SPID-Join: A Skew-resistant Processing-in-DIMM Join Algorithm Exploiting the Bank- and Rank-level Parallelisms of DIMMs

SUHYUN LEE, Yonsei University, South Korea
CHAEMIN LIM, Yonsei University, South Korea
JINWOO CHOI, Yonsei University, South Korea
HEELIM CHOI, Yonsei University, South Korea
CHAN LEE, Yonsei University, South Korea
YONGJUN PARK, Yonsei University, South Korea
KWANGHYUN PARK, Yonsei University, South Korea
HANJUN KIM, Yonsei University, South Korea
YOUNGSOK KIM, Yonsei University, South Korea

Recent advances in Dual In-line Memory Modules (DIMMs) allow DIMMs to support Processing-In-DIMM (PID) by placing In-DIMM Processors (IDPs) near their memory banks. Prior studies have shown that in-memory joins can benefit from PID by offloading their operations onto the IDPs and exploiting the high internal memory bandwidth of DIMMs. Aimed at evenly balancing the computational loads between the IDPs, the existing algorithms perform IDP-wise global partitioning on input tables and then make each IDP process a partition of the input tables. Unfortunately, we find that the existing PID join algorithms achieve low performance and scalability with skewed input tables. With skewed input tables, the IDP-wise global partitioning incurs imbalanced loads between the IDPs, making the IDPs remain idle until the heaviest-load IDP completes processing its partition. To fully exploit the IDPs for accelerating in-memory joins involving skewed input tables, therefore, we need a new PID join algorithm which achieves high skew resistance by mitigating the imbalanced inter-IDP loads.

In this paper, we present *SPID-Join*, a skew-resistant PID join algorithm which exploits two parallelisms inherent in DIMM architectures, namely bank- and rank-level parallelisms. By replicating join keys across the banks within a rank and across ranks, SPID-Join significantly increases the internal memory bandwidth and computational throughput allocated to each join key, improving the load balance between the IDPs and accelerating join executions. SPID-Join exploits the bank- and the rank-level parallelisms to minimize join key replication overheads and support a wider range of join key replication ratios. Despite achieving high skew resistance, SPID-Join exhibits a trade-off between the join key replication ratio and the join execution

Authors' Contact Information: Suhyun Lee, su_hyun@yonsei.ac.kr, Department of Computer Science and Engineering, Yonsei University, Seoul, South Korea; Chaemin Lim, cmlim@yonsei.ac.kr, Department of Computer Science and Engineering, Yonsei University, Seoul, South Korea; Jinwoo Choi, jinwoo1029@yonsei.ac.kr, Department of Computer Science and Engineering, Yonsei University, Seoul, South Korea; Heelim Choi, heelim@yonsei.ac.kr, School of Electrical and Electronic Engineering, Yonsei University, Seoul, South Korea; Chan Lee, chan.lee@yonsei.ac.kr, School of Electrical and Electronic Engineering, Yonsei University, Seoul, South Korea; Yongjun Park, yongjunpark@yonsei.ac.kr, Department of Computer Science and Engineering, Yonsei University, Seoul, South Korea; Kwanghyun Park, kwanghyun.park@yonsei.ac.kr, Department of Computer Science and Engineering, Yonsei University, Seoul, South Korea; Hanjun Kim, hanjun@yonsei.ac.kr, School of Electrical and Electronic Engineering, Yonsei University, Seoul, South Korea; Youngsok Kim, youngsok@yonsei.ac.kr, Department of Computer Science and Engineering, Yonsei University, Seoul, South Korea.

ACM 2836-6573/2024/12-ART251
https://doi.org/10.1145/3698827

latency, making the best-performing join key replication ratio depend on join and PID system configurations. We, therefore, augment SPID-Join with a cost model which identifies the best-performing join key replication ratio for given join and PID system configurations. By accurately modeling and scaling the IDPs' throughput and the inter-IDP communication bandwidth, the cost model accurately captures the impact of the join key replication ratio on SPID-Join. Our experimental results using eight UPMEM DIMMs, which collectively provide a total of 1,024 IDPs, show that SPID-Join achieves up to 10.38× faster join executions over PID-Join, the state-of-the-art PID join algorithm, with highly skewed input tables.

CCS Concepts: • **Information systems** → **Join algorithms**; **Main memory engines**; • **Hardware** → **Dynamic memory**; *Memory and dense storage*.

Additional Key Words and Phrases: in-memory join, processing-in-memory, processing-in-DIMM, data skew

ACM Reference Format:
Suhyun Lee, Chaemin Lim, Jinwoo Choi, Heelim Choi, Chan Lee, Yongjun Park, Kwanghyun Park, Hanjun Kim, and Youngsok Kim. 2024. SPID-Join: A Skew-resistant Processing-in-DIMM Join Algorithm Exploiting the Bank- and Rank-level Parallelisms of DIMMs. *Proc. ACM Manag. Data* 2, 6 (SIGMOD), Article 251 (December 2024), 27 pages. https://doi.org/10.1145/3698827

1 Introduction

Recent advances in Dual In-line Memory Modules (DIMMs), currently the de-facto standard for implementing host memory, allow DIMMs to support Processing-In-DIMM (PID) by placing In-DIMM Processors (IDPs) near their memory banks [19, 36]. PID can accelerate applications suffering from the memory wall by letting them offload memory-intensive operations onto the IDPs. By offloading the operations onto the IDPs, the applications can exploit the high internal memory bandwidth of DIMMs and minimize the data movement between host Central Processing Units (CPUs) and DIMMs [26]. Commodity PID-enabled DIMMs have not been available until recently. However, with the introduction of UPMEM DIMM [19] and Samsung AxDIMM [36], PID is gaining increasing interest in various application domains (e.g., bioinformatics [14, 21, 39], machine learning [18, 24, 25, 34, 52], and security [27, 52]).

In-memory databases, which frequently suffer from the memory wall [10, 22, 28, 29, 49, 50, 64], have been shown to greatly benefit from PID [6–8, 26, 33, 44, 61]. In particular, prior studies [44, 61] accelerate in-memory joins, a key operation of the databases [5, 9, 13, 46–48, 60], by proposing PID join algorithms which offload the operations of the joins onto the IDPs. Given two tables R and S, where $|R| \leq |S|$ and R consists of unique join keys, the algorithms perform $R \bowtie S$ as follows. First, host CPUs evenly scatter the tuples of R and S to all the IDPs and make the IDPs perform IDP-wise global partitioning on their tuples. Then, the CPUs shuffle the tuples between the IDPs so that each of the IDPs receives all the tuples belonging to its R and S partitions. After that, the IDPs perform single-IDP joins on their R and S partitions, followed by the CPUs gathering the output tuples from all the IDPs. In this way, the algorithms achieve fast in-memory join executions by exploiting the high internal memory bandwidth of DIMMs for all the steps of an in-memory join [9, 11, 47, 56, 60].

Unfortunately, we find that the existing PID join algorithms achieve low performance and scalability with skewed input tables, making it difficult for in-memory databases to employ the algorithms for accelerating joins. The algorithms employ IDP-wise global partitioning to evenly balance the computational loads between the IDPs [44, 61]. With skewed input tables, however, the IDP-wise global partitioning incurs severe inter-IDP load imbalance, making all the IDPs remain idle until the heaviest-load IDP completes processing its R and S partitions. This is due to the IDP-wise global partitioning allowing a join key, regardless of its popularity, to be processed by only a single IDP's limited internal memory bandwidth and computational throughput. Furthermore, as the algorithms make all the IDPs synchronously perform each step of a join [44, 61], the heaviest-load IDP's high execution latency gets exposed and slows down every step of a join. Therefore, to

Table 1. Characteristics of the existing processing-in-DIMM join algorithms and SPID-Join

PID Join Algorithm	Skew Resistance	Input Tuple Distribution ($\|R\| \leq \|S\|$)		$\|R\| : \|S\|$ Ratios
		R	S	
UPMEM-Join [61]	Low	IDP-wise Global Partitioning	IDP-wise Global Partitioning	1:1†
PID-Join [44]	Low	IDP-wise Global Partitioning	IDP-wise Global Partitioning	Any
SPID-Join (This work)	**High**	**Bank- & Rank-wise Replication+Partitioning**	**Replication-aware Global Partitioning**	**Any**

†The only $|R| : |S|$ ratio supported by the publicly-available artifact of UPMEM-Join [61]

facilitate wider adoption of PID for accelerating in-memory joins, we need a new PID join algorithm which achieves high skew resistance by mitigating the severe inter-IDP load imbalance.

In this paper, we present *SPID-Join*, a skew-resistant PID join algorithm which exploits two parallelisms inherent in DIMMs, namely bank- and rank-level parallelisms, to mitigate the inter-IDP load imbalance and achieve high skew resistance. The key idea of SPID-Join is to allocate higher internal memory bandwidth and computational throughput to each join key by replicating join keys across ranks and across the banks within a rank. This allows SPID-Join to relax the significant burden imposed on the heaviest-load IDP due to skewed input tables and process a popular join key using multiple IDPs and their high aggregate internal memory bandwidth and computational throughput. When replicating join keys, SPID-Join first aims to replicate join keys across the banks within a rank by leveraging the burst length of DIMMs (i.e., the minimum data access granularity of DIMMs) which provides high join key replication bandwidth. SPID-Join then further extends its supported range of join key replication degrees by exploiting the rank-level parallelism and replicating join keys across the memory banks of different ranks. In this way, SPID-Join can mitigate the severe load imbalance between the IDPs of the existing algorithms and greatly accelerate join executions even with skewed input tables.

We then augment SPID-Join with a cost model which identifies the best-performing replication ratio by taking as input the target join and PID system configurations. SPID-Join exhibits a trade-off between the join key replication ratio and the join execution latency as replicating join keys across ranks and across the banks within a rank involves different inter-IDP communication patterns and join key replication bandwidths. The cost model, therefore, is essential for not only minimizing the join execution latency by identifying the best-performing replication ratio but also for allowing SPID-Join to be easily employed by the existing in-memory databases. By leveraging its detailed modeling of the IDPs' throughput and the inter-IDP communication bandwidth, the cost model accurately identifies the best-performing replication ratio using only a few characteristics of input tables (e.g., the Zipf factor of S) and the underlying PID system (e.g., the number of PID-enabled ranks). SPID-Join uses the cost model to identify the best-performing replication ratio first for a given join, and then executes the join with the replication ratio by exploiting the bank- and rank-level parallelisms.

We evaluate SPID-Join on a real system equipped with eight UPMEM DIMMs [19], currently the only publicly-available commodity PID-enabled DIMMs. The eight UPMEM DIMMs provide a total of 1,024 IDPs and 716.8 GB/s of aggregate internal memory bandwidth. Our evaluation using 80 benchmarks with varying Zipf factors shows that SPID-Join achieves up to 10.38× faster join executions over PID-Join [44], the state-of-the-art PID join algorithm. The evaluation further shows that SPID-Join accelerates the joins of TPC-H queries [65], when their input tables are skewed, by up to 4.88× over PID-Join. The results clearly demonstrate SPID-Join's high skew resistance, performance, and scalability achieved by successfully mitigating the severe inter-IDP load imbalance.

To the best of our knowledge, SPID-Join is the first PID join algorithm to tackle skewed input tables. Table 1 summarizes the characteristics of the existing PID join algorithms and SPID-Join. In summary, this paper makes the following key contributions:

- We reveal that the existing PID join algorithms achieve limited performance and scalability with skewed input tables by incurring severe load imbalance between the IDPs.
- We propose SPID-Join, a skew-resistant PID join algorithm which mitigates skewed input tables by allocating higher internal memory bandwidth and computational throughput to each join key using the bank- and rank-level parallelisms of DIMMs.
- We augment SPID-Join with a cost model which models SPID-Join's trade-off between the join key replication ratio and the join execution latency and can identify the best-performing join key replication ratio for given join and PID system configurations.
- Using a real system equipped with UPMEM DIMMs, we show that SPID-Join greatly outperforms the state-of-the-art PID-Join [44] when performing joins on skewed input tables.

2 Background

2.1 Dual In-line Memory Modules

Dual In-line Memory Modules (DIMMs), currently the de facto standard for implementing host memory, are equipped with multiple banks to provide high memory bandwidth and capacity. The DIMM has a hierarchical structure of banks and ranks (i.e., a DIMM consists of multiple ranks and each of the ranks consists of multiple banks). The host CPUs access the data stored in a DIMM through a memory channel using bursts; a burst is the minimum data access granularity supported by DIMMs. The burst length is set to 64 bits (or eight bytes) for Double Data Rate 4 (DDR4) DIMMs [32]. To increase the memory bandwidth, DIMMs employ byte-interleaving which interleaves the sequential bytes of a burst to memory banks [32]. For example, a DDR4 DIMM distributes the eight bytes of a burst to and serves the burst using eight different memory banks.

Due to their hierarchical bank organization, DIMMs exhibit two architecture-level parallelisms: bank-level parallelism and rank-level parallelism. First, the bank-level parallelism stems from the fact that a DIMM consists of several memory banks and serves a memory request by accessing a set of the memory banks in parallel (e.g., access eight memory banks in parallel for serving a burst in a DDR4 DIMM). Second, the rank-level parallelism refers to the parallelism between the ranks of a DIMM. Each rank of a DIMM owns its dedicated set of control signals, so the ranks of the DIMM can operate independently and concurrently serve different memory requests. Using the two parallelisms, DIMMs achieve high memory bandwidth by serving multiple memory requests in parallel.

