Evaluating In-Memory Hash Joins on Persistent Memory

Tobias Maltenberger*, Till Lehmann*, Lawrence Benson, Tilmann Rabl
{tobias.maltenberger,till.lehmann}@student.hpi.de,{lawrence.benson,tilmann.rabl}@hpi.de
Hasso Plattner Institute, University of Potsdam

ABSTRACT

Steady advances in processor and memory technologies have driven continuous tuning and redesigning of in-memory hash joins for decades. Over the years, research has shown advantages of both hardware-conscious radix joins and hardware-oblivious hash joins for different workloads. In this paper, we evaluate both join types on persistent memory (PMem) as an emerging memory technology offering close-to-DRAM speed at significantly higher capacities. We study the no partitioning join (NPO) and the parallel radix join (PRO) in PMem and analyze how their performance differs from DRAM-based execution. Our results show that the PRO is always at least as fast as the NPO in DRAM. However, in PMem, the NPO outperforms the PRO by up to 1.7×. Based on our findings, we provide an outlook into crucial design choices for PMem-optimized hash join implementations.

1 INTRODUCTION

Modern in-memory database systems heavily utilize hash joins. The interplay of state-of-the-art hardware features and database workloads serves as a broad research field for tuning parameters and (re-)designing hash join algorithms. Over the past decades, fundamental shifts in processor and storage technology regularly challenged researchers to revisit the search for the optimal hash join implementation [1, 4, 5, 9, 12, 15–17, 22, 22, 23, 27, 30]. In general, there are two types of hash joins: *hardware-conscious* algorithms and *hardware-oblivious* approaches. While hardware-conscious hash joins entail a platform-specific partition phase to reduce the number of cache and translation lookaside buffer (TLB) misses [4, 5, 27], the less complex hardware-oblivious approaches come entirely without a partition phase [9]. Some researchers claim that hardware-conscious hash joins are superior to hardware-oblivious ones in terms of performance [5, 28, 31]. Others show that in a real-world database system, hardware-conscious hash joins outperform their non-partitioned hardware-oblivious counterparts only for a few workloads [7].

Common to the majority of published in-memory hash joins is that they operate exclusively in DRAM. Yet, persistent memory (PMem) promises higher capacity at lower cost and Byte-addressability at close-to-DRAM performance [14, 35, 36, 38]. Intel Optane DC Persistent Memory Modules (Optane DC PMM) is the first commercially available PMem technology [20]. Since the type of I/O operation, access size, access pattern, and the number of threads heavily impact Intel Optane DC PMM's performance, best practices for its usage have been established [8, 13, 36, 38]. Researchers proposed various PMem-based index structures [2, 11, 24], hash maps [25, 26], and database components [3, 34, 37]. Consequently, PMem might be a cheaper alternative or extension to DRAM for efficient in-memory hash joins as well.

*Both authors contributed equally.

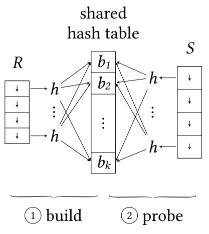


Figure 1: No partitioning join

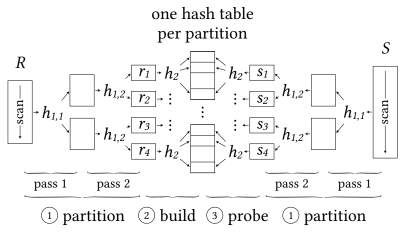


Figure 2: Parallel radix join

In this paper, we adapt two in-memory hash joins by Balkesen et al. to PMem: the optimized hardware-oblivious no partitioning join (NPO) and the optimized hardware-conscious parallel radix join (PRO) [5]. We choose these baselines as they are widely used and clearly show the differences between their concepts in DRAM and PMem, making our results applicable to variations of both algorithms. For both hash joins, we evaluate three different variants via embedded performance counters. First, DRAM-only, where all data resides in DRAM. Second, PMem-only, where both input relations and the hash tables are stored in PMem. Third, PMem-relations, where the two relations are located in PMem, but the (partitioned) hash tables reside in DRAM.

