becomes more striking for particular storage technologies [26, 78]. Therefore, the asymmetric partitioning idea provides a certain significance and deserves more thorough investigations for future storage systems.

[16] Partitioning at this scale exceeds the cache/TLB limit; hence, we reduce S partition fanouts (i.e., increase k to 4) to mitigate the thrashing overhead.

[17] We also have the same conclusion for 1-pass RDX and ASYM but omit the function/derivative for brevity.

9 RELATED WORK

We now briefly review some existing works. Generally, our research relates to two camps of studies: main-memory hash joins and SCM system studies.

Main-Memory Hash Joins. Main-memory hash joins have been rigorously studied for almost thirty years. Shatdal et al. [92] 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 el al. [14, 75, 76] 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 [6–9, 58] extend their idea to parallel query processing and develop a performant PHJ implementation. Meanwhile, Blanas et al. [13] maintain that modern hardwares effectively conceal the cache miss overhead, which makes partitioning unnecessary and leads NPHJ in beating PHJ. Afterward, Schuh et al. [88] compare PHJ with NPHJ in microbenchmarks and proclaim that PHJ generally outperforms NPHJ. However, Bandle et al. [10] later conduct the comparison in TPC-H [24] 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 [12, 25, 34, 97, 101, 104, 106] characterize its access profile, providing several practices for better utilizing the hardware. Some other researchers develop SCM-friendly data structures [37, 39, 59, 71], which exploit SCM's non-volatility for fast recovery. Other researches focus more on general SCM technologies rather than Optane DIMMs; they mostly follow the NVDIMM-P specification and propose designs for logging [4], file system [105], memory security [35], etc.

Unfortunately, few efforts have been made for SCM-based join processing. Viglas [98] first studies the read/write asymmetry impact in join and Shanbhag et al. [90] revisit his findings in Optane DIMMs. Besides, Daase et al. [25] and Lasch et al. [64] reexamine query benchmarks in Optane DIMMs. Nonetheless, they target traditional external joins, failing to exploit the SCM's byte-addressability in join processing. Maltenberger at el. [74] 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.

10 CONCLUSION

This paper revisits main-memory hash joins in the context of 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 hardwares. 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.

REFERENCES

[1] Ahmed H. M. O. Abulila, Vikram Sharma Mailthody, Zaid Qureshi, Jian Huang, Nam Sung Kim, Jinjun Xiong, and Wen-Mei W. Hwu. 2019. FlatFlash: Exploiting the Byte-Accessibility of SSDs within a Unified Memory-Storage Hierarchy. In *ASPLOS*. ACM, 971–985.

[2] AgigaTech. 2022. AGIGARAM® NVDIMM-N. http://agigatech.com/products/agigaram-nvdimms/

[3] 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.

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

[5] Shuhan Bai, Hu Wan, Yun Huang, Xuan Sun, Fei Wu, Changsheng Xie, Hung-Chih Hsieh, Tei-Wei Kuo, and Chun Jason Xue. 2022. Pipette: efficient fine-grained reads for SSDs. In *DAC*. ACM, 385–390.

[6] 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.

[7] 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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] Lawrence Benson, Leon Papke, and Tilmann Rabl. 2022. PerMA-Bench: Benchmarking Persistent Memory Access. *Proc. VLDB Endow.* 15, 11 (2022), 2463–2476.

[13] Spyros Blanas, Yinan 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.

[14] 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.

[15] 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.

[16] Daniel Castro, Alexandro Baldassin, João Barreto, and Paolo Romano. 2021. SPHT: Scalable Persistent Hardware Transactions. In *FAST*. USENIX Association, 155–169.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] Jungsik Choi, Jiwon Kim, and Hwansoo Han. 2017. Efficient Memory Mapped File I/O for In-Memory File Systems. In *HotStorage*. USENIX Association.

[22] CXL Consortium. 2022. Compute Express Link (CXL) Specification. https://www.computeexpresslink.org/_files/ugd/0c1418_1798ce97c1e6438fba818d760905e43a.pdf

[23] 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

[24] Transaction Processing Performance Council. 2021. TPC BENCHMARKTM H (Decision Support) Standard Specification Revision 3.0.0. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf

[25] 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.

[26] Tim Daulby, Anand Savanth, Alex S. Weddell, and Geoff V. Merrett. 2020. Comparing NVM Technologies through the Lens of Intermittent Computation. In *ENSsys@SenSys*. ACM, 77–78.

[27] 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).

[28] 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.

[29] Alexandra Fedorova, Keith Smith, Keith Bostic, Alexander Gorrod, Sue LoVerso, and Michael Cahill. 2022. Writes Hurt: Lessons in Cache Design for Optane NVRAM. *CoRR* abs/2205.14122 (2022).

[30] Ivan Fernandez, Aditya Manglik, Christina Giannoula, Ricardo Quislant, 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).

[31] Bill Gervasi. 2019. Will Carbon Nanotube Memory Replace DRAM? *IEEE Micro* 39, 2 (2019), 45–51.

[32] 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.

[33] Caixin Gong, Chengjin Tian, Zhengheng Wang, Sheng Wang, Xiyu Wang, Qiulei 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.

[34] Shashank Gugnani, Arjun Kashyap, and Xiaoyi Lu. 2020. Understanding the Idiosyncrasies of Real Persistent Memory. *Proc. VLDB Endow.* 14, 4 (2020), 626–639.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] Intel. 2014. Intel® VTune Profiler. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html

[42] Intel. 2020. Directory Structure in Skylake Server CPUs. https://community.intel.com/t5/Software-Tuning-Performance/Directory-Structure-in-Skylake-Server-CPUs/td-p/1185376

[43] Intel. 2020. Intel® PMWatch. https://github.com/intel/intel-pmwatch/

[44] 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,billion%2C%20down%2017%25%20YoY.&text=Intel's%20Client%20Computing%20and%20Datacenter,Mobileye%20achieved%20record%20quarterly%20revenue.