2.2 Processing-in-DIMM

Thanks to the recent advances in the internal architecture of DIMMs, modern commodity DIMMs such as UPMEM DIMM [19] and Samsung AxDIMM [41] now support Processing-In-DIMM (PID) by placing In-DIMM Processors (IDPs) near their memory banks. One key advantage of PID is that it allows applications to exploit the high internal memory bandwidth of a DIMM, much higher than the memory bandwidth provided to the host CPUs through a memory channel. For example, a standard DDR4-2400 DIMM provides 19.2 GB/s of memory bandwidth, while 128 IDPs of an UPMEM DIMM provide 89.6 GB/s of internal memory bandwidth, theoretically.

Fig. 1 shows the internal architecture of UPMEM DIMM, currently the only publicly-available commercial PID-enabled DIMM. An UPMEM DIMM consists of two ranks, eight chips per rank, and eight memory banks per chip. Each of the memory banks provides 64 MB of capacity and 700 MB/s of memory bandwidth. The DIMM extends a standard DDR4-2400 DIMM by associating an IDP

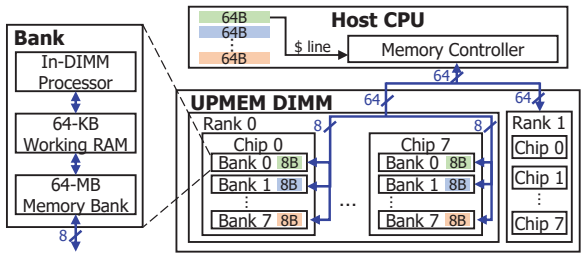


Fig. 1. The internal architecture of UPMEM DIMM

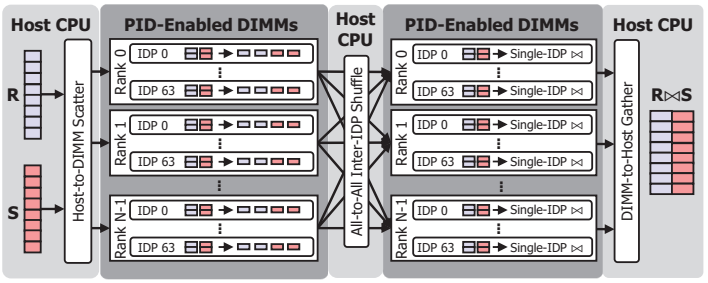


Fig. 2. End-to-end working model of PID-Join [44]

and a 64-KB Working RAM (WRAM) to each memory bank. Each of the 350-MHz scalar, in-order, and multi-threaded IDPs implements a load-store architecture and supports up to 24 concurrent hardware threads. An IDP can directly access the data stored in its WRAM. Accessing the data stored in the memory bank requires the IDP to load the data to the WRAM in advance.

As PID-enabled DIMMs are typically designed to be a drop-in replacement for standard DIMMs, all the architectural characteristics of the standard DIMMs, including the bank- and the rank-level parallelisms, equally apply to PID-enabled DIMMs. One of the architectural characteristics is the CPU-mediated inter-IDP data transfer. Due to the tight chip area, timing, and power consumption constraints imposed on DIMMs [1], the IDPs implement a shared-nothing architecture and thus cannot directly access the data stored in the other IDPs' WRAMs and memory banks. The lack of fast hardware-level inter-IDP data transfer paths requires the host CPUs to mediate any inter-IDP data transfer [26]. For example, when transferring data between the IDPs of a rank, the host CPUs first collect the data from the (source) banks, transpose the collected data according to the bank interleaving, and send the data to the (destination) banks [44, 66]. The host CPU-mediated data transfer frequently bottlenecks the performance of PID-accelerated applications [26, 40], so careful inter-IDP data transfer optimizations are necessary for fully exploiting the large potential of PID.

2.3 Processing-in-DIMM Join Algorithms

To exploit the benefits of PID for in-memory joins, a key operation of in-memory databases [5, 9, 13, 46–48, 60], prior studies [44, 61] have proposed PID join algorithms which offload the operations of an in-memory join onto the IDPs. The algorithms employ IDP-wise global partitioning to balance the computational loads between the IDPs and fully exploit all the aggregate internal memory bandwidth and computational throughput of the IDPs. By leveraging all the IDPs in every step of an in-memory join, the algorithms have been shown to outperform CPU-based in-memory join algorithms.

Assuming that all the IDPs' loads are evenly balanced, the algorithms break down the execution of a join into five steps and make all the IDPs synchronously perform each of the steps. Fig. 2 depicts how PID-Join [44], the state-of-the-art PID join algorithm, performs $R \bowtie S$ on two input tables R

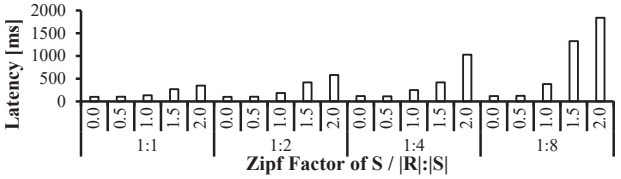


Fig. 3. $R \bowtie S$ execution latencies of PID-Join on the PID-enabled system with varying Zipf factors of S ($|R|$ = 0.5 M)

and S, initially stored in the host memory, where $|R| \leq |S|$. First, the host CPUs evenly scatter the tuples of R and S from the host memory to all the IDPs. Second, the IDPs perform IDP-wise global partitioning on their R and S tuples. Third, the host CPUs make all the IDPs shuffle their tuples so that the tuples get transferred to their target IDPs according to the global partitioning. Fourth, the IDPs perform single-IDP join on their R and S partitions and produce the join results for the partitions. For example, a single-IDP hash join further partitions the given R and S partitions to fit within the WRAM size. Then, 24 hardware threads build and probe the WRAM-fit hash table with the partitions in parallel. Lastly, the host CPUs gather the per-IDP join results from all the IDPs and complete executing the join.

3 Motivation

3.1 Slow Join Executions with Skewed Tables

Despite achieving large speedups over CPU-based in-memory join algorithms, we find that the existing PID join algorithms suffer from low performance and scalability with skewed input tables. To examine and quantify the impact of skewed input tables on the existing algorithms, we profile and analyze their join execution latencies on a real PID-enabled system. We configure the PID-enabled system to match that used by Lim et al. [44] for evaluating PID-Join. The system is equipped with eight UPMEM DIMMs, which collectively provide 16 ranks, 1,024 IDPs, and 716.8 GB/s of internal memory bandwidth. The system uses an Intel Xeon Gold 5222 CPU [31] as its host CPU and two standard DDR4-2400 DIMMs for the host memory. The system has five memory channels, each attached with two DIMMs and providing 19.2 GB/s of channel bandwidth. The eight UPMEM DIMMs are attached to four memory channels and the standard DIMMs are attached to the other memory channel. We focus on a join scenario with a table containing unique keys, showing a uniform distribution, and the other table with non-unique keys, which may exhibit skew (e.g., PK-FK join) [12]. This setup is common across real-world benchmarks spanning all queries of TPC-DS, TPC-H, and Join Order Benchmark [43, 51, 71].

Using the PID-enabled system, we first find that the performance of the existing PID join algorithms gets severely degraded with skewed input tables. Fig. 3 shows the impact of the Zipf factor of S on the $R \bowtie S$ execution latencies of PID-Join. We configure R to have 0.5-M uniformly-distributed tuples and consider four $|R| : |S|$ ratios (from 1:1 to 1:8). The results indicate that PID-Join achieves low skew resistance and thus suffers from skewed input tables. Ideally, regardless of an increase in the Zipf factor of S, the join execution latencies should remain consistent as long as the $|R| : |S|$ ratio remains the same; only the changes in $|R|$ and $|S|$ affect the amount of compute the IDPs should process for a join. However, the results show that the join execution latencies significantly increase as the Zipf factor of S increases. For example, when $|R| : |S|$ is 1:8, increasing the Zipf factor of S from 0.0 to 2.0 greatly increases PID-Join's join execution latency by 16.2× (118 ms vs 1,909 ms).

We also find that skewed input tables greatly limit the scalability of the existing PID join algorithms. Fig. 4 shows how the Zipf factor of S affects PID-Join's join execution latencies when

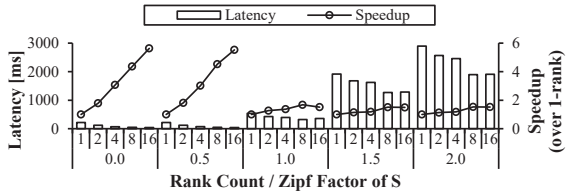
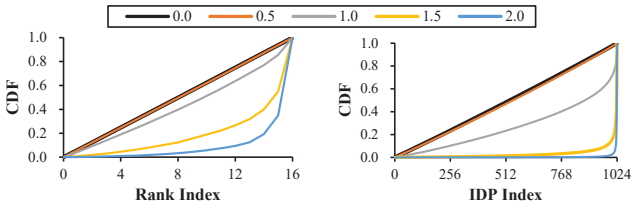


Fig. 4. PID-Join's join execution latencies with varying Zipf factors of S and rank counts ($|R|$ = 0.5 M, $|R|$: $|S|$ = 1:8)



(a) Inter-rank load imbalance (b) Inter-IDP load imbalance

Fig. 5. Distribution of the per-rank and -IDP S tuple counts of PID-Join on the 16-rank PID-enabled system with varying Zipf factors of S ($|R|$ = 0.5 M, $|R|$: $|S|$ = 1:8)

increasing the number of PID-enabled ranks from one to 16. The join execution latencies of PID-Join should decrease when we increase the PID-enabled rank count; increasing the rank count increases the number of IDPs, and thus the aggregate computational throughput and internal memory bandwidth PID-Join can be exploited. The results show that PID-Join achieves good scalability with less skewed Ss. Increasing the rank count from one to 16 allows PID-Join to achieve a speedup of 5.63× when the Zipf factor of S is 0.0. However, as the Zipf factor of S increases, PID-Join no longer scales with the number of PID-enabled ranks; the join execution latencies remain similar to those of single-rank configuration despite increases in the rank count. For example, when S's Zipf factor is set to 2.0, PID-Join achieves a limited speedup of 1.52× with 16 PID-enabled ranks.

3.2 Severe Inter-IDP Load Imbalance

Our further analyses reveal that the IDP-wise global partitioning of the existing PID join algorithms is a key contributor to their slow join executions with skewed input tables. The existing algorithms employ global partitioning to evenly balance the computational loads between the IDPs; however, with skewed records, the global partitioning causes a severe imbalance between the IDPs due to varying join keys' popularity. Fig. 5 depicts the severe inter-IDP load imbalance due to the IDP-wise global partitioning. To balance the loads between the IDPs, similar numbers of join keys should be assigned to each rank and the IDPs. However, the results show that, with skewed input tables, IDP-wise global partitioning incurs large discrepancies in the numbers of join keys assigned to the ranks and the IDPs. Furthermore, as the Zipf factor increases, the inter-rank and the inter-IDP load imbalances become more severe. For example, when the Zipf factor of S increases to 2.0, the heaviest-load IDP gets assigned more than 2 M tuples, whereas there are 66 lightest-load IDPs that do not get assigned any tuples of S.

As the existing algorithms make all the IDPs synchronously perform each step of an in-memory join, the severe inter-IDP load imbalance forces all the IDPs to remain idle until the heaviest-load IDP completes its partitions at every step of a join. Furthermore, as the skew of input tables increases, the inter-IDP load imbalance, and thus the slowdown of the existing algorithms, becomes more severe. Fig. 6 depicts the per-rank and -IDP join execution timelines of PID-Join for an example $R \bowtie S$ involving a high-skew S. As the timelines demonstrate, the severe inter-rank and -IDP load imbalances translate to significant differences in the per-rank and -IDP join execution latencies and

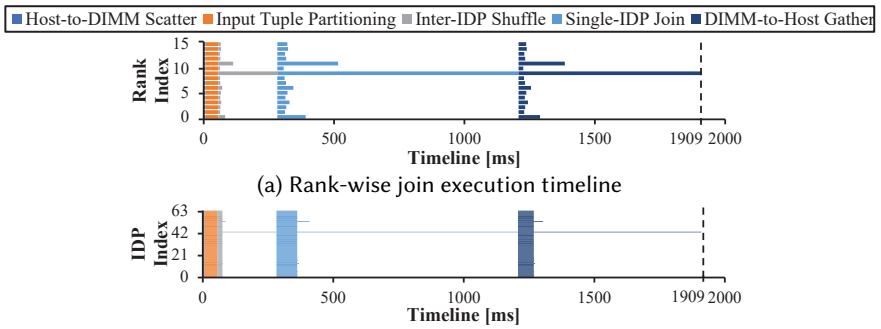


Fig. 6. Per-rank and per-IDP join execution timelines of PID-Join on the 16-rank PID-enabled system ($|R|$ = 0.5 M, $|R|$: $|S|$ = 1:8, S's Zipf factor = 2.0)

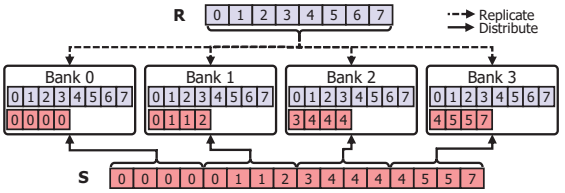


Fig. 7. Our prototype PID broadcasting algorithm's replication of join keys across all memory banks

make the heaviest-load rank and IDP slow down every step of the join. The load imbalances become significant from the inter-IDP shuffle step of PID-Join; the IDPs incur similar execution latencies for the prior steps (i.e., the host-to-DIMM scatter, the IDP-wise global partitioning) as the tuples of R and S get evenly distributed across the IDPs. After the IDP-wise global partitioning, however, the IDPs operate on their partitions of R and S and exhibit the severe inter-IDP load imbalance.