We show that the number of writes is the deciding factor of both joins' overall performance in PMem. The NPO outperforms the PRO by 1.7× in PMem due to fewer writes. In DRAM, the PRO outperforms the NPO, as DRAM-writes perform significantly better. Our measurements reveal the NPO's and PRO's PMem-only variant to be 3.3-6.0× and 5.5× slower than their DRAM-only counterpart. However, the hash joins' PMem-relations variants, where PMem is used as read-only memory for the two input relations, reach competitive performance to the DRAM-only mode. Their performance is slower by only 1.1× for the NPO and 1.4× for the PRO. Guided by our experimental analysis, we outline crucial design goals for future PMem-aware hash joins.

With this paper, we make the following contributions. First, we adapt two in-memory hash join implementations by Balkesen et al. – the optimized no partitioning join (NPO) and the parallel radix join (PRO) – to PMem. Second, we evaluate both algorithms in DRAM and PMem. Third, we provide insights into key design choices for PMem-aware hash join implementations.

2 IN-MEMORY HASH JOINS

There are two classes of in-memory hash joins: *hardware-oblivious* implementations that are independent of the platform characteristics and *hardware-conscious* approaches that exploit certain platform features via tunable performance knobs [6]. Balkesen et al. analyze whether hardware-oblivious hash joins are competitive to fine-tuned alternatives on modern hardware with prefetching and out-of-order execution [5]. We evaluate whether the authors' findings hold when working in PMem by analyzing the no partitioning join and the parallel radix join.

No Partitioning Join. Blanas et al. propose the *no partitioning join* as a parallel adaptation of the canonical hash join (see Figure 1) [9]. It is independent of platform-specific parameters and operates without data partitioning. Besides, it splits both relations, R and S, across multiple threads into equal-sized portions. During the build phase, the threads populate a shared hash table from R and synchronize via a barrier ①. In the probe phase, the worker threads find join partners for their portion of S in the hash table that they access in a read-only fashion ②.

Parallel Radix Join. Cache misses through random memory lookups are a major drawback of hardware-oblivious joins [10]. Shatdal et al. find that partitioning the hash table into small cache-sized chunks reduces cache misses [33]. The authors' partitioned hash join divides the input relations R and S with $|R| < |S|$ into partitions r_i and s_j using hash partitioning. In the build phase, it creates a separate hash table for each r_i partition that fits into the CPU cache. In the probe phase, it scans each s_j partition and probes the hash table for matching tuples. Since the partitions reside in different virtual memory pages, the hash join algorithm suffers from a large number of TLB misses [5].

Manegold et al. [27] propose the in-memory radix join, as depicted in Figure 2. In the partition phase, it partitions the two input relations R and S through a two-phase radix partitioning with the hash functions $h_{1,1}$ and $h_{1,2}$, respectively ①. After that, the radix join builds the hash table over all r_i partitions of R ②. In the probe phase, it scans all s_i partitions of S and probes the corresponding r_i hash tables to obtain the join matches. Consequently, the radix join does $log(|R|)$ passes over the input relations R and S as the maximum fan-out of each pass is limited by the CPU's number of TLB entries. Another configuration parameter is the partition size. It should roughly align with the CPU's cache size [5]. Kim et al. propose the *parallel radix join* as an extension for multi-core systems where the relations R and S are divided into chunks and processed by individual threads [23].

3 PERSISTENT MEMORY

PMem is an emerging class of memory devices bridging the gap between DRAM and flash storage. It provides Byte-addressable random access and data persistence. PMem's bandwidth and latency characteristics are comparable to DRAM's, while its capacity is closer to that of flash-based devices. Intel Optane DC PMM DIMMs are available with larger capacities than DRAM DIMMs and lower cost per GiB [18]. We use the terms PMem and Intel Optane DC PMM interchangeably.

The CPU communicates with PMem via an integrated memory controller at 64 Byte cache line granularity. However, Intel Optane DC PMM DIMMs have an internal granularity of 256 Byte, causing read and write amplification for smaller access sizes. To mitigate this, the DIMMs contain write-combining buffers that buffer adjacent writes before flushing them [36]. In Table 1, we show the peak bandwidth and latency for random data access of DRAM and PMem. We observe that random read access in PMem has a 2.4× higher latency than in DRAM due to slower media access. The latency of random writes is 5.3× higher in PMem than in DRAM. Consequently, algorithms should aim for writing sequentially into PMem. PMem achieves 40% for reads and 19% for write operations of DRAM's peak bandwidth.