[45] Intel. 2022. Intel® Optane™ Persistent Memory. https://www.intel.sg/content/www/xa/en/architecture-and-technology/optane-dc-persistent-memory.html

[46] JEDEC. 2018. DDR4 NVDIMM-N DESIGN SPECIFICATION. https://www.jedec.org/standards-documents/docs/jesd248

[47] JEDEC. 2021. DDR4 NVDIMM-P BUS PROTOCOL. https://www.jedec.org/system/files/docs/JESD304-4-01.pdf

[48] 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.

[49] Rohan Kadekodi, Saurabh Kadekodi, Soujanya Ponnapalli, Harshad Shirwadkar, Gregory R. Ganger, Aasheesh Kolli, and Vijay Chidambaram. 2021. WineFS: a hugepage-aware file system for persistent memory that ages gracefully. In *SOSP*. ACM, 804–818.

[50] 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.

[51] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alex Rasin, Stanley B. Zdonik, Evan P. C. Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J. Abadi. 2008. H-store: a high-performance, distributed main memory transaction processing system. *Proc. VLDB Endow.* 1, 2 (2008), 1496–1499.

[52] Vasileios Karakostas, Jayneel Gandhi, Furkan Ayar, Adrián Cristal, Mark D. Hill, Kathryn S. McKinley, Mario Nemirovsky, Michael M. Swift, and Osman S. Unsal. 2015. Redundant memory mappings for fast access to large memories. In *ISCA*. ACM, 66–78.

[53] 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 *USENIX Annual Technical Conference*. USENIX Association, 821–837.

[54] 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.

[55] Alfons Kemper and Thomas Neumann. 2011. HyPer: A hybrid OLTP&OLAP main memory database system based on virtual memory snapshots. In *ICDE*. IEEE Computer Society, 195–206.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] Dimitrios Koutsoukos, Raghav Bhartia, Ana Klimovic, and Gustavo Alonso. 2021. How to use Persistent Memory in your Database. *CoRR* abs/2112.00425 (2021).

[63] 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.

[64] 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.

[65] 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.

[66] Sekwon Lee, Soujanya Ponnapalli, 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).

[67] 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.

[68] 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.

[69] 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.

[70] Sihang Liu, Suraaj 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)*.

[71] 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.

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

[73] 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.

[74] 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.

[75] 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.

[76] 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.

[77] 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.

[78] 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.

[79] Yoon-Min Nam, Donghyoung Han, and Min-Soo Kim. 2020. SPRINTER: A Fast n-ary Join Query Processing Method for Complex OLAP Queries. In *SIGMOD Conference*. ACM, 2055–2070.

[80] 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.

[81] 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.

[82] Oracle. 2022. MySQL Database. https://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v3.0.0.pdf

[83] Ismail Oukid. 2018. *Architectural Principles for Database Systems on Storage-Class Memory*. Ph. D. Dissertation. Dresden University of Technology, Germany.

[84] Ismail Oukid. 2019. Architectural Principles for Database Systems on Storage-Class Memory. In *BTW (LNI, Vol. P-289)*. Gesellschaft für Informatik, Bonn, 477–486.

[85] 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

[86] Moinuddin K. Qureshi. 2014. Memory Scaling is Dead, Long Live Memory Scaling. https://hps.ece.utexas.edu/yale75/qureshi_slides.pdf

[87] 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.

[88] 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.

[89] 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.

[90] 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.

[91] 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/

[92] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. 1994. Cache Conscious Algorithms for Relational Query Processing. In *VLDB*. Morgan Kaufmann, 510–521.

[93] Ali Sheikholeslami and P. Glenn Gulak. 2000. A survey of circuit innovations in ferroelectric random-access memories. *Proc. IEEE* 88, 5 (2000), 667–689.

[94] 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

[95] 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.

[96] Daniel Terpstra, Heike Jagode, Haihang You, and Jack J. Dongarra. 2009. Collecting Performance Data with PAPI-C. In *Parallel Tools Workshop*. Springer, 157–173.

[97] 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.

[98] Stratis Viglas. 2014. Write-limited sorts and joins for persistent memory. *Proc. VLDB Endow.* 7, 5 (2014), 413–424.

[99] 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.

[100] 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.

[101] 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.

[102] 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)*.

[103] 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.

[104] 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.

[105] 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.

[106] 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.

[107] Jifei Yi, Mingkai Dong, Fangnuo Wu, and Haibo Chen. 2022. HTMFS: Strong Consistency Comes for Free with Hardware TBBDSANBADBHABDH ransactional Memory in Persistent Memory File Systems. In *FAST*. USENIX Association, 17–34.

[108] Bowen Zhang, Shengan Zheng, Zhenlin Qi, and Linpeng Huang. 2022. NBTree: a Lock-free PM-friendly Persistent B+-Tree for eADR-enabled PM Systems. *Proc. VLDB Endow.* 15, 6 (2022), 1187–1200.

[109] 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.

[110] Zuyu Zhang, Harshad Deshmukh, and Jignesh M. Patel. 2019. Data Partitioning for In-Memory Systems: Myths, Challenges, and Opportunities. In *CIDR*. www.cidrdb.org.

[111] 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.

[112] Xinjing Zhou, Joy Arulraj, Andrew Pavlo, and David Cohen. 2021. Spitfire: A Three-Tier Buffer Manager for Volatile and Non-Volatile Memory. In *SIGMOD Conference*. ACM, 2195–2207.