4 SPID-Join

4.1 Large Potential of Join Key Replication

As the first step toward achieving high skew resistance, we make a key finding that replicating join keys across the IDPs is a promising solution for mitigating the severe inter-IDP load imbalance. Unlike the IDP-wise global partitioning which partitions each join key to only one IDP, replicating join keys across the IDPs can significantly relax the burden of the heaviest-load IDP by allowing a popular join key to be processed by multiple IDPs. Once a popular join key gets replicated across multiple IDPs, we can evenly distribute the tuples of a skewed input table to the IDPs and thus accelerate the join execution for the popular key. The high internal memory bandwidth of a DIMM makes join key replication even more promising for PID-enabled DIMMs. PID can leverage the abundant aggregate internal memory bandwidth of all the IDPs attached to a PID-enabled system, unlike many hardware platforms which limit the potential of join key replication by providing limited memory bandwidth (e.g., host CPUs with only 10s of GB/s of host memory bandwidth). For example, replicating join keys across all the 1,024 IDPs of eight UPMEM DIMMs allows each join key to be processed with 716.8 GB/s of internal memory bandwidth. Such abundant internal memory bandwidth can greatly benefit join key replication which essentially provides higher access bandwidth to each join key for mitigating skewed input tables.

To quantify the potential of join key replication, we prototype a broadcast PID join algorithm by extending PID-Join to broadcast its join keys to all the IDPs. Instead of performing the IDP-wise global partitioning, the algorithm performs $R \bowtie S$ where $|R| \leq |S|$ and R consists of unique join

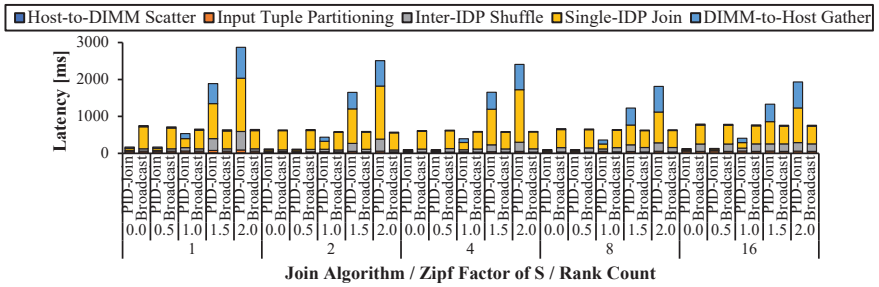


Fig. 8. Potential of replicating join keys across the IDPs

keys by replicating R across all the IDPs and evenly distributing the tuples of S to all the IDPs (Fig. 7). Fig. 8 compares the join execution latencies of PID-Join and the broadcast PID join algorithm on our PID-enabled system when $|R|$ = 0.5 M and $|R| : |S|$ = 1:8. The results show that, with highly skewed Ss whose Zipf factors reach 2.0, replicating join keys significantly across all the IDPs reduces the join execution latencies. The burden of processing popular join keys gets evenly distributed to all the IDPs, balancing the loads between the IDPs and reducing the execution latencies of the inter-IDP shuffle, the single-IDP join, and the DIMM-to-host gather steps. PID-Join, however, suffers from inter-IDP load imbalances and thus incurs higher join execution latencies.

At the same time, however, the results show that a trade-off exists between the join execution latency and the join key replication ratio. The results demonstrate that replicating R to all the IDPs does not guarantee the fastest join executions. With low Zipf factors of S ($\leq$ 1.0), PID-Join outperforms the broadcast PID join algorithm as replicating R to all the IDPs imposes larger computational overhead to each IDP than the IDP-wise global partitioning. As exposed through larger inter-IDP shuffle and single-IDP join latencies, replicating R to all the IDPs increases both the amount of inter-IDP data transfer and the number of R tuples each IDP needs to perform single-IDP join on. The inter-IDP data transfer increases as each tuple of R needs to be broadcasted to all the IDPs, and each IDP needs to construct a hash table with all the tuples of R rather than a partition of R. The trade-off, therefore, necessitates a new PID join algorithm not only for achieving high skew resistance and fast join executions but also for exploring how large the performance benefits of various join key replication ratios are on the join execution latencies with respect to the skew of input tables.

4.2 Overview

We propose *SPID-Join*, a skew-resistant PID join algorithm which replicates join keys across multiple IDPs by exploiting the bank- and the rank-level parallelisms of DIMMs to achieve high skew resistance and fast join executions. SPID-Join replicates join keys across multiple IDPs to achieve high skew resistance and fast join executions with skewed input tables. Motivated by the large potential of join key replication and the trade-off between the join execution latency and the join key replication ratio, SPID-Join aims to minimize the execution latency of $R \bowtie S$ for two input tables, a uniform R and a skewed S where $|R| \leq |S|$, and join key replication ratio. Taking the replication ratio as input allows us to further explore the trade-off using SPID-Join; we explore and discuss the trade-off in detail later in Section 4.4.

SPID-Join supports a wide range of join key replication ratios by exploiting the bank- and the rank-level parallelisms of DIMMs. SPID-Join groups the IDPs of PID-enabled DIMMs into *rank set*s followed by *bank set*s, and then replicates R across the sets and evenly distributes the tuples of S to the sets. One or more IDPs within a rank form a bank set, and a rank set comprises one or more ranks. After that, for each of the sets, SPID-Join partitions the set's R and S tuples to the IDPs

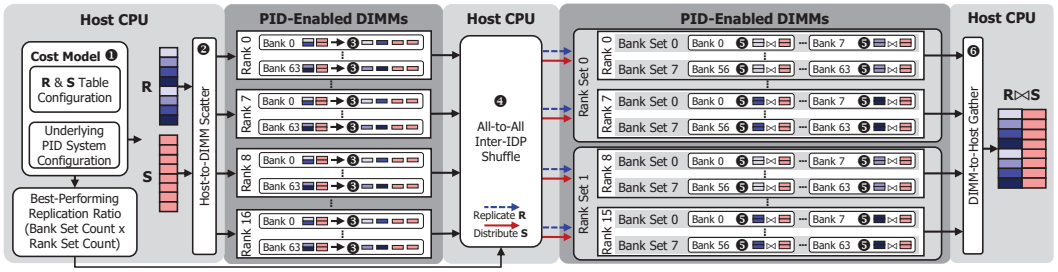


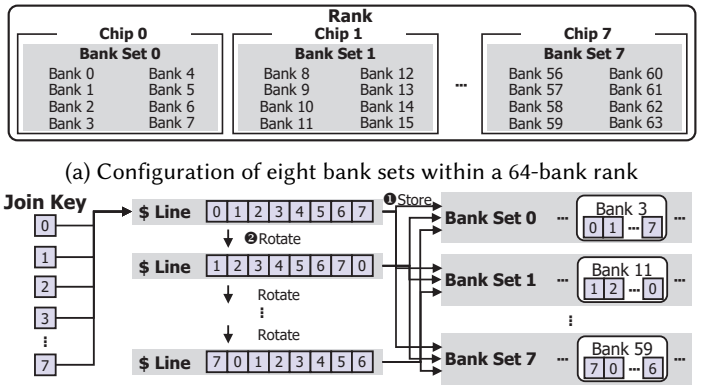
Fig. 9. End-to-end working model of SPID-Join

Algorithm 1 SPID-Join

1: $\#IDPs_{rank}$: The number of IDPs in a single rank (i.e.,64)
2: $\#banks_{BL}$: The number of banks serving a burst length (i.e., 8)
3: $\#ranks_{RS}$: The number of ranks in a single rank set
4: $\#banks_{BS}$: The number of banks in a single bank set
5: C_{data}, C_{system}: Configurations of the input data and the underlying PID-enabled system (e.g., skewness, #rank)
6: $RankwisePartition_R$, $RankwisePartition_S$: Partitions of R and S created after rank set-wise partitioning
7: $BankwisePartition_R$, $BankwisePartition_S$: Partitions of R and S created after bank set-wise partitioning
8:
9: **procedure** SPID-Join
10: $RR_{optimal} \leftarrow$ CostModel$(R, S, C_{data}, C_{system})$ ▷ ❶
11: **if** $RR_{optimal} < \#banks_{BL}$ **then**
12: $\#rankSet = RR_{optimal}$
13: **else if** $RR_{optimal} \leq \#IDPs_{rank}$ **then**
14: $\#bankSet = RR_{optimal}$
15: **else**
16: $\#bankSet = \#IDPs_{rank}$
17: $\#rankSet = RR_{optimal}/\#IDPs_{rank}$
18: **end if**
19: Host-to-DIMM Scatter(R, S) ▷ ❷
20: **for** *tuple in* R **do** ▷ ❸
21: $RankwisePartition_R \leftarrow$ RadixPartition$(tuple, \#rankSet)$
22: **for** *tuple in* $RankwisePartition_R$ **do**
23: $BankwisePartition_R \leftarrow$ RadixPartition$(tuple, \#bankSet)$
24: **end for**
25: **end for**
26: **for** *tuple in* S **do**
27: $RankwisePartition_S \leftarrow$ RadixPartition$(tuple, \#rankSet \times \#ranks_{RS})$
28: **for** *tuple in* $RankwiseParition_S$ **do**
29: $BankwiseParittion_S \leftarrow$ RadixPartition$(tuple, \#bankSet \times \#banks_{BS})$
30: **end for**
31: **end for**
32: ALLtoALL Inter-IDP Shuffle(R, S) ▷ ❹
33: SingleIDP-Join$(BankwisePartition_R, BankwisePartition_S)$ ▷ ❺
34: DIMM-to-Host Gather(R, S) ▷ ❻
35: **end procedure**

of the set, shuffles the tuples between the IDPs, and makes the IDPs perform single-IDP join on their partitions of the tuples. Replicating R across the sets increases the IDP count (and thus the internal memory bandwidth and computational throughput) allocated to each tuple of R. Evenly distributing the tuples of S to the sets reduces the inter-IDP load imbalance caused by the skew of S. As each set receives one replica of R, the total number of the sets (i.e., the bank set count ×the rank set count) becomes the join key replication ratio of SPID-Join. This allows SPID-Join to support various join key replication ratios by configuring the bank and rank counts in a way that the total number of the sets matches a given replication ratio. For example, with eight UPMEM DIMMs which collectively provide 16 ranks and 64 banks per rank, SPID-Join supports join replication ratios from one to 1,024 (up to 16 rank sets and up to 64 bank sets per rank).

SPID-Join extends PID-Join [44], the state-of-the-art PID join algorithm using the IDP-wise global partitioning, to implement the bank and rank set-based join key replication. SPID-Join replaces the



(a) Configuration of eight bank sets within a 64-bank rank

(b) Replicating join keys to the eight bank sets

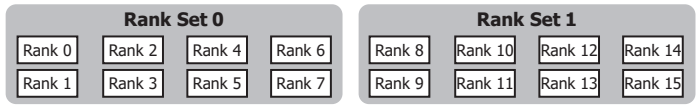
Fig. 10. SPID-Join's bank set-based join key replication

IDP-wise global partitioning with bank and rank set-aware partitioning; the other steps of PID-Join remain as-is. Fig. 9 and Algorithm 1 detail the end-to-end working model of SPID-Join. Given two input tables, a uniform R and a skewed S, ❶ SPID-Join first identifies the best-performing join key replication ratio using its cost model (Section 4.4). Rank set and bank set counts are determined based on the selected replication ratio (Section 4.3.3). Then, ❷ SPID-Join performs host-to-DIMM scatter which evenly distributes the tuples of R and S to all the IDPs. ❸ Each IDP then performs the bank and rank set-aware partitioning of R and S tuples, determining their destination IDPs based on the bank set and the rank set counts. S partitions are evenly distributed across ranks and banks within the rank set and bank set, whereas R partitions are identically replicated across all sets. After that, ❹ SPID-Join performs all-to-all inter-IDP shuffle for transferring the R and S tuples from their source IDPs to their destination IDPs. ❺ The IDPs then perform single-IDP join (e.g., hash, sort-merge) on their R and S partitions. Lastly, ❻ the host CPUs gather the join results from the IDPs using DIMM-to-host gather.