Table 1: Peak bandwidth and latency

	READ		WRITE	
	Bandwidth	**Latency**	**Bandwidth**	**Latency**
DRAM	100 GiB/s	190 ns	70 GiB/s	170 ns
PMem	40 GiB/s	450 ns	13 GiB/s	900 ns

4 EXPERIMENTAL SETUP

In this section, we outline the implementations as well as workload and platform configurations for our evaluation.

Implementations. We adapt Balkesen et al.'s NPO (hardware-oblivious) and PRO (hardware-conscious) to PMem [5].

DRAM-only. The DRAM-only implementation is the original implementation by Balkesen et al. and serves as a baseline. We generate both input relations R and S randomly and store them together with all auxiliary data structures in DRAM.

PMem-only. In the PMem-only variant, both input relations R and S as well as the temporary data structures reside in PMem. We replace all dynamic memory allocations with calls to a custom PMem allocator. Our PMem allocator initially reserves consecutive PMem space and returns pointers to PMem chunks of the requested size. Only a negligible constant overhead of < 5 MiB stack space resides in DRAM regardless of the workload.

PMem-relations. The PMem-relations variant represents a trade-off between the DRAM-only and the PMem-only variants. It harnesses the limited, but fast DRAM for the hash joins' probing tables and the large but slower PMem for the input relations R and S. Since all random writes during radix partitioning or probing the hash tables are targeted to DRAM, PMem is used in a read-only fashion. The PMem-relations variant represents a database system where data resides persistently in PMem, but query processing happens in DRAM. The overall storage capacity of the (in-memory) database system can exceed DRAM capacity while maintaining a reasonable read bandwidth to the data.

Workload Configuration. We use the workload by Blanas et al. [9]. It entails a column-oriented storage model with $\langle key, payload \rangle$ tuples, where key and $value$ are each 8 Byte. The build relation R contains 16×2^{20} (~ 16M, 256 MiB) tuples. The probe relation S contains 256×2^{20} (~ 268M, 4096 MiB) tuples. The uniformly distributed keys of R and S follow a foreign key relationship – every tuple in S has exactly one join partner in R.

System Configuration. We benchmark both hash joins on a system with an Intel Xeon Gold 5220S CPU featuring 18 cores at 2.7 GHz. The CPU's L1, L2, and L3 cache sizes are 1.1 MiB, 18 MiB, and 24.75 MiB. The CPU comes with L1 TLBs (32 entries) and a shared L2 TLB (1536 entries), utilizing 2 MiB pages. The system has 6× 128 GiB Intel Optane DC PMM 100 Series DIMMs and 6× 16 GiB DDR4 DRAM DIMMs. PMem is interleaved and used in App Direct mode. The system runs on Ubuntu 20.04 LTS. We compile with `g++` 9.3.0 and the `-O3` optimization level. We instrument our implementations with the Intel Performance Counter Monitor [21]. Besides, we conduct three runs for each experiment and show the arithmetic mean of the results.

5 EXPERIMENTAL EVALUATION

In this section, we analyze the NPO and the PRO in their DRAM-based and PMem-based implementations. Figure 3 shows the algorithms' runtimes. We see that PMem-only variants perform worse than the DRAM-based ones due to PMem's decreased read/write bandwidth. For the PMem-relations variant, we see that it is slower in single-threaded execution but almost as fast as DRAM-based variants in multi-threaded execution given the favorable scaling of sequential PMem read access. In DRAM, NPO and PRO perform very similarly. When moving to PMem, the PRO performs considerably worse than the NPO because of increased write access to PMem during partitioning. *Due to the different performance characteristics of reads and writes, DRAM optimizations for joins cannot be applied to PMem directly.*

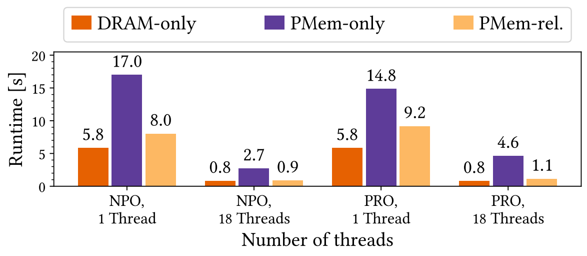


Figure 3: Comparison of the runtime between single and multi-threaded variants of NPO and PRO

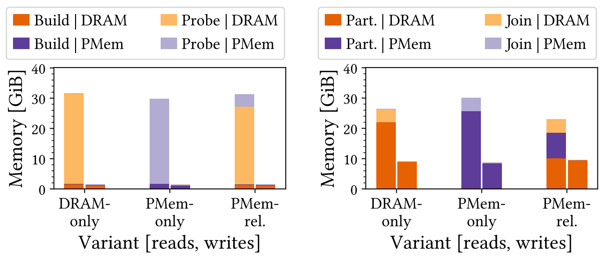
5.1 Memory Access

In Figure 4, we study memory access patterns. Comparing the amount of data read to and written from the memory controller by the NPO (see Figure 4a), we observe that memory access is dominated by read operations. The absence of write accesses is why the NPO outperforms the PRO in PMem. Write operations to PMem have significantly lower bandwidth and higher latency than to DRAM. The NPO performs write operations only for the smaller relation – to create the probing table. Since the size of both input relations differs by a factor of 16, the probing phase's read operations dominate the overall runtime.

In contrast, the PRO needs to perform multi-pass partitioning for both relations before joining them, writing the relations to PMem twice. In Figure 4b, we see that the PRO writes almost 10 GiB to the memory controller during the partition phase while the NPO writes less than 1.5 GiB. The additional data written to the memory controller is the bottleneck of the PRO.

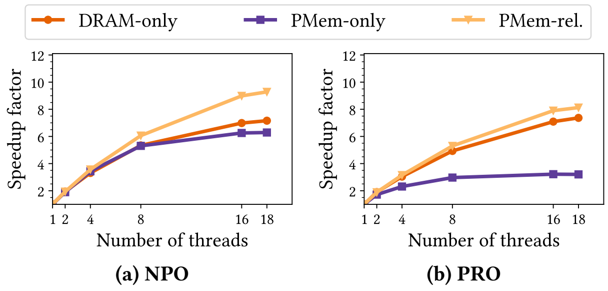
The memory access pattern of the PMem-relations variant differs only slightly from that of the DRAM-only implementation. Figure 4 reveals that no data is written to PMem since it is used in read-only mode. Sequentially reading both input relations from PMem in the NPO accounts for only 15% of all read operations, limiting the impact of PMem's lower bandwidth and higher latency on the overall execution time. However, during the PRO's first partitioning pass, the relations are read twice – once for calculating the histogram and once for partitioning the data. Consequently, the amount of data read from PMem is doubled. In the PMem-relations implementation of the PRO, 35% of read accesses are served from PMem, which leads to a 40% increase in the overall runtime compared to DRAM.

In conclusion, for the sequential read operations of both in-memory hash join algorithms, PMem performs close to DRAM. However, when writing during the hash joins, PMem performs worse due to its asymmetric bandwidth characteristics (see Table 1). While DRAM's bandwidth is 2.5× higher than PMem's for read accesses, it is 5.5× higher for write operations.



(a) NPO **(b) PRO**

Figure 4: Amount of data read from and written to the memory controller



(a) NPO **(b) PRO**

Figure 5: Speedup achieved through multi-threading compared to single-threaded executions

5.2 Thread Scalability

In Figure 5, we study the thread scaling behavior of both hash joins. All NPO implementations achieve a noticeable speedup by utilizing multiple threads (see Figure 5a). With 18 threads, the PMem-relations variant achieves the highest speedup of 9.3×, considerably reducing the performance gap between the DRAM-only and PMem-relations implementations compared to the single-threaded execution. In the single-threaded execution, the PMem-relations variant takes 38% more time than the DRAM-only version. With 18 threads, the remaining runtime overhead of the PMem-relations variants decreases to only 6%.

During the single-threaded execution, the performance is bound by the bandwidth of sequential reads from PMem. Since sequential PMem reads scale better than random DRAM writes, the bottleneck shifts towards the bandwidth of random writes to DRAM when utilizing more threads. Since the bottleneck of the DRAM-only variant is the same, their runtimes converge.

The NPO's PMem-only variant has a lower peak speedup of 6.3× than its DRAM-only variant at 7.2×, leading to a higher performance gap when utilizing the entire CPU's compute power. Up until 18 threads, though, all variants show desirable multi-threading scaling. Since the NPO is read-dominant, we observe that it benefits from parallelization in PMem and DRAM.