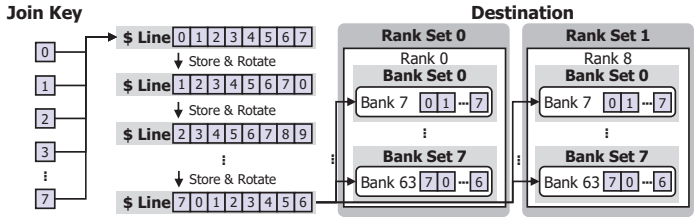
4.3 Bank- and Rank-wise Join Key Replication

4.3.1 Maximizing the Join Key Replication Bandwidth. To minimize the overhead of replicating join keys across the IDPs of a bank set, SPID-Join leverages the burst length of a DIMM to achieve high join key replication bandwidth. Rather than arbitrarily grouping the IDPs into bank sets, SPID-Join uses as the bank set granularity a set of the IDPs to which the sequential bytes of a burst-length-sized piece of data get byte-interleaved. As discussed in Section 2.1, the burst length serves as the granularity of accessing the data stored in a DIMM, and a sequential burst-length-sized piece of data gets interleaved across multiple memory banks. For example, on a DDR4 UPMEM DIMM with two 64-bank ranks, the burst length is 64 bytes and a sequential piece of 64-byte data gets byte-interleaved across eight memory banks of a rank. Therefore, for the UPMEM DIMM, SPID-Join uses the eight memory banks (and thus the eight IDPs) which collectively serve a single burst-length-sized data access as the granularity of its bank sets; the allowed bank set counts are thus 64, 32, 16, 8, and 1 each having 1, 2, 4, 8, and 64 IDPs, respectively.

Aligning the bank set granularity with the burst length allows us to rapidly replicate join keys across the IDPs of bank sets using only a few vector instructions and a single cache line of the host CPUs. Assuming that the 64 banks are grouped into eight bank sets and each join key is eight-byte large (Fig. 10a), Fig. 10b depicts how SPID-Join replicates join keys across the bank sets. SPID-Join first makes the host CPUs issue a single burst-length memory request to load eight join keys from



(a) Configuration of the two rank sets within 16 ranks



(b) Replicating join keys to the rank sets

Fig. 11. SPID-Join's rank set-based join key replication

eight memory banks to a 64-byte vector register of the host CPUs. Then, SPID-Join performs eight iterations of join key replication and vector register rotations, replicating the join keys to all eight bank sets. In each iteration, the eight join keys stored in the vector register get scattered to the eight memory banks, each belonging to different bank sets, using a burst-length memory request. As each bank set should retrieve all the eight join keys according to the byte-interleaving, SPID-Join rotates the vector register by eight bytes (i.e., the size of a tuple containing 4-byte join key and 4-byte tuple index) and then moves on to the next iteration. In the next iteration, SPID-Join again uses a burst-length memory request to scatter the join keys; each bank set retrieves the join key which appears next in the set of the eight join keys stored in the vector register. Performing all the iterations places a replica of all the eight join keys to each bank set, completing the replication of the eight join keys to the eight bank sets. This allows SPID-Join to complete the join key replication using only nine burst-length memory requests and seven vector register rotations. SPID-Join can replicate larger numbers of join keys to more bank sets by increasing the iteration count and adjusting the target banks.

When replicating join keys across rank sets, SPID-Join exploits the rank-level parallelism which allows ranks to operate independently and concurrently serve independent memory requests. The rank-level parallelism introduces another opportunity for maximizing join key replication bandwidth; concurrently accessing the ranks attached to different memory channels can exploit the high memory bandwidth collectively provided by the memory channels. To leverage the high multi-channel memory bandwidth, after loading a set of join keys to a vector register of the host CPUs, SPID-Join performs the replication of join keys to different rank sets in parallel using multiple host CPU threads. Fig. 11 depicts how SPID-Join exploits the rank-level parallelism to accelerate the replication of join keys to rank sets with an example rank set configuration of two rank sets and eight ranks per rank set. Similar to replicating join keys across the bank sets, SPID-Join first loads join keys to a vector register. Then, SPID-Join scatters the join keys to two rank sets in parallel; one host CPU thread scatters the join keys to the bank sets of one rank set, and another host CPU thread does the same to the bank sets of the other rank set. After that, the host CPUs execute multiple iterations of rotating the vector register by one join key and concurrently scattering the join keys to the bank sets of the two rank sets. In this way, SPID-Join can increase the join key replication bandwidth by up to a factor of the rank set count; when the rank set count is set to the number of PID-enabled ranks, SPID-Join can fully exploit the high aggregate memory bandwidth provided by all the memory channels of a PID-enabled system.

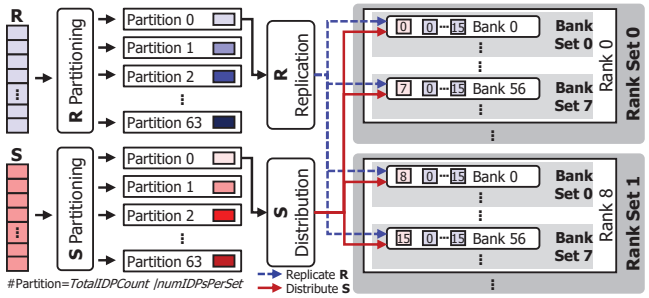


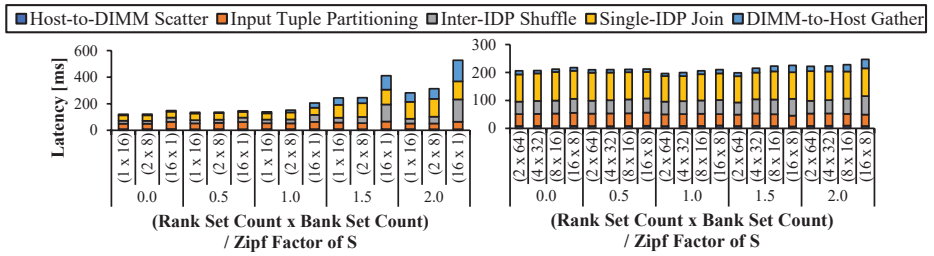
Fig. 12. Illustration of SPID-Join's bank and rank set-aware partitioning on an IDP and its input tuples

4.3.2 Bank and Rank Set-aware Partitioning. Even with the join key replication, the tuples of R and S still need to be partitioned to the IDPs as the key idea of SPID-Join is to place one replica of R to each bank set of a rank set. For this purpose, SPID-Join performs bank and rank set-aware partitioning to replicate R across the sets and evenly distribute S to the IDPs with respect to the bank and rank set counts. The bank and rank set-aware partitioning replaces PID-Join's IDP-wise global partitioning and gets performed by each IDP on its R and S tuples after the host-to-DIMM scatter step evenly scatter the tuples to all the IDPs. Once all the IDPs complete the bank and rank set-aware partitioning on their R and S tuples, the tuples get transferred to their destination IDPs in the next step of a join execution (i.e., the all-to-all inter-IDP shuffle).

Fig. 12 illustrates how an IDP performs the bank and rank set-aware partitioning on its R and S tuples with a bank set count of eight and a rank set count of two. The IDP first partitions its R and S into *numIDPsPerSet* partitions by radix partitioning. *numIDPsPerSet* is set as the number of all the IDPs available on a PID-enabled system divided by the bank and the rank set counts. For example, with our 1024-IDP PID-enabled system, *numIDPsPerSet* becomes 64 ($= 1024/(2 \times 8)$) as $rankSetCount = 2$ and $bankSetCount = 8$. The IDP then performs R replication and S distribution. The R replication replicates the tuples of each R partition to the corresponding ($bankSetCount \times rankSetCount$) IDP-wise partitions, and the S distribution distributes the tuples of each S partition to the IDP-wise partitions. The IDPs associated with the IDP-wise partitions belong to different bank and rank sets. The bank and rank set-aware partitioning terminates once the IDP processes all the *numIDPsPerSets* partitions of R and S. After that, the next step of a join execution (i.e., the inter-IDP shuffle) transfers the tuples stored in the IDP-wise partitions to their corresponding (destination) IDPs.

4.3.3 Prioritizing Bank Sets over Rank Sets. To maximize the join key replication bandwidth, SPID-Join leverages the burst length for bank sets and the rank-level parallelism for rank sets. However, as the two orthogonal dimensions for replicating join keys across the IDPs exist, SPID-Join can implement a given join key replication ratio with multiple bank and rank set configurations. For example, when a given join key replication ratio is 16 and the underlying PID-enabled system consists of 16 ranks with 64 memory banks per rank, the set of all the possible ratios between the rank set count and the bank set count becomes $\{16:1, 2:8, 1:16\}$. Being able to implement a join key replication ratio with multiple bank set and rank set configurations demonstrates SPID-Join's high flexibility. At the same time, however, the high flexibility raises a question of which bank set and rank set configuration SPID-Join should select for a given join key replication ratio.

As achieving fast join executions is a key design goal of SPID-Join, we conduct an experimental study to analyze the impact of the bank and rank set counts on the join execution latencies. The study reveals that for SPID-Join to achieve fast join executions for a given join key replication ratio, SPID-Join should prioritize bank sets over rank sets. Fig. 13 shows how the join execution latencies



(a) Join key replication ratio = 16 (b) Join key replication ratio = 128

Fig. 13. Join execution latencies of SPID-Join with varying ratios of bank and rank set counts ($|R|$ = 0.5 M, $|R|$: $|S|$ = 1:8)

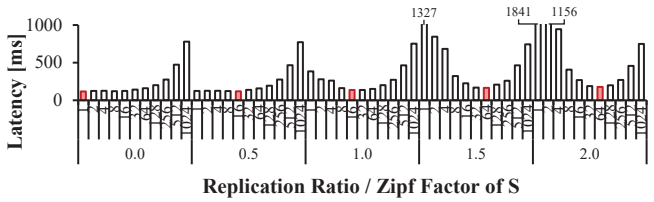


Fig. 14. Sensitivity of SPID-Join to the join key replication ratio and the Zipf factor of S ($|R|$ = 0.5 M, $|R|$: $|S|$ = 1:8). The red bars denote the best-performing replication ratios.

of SPID-Join on the PID-enabled system change with respect to the possible rank set and bank set configurations for two join key replication ratios: 16 and 128. 16 has been chosen to show the cases where either the bank or rank set counts are set to 1, and 128 demands both the bank and the rank set counts to be greater than one on the 16-rank PID-enabled system. The results show that, for both of the replication ratios, increasing the bank set count provides larger reductions in the join execution latencies than increasing the rank set count. In particular, when we increase the rank set count, the large increases in the inter-IDP shuffle and DIMM-to-host gather latencies offset the reductions in the single-IDP join latencies. This tendency shows that mitigating the inter-IDP load imbalance within a rank is more effective than addressing the less severe inter-rank load imbalance between the ranks. Motivated by the larger benefits of prioritizing the bank sets over the rank sets, SPID-Join implements a given join replication ratio by increasing the bank set count first, followed by the rank set count.

4.4 Cost-driven Replication Ratio Selection

Using its bank and rank set-based join key replications, SPID-Join can implement various replication ratios to mitigate the inter-IDP load imbalance. However, due to a trade-off between the join execution latency and the join key replication ratio, SPID-Join demands a cost model to decide the best-performing replication ratio for given join and PID system configurations. The trade-off is due to the fact that replicating join keys across multiple IDPs incurs larger numbers of R tuples being processed by each IDP, potentially offsetting the benefits of allocating higher computational throughput and internal memory bandwidth to each tuple of R. The best-performing replication ratio changes depending on various features of a join execution, including $|R|$, $|R|$: $|S|$, the Zipf factor of S, and the number of IDPs available on the underlying PID-enabled system. Fig. 14 demonstrates the trade-off by showing that the best-performing replication ratios differ with varying Zipf factors. SPID-Join achieves the fastest join execution using a replication ratio of 8 for a Zipf factor of 0.5, whereas the best-performing replication ratio becomes 64 when the Zipf factor increases to 2.0.

Table 2. Notations for SPID-Join's cost model

Variable	Description
TP_{SP}	Throughput of bank and rank set-aware partitioning of an IDP [tuples/sec]
TP_{LP}	Throughput of local partitioning of an IDP [tuples/sec]
TP_{Build}	Throughput of hash table build of an IDP [tuples/sec]
TP_{Probe}	Throughput of hash table probe of an IDP [tuples/sec]
BW_{HtoD}	Bandwidth of host-to-DIMM scatter [tuples/sec]
$BW_{Shuffle}$	Bandwidth of inter-IDP shuffle [tuples/sec]
BW_{DtoH}	Bandwidth of DIMM-to-host gather [tuples/sec]

Motivated by the trade-off, we develop a cost model for SPID-Join, whose objective is to find the best-performing replication ratio for given join and PID system configurations. The cost model builds on top of the variables shown in Table 2. Let $\mathbb{P}$ be a set of the replication ratios which SPID-Join can support according to the underlying PID system configuration (i.e., rank count, per-rank bank count). We define $Latency_{SPID-Join}$ as a cost evaluation function which predicts the join execution latency of SPID-Join and $Capacity_{SPID-Join}$ as a function which calculates the memory capacity SPID-Join needs. Then, we seek to find the element of $\mathbb{P}$ such that the cost $Latency_{SPID-Join}(rr)$ becomes minimum and $Capacity_{SPID-Join}(rr)$ does not exceed $Capacity_{MemoryBank}$, a memory bank size, where $rr \in \mathbb{P}$. The model then outputs the identified rr as the best-performing replication ratio for SPID-Join.