Regarding the PRO's thread scaling behavior, we notice a significant speedup of the PMem-relations and DRAM-only variants (see Figure 5b). At the highest CPU utilization with 18 threads, both implementations achieve a speedup of 8.1× and 7.3×, respectively, compared to the single-threaded execution. The performance improvements of thread scaling in the PMem-only variant are significantly lower. With 8 threads, it reaches a speedup of merely 3× – without any considerable increase upwards. We observe the maximum speedup of 3.2× with 16 threads.

The PRO's partitioning phases are dominated by write accesses to PMem to re-partition both input relations. Recent studies show that PMem writes quickly suffer from over-saturation when using numerous threads [13, 36]. However, since multiple threads do not negatively affect PMem and DRAM read operations, the DRAM-only and PMem-relations implementations do not reflect this behavior. Write operations, thus, impose a disadvantage for the PRO in PMem. In conclusion, when employing the PRO in PMem, few parallel threads are sufficient to achieve close to optimal performance during the partitioning phase.

5.3 Data Partitioning vs. Random Writes

In Figure 6, we analyze the NPO's and the PRO's performance for a workload with equal-sized relations. If both relations have the same size, the probing table becomes larger, and the NPO incurs more random write accesses. Although both input relations have the same size, building a probing table requires more time than

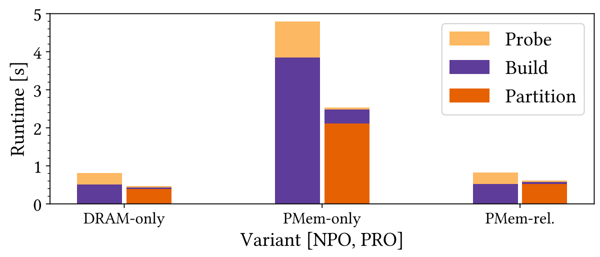


Figure 6: Runtime for workload with equal-sized relations

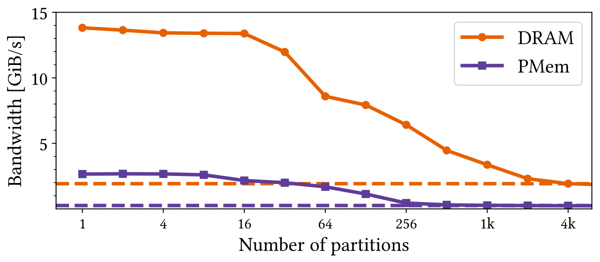


Figure 7: Write bandwidth (solid lines: data partitioning, dashed lines: random writes)

probing a relation of the same size. We observe a drastic decline in NPO's performance, caused by a slower build phase.

The performance benefit of the PRO mainly originates from the partition phase. It enables the join operation to benefit from high cache utilization. However, as a fundamental premise, partitioning the input relations has to perform quicker than building the probing table from an unpartitioned relation. In order to investigate whether building a probing table of a relation is inherently slower than partitioning the relation, we conduct a micro-benchmark comparing their write access patterns.

We initially allocate a contiguous chunk of values with a total size of 2 GiB. A value is 8 Bytes long, resembling a $\langle key, payload \rangle$ tuple where *key* and *value* is each 4 Byte. First, to mimic the access pattern of building a probing table, we write each value exactly once but in random order. Without collisions, we assume the best-case scenario of a uniform distribution in the probing table. Second, to emulate the access pattern of data partitioning, we divide the array into N partitions. The data is written sequentially inside each partition. However, the order in which partitions are written in each iteration is random.

In Figure 7, we study the write bandwidth of DRAM and PMem. Dashed lines represent the write bandwidth of random writes at 2 GiB/s for DRAM and 0.3 GiB/s for PMem. Solid lines depict the bandwidth of data partitioning, which highly depends on the number of partitions. With only one partition, we measure the performance of sequential writes to be 13.8 GiB/s for DRAM and 2.6 GiB/s for PMem. Therefore, applying data partitioning to mitigate random write overhead remains viable in PMem.

The cores on our system have ten write buffers [19]. Consequently, a core can combine 8 Byte writes into 64 Byte ones without switching the cache line for up to ten partitions. Starting at 8 to 16 partitions, the write bandwidth decreases gradually. Beginning at 32 partitions, the number of partitions reaches the number of entries in the L1 TLB. After this point, TLB misses become more frequent and negatively impact the performance. More partitions further reduce the bandwidth until it converges to the minimum bandwidth of random writes.

To enable high cache utilization during the join phase, each partition must be small enough to fit into the core-local cache. When an input relation becomes too big, partitioning cannot be conducted efficiently. Therefore, the PRO introduces multi-pass partitioning, which reduces the number of partitions.

6 DISCUSSION

We observe that adapting a DRAM-optimized hash join to PMem does not automatically yield the best performance. Our results show that writes to PMem constitute the central performance bottleneck in PMem-based hash joins. Due to PMem's read/write asymmetry, slower and limited writes negatively impact performance more than in DRAM. Future PMem-aware hash joins must put a focus on optimizing the build phase to achieve better performance, e.g., by adopting published DRAM optimizations [29, 32]. One idea is to perform only one partition pass or buffer before writing to PMem. The partitioning concepts of the PRO can further be incorporated to avoid low random write performance as the hash table stays in each CPU's cache without entailing random flushes to PMem. A precondition for this is to not have any PMem-resident data in the caches before building the tables since that would cause random cache evictions to PMem.

Performing another write phase before building the hash table is unnecessary if the build side is small. While the additional partitioning pays off in DRAM, it does not in PMem. To improve the performance of partitioning in PMem, in-memory hash joins should improve write accesses for small sequential blocks (e.g., by varying the store operations and the size of flushed blocks). It requires optimizing for the case that sequential PMem bandwidth drops for more than 8 threads. While the probe phase can scale to many threads, the build side's thread scaling behavior is upper bounded. Consequently, more fine-grained thread scaling in the build phase may yield further performance gains.

Recent work shows that radix partitioned hash joins often do not perform better than their non-partitioned counterparts in real-world main memory database systems [7]. To understand how PMem is most efficiently utilized in modern database systems, future work should further investigate PMem-aware hash joins in the context of larger systems and more workloads.

7 CONCLUSION

In this paper, we adapt two in-memory hash joins to PMem: the NPO, a hardware-oblivious join, and the PRO, a hardware-conscious algorithm. We benchmark the two hash joins in PMem and DRAM to show their performance characteristics on both memory technologies and re-evaluate whether common wisdom on optimized implementations applies to PMem.

We show that the number and type of writes to PMem is the deciding factor for both hash joins' performance. Although the PRO outperforms the NPO in DRAM, it does not in PMem. Due to fewer write operations, the NPO performs up to 1.7× better than the PRO. Furthermore, we demonstrate that the PMem-relations variants of NPO and PRO, where PMem is used in a read-only mode, reach competitive performance to their respective DRAM-only implementations. Storing the relations in PMem enables joining significantly larger tables, as DRAM is not used for storage. Since different memory technology characteristics impact hash join performance, we provide insights for designing future PMem-optimized hash join implementations.

ACKNOWLEDGMENTS

This work was partially funded by the European Union's Horizon 2020 research and innovation program (ref. 957407).

REFERENCES

[1] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. 1999. DBMSs on a Modern Processor: Where Does Time Go?. In *Proc. VLDB Endow. (VLDB '99)*. ACM, New York, NY, USA, 266–277. https://doi.org/10.5555/645925.671662

[2] J. Arulraj, J. Levandoski, U. F. Minhas, and P.-A. Larson. 2018. Bztree: A High-Performance Latch-Free Range Index for Non-Volatile Memory. *Proc. VLDB Endow.* 11, 5 (January 2018), 553–565. https://doi.org/10.14778/3164135.3164147

[3] J. Arulraj, A. Pavlo, and S. R. Dulloor. 2015. Let's Talk about Storage and Recovery Methods for Non-volatile Memory Database Systems. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. ACM, New York, NY, USA, 707–722. https://doi.org/10.1145/2723372.2749441

[4] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. 2013. Multi-Core, Main-Memory Joins: Sort vs. Hash Revisited. *Proc. VLDB Endow.* 7, 1 (September 2013), 85–96. https://doi.org/10.14778/2732219.2732227

[5] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. 2013. Main-Memory Hash Joins on Multi-Core Cpus: Tuning to the Underlying Hardware. In *2013 IEEE 29th International Conference on Data Engineering (ICDE '13)*. IEEE, New York, NY, USA, 362–373. https://doi.org/10.1109/ICDE.2013.6544839