$\mathbb{P}$ can be defined using the combinations of the possible rank and bank set counts. Let $\mathbb{R}$ and $\mathbb{B}$ be the sets of available rank and bank set counts, respectively. As the cost model considers the power-of-two rank set counts and SPID-Join groups the banks within each rank exploiting the burst length (*numBanksPerBL*; 8 banks serve a burst on UPMEM DIMM), $\mathbb{R}$ and $\mathbb{B}$ are defined as follows:

$$\mathbb{R} = \{x | x = 2^k, 0 \leq k \leq \log_2 \#Rank\}$$
$$\mathbb{B} = \{1, 8(= 1 * numBanksPerBL), 16, 32, 64\}$$

Then, $\mathbb{P}$ is defined as a combination of $\mathbb{R}$ and $\mathbb{B}$:

$$\mathbb{P} = \{x | x = ab, a \in \mathbb{R}, b \in \mathbb{B},\ \ Capacity_{SPID-Join}(x) < Capacity_{MemoryBank}\}$$

To find the lowest join execution latency, we model SPID-Join's join execution latency (i.e., $Latency_{SPID-Join}$) as the sum of the latencies of the five join execution steps. The five steps are further grouped into two: PID execution and inter-IDP communication. If SPID-Join uses hash join as the single-IDP join algorithm, we get:

$$Latency_{SPID-Join} = Latency_{PID} + Latency_{Comm}$$
$$Latency_{PID} = Latency_{SP} + Latency_{LP} + Latency_{Build} + Latency_{Probe}$$
$$Latency_{Comm} = Latency_{HtoD} + Latency_{Shuffle} + Latency_{DtoH}$$

where *SP* refers to the bank and rank set-aware partitioning, *LP*, *Build*, and *Probe* correspond to the three internal steps of a single-IDP hash join (i.e., local partitioning, hash table build, hash table probe), *HtoD* is the host-to-DIMM scatter, $Shuffle$ is the inter-IDP shuffle, and *DtoH* denote the DIMM-to-host gather.

To model the latencies of the five steps and the required memory capacity within a single bank (i.e., $Capacity_{SPID-Join}$), the cost model needs to calculate the total and the per-IDP R and S tuple counts being involved in each of the steps. As SPID-Join's join key replication increases the R tuple count by replicating the R tuples, the per-IDP and the total R tuple counts become:

$$R_{IDP} = |R| \times rr \ / \ \#IDPs, \quad R_{Total} = R_{IDP} \times \#IDPs$$

where $\#IDPs$ denotes the total number of IDPs available on the underlying PID-enabled system. The tuples of S, on the other hand, get distributed to the bank and rank sets. Rather than precisely calculating the total and the per-IDP sizes of the S tuples, we model the sizes by leveraging the

fact that the heaviest-load IDP, and thus the IDP having the largest number of S tuples dominates the per-step execution latencies; all the IDPs must wait for the heaviest-load IDP to complete its execution in each step of a join execution. Accordingly, the per-IDP and the total S tuples counts become:

$$S_{IDP} = Heaviest\text{-}Load \;/\; rr, \quad S_{Total} = S_{IDP} \times \#IDPs$$

where *Heaviest-Load* is the S tuple count of the heaviest-load IDP which gets assigned the largest number of S tuples by SPID-Join according to the bank and rank set configuration. Then, to determine the memory required by SPID-Join, we model $Capacity_{SPID-Join}$ as the sum of R, S tuples of the heaviest-load IDP, and α, the size of the intermediate data. For hash join, α is the size of the hash table, twice the size of R, setting a fill rate of 50%. For sort-merge join, *alpha* is equivalent to the sizes of R and S to store the sorted data:

$$Capacity_{SPID-Join} = R_{IDP} + S_{IDP} + \alpha$$

For modeling the heaviest-load IDP's S tuple count, we make a key insight that SPID-Join's distribution of S tuples distributes the loads of not only the most popular join key, but also all the other join keys. This makes the most popular join key of S still remain the most popular even after distributing the S tuples to the IDPs. Based on the key insight, we make an assumption that all the join keys, except the most popular one, have negligible impact on the load of the heaviest-load IDP regardless of the replication ratio. The cost model thus calculates the heaviest-load IDP's S tuple count as:

$$Heaviest\text{-}Load = MostPopularJoinKeyCount + \frac{|S| - MostPopularJoinKeyCount}{\#IDPs}$$

where $MostPopularJoinKeyCount$ denotes the number of the S tuples which have the most popular join key.

$MostPopularJoinKeyCount$ can be determined in two ways: by constructing a global histogram and by using an analytical model with statistics of the input table [16, 53, 67, 69]. If the input tables are new to the cost model without preliminary statistics, SPID-Join allows IDPs to build local histograms for S. The host CPUs then collect the local histograms to construct the global histogram and identify the most frequent join key. This incurs minimal overhead as it is performed during the bank and rank set-aware partitioning of S, enabling the optimal replication ratio to be determined before the distribution of S. In addition, if the statistics of S are available, $MostPopularJoinKeyCount$ can be analytically calculated. For example, if S follows a Zipfian distribution and has a Zipf factor of $Zipf$, $MostPopularJoinKeyCount$ can be calculated as:

$$MostPopularJoinKeyCount = |S| \times \frac{\frac{1}{1^{Zipf}}}{\sum_{k=1}^{|R|}(\frac{1}{k^{Zipf}})} = \frac{|S|}{\sum_{k=1}^{|R|}(\frac{1}{k^{Zipf}})}$$

As the Zipf factor characterizes the probability of the n-th popular value as $\frac{1}{n^{Zipf}} \;/\; \sum_{k=1}^{N}(\frac{1}{k^{Zipf}})$, multiplying $|S|$ with the probability of the most popular join key (i.e., $n = 1$) derives the number of the S tuples having the most popular join key.

Using the aforementioned equations, the cost model can now estimate all the per-step execution latencies of a join execution. As the first two steps, the host-to-DIMM scatter and the bank and rank set-aware partitioning, are not affected by SPID-Join's join key replication ratio, their latencies are modeled as:

$$Latency_{HtoD} = (|R| + |S|) \;/\; BW_{HtoD}$$
$$Latency_{SP} = (|R| + |S|) \;/\; TP_{SP}$$

All the remaining steps occur after the bank and rank set-aware partitioning, so SPID-Join's bank and rank set counts need to be taken into account. Based on the aforementioned equations, the

remaining steps of the join execution are modeled as:

$$Latency_{Shuffle} = (R_{Total} + S_{Total}) \;/\; BW_{Shuffle}$$
$$Latency_{LP} = (R_{IDP} + S_{IDP}) \;/\; TP_{LP}$$
$$Latency_{Build} = R_{IDP} \;/\; TP_{Build}$$
$$Latency_{Probe} = S_{IDP} \;/\; TP_{Probe}$$
$$Latency_{DtoH} = S_{Total} \;/\; BW_{DtoH}$$

The per-IDP throughputs and the host-to-DIMM, inter-IDP, and DIMM-to-host bandwidths depend on the underlying PID-enabled system configuration; we can easily obtain all the throughputs and the bandwidths through one-time profiling on the system.

5 Evaluation

5.1 Experimental Setup

We evaluate SPID-Join against PID-Join [44] and UPMEM-Join [61], the state-of-the-art PID join algorithms. We execute an equi-join query, `SELECT * FROM R, S WHERE R.key = S.key`, and measure their join execution latencies. As benchmarks, we use 80 synthetic join configurations derived from various studies on in-memory join [45, 55, 58, 62, 68]. We set up the benchmarks with four sizes of R consisting of unique join keys (0.5 M, 2 M, 8 M, and 32 M tuples) covering both cases in which R tuples fit in or exceed a single-bank capacity, four $|R| : |S|$ ratios (1:1, 1:2, 1:4, and 1:8), and five Zipf factors of S (0.0, 0.5, 1.0, 1.5, and 2.0); the Zipf factor is widely used for characterizing and generating skewed input tables [4, 57]. Each tuple of the R and S consists of a 32-bit integer join key and a 32-bit tuple index. We also evaluate SPID-Join with the TPC-H dataset (SF = 10), a representative benchmark for analytical workloads [65]. Our experimental results are evaluated on Ubuntu 18.04, and we compile PID join algorithms using g++-11 with the -O3 compiler optimization option. To compile PID kernels for UPMEM DIMMs, we utilize a compiler provided by the UPMEM SDK v2021.3 [66].

5.2 Fast Join Executions with Skewed Tables

We first compare the join execution latencies of SPID-Join against PID-Join. We also compare SPID-Join with SPID-Join (Oracle), which is SPID-Join configured to always use the best-performing replication ratio through offline profiling. Fig. 15 shows the join execution latencies of two different PID join algorithms. The results show that SPID-Join outperforms PID-Join across all the highly skewed scenarios (Zipf factors of 1.0, 1.5, 2.0) for all the sizes of R and $|R| : |S|$ ratios. For a Zipf factor of 2.0, SPID-Join achieves up to 10.38× faster latency over PID-Join. This indicates that SPID-Join effectively mitigates the inter-IDP load imbalance of PID-Join. In addition, we evaluate the latencies of UPMEM-Join; it can only perform join with $|R| : |S|$ = 1:1 and Zipf factor of 0.0 and shows higher latencies than PID-Join and SPID-Join.

The comparison between SPID-Join and SPID-Join (Oracle) shows that, as the Zipf factor increases from 0.0 to 2.0, SPID-Join achieves geometric mean speedups of 1.0×, 0.97×, 2.46×, 4.96×, and 5.60× respectively, over PID-Join. SPID-Join (Oracle) achieves geometric mean speedups of 1.0×, 1.02×, 2.48×, 4.97×, and 5.72× over PID-Join. SPID-Join shows minimal speedup differences, achieving a Mean Absolute Percentage Error (MAPE) of only 2.62% compared to SPID-Join (Oracle). The results show that SPID-Join's cost-driven replication ratio selection correctly selects the best-performing replication ratio for the high Zipf factors. For lower Zipf factors (i.e., 0.5), SPID-Join shows a slightly different geometric mean speedup compared to SPID-Join (Oracle). We further analyze this and the efficiency of the cost model in Section 5.5.

SPID-Join not only shows superior performance but also shows the highest capability than PID-Join for skewed input tables. For many cases of our benchmarks with high Zipf factors, PID-Join

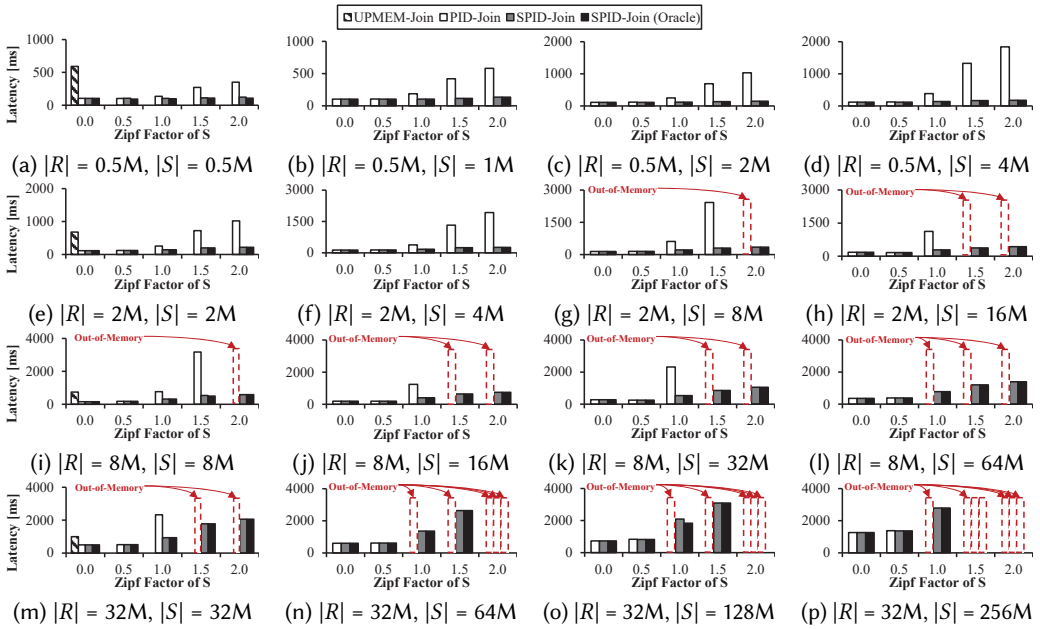


Fig. 15. Join execution latencies of PID-Join and SPID-Join with varying Zipf factors of S, tuple counts of R and $|R| : |S|$ ratios from 1:1 to 1:8. The red dotted bars denote the joins which fail to be executed due to the out-of-memory error caused by assigning too many input tuples, whose aggregate size exceeds the limited memory capacity of a 64-MB memory bank, to an IDP.

fails the join execution due to an out-of-memory error, unable to perform 22 over a total of 80 benchmarks. This is because the memory capacity of the heaviest-load IDP's bank is insufficient to execute the join for the given R and S tuples due to inter-IDP load imbalance by IDP-wise global partitioning. For 4 out of 80 cases, SPID-Join experiences out-of-memory issues with 32-M R tuples and S tuples with the highest Zipf factor we evaluated (i.e., 2.0) since the cost model finds no feasible replication ratio that allows both R and S to fit into the memory bank. However, SPID-Join can alleviate the out-of-memory issues by exploiting more PID-enabled DIMMs. Although we evaluate SPID-Join with 8 PID-enabled DIMMs within 4 memory channels, common server CPUs have 6 channels with 12 slots, and it can be further extended to 24 using NUMA CPUs. More DIMMs increase the range of possible replication ratios, enabling SPID-Join to find a suitable ratio that fits both R and S in memory.