[6] C. Balkesen, J. Teubner, G. Alonso, and M. T. Özsu. 2015. Main-Memory Hash Joins on Modern Processor Architectures. *IEEE Transactions on Knowledge and Data Engineering* 27, 7 (2015), 1754–1766. https://doi.org/10.1109/TKDE.2014.2313874

[7] M. Bandle, J. Giceva, and T. Neumann. 2021. To Partition, or Not to Partition, That Is the Join Question in a Real System. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. ACM, New York, NY, USA, 168–180. https://doi.org/10.1145/3448016.3452831

[8] L. Benson, H. Makait, and T. Rabl. 2021. Viper: An Efficient Hybrid PMem-DRAM Key-Value Store. *Proc. VLDB Endow.* 14, 9 (2021), 1544–1556. https://doi.org/10.14778/3461535.3461543

[9] S. Blanas, Y. Li, and J. M. Patel. 2011. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. ACM, New York, NY, USA, 37–48. https://doi.org/10.1145/1989323.1989328

[10] P. A. Boncz, S. Manegold, and M. L. Kersten. 1999. Database Architecture Optimized for the New Bottleneck: Memory Access. In *Proc. VLDB Endow. (VLDB '99)*. ACM, New York, NY, USA, 54–65. https://doi.org/10.5555/645925.671364

[11] S. Chen and Q. Jin. 2015. Persistent B+-Trees in Non-Volatile Main Memory. *Proc. VLDB Endow.* 8, 7 (February 2015), 786–797. https://doi.org/10.14778/2752939.2752947

[12] X. Cheng, B. He, X. Du, and C. T. Lau. 2017. A Study of Main-Memory Hash Joins on Many-Core Processor: A Case with Intel Knights Landing Architecture. In *Proceedings of the 2017 ACM on Conference on Information and Knowledge Management (CIKM '17)*. ACM, New York, NY, USA, 657–666. https://doi.org/10.1145/3132847.3132916

[13] B. Daase, L. J. Bollmeier, L. Benson, and T. Rabl. 2021. Maximizing Persistent Memory Bandwidth Utilization for OLAP Workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD '21)*. ACM, New York, NY, USA, 339–351. https://doi.org/10.1145/3448016.3457292

[14] P. Götze, A. K. Tharanatha, and K.-U. Sattler. 2020. Data Structure Primitives on Persistent Memory: An Evaluation. In *Proceedings of the 16th International Workshop on Data Management on New Hardware (DaMoN '20)*. ACM, New York, NY, USA, 1–3. https://doi.org/10.1145/3399666.3399900

[15] B. He, M. Lu, K. Yang, R. Fang, N. K. Govindaraju, Q. Luo, and P. V. Sander. 2009. Relational Query Coprocessing on Graphics Processors. *ACM Trans. Database Syst.* 34, 4 (December 2009), 1–39. https://doi.org/10.1145/1620585.1620588

[16] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. 2008. Relational Joins on Graphics Processors. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 511–524. https://doi.org/10.1145/1376616.1376670

[17] M. Heimel, M. Saecker, H. Pirk, S. Manegold, and V. Markl. 2013. Hardware-Oblivious Parallelism for In-Memory Column-Stores. *Proc. VLDB Endow.* 6, 9 (July 2013), 709–720. https://doi.org/10.14778/2536360.2536370

[18] Intel. 2019. *Intel® Optane™ DC Persistent Memory Product Brief*. Intel. Retrieved June 28, 2021 from https://www.intel.de/content/dam/www/public/us/en/documents/product-briefs/optane-dc-persistent-memory-brief.pdf

[19] Intel. 2021. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*. Intel. Retrieved June 28, 2021 from https://software.intel.com/content/dam/develop/external/us/en/documents-tps/64-ia-32-architectures-optimization-manual.pdf

[20] Intel. 2021. *Intel® Optane™ Persistent Memory*. Intel. Retrieved June 28, 2021 from https://www.intel.com/content/www/us/en/architecture-and-technology/optane-dc-persistent-memory.html

[21] Intel. 2021. *PCM: Processor Counter Monitor*. Intel. Retrieved June 28, 2021 from https://github.com/opcm/pcm

[22] S. Jha, B. He, M. Lu, X. Cheng, and H. P. Huynh. 2015. Improving Main Memory Hash Joins on Intel Xeon Phi Processors: An Experimental Approach. *Proc. VLDB Endow.* 8, 6 (February 2015), 642–653. https://doi.org/10.14778/2735703.2735704

[23] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. 2009. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. *Proc. VLDB Endow.* 2, 2 (August 2009), 1378–1389. https://doi.org/10.14778/1687553.1687564

[24] L. Lersch, X. Hao, I. Oukid, T. Wang, and T. Willhalm. 2019. Evaluating Persistent Memory Range Indexes. *Proc. VLDB Endow.* 13, 4 (December 2019), 574–587. https://doi.org/10.14778/3372716.3372728

[25] B. Lu, X. Hao, T. Wang, and E. Lo. 2020. Dash: Scalable Hashing on Persistent Memory. *Proc. VLDB Endow.* 13, 8 (April 2020), 1147–1161. https://doi.org/10.14778/3389133.3389134

[26] B. Lu, X. Hao, T. Wang, and E. Lo. 2021. Scaling Dynamic Hash Tables on Real Persistent Memory. *SIGMOD Rec.* 50, 1 (June 2021), 87–94. https://doi.org/10.1145/3471485.3471506

[27] S. Manegold, P. Boncz, and M. Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering* 14, 4 (2002), 709–730. https://doi.org/10.1109/TKDE.2002.1019210

[28] P. Menon, T. C. Mowry, and A. Pavlo. 2017. Relaxed Operator Fusion for In-Memory Databases: Making Compilation, Vectorization, and Prefetching Work Together at Last. *Proc. VLDB Endow.* 11, 1 (September 2017), 1–13. https://doi.org/10.14778/3151113.3151114

[29] Ingo Müller, Peter Sanders, Arnaud Lacurie, Wolfgang Lehner, and Franz Färber. 2015. Cache-Efficient Aggregation: Hashing Is Sorting. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1123–1136. https://doi.org/10.1145/2723372.2747644

[30] C. Pohl and K.-U. Sattler. 2018. Joins in a Heterogeneous Memory Hierarchy: Exploiting High-Bandwidth Memory. In *Proceedings of the 14th International Workshop on Data Management on New Hardware (DAMON '18)*. ACM, New York, NY, USA, 1–10. https://doi.org/10.1145/3211922.3211929

[31] S. Schuh, X. Chen, and J. Dittrich. 2016. An Experimental Comparison of Thirteen Relational Equi-Joins in Main Memory. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1961–1976. https://doi.org/10.1145/2882903.2882917

[32] 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 (may 2015), 934–937. https://doi.org/10.14778/2777598.2777602

[33] A. Shatdal, C. Kant, and J. F. Naughton. 1994. Cache Conscious Algorithms for Relational Query Processing. In *Proc. VLDB Endow. (VLDB '94)*. ACM, New York, NY, USA, 510–521. https://doi.org/10.5555/645920.758363

[34] A. van Renen, K. Leis, A. Kemper, T. Neumann, T. Hashida, K. Oe, Y. Doi, L. Harada, and M. Sato. 2018. Managing Non-Volatile Memory in Database Systems. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD '18)*. ACM, New York, NY, USA, 1541–1555. https://doi.org/10.1145/3183713.3196897

[35] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. 2019. Persistent Memory I/O Primitives. In *Proceedings of the 15th International Workshop on Data Management on New Hardware (DaMoN'19)*. ACM, New York, NY, USA, 1–7. https://doi.org/10.1145/3329785.3329930

[36] A. van Renen, L. Vogel, V. Leis, T. Neumann, and A. Kemper. 2020. Building Blocks for Persistent Memory. *The VLDB Journal* 29, 6 (2020), 1223–1241. https://doi.org/10.1007/s00778-020-00622-9

[37] Y. Wu, K. Park, R. Sen, B. Kroth, and J. Do. 2020. Lessons Learned from the Early Performance Evaluation of Intel Optane DC Persistent Memory in DBMS. In *Proceedings of the 16th International Workshop on Data Management on New Hardware (DaMoN '20)*. ACM, New York, NY, USA, 1–3. https://doi.org/10.1145/3399666.3399898

[38] J. Yang, J. Kim, M. Hoseinzadeh, J. Izraelevitz, and S. Swanson. 2020. An Empirical Guide to the Behavior and Use of Scalable Persistent Memory. In *18th USENIX Conference on File and Storage Technologies (FAST '20)*. USENIX, Santa Clara, CA, USA, 169–182.