5.3 Scalability Analysis

To evaluate the scalability of SPID-Join, we compare the join execution latencies of SPID-Join and PID-Join using 0.5 M tuples of R and $|R| : |S|$ ratio of 1:8 with varying the rank counts from 1 to 16 and Zipf factors from 0.0 to 2.0. Fig. 16 shows that SPID-Join achieves high scalability and outperforms PID-Join for every Zipf factor and every rank count. SPID-Join with 16 ranks achieves a geometric mean speedup of 7.49× over the latency of PID-Join with 1 rank for all Zipf factors. SPID-Join also shows 15.53× faster latency with 16 ranks and a Zipf factor of 2.0, compared to the latency of PID-Join with 1 rank. SPID-Join achieves high scalability by flexibly applying bank- and rank-wise set join key replication to mitigate the skewness in S for any rank counts. In contrast, PID-Join fails to achieve latency improvement with increased rank counts, except when the Zipf factors are relatively low (0.0 and 0.5). PID-Join achieves a speedup of 1.52× with 16 ranks and a

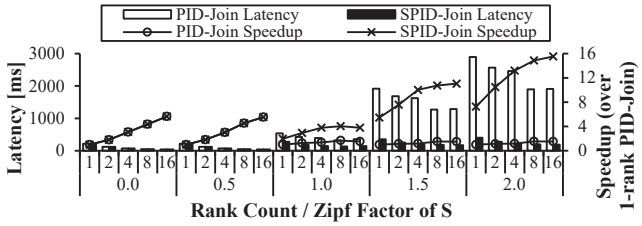
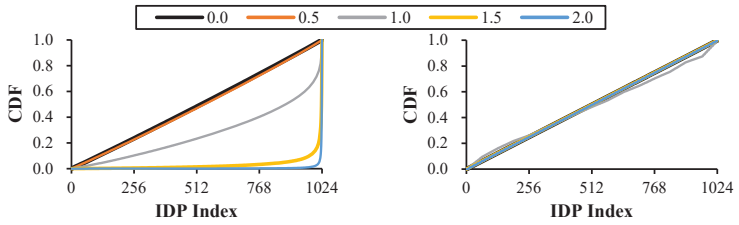
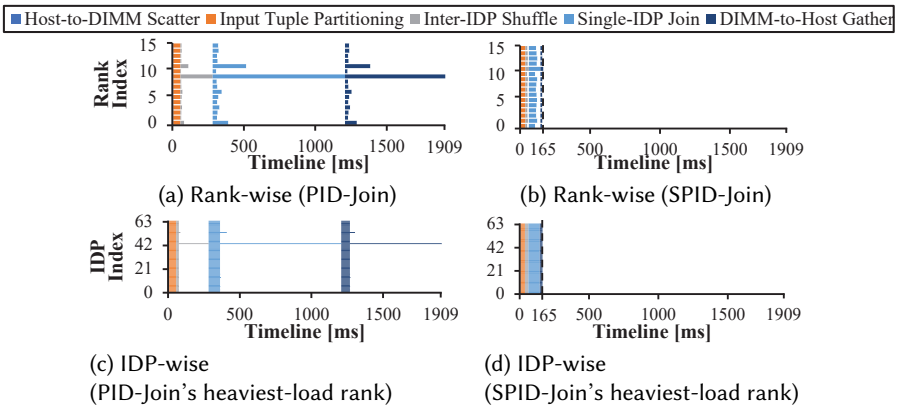


Fig. 16. Scalability of PID-Join and SPID-Join with varying rank counts ($|R|$ = 0.5 M, $|R|$: $|S|$ = 1:8)



(a) PID-Join (b) SPID-Join

Fig. 17. Per-IDP S tuple count distributions with varying Zipf factors of S ($|R|$ = 0.5 M, $|R|$: $|S|$ = 1:8)



(a) Rank-wise (PID-Join) (b) Rank-wise (SPID-Join)

(c) IDP-wise (PID-Join's heaviest-load rank) (d) IDP-wise (SPID-Join's heaviest-load rank)

Fig. 18. Per-rank and per-IDP join execution timelines of SPID-Join and PID-Join on the 16-rank PID-enabled system ($|R|$ = 0.5 M, $|R|$: $|S|$ = 1:8, S's Zipf factor = 2.0)

Zipf factor of 2.0 compared to the latency of PID-Join with 1 rank. This is due to the bottleneck caused by the heaviest-load IDP, which results in the severe underutilization of the other IDPs. As a result, SPID-Join with 16 ranks achieves a geometric mean speedup of 5.44× than PID-Join with 16 ranks with high Zipf factors (i.e., 1.0, 1.5, 2.0).

5.4 Mitigation of Inter-IDP Load Imbalance

To evaluate the efficacy of SPID-Join in addressing the inter-IDP load imbalance, we compare the distributions of S tuple counts assigned to each IDP by PID-Join and SPID-Join, using 0.5 M tuples of R and $|R|$: $|S|$ ratio of 1:8 with varying Zipf factors of S from 0.0 to 2.0. Fig. 17 illustrates the inter-IDP tuple distributions of PID-Join and SPID-Join, showing that SPID-Join successfully mitigates the inter-IDP load imbalance across Zipf factors. The standard deviations of the tuple distributions of PID-Join and SPID-Join are 78,974 and 6,223, respectively, with a Zipf factor of 2.0. SPID-Join's bank- and rank-wise join key replication effectively manages the inter-IDP load imbalance, while PID-Join's IDP-wise global partitioning exhibits the severe inter-IDP load imbalance.

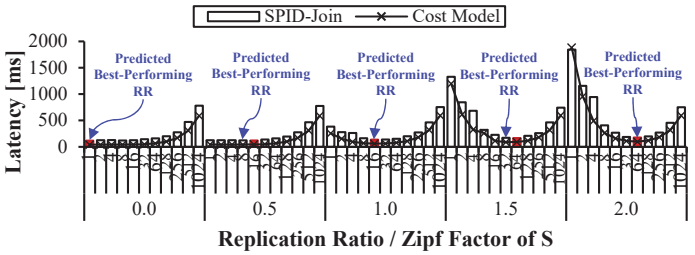


Fig. 19. Comparison of cost-driven predicted and measured latencies of SPID-Join ($|R|$ = 0.5 M, $|R| : |S|$ = 1:8)

To verify the effect of mitigation of inter-IDP load imbalance of SPID-Join, we measure the join execution latencies for each rank and IDP. The rank-wise join key replication alleviates rank underutilization by avoiding stalls waiting for the heaviest-load rank. The bank-wise join key replication ensures IDPs within a rank process similar amounts of tuples from S. Fig. 18 shows the timeline of SPID-Join with respect to ranks and IDPs of PID-Join and SPID-Join. The results show that the join execution latencies of ranks and IDPs reflect the inter-IDP load imbalance of S, showing the latency reduction from 1909 ms (PID-Join) to 165 ms (SPID-Join). For PID-Join, idle times account for 85.58% and 87.13% of the total execution time for ranks and IDPs respectively. SPID-Join has much lower idle times of 24.39% and 0.46% for ranks and IDPs repectively.

5.5 Validation of Cost-driven Replication Ratio Selection

To validate SPID-Join's cost-driven replication ratio selection, we evaluate the accuracy of the cost model in selecting the best-performing replication ratio and thus predicting the join latency of SPID-Join using varying Zipf factors of S ranging from 0.0 to 2.0. Fig. 19 shows the measured latencies of SPID-Join and the predicted latencies by the cost model. The results show that the join latency with the cost-driven best-performing replication ratio is only 0.85% higher than the oracle latency; the cost model correctly selects the best-performing replication ratio for three out of five Zipf factors evaluated. Furthermore, across all the 60 benchmarks with different table sizes, $|R| : |S|$, and Zipf factors, the cost model shows only 2.42% higher latency compared to the latency using the best-performing replication ratio. The high accuracy of the cost model stems from its ability to capture the trade-off between the benefits of join key replication and the overheads of the R tuple replication.

While the cost model accurately captures the overall trend of join execution latency changes with varying join key replication ratios, we observe a tendency for the measured latencies to be slight higher than the predicted ones with low replication ratios (e.g., 4). Our analysis reveals that the tendency is due to the slight errors caused by the cost model's inter-IDP communication modeling. The cost model does not discriminate the inter-IDP communications within a rank and across multiple ranks; however, we observe that the intra-rank communication is slightly faster than the inter-rank communication in most of the 60 benchmarks. For example, with low replication ratios, most of the inter-IDP communication becomes inter-rank due to SPID-Join's constraint on its bank set count to be a multiple of the number of the IDPs serving a single burst-length memory request. The cost model, on the other hand, uses a single value for modeling both the intra- and the inter-rank communications, and thus incurs slight modeling errors for the low replication ratios involving only the inter-rank communication (which is typically slower than the intra-rank communication [44]). Nevertheless, the cost model accurately captures the overall trend and identifies the (near-)best-performing replication ratios incurring similar join execution latencies to those of the best-performing replication ratios in all the 60 benchmarks.

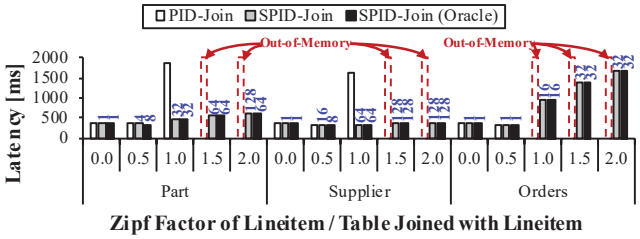


Fig. 20. Join execution latencies of PID-Join and SPID-Join using TPC-H dataset (SF = 10). The number above each bar denotes the replication ratio applied for the execution.

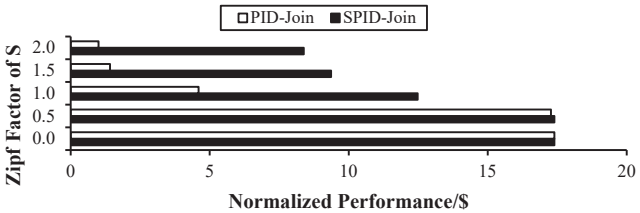


Fig. 21. Performance/$ of PID-Join and SPID-Join

5.6 Fast Join Executions with TPC-H Dataset

To evaluate the performance of SPID-Join on real-world datasets, we use the TPC-H dataset [65], a widely used real-world dataset [15, 23, 62]. We compare the join execution latencies for three tables (*Part*, *Supplier*, *Orders*) which have primary key-foreign key relationships with the *Lineitem* table. We set up the TPC-H dataset with a scale factor of 10; the *Lineitem* table has 60 M tuples, the *Part* table has 2 M tuples, the *Supplier* table has 0.1 M tuples, and the *Orders* table has 15 M tuples. We vary the Zipf factor from 0.0 to 2.0 for the *Lineitem* table to simulate different data skewness scenarios.

Fig. 20 shows that SPID-Join outperforms PID-Join significantly in real-world benchmarks. The speedups of SPID-Join and SPID-Join (Oracle) over PID-Join increase as the Zipf factor rises, both reaching up to 4.88×. Moreover, the latency difference between SPID-Join and SPID-Join (Oracle) with an MAPE is only 0.72%, highlighting that our cost-driven replication ratio selection can choose a replication ratio achieving minimal latency difference.

For *Orders* $\bowtie$ *Lineitem*, SPID-Join shows a slight increase in latencies with increasing Zipf factor even though the cost model selects the optimal replication ratio. The size of the *Orders* table (15 M tuples) restricts the range of possible replication ratios up to 64 in the evaluated PIM system configuration. This prevents SPID-Join from further replicating the table, leading to slightly less robust performance for highly skewed data. On the other hand, PID-Join fails its executions due to out-of-memory constraints when the Zipf factor is 1.5 or higher, regardless of the joined table size, due to the inter-IDP load imbalance. This diminishes PID-Join's utility for various real-world datasets with diverse Zipf factors.

5.7 System Cost

To compare the system costs of SPID-Join and PID-Join, we calculate the Manufacturer's Suggested Retail Price (MSRP) of our PID-enabled system. The MSRP of our PID-enabled system is a total of $4,020, which includes the cost of $1,500 for an Intel Xeon 5222 CPU, $120 for two standard DIMMs ($60 each), and $2,400 for eight UPMEM DIMMs ($300 each). Then, we compare the system costs using the geometric mean join processing throughput of SPID-Join and PID-Join from Zipf factor 0.0 to 2.0. Fig. 21 illustrates the normalized performance/$ of SPID-Join and PID-Join, normalized to

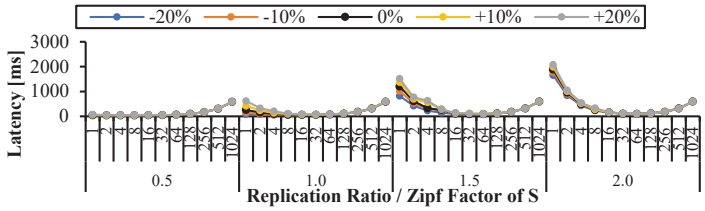
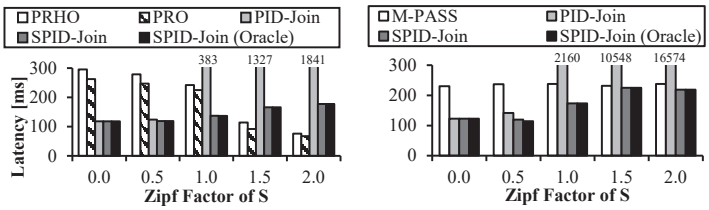


Fig. 22. Predicted join execution latencies of SPID-Join by applying errors to the Zipf factors ($|R|$ = 0.5 M, $|R|$: $|S|$ = 1:8)



(a) Hash Join (b) Sort-merge Join

Fig. 23. Comparison of execution latencies between CPU join algorithms and SPID-Join ($|R|$ = 0.5 M, $|R|$: $|S|$ = 1:8)

PID-Join with the Zipf factor of 2.0. As the Zipf factor increases, the performance/$ of SPID-Join is relatively maintained, whereas that of PID-Join significantly decreases. With the Zipf factor of 2.0, SPID-Join achieves 5,938 tuples/sec/$, which is 8.37× higher than that of PID-Join (709 tuples/sec/$). When the Zipf factor is 0.0, the system cost of SPID-Join is the same as that of PID-Join. The results show that SPID-Join makes better use of the PID-enabled system by effectively addressing severe inter-IDP load imbalances.

5.8 Impact of the Zipf Factor Mismatch

To study the impact of inaccurately estimated Zipf factors on the cost-driven replication ratio selection, we introduce -20% to +20% errors to the Zipf factors. For example, a -10% error rate adjusts a Zipf factor of 1.0 to 0.9. Fig. 22 shows that despite these errors, our cost model selects the optimal replication ratio in 13 out of 20 cases. The predicted latencies follow a consistent pattern that the latency increases at low replication ratios, decreases as the ratio increases, and increases again at higher ratios. Even when sub-optimal ratios are chosen, the execution latencies do not significantly differ from the optimal latencies, showing an MAPE of only about 0.51%. Sub-optimal ratios are selected when Zipf factors are low, where the performance of various replication ratios is similar.

5.9 Comparison with CPU Join Algorithms

Fig. 23 shows latency comparison of SPID-Join with PID-Join and CPU join algorithms (i.e., PRO [3] and PRHO [2] for hash join, and M-PASS [2] for sort-merge join) using 0.5 M tuples of R, $|R|$: $|S|$ ratio of 1:8 and varying the Zipf factors from 0.0 to 2.0. The results show that SPID-Join outperforms M-PASS across all Zipf factors for sort-merge join, and achieves lower latency than PRO and PRHO up to a Zipf factor of 1.0 whereas PID-Join shows much higher latencies compared to SPID-Join with high skews. This emphasizes the importance of skew mitigation in exploiting PID-enabled DIMMs. SPID-Join shows higher latencies with high Zipf factors (1.5, 2.0) than PRO and PRHO. CPU hash joins exhibit better performance by frequently accessing specific hash table entries due to high skewness (e.g., 60.8% of tuples have the same join key in Zipf factor 2.0), leading to high cache hits. These findings highlight the potential of CPU-PID heterogeneous join execution, a topic for future work.

6 Discussions

6.1 Complex Join Predicates

While the evaluated performance enhancement of SPID-Join may not be equal for complex joins involving variable-length tuples and intricate equality rules compared to straightforward 32-bit integer key cases, SPID-Join is not limited to simple predicates. SPID-Join remains effective for complex joins by extending its cost model with the appropriate single-IDP join implementation's throughput model. SPID-Join can effectively handle complex join predicates with different single-IDP joins (i.e., sort-merge and nested-loop) [44].

6.2 Applicability to Other PID-enabled DIMMs

In this paper, we propose and evaluate SPID-Join using UPMEM DIMMs. The key ideas of SPID-Join focus on exploiting bank- and rank-level parallelism of the DIMMs to mitigate the severe inter-IDP load imbalance, and this is not limited to a specific PID-enabled DIMM architecture. Samsung AxDIMM [41] is another commodity PID-enabled DIMM. As part of the DDR family, AxDIMM follows a shared-nothing architecture, with the processor positioned near ranks. Given the similarities between AxDIMM and UPMEM DIMMs, SPID-Join can be applied with minimal adjustments. The challenges of inter-IDP load imbalance persist due to the concentration of skewed data in a specific rank on AxDIMM.

6.3 Other Processing-in-memory Architectures

The key ideas of SPID-Join can be applied not only for PID-enabled DIMM but also for other Processing-In-Memory (PIM) devices. The load imbalance across the In-Memory Processors (IMPs) is due to the skewness of input tables, which are not specific to a particular architecture. For example, Samsung Aquabolt-XL [42] and SK Hynix AiM [38] are recent real-world PIM devices. Due to their common power and area budget constraints, PIM devices commonly employ shared-nothing architecture. Therefore, inter-IMP load imbalance of PIM devices can be mitigated by SPID-Join's key ideas.

6.4 Scalability of SPID-Join

Although our experiments were conducted with S sizes that fit within the aggregate memory bank capacity of our setup, the size of input tuples is not strictly limited by the memory capacity. Since S tuples do not impact each other in deriving the join result, SPID-Join can perform the join with larger S by dividing the S tuples into several partitions. Then, it can keep the R table to join on the memory banks and probe the partition of S tuples in a streaming manner. As long as the matching R and S tuples are loaded at the same time, R tuples to process the single-IDP join can be stored in the memory bank, enabling S to be performed sequentially as a pipelined join.

7 Related Work

7.1 Skew Mitigation for PID-enabled DIMMs

There exists a prior study presenting the skew mitigation method for PID-enabled DIMMs. Kang et al. [33] proposed a skip list-based ordered index structure, utilizing the host CPUs and PID for point queries, updates, and range scans. However, their approach is hard to apply to joins. Sending skewed tuples of the input table to the host CPUs and processing the rest in PID cannot fully utilize the high bandwidth of PID and incur additional data transfer of non-joined tuples to the host. In contrast, SPID-Join effectively manages high skewness for joins by mitigating inter-IDP load imbalance through the high bandwidth of PID-enabled DIMMs.

7.2 PID-accelerated Databases

As PID-enabled DIMMs provide much higher memory bandwidth than standard DIMMs, they greatly fit with accelerating database operations. Lim et al.[44] accelerate join algorithms and optimize host-to-DIMM and inter-IDP data communication. However, they do not handle the effect of high skewness which limits the performance and scalability of PID-enabled DIMM. In addition to the join operation, there have been efforts to improve various relational operations using PID-enabled DIMMs [6–8, 33, 41]. Bernhardt et al. [8] conduct performance analysis for scan, selection, aggregation, projection, and record materialization. Baumstark et al. [6] evaluate the effectiveness of the scans with varying table sizes. Lee et al. [41] accelerate a defined range and in-list scan operation. Kang et al. [33] propose an ordered index that provides high load balance and low communication under workload skew. Baumstark et al. [7] accelerate the scan-and-selection operation in the query pipeline.

7.3 Skew-resistant Join Algorithms

Prior approaches have been proposed to address high skewness in join executions. Balkeson et al. [2] divide skewed data into chunks with all threads processing them together. Kim et al. [35] partition the entire set of tuples for parallel execution, sharing information about R. Paul et al. [54] divide tuples from all partitions into smaller chunks, distributing them to each thread block in a round-robin manner. Rui et al. [59] enable block processing of skewed tuples by generating additional kernels for concurrent handling. Most prior works dynamically redistribute data in more fine-grained partitions among threads, but it cannot be applied to SPID-Join due to the data transfer overhead required for dynamic allocation.

7.4 Joins on Shared-nothing Environment

Various shared-nothing join algorithms [20, 30, 37, 63] address skews by balancing the workload across processing nodes. Zhou et al. [70] identify the heaviest-load processor and dynamically balance the workload across the nodes. Rödiger et al. [57] keep locally detected skewed tuples in each node, redistribute only the remaining tuples, and broadcast the required R tuples among the nodes. However, they cannot be directly applied to PID-enabled DIMMs since the system lacks a communication path between IDPs which enables fine-grained workload balancing. Xu et al. [67] and Cheng et al. [17] propose that each node identifies skewed tuples and broadcasts them to all nodes. However, it is not applicable to PID-enabled DIMMs due to their limited memory bank size.

8 Conclusion

We proposed *SPID-Join*, a skew-resistant PID join algorithm that mitigates inter-IDP load imbalance by leveraging bank- and rank-level parallelisms of DIMMs through bank- and rank-wise join key replication. First, SPID-Join groups ranks into rank sets and replicates join keys across them. Second, It also groups banks of each rank into bank sets, enabling efficient intra-rank replication by sharing burst lengths across bank sets. Third, SPID-Join incorporates a cost model to find the best-performing replication ratio based on predicted join latency for given join and system configurations. Our evaluation shows SPID-Join successfully mitigates inter-IDP load imbalance, achieving up to 10.38× speedup over PID-Join.

Acknowledgments

This work was supported by the National Research Foundation of Korea (NRF) grant (NRF-2022R1C1C1008131) and the Institute for Information & communications Technology Promotion (IITP) grants (RS-2020-II201361, RS-2022-II220050, RS-2024-00395134) funded by the Korea government (MSIP). Youngsok Kim is the corresponding author of this paper.

References

[1] Rajeev Balasubramonian, Andrew B Kahng, Naveen Muralimanohar, Ali Shafiee, and Vaishnav Srinivas. 2017. CACTI 7: New tools for interconnect exploration in innovative off-chip memories. *ACM Transactions on Architecture and Code Optimization (TACO)* (2017).

[2] Cagri Balkesen, Gustavo Alonso, Jens Teubner, and M Tamer Özsu. 2013. Multi-core, main-memory joins: Sort vs. hash revisited. *Proc. VLDB Endowment (PVLDB)* (2013).

[3] 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 *Proc. 29th IEEE International Conference on Data Engineering (ICDE)*.

[4] Maximilian Bandle, Jana Giceva, and Thomas Neumann. 2021. To partition, or not to partition, that is the join question in a real system. In *Proc. 2021 ACM SIGMOD International Conference on Management of Data (SIGMOD)*.

[5] Ronald Barber, Guy Lohman, Ippokratis Pandis, Vijayshankar Raman, Richard Sidle, Gopi Attaluri, Naresh Chainani, Sam Lightstone, and David Sharpe. 2014. Memory-efficient hash joins. *Proc. VLDB Endowment (PVLDB)* (2014).

[6] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Accelerating Large Table Scan using Processing-In-Memory Technology. *Proc. 20th Conference on Database Systems for Business, Technology and Web (BTW)* (2023).

[7] Alexander Baumstark, Muhammad Attahir Jibril, and Kai-Uwe Sattler. 2023. Processing-in-Memory for Databases: Query Processing and Data Transfer. In *Proc. 19th International Workshop on Data Management on New Hardware (DaMoN)*.

[8] Arthur Bernhardt, Andreas Koch, and Ilia Petrov. 2023. pimDB: From Main-Memory DBMS to Processing-In-Memory DBMS-Engines on Intelligent Memories. In *Proc. 19th International Workshop on Data Management on New Hardware (DaMoN)*.

[9] Spyros Blanas and Jignesh M Patel. 2013. Memory footprint matters: efficient equi-join algorithms for main memory data processing. In *Proc. 4th annual Symposium on Cloud Computing*.

[10] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. 2008. Breaking the Memory Wall in MonetDB. *Communications of the ACM (CACM)* (2008).

[11] Peter A Boncz, Stefan Manegold, Martin L Kersten, et al. 1999. Database architecture optimized for the new bottleneck: Memory access. In *VLDB*.

[12] Sebastian Breß. 2014. The design and implementation of CoGaDB: A column-oriented GPU-accelerated DBMS. *Datenbank-Spektrum* (2014).

[13] Shasank Chavan, Albert Hopeman, Sangho Lee, Dennis Lui, Ajit Mylavarapu, and Ekrem Soylemez. 2018. Accelerating joins and aggregations on the oracle in-memory database. In *Proc. 34th IEEE International Conference on Data Engineering (ICDE)*.

[14] Liang-Chi Chen, Chien-Chung Ho, and Yuan-Hao Chang. 2023. UpPipe: A Novel Pipeline Management on In-Memory Processors for RNA-seq Quantification. In *Proc. 60th ACM/IEEE Design Automation Conference (DAC)*.

[15] Yu Chen and Ke Yi. 2017. Two-level sampling for join size estimation. In *Proc. 2017 ACM International Conference on Management of Data (SIGMOD)*. 759–774.

[16] Long Cheng, Spyros Kotoulas, Tomas E Ward, and Georgios Theodoropoulos. 2014. Efficiently handling skew in outer joins on distributed systems. In *Proc. 14th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*.

[17] Long Cheng, Spyros Kotoulas, Tomas E Ward, and Georgios Theodoropoulos. 2014. Robust and skew-resistant parallel joins in shared-nothing systems. In *Proc. 23rd ACM International Conference on Conference on Information and Knowledge Management*.

[18] Prangon Das, Purab Ranjan Sutradhar, Mark Indovina, Sai Manoj Pudukotai Dinakarrao, and Amlan Ganguly. 2022. Implementation and Evaluation of Deep Neural Networks in Commercially Available Processing in Memory Hardware. In *Proc. 35th International System-on-Chip Conference (SOCC)*.

[19] Fabrice Devaux. 2019. The true processing in memory accelerator. In *Proc. IEEE Hot Chips 31 Symposium (HCS)*.

[20] David J. DeWitt, Jeffrey F. Naughton, Donovan A. Schneider, and S. Seshadri. 1992. Practical Skew Handling in Parallel Joins. In *Proc. 18th International Conference on Very Large Data Bases (VLDB)*.

[21] Safaa Diab, Amir Nassereldine, Mohammed Alser, Juan Gómez Luna, Onur Mutlu, and Izzat El Hajj. 2023. A framework for high-throughput sequence alignment using real processing-in-memory systems. *Bioinformatics* (2023).

[22] Sairo R dos Santos, Francis B Moreira, Tiago R Kepe, and Marco AZ Alves. 2022. Advancing Database System Operators with Near-Data Processing. In *Proc. 30th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*.

[23] Hao Gao and Nikolai Sakharnykh. 2021. Scaling Joins to a Thousand GPUs. In *ADMS@ VLDB*.

[24] Juan Gómez-Luna, Yuxin Guo, Sylvan Brocard, Julien Legriel, Remy Cimadomo, Geraldo F. Oliveira, Gagandeep Singh, and Onur Mutlu. 2022. Machine Learning Training on a Real Processing-in-Memory System. In *Proc. IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*.

[25] Juan Gómez-Luna, Yuxin Guo, Sylvan Brocard, Julien Legriel, Remy Cimadomo, Geraldo F. Oliveira, Gagandeep Singh, and Onur Mutlu. 2023. Evaluating Machine Learning Workloads on Memory-Centric Computing Systems. In *Proc. IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*.

[26] Juan Gómez-Luna, Izzat El Hajj, Ivan Fernandez, Christina Giannoula, Geraldo F. Oliveira, and Onur Mutlu. 2022. Benchmarking a New Paradigm: Experimental Analysis and Characterization of a Real Processing-in-Memory System. *IEEE Access* (2022).

[27] Harshita Gupta, Mayank Kabra, Juan Gómez-Luna, Konstantinos Kanellopoulos, and Onur Mutlu. 2023. Evaluating Homomorphic Operations on a Real-World Processing-In-Memory System. In *Proc. 2023 IEEE International Symposium on Workload Characterization (IISWC)*.

[28] Robert J Halstead, Ildar Absalyamov, Walid A Najjar, and Vassilis J Tsotras. 2015. FPGA-based Multithreading for In-Memory Hash Joins.. In *Proc. 7th bieennial Conference on Innovative Data Systems Research (CIDR)*.

[29] Wei He, Minqi Zhou, Xueqing Gong, and Xiaofeng He. 2013. Massive parallel join in NUMA architecture. In *Proc. 2013 IEEE International Congress on Big Data*.

[30] Kien A. Hua and Chiang Lee. 1991. Handling Data Skew in Multiprocessor Database Computers Using Partition Tuning. In *Proc. 17th International Conference on Very Large Data Bases (VLDB)*.

[31] Intel. 2019. Intel® Xeon® Gold 5222 Processor. https://www.intel.com/content/www/us/en/products/sku/192445/intel-xeon-gold-5222-processor-16-5m-cache-3-80-ghz/specifications.html

[32] JEDEC Solid State Technology Association. 2012. *DDR4 SDRAM STANDARD*.

[33] Hongbo Kang, Yiwei Zhao, Guy E. Blelloch, Laxman Dhulipala, Yan Gu, Charles McGuffey, and Phillip B. Gibbons. 2022. PIM-tree: A Skew-resistant Index for Processing-in-Memory. *Proc. VLDB Endowment (PVLDB)* (2022).

[34] Liu Ke, Xuan Zhang, Jinin So, Jong-Geon Lee, Shin-Haeng Kang, Sukhan Lee, Songyi Han, YeonGon Cho, Jin Hyun Kim, Yongsuk Kwon, KyungSoo Kim, Jin Jung, Ilkwon Yun, Sung Joo Park, Hyunsun Park, Joonho Song, Jeonghyeon Cho, Kyomin Sohn, Nam Sung Kim, and Hsien-Hsin S. Lee. 2022. Near-Memory Processing in Action: Accelerating Personalized Recommendation With AxDIMM. *IEEE Micro* (2022).

[35] Changkyu Kim, Tim Kaldewey, Victor W Lee, Eric Sedlar, Anthony D Nguyen, Nadathur Satish, Jatin Chhugani, Andrea Di Blas, and Pradeep Dubey. 2009. Sort vs. hash revisited: Fast join implementation on modern multi-core CPUs. *Proc. VLDB Endowment (PVLDB)* (2009).

[36] Jin Hyun Kim, Shin-Haeng Kang, Sukhan Lee, Hyeonsu Kim, Yuhwan Ro, Seungwon Lee, David Wang, Jihyun Choi, Jinin So, YeonGon Cho, JoonHo Song, Jeonghyeon Cho, Kyomin Sohn, and Nam Sung Kim. 2022. Aquabolt-XL HBM2-PIM, LPDDR5-PIM With In-Memory Processing, and AXDIMM With Acceleration Buffer. *IEEE Micro* (2022).

[37] Masaru Kitsuregawa and Yasushi Ogawa. 1990. Bucket Spreading Parallel Hash: A New, Robust, Parallel Hash Join Method for Data Skew in the Super Database Computer (SDC). In *Proc. 16th International Conference on Very Large Data Bases (VLDB)*.

[38] Yongkee Kwon, Kornijcuk Vladimir, Nahsung Kim, Woojae Shin, Jongsoon Won, Minkyu Lee, Hyunha Joo, Haerang Choi, Guhyun Kim, Byeongju An, et al. 2022. System architecture and software stack for GDDR6-AiM. In *Proc. IEEE Hot Chips 34 Symposium (HCS)*.

[39] Dominique Lavenier, Remy Cimadomo, and Romaric Jodin. 2020. Variant Calling Parallelization on Processor-in-Memory Architecture. In *Proc. IEEE International Conference on Bioinformatics and Biomedicine (BIBM)*.

[40] Dongjae Lee, Bongjoon Hyun, Taehun Kim, and Minsoo Rhu. 2024. Analysis of Data Transfer Bottlenecks in Commercial PIM Systems: A Study with UPMEM-PIM. *IEEE Computer Architecture Letters* (2024).

[41] Donghun Lee, Jinin So, MINSEON AHN, Jong-Geon Lee, Jungmin Kim, Jeonghyeon Cho, Rebholz Oliver, Vishnu Charan Thummala, Ravi shankar JV, Sachin Suresh Upadhya, Mohammed Ibrahim Khan, and Jin Hyun Kim. 2022. Improving In-Memory Database Operations with Acceleration DIMM (AxDIMM). In *Proc. 18th International Workshop on Data Management on New Hardware (DaMoN)*.

[42] Sukhan Lee, Shin-haeng Kang, Jaehoon Lee, Hyeonsu Kim, Eojin Lee, Seungwoo Seo, Hosang Yoon, Seungwon Lee, Kyounghwan Lim, Hyunsung Shin, Jinhyun Kim, Seongil O, Anand Iyer, David Wang, Kyomin Sohn, and Nam Sung Kim. 2021. Hardware Architecture and Software Stack for PIM Based on Commercial DRAM Technology. In *Proc. 48th IEEE/ACM International Symposium on Computer Architecture (ISCA)*.

[43] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2015. How good are query optimizers, really? *Proc. VLDB Endowment (PVLDB)* (2015).

[44] Chaemin Lim, Suhyun Lee, Jinwoo Choi, Jounghoo Lee, Seongyeon Park, Hanjun Kim, Jinho Lee, and Youngsok Kim. 2023. Design and Analysis of a Processing-in-DIMM Join Algorithm: A Case Study with UPMEM DIMMs. *Proc. ACM on Management of Data (PACMMOD)* (2023).

[45] Clemens Lutz, Sebastian Breß, Steffen Zeuch, Tilmann Rabl, and Volker Markl. 2022. Triton join: Efficiently scaling to a large join state on gpus with fast interconnects. In *Proc. 2022 International Conference on Management of Data (SIGMOD)*.

[46] Kamesh Madduri and Kesheng Wu. 2009. Efficient joins with compressed bitmap indexes. In *Proc. 18th ACM International Conference on Conference on Information and Knowledge Management.*
[47] Darko Makreshanski, Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. 2018. Many-query join: efficient shared execution of relational joins on modern hardware. *The VLDB Journal* (2018).
[48] S. Manegold, P. Boncz, and M. Kersten. 2002. Optimizing Main-Memory Join on Modern Hardware. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* (2002).
[49] Jun Miyazaki. 2005. Hardware supported memory access for high performance main memory databases. In *Proc. 1st international workshop on Data management on new hardware.*
[50] Jun Miyazaki. 2006. A Memory Subsystem with Comparator Arrays for Main Memory Database Operations. In *Proc. 2006 ACM Symposium on Applied Computing (SAC).*
[51] Raghunath Othayoth Nambiar and Meikel Poess. 2006. The Making of TPC-DS.. In *VLDB.*
[52] Joel Nider, Craig Mustard, Andrada Zoltan, John Ramsden, Larry Liu, Jacob Grossbard, Mohammad Dashti, Romaric Jodin, Alexandre Ghiti, Jordi Chauzi, and Alexandra Fedorova. 2021. A Case Study of Processing-in-Memory in off-the-Shelf Systems. In *Proc. 2021 USENIX Annual Technical Conference (USENIX ATC).*
[53] Oracle. 2024. MySQL 8.0 Reference Manual - The INFORMATION_SCHEMA STATISTICS Table. https://dev.mysql.com/doc/refman/8.0/en/information-schema-statistics-table.html.
[54] Johns Paul, Bingsheng He, Shengliang Lu, and Chiew Tong Lau. 2020. Revisiting hash join on graphics processors: A decade later. *Distributed and Parallel Databases* (2020).
[55] Johns Paul, Shengliang Lu, Bingsheng He, and Chiew Tong Lau. 2021. MG-Join: A Scalable Join for Massively Parallel Multi-GPU Architectures. In *Proc. International Conference on Management of Data (SIGMOD).*
[56] Danila Piatov, Sven Helmer, and Anton Dignös. 2016. An interval join optimized for modern hardware. In *Proc. 32nd IEEE International Conference on Data Engineering (ICDE).*
[57] Wolf Rödiger, Sam Idicula, Alfons Kemper, and Thomas Neumann. 2016. Flow-join: Adaptive skew handling for distributed joins over high-speed networks. In *Proc. 32nd IEEE International Conference on Data Engineering (ICDE).*
[58] Ran Rui, Hao Li, and Yi-Cheng Tu. 2015. Join algorithms on GPUs: A revisit after seven years. In *Proc. 2015 IEEE International Conference on Big Data (Big Data).*
[59] Ran Rui and Yi-Cheng Tu. 2017. Fast Equi-Join Algorithms on GPUs: Design and Implementation. In *Proc. 29th International Conference on Scientific and Statistical Database Management (SSDBM).*
[60] Ibrahim Sabek and Tim Kraska. 2023. The Case for Learned In-Memory Joins. *Proc. VLDB Endowment (PVLDB)* (2023).
[61] Roee Shlomo, Julien Legriel, Aphélie Moisson, and Sylvan Brocard. 2023. UPMEM-PIM Evaluation for SQL Query Acceleration. https://github.com/upmem/dpu_olap.
[62] Panagiotis Sioulas, Periklis Chrysogelos, Manos Karpathiotakis, Raja Appuswamy, and Anastasia Ailamaki. 2019. Hardware-Conscious Hash-Joins on GPUs. In *Proc. 35th IEEE International Conference on Data Engineering (ICDE).*
[63] Michael Stonebraker. 1986. The case for shared nothing. *IEEE Database Eng. Bull.* (1986).
[64] Diego G Tomé, Tiago Rodrigo Kepe, Marco AZ Alves, and Eduardo C de Almeida. 2018. Near-Data Filters: Taking Another Brick from the Memory Wall.. In *ADMS@ VLDB.*
[65] Transaction Processing Performance Council (TPC). 2022. TPC Benchmark H. https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-H_v3.0.1.pdf.
[66] UPMEM SAS. 2021. UPMEM SDK. https://sdk.upmem.com/2021.3.0/index.html
[67] Yu Xu, Pekka Kostamaa, Xin Zhou, and Liang Chen. 2008. Handling data skew in parallel joins in shared-nothing systems. In *Proc. 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD).*
[68] Makoto Yabuta, Anh Nguyen, Shinpei Kato, Masato Edahiro, and Hideyuki Kawashima. 2017. Relational Joins on GPUs: A Closer Look. *IEEE Transactions on Parallel and Distributed Systems (TPDS)* 28 (2017).
[69] Wangda Zhang and Kenneth A Ross. 2020. Exploiting data skew for improved query performance. *IEEE Transactions on Knowledge and Data Engineering (TKDE)* (2020).
[70] Shunjie Zhou, Fan Zhang, Hanhua Chen, Hai Jin, and Bing Bing Zhou. 2019. FastJoin: A Skewness-Aware Distributed Stream Join System. In *Proc. 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS).*
[71] Zichen Zhu, Xiao Hu, and Manos Athanassoulis. 2023. NOCAP: Near-Optimal Correlation-Aware Partitioning Joins. *Proc. ACM on Management of Data (SIGMOD)* (2023).

Received April 2024; revised July 2024; accepted August 